

Large-Scale Development (LSD)

Assignment 3: Untersuchung der Architektur von Tomcat 6.0.53

Autoren: Felix Hefner, Max Jando, Severin Kohler

Stand: 12.10.2017

1 Einleitung

Im Rahmen der Vorlesung LSD (Large-Scale-Development) sollten die Autoren dieses Dokuments, die Architektur von Tomcat genauer untersuchen. Da die gesamte Analyse von Tomcat 6.0.53 zu umfangreich für die Vorlesung wäre, beschränkt sich diese Analyse auf die Beziehungen der Pakete beim Aufruf eines Requests an den Tomcat.

2 Vorgehen

Um die Architektur zu untersuchen, haben wir uns sowohl auf die statische Struktur und die Abhängigkeiten zwischen den Paketen als auch auf das dynamische Verhalten des Systems während der Laufzeit fokussiert.

2.1 Statische Analyse

Folgende Schritte wurden bei der statischen Analyse ausgeführt:

1. Mit dem Linux-Tool *grep*, wurde durch alle Dateien des *Source Code* iteriert und nach dem Schlüsselwort *import* gesucht.
2. Die Ergebnisse wurden mittels *sed* in ein geeignetes Format ($Klasse_a \rightarrow Klasse_b$) gebracht, das veranschaulicht, welche Klasse eine andere Klasse importiert.
3. Mit weiteren *grep*-Ausführungen wurden nach bestimmten Paketen gefiltert, bzw. nach allen Paketen, die **nicht** (*grep -v*) den Kriterien entsprechen.

4. Alle bisherigen Ausgaben wurden in eine .dot-Datei geleitet und mit dem Header `digraph {` und einer schließenden Klammer versehen, um sie für das Tool *dot* von *Graphviz*¹ vorzubereiten.
5. Abschließend wurde das Kommando `dot -Dpng input.dot output.png` ausgeführt, um einen Abhängigkeitsgraphen als .png zu erstellen.

2.2 Dynamische Analyse

Um das Verhalten von Tomcat während der Laufzeit zu untersuchen, haben wir zunächst ein paar Vorbereitungen durch Modifizierung des Codes vorgenommen. Wie wir aus der folgenden Abbildung entnehmen konnten, ist offenbar das Paket *org.catalina.connector* wichtig für Tomcat und hängt mit vielen anderen Paketen zusammen.

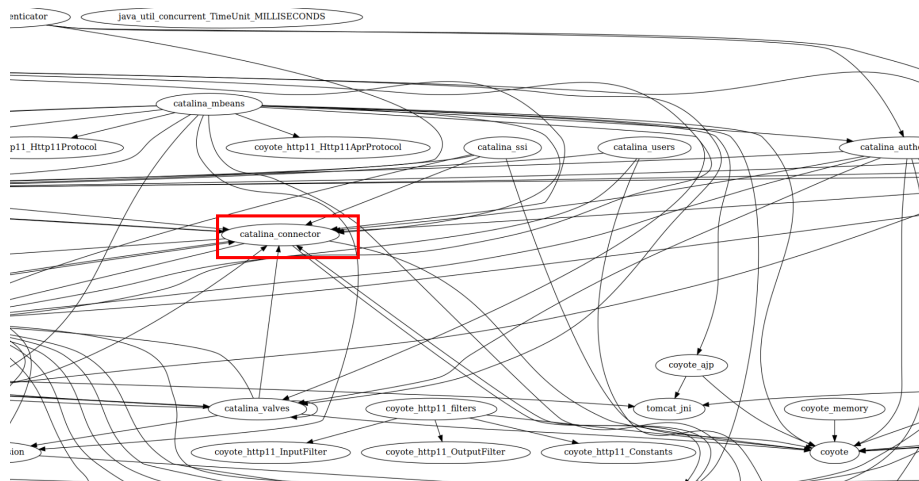


Abbildung 1: Ausschnitt aus dem Graphen der Abhängigkeiten sämtlicher Klassen

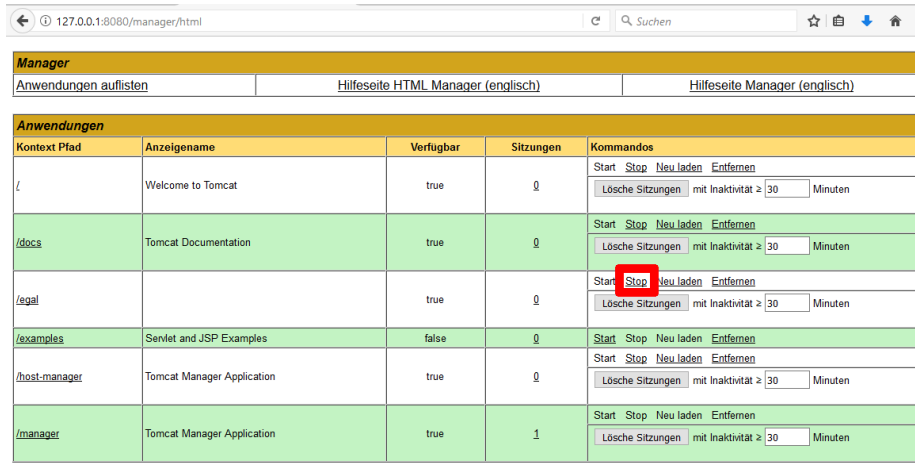
Deswegen hatten wir uns dazu entschlossen, dieses Paket als erstes zu untersuchen. Innerhalb des Paketes in der Klasse *Connector* haben wir uns mit einem Iterator sämtliche Bohnen (engl. Beans), die beim Starten angelegt werden, ausgegeben. Die auffälligste Bohne *RequestDumperValve*, aufgrund des im Namen enthaltenen Wortes *Request*², haben wir uns diese Klasse genauer angeschaut. Da die *RequestDumperValve* bei der Funktion *invoke()* mit Hilfe einer *ServletException* einen Fehler im Programmablauf aufzeigen kann, haben wir uns ein spezielles *Servlet*³ rausgesucht und näher betrachtet. Hierfür haben wir auf der

¹www.graphviz.org

²deutsch: Anfrage

³Tomcat Manager (stop-Funktion für Servlets)

Weboberfläche von Tomcat unter localhost:8080/ den Servlet-Manager aufrufen und bei unserem Test-Servlet *egal* auf den Stop-Link geklickt, wodurch die Stop-Funktion des *HTTPManagerServlet* aufgerufen wurde (siehe Abb. 2).



Manager				
Anwendungen auflisten		Hilfeseite HTML Manager (englisch)		Hilfeseite Manager (englisch)
Anwendungen				
Kontext Pfad	Anzeigenname	Verfügbar	Sitzungen	Kommandos
/	Welcome to Tomcat	true	0	Start Stop Neu laden Entfernen Lösche Sitzungen mit Inaktivität ≥ <input type="text" value="30"/> Minuten
/docs	Tomcat Documentation	true	0	Start Stop Neu laden Entfernen Lösche Sitzungen mit Inaktivität ≥ <input type="text" value="30"/> Minuten
/egal		true	0	Start Stop Neu laden Entfernen Lösche Sitzungen mit Inaktivität ≥ <input type="text" value="30"/> Minuten
/examples	Servlet and JSP Examples	false	0	Start Stop Neu laden Entfernen
/host-manager	Tomcat Manager Application	true	0	Start Stop Neu laden Entfernen Lösche Sitzungen mit Inaktivität ≥ <input type="text" value="30"/> Minuten
/manager	Tomcat Manager Application	true	1	Start Stop Neu laden Entfernen Lösche Sitzungen mit Inaktivität ≥ <input type="text" value="30"/> Minuten

Abbildung 2: Web-Oberfläche des Tomcat-Servlets-Manager

Da wir außerdem die Aufrufhierarchie bei einer Anfrage von eigenen *Java Server Pages*(JSP)-Files untersuchen wollten, haben wir ein kleines JSP erstellt und in den Java-Abschnitt das Auftreten einer absichtlichen Fehlermeldung⁴ eingebaut, welches die Aufrufhierarchie aller Klassen bis zur Stelle der Fehlermeldung ausgibt. Das nachfolgende Listing verdeutlicht die Aufrufhierarchie.

```
<html><body>
Es ist:
<% out.println((new java.util.Date()).toString());

    try {
        throw new Exception();
    } catch (Exception e) {
        e.printStackTrace();
    }
};
%>

</body></html>
```

Listing 1: Einfaches Servlet, in das eine Fehlermeldung eingebaut wurde.

⁴engl: exception

3 Ergebnisse der Architekturanalyse

Die folgenden Abhängigkeitsgraphen zeigen die Ergebnisse unserer dynamischen Architektur-Untersuchungen. Diese wurden mit dem Tool *dot* aus *Graphviz* gezeichnet.



Abbildung 3: Handling des Requests der Funktion stop der Servlet-Manager-Web-Oberfläche

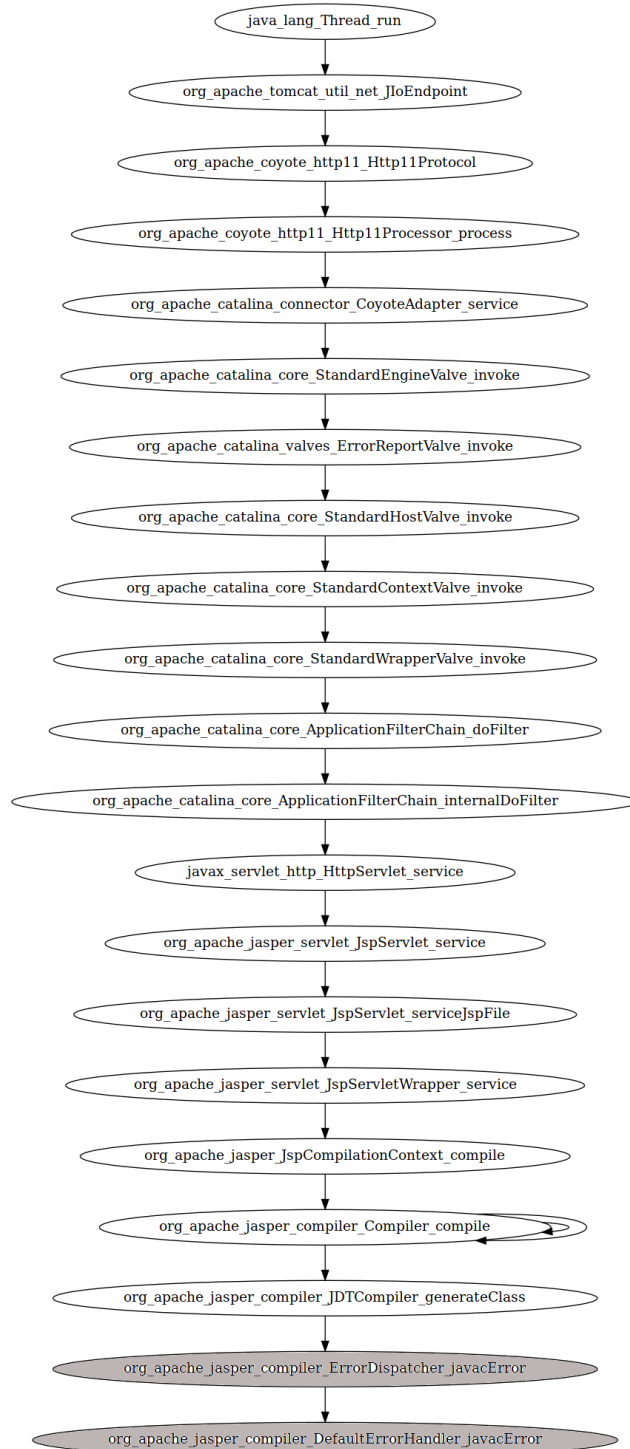


Abbildung 4: Handling des Requests beim Aufruf eines Servlets

4 Probleme

Während der Bearbeitung dieses Assignments ergaben sich unter anderem aufgrund der relativ weit gefassten Aufgabenstellung ein paar Probleme:

- Die generierten Graphen wurden schnell sehr komplex und zu unübersichtlich, sodass man sich auf die relevanten Teile fokussieren muss
- Wichtige Klassen als Einstiegspunkte für weitere Untersuchungen zu identifizieren bei solch einem komplexen Softwareprojekt ist nicht trivial.
- Bei der Aufbereitung der Ergebnisse mussten manche ausgeführten Schritte rekonstruiert werden, da nicht direkt während der Untersuchungen alles dokumentiert wurde.