Sparen mal anders

Testing von Apache Thrift im Vergleich zu REST-Services

Roger Meier, Michael C. Jaeger

Apache Thrift ist ein Framework, das eine Kommunikationsinfrastruktur für eine Vielzahl von Programmiersprachen bereitstellt. Es hat ähnliche Vorteile wie REST, nämlich ein leichtgewichtiges Erzeugen und Verarbeiten der Nachrichten, Verfügbarkeit als Open Source und Einsetzbarkeit für relativ viele Plattformen. Ist Thrift aber wirklich eine interessante Alternative zu REST-basierten Services? Wie lassen sich insbesondere Thrift-Dienste testen?

Interprozesskommunikation

Heutige Softwareservices, sei es als Webservices in Softwarelösungen oder bei Software-as-a-Service-Angeboten im Cloud-Umfeld, scheinen sich von SOAP abzuwenden, während leichtgewichtige Lösungen wie REST Zuspruch ernten. Im Gegensatz zu SOAP hat REST grundsätzlich den Vorteil, dass die ausgetauschten Informationen mit sehr einfachen Mitteln verarbeitet werden können: Statt einer XML-Nachricht, die durch SOAP definiert wird, werden bei REST die Informationen für einen Aufruf durch einen Ressourcen-Pfad und gegebenenfalls zusätzliche Parameter codiert. Weitere Daten können optional in den Nutzdaten, der Payload, der HTTP-Nachricht untergebracht werden. Bei der traditionellen Benutzung von SOAP muss jeder noch so kleine Aufruf in einer relativ wortreichen XML-Nachricht ausgedrückt werden. Dies bringt unter Umständen einiges an Aufwand bei der Erstellung und Verarbeitung der Nachrichten mit sich. Im Embedded-Umfeld ist dies unerwünscht, weil ein größerer Ressourcenverbrauch direkt in höherem Stromverbrauch resultiert - im Cloud-Umfeld mit vielen Nutzern erzeugt eine höhere Last direkte Mehrkosten bei einer lastabhängigen Abrechnung.

Mitarbeiter von Facebook haben eine weitere Technologie entwickelt, um eine effiziente und performante Kommunikation zwischen Prozessen zu bewerkstelligen: Es handelt sich um Thrift, ein Framework, dass eine IDL (interface definition language) definiert und gleichzeitig eine Kommunikationsinfrastruktur bereitstellt. Thrift ist ein Apache-Projekt, dem einiges Potenzial für eine breitere Verwendung bescheinigt wird. Wie die Autoren von Thrift auch im lesenswerten [ThriftWhitePaper] formulieren, gibt es bereits mehrere Technologien, die einen Methodenaufruf codieren, um die Informationen zwischen den Prozessgrenzen zu übertragen und dann am Zielort wieder zur Weiterverarbeitung zu decodieren. Die Besonderheiten bei Thrift im Gegensatz zu SOAP, Google Protocol Buffers, CORBA oder COM liegen wie bei REST im leichtgewichtigen Design bei der Erzeugung, Verarbeitung und Übertragung der Nachrichten, in der Verfügbarkeit als Open Source und in der gleichzeitigen Einsetzbarkeit für relativ viele Plattformen und Programmiersprachen.

Funktionstest von Thrift-Diensten

Aber wie bewährt sich Thrift im Alltag in eigenen Projekten? Für den technologieoffenen und neugierigen Softwareentwickler ist das hoch-performante Thrift bestimmt eine interessante Alternative zu REST-basierten Services. Schließlich setzt Facebook Thrift zur Kommunikation zwischen ihren Komponenten ein und beweist daher die Eignung in einem sehr anspruchsvollen Anwendungsfall. Die IDL, auf die hier nicht näher eingegangen werden soll, ist übersichtlich und profitiert sicherlich von Erfahrungen und Konzepten bestehender IDLs. Für einen Service lassen sich schnell die Schnittstellen anhand der Thrift IDL definieren. Auch wenn die Thrift-Dokumentation im Internet keine Rekorde bricht, stellt sie – immerhin rudimentär vorhanden – auch kein Hindernis für den Einstieg dar.

Da sich Thrift in die bestehende Entwicklungsinfrastruktur einfügen soll, müssen Thrift-Services im gleichen Maße getestet werden können, wie es auch Standard im Softwareentwickler-Alltag ist. Für REST-basierte Webservices können Lösungen für unterschiedliche Testziele als gemeinhin vorhanden vorausgesetzt werden.

Die REST-Seite

Wir betrachten zur Einführung Funktionstests gegen die Serviceschnittstelle: In der REST-Welt kann ein bewährtes Komponententest-Framework eingesetzt und mit einem HTTP-Client-Framework gekreuzt werden. Das Deployment des Service kann die Build-Umgebung übernehmen. Automatisiertes Deplyoment von Servicekomponenten ist in vielen Umgebungen Standard, um die automatisierte Ausführung von Tests zu ermöglichen. Soll eine Datenbank oder eine andere Form von Cloud-Speicher ins Spiel kommen, dann muss diese ebenfalls mit aufgestellt werden. Hierfür existieren alternative Ansätze, um benötigte Infrastruktur für automatisierte Softwaretests zu mocken.

Für Aufrufe der REST-Schnittstellen ist als Beispiel der Einsatz des Java-Projektes HTTPUnit von Meterware sehr kompakt und übersichtlich:

```
WebConversation wc = new WebConversation();
WebRequest     req = new GetMethodWebRequest( serviceUrl );
WebResponse   resp = wc.getResponse( req );
assertEquals(expectedValue, resp.getText());
assertTrue("Response code should be 2XX", resp.getResponseCode() < 300);
```

Der Aufruf unter Benutzung einer Javax-RS-Bibliothek wie zum Beispiel Jersey ist indes nicht viel komplizierter:

```
Client          client = Client.create();
WebResource webResource = client.resource( serviceUrl );
ClientResponse response = webResource.put(ClientResponse.class);
assertEquals(expectedValue, response.getEntity(String.class))
assertTrue("Response code should be 2XX", response.getStatus() < 300)
```

Unterschiedliche Client-Implementierungen nehmen sich aus der Perspektive des Testens nicht viel, wenn man sich im Standardrahmen bewegt.

Die Thrift-Seite

Da Thrift den Einsatz verschiedener Sprachen zulässt, wählen wir für unser Beispiel eine leichtgewichtige Variante, nämlich JavaScript. JavaScript ist mittlerweile eine der populärsten Programmiersprachen und eine von zurzeit neunzehn Sprachen, die von Thrift unterstützt werden. Das Testing unter JavaScript war zu Beginn sehr rudimentär aufgebaut. Doch mittlerweile gibt es auch für JavaScript einige etablierte Testsuites. QUnit besteht aus einer JavaScript- und einer CSS-Datei, die Tests sind sehr einfach aufgebaut:

```
test("Double", function() {
  equals(client.testDouble(3.14), 3.14);
});
```

Die erste Frage, die sich jedoch beim Einsatz von JavaScript stellt, ist die nach der Ausführungsumgebung. Niemand mag wiederholt manuell den Browser öffnen, um die Test-HTML-Seite aufzurufen. Im Weiteren dürfen auch vorhergehende Schritte wie das Starten des Servers nicht vergessen werden. In einem modernen Entwicklungsumfeld sollen diese Schritte automatisiert und in den Build-Prozess integriert werden, sodass die Funktionalität der JavaScript-Bibliothek und die generierten JavaScripts ohne manuelle Eingriffe verifiziert werden können.

Das Tolle an dem Ansatz ist, dass wir die notwendigen Funktionen bereits in der Open-Source-Community vorfinden. Im Einzelnen werden folgende Komponenten hierzu verwendet:

- Headless Browser (Projekt phantomjs) und Virtual Framebuffer (Projekt Xvfb),
- http-Server zur Auslieferung der JavaScript-, CSS- und HTML-Dateien basierend auf HttpComponents,
- HTTP/JSON Thrift Server (JavaScript unterstützt zurzeit nur Thrift JSON Protocol),
- ein ant-basiertes Build-Skript („build.xml“), welches von den darüber liegenden Makefiles aufgerufen wird, es soll die Abhängigkeiten prüfen sowie den Server und Client starten bzw. beenden.

Abbildung 1 veranschaulicht den Aufbau der Testsuite, welche nach dem Überprüfen des Vorhandenseins der Abhängigkeiten (phantomjs und Xvfb) den virtuellen Framebuffer, den Test-Server und anschließend den Browser startet. Das Resultat des Tests wird aktuell mit SUCCESS/FAILURE ausgewertet. Hier wäre künftig noch ein zu xUnit kompatibles Format wünschenswert.

Die Testsuite ist in der Datei lib/js/test/build.xml definiert, so werden für die Abhängigkeiten entsprechend des üblichen Vorgehens Properties gesetzt, phantomjs benötigt Xvfb und wird beispielsweise wie folgt detektiert:

```
<target name="phantomjs" depends="xvfb" if="xvfb.present">
  <echo>check if phantomjs is available:</echo>
```

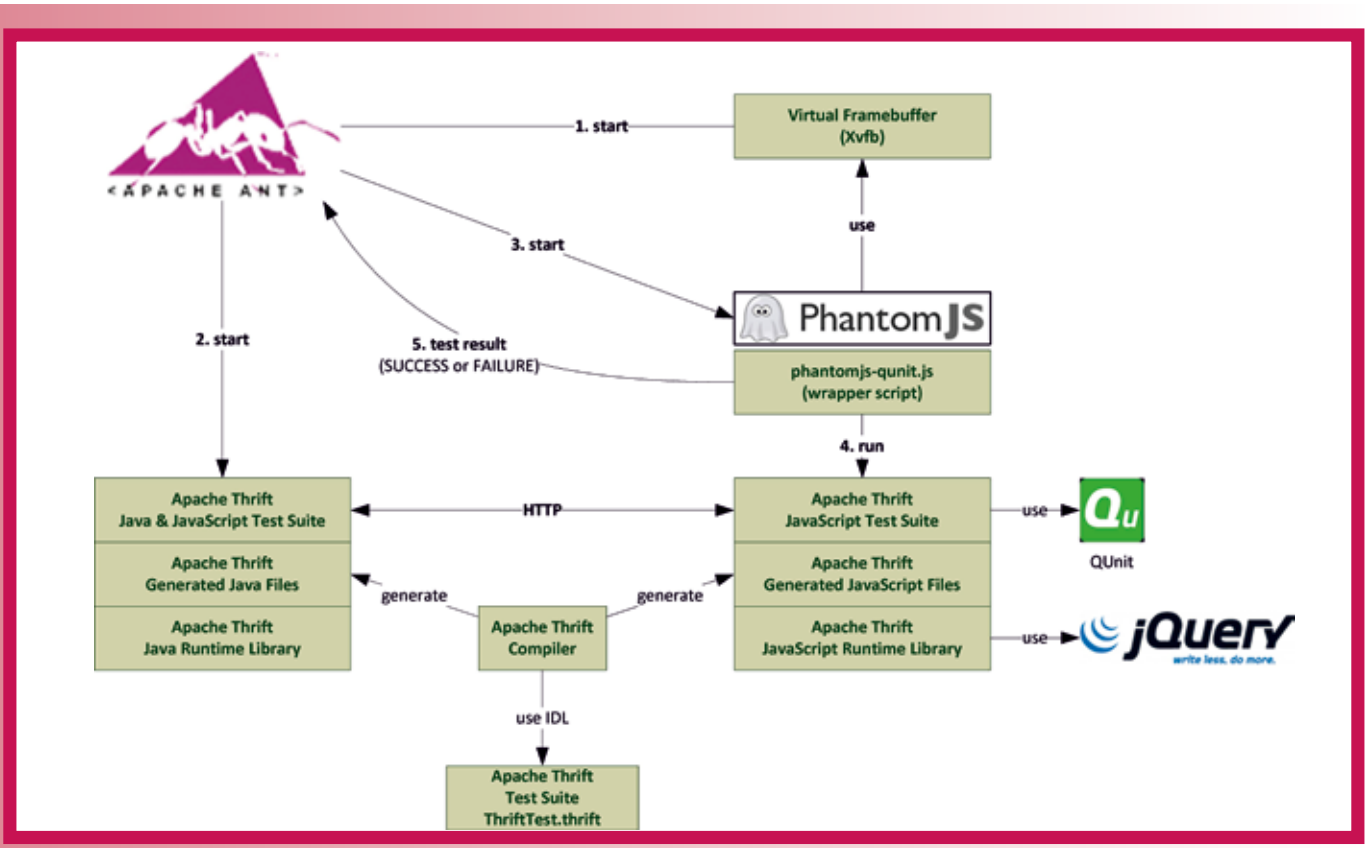

Abb. 1: Aufbau der Testsuite

```
    <exec executable="phantomjs"
          failifexecutionfails="no"
          resultproperty="phantomjs.present"
          failonerror="false"
          output="${build}/log/phantomjs.log">
      <arg line="--version"/>
    </exec>
  </target>
```

Der Thrift-Server, welcher für JavaScript das JSON-Protokoll auf dem http-Transport nutzt, basiert auf dem Apache-Http-Components-HTTP-Server und verwendet den bereits verfügbaren TestHandler der Java-TestSuite von Thrift.

Der ant-Task für den Unit-Test nutzt den ant-parallel-Task, um den Server und mit einer Verzögerung von zwei Sekunden den Client(Browser) zu starten. Dies sieht dann wie folgt aus:

```
<target name="unittest"
        description="do unit tests with headless browser phantomjs"
        depends="init, phantomjs, jstest, jslibs"
        if="phantomjs.present">
  <parallel>
    <exec executable="Xvfb" spawn="true" failonerror="false">
      <arg line=":99" />
    </exec>
    <java classname="test.Httpd" fork="true" timeout="10000"
      classpathref="test.classpath" failonerror="false"
                  output="${build}/log/unittest.log">
      <arg value="../" />
    </java>
    <sequential>
      <sleep seconds="2"/>
      <echo>Running Unit Tests with headless browser!</echo>
      <exec executable="phantomjs" failonerror="true">
        <env key="DISPLAY" value=":99"/>
        <arg line=
          "phantomjs-qunit.js http://localhost:8088/test/test.html" />
      </exec>
    </sequential>
  </parallel>
</target>
```

Das Resultat zeigt sich dann folgendermaßen:

```
unittest:
  [echo] Running Unit Tests with headless browser!
  [exec] 'waitFor()' finished in 1579ms.
  [exec] Tests completed in 1440 milliseconds.
  [exec] 90 tests of 90 passed, 0 failed.
  [java] Timeout: killed the sub-process
  [java] Java Result: -1

BUILD SUCCESSFUL
Total time: 1 minute 19 seconds
```

Eine weiterer erwähnenswerter Teil der JavaScript-Testsuite für Thrift ist die Integration von JSLint. Mit Hilfe von jslint-4java, einem Wrapper für JSLint von Douglas Crockford, können auch die Codequalität-Verifikationen in den Build integriert werden:

```
<target name="jslint" depends="resolve">
  <taskdef uri="antlib:com.googlecode.jslint4java"
           resource="com/googlecode/jslint4java/antlib.xml"
           classpathref="libs.classpath" />
  <jsl:jslint options="evil,forin,browser,bitwise,regexp,newcap,immed"
              encoding="UTF-8">
    <formatter type="plain" />
    <fileset dir="${genjs}" includes="**/*.js" />
    <fileset dir=".." includes="thrift.js" />
  </jsl:jslint>
</target>
```

Ausblick

Wir haben ein Thrift-Test-Framework auf Basis von Open-Source-Projekten aufgebaut und in das Projekt zurückgespeist. Durch den Einsatz von Open-Source-Projekten zeigen wir auch, dass es sich lohnt, aktiv an der Community teilzunehmen und Projekte zu unterstützen, um beispielsweise die Qualität von verwendeten Open-Source-Komponenten weiterzuentwickeln und sicherzustellen.

Zudem gewinnt JavaScript zunehmend an Bedeutung. Mit node.js wird JavaScript neu auch auf der Server-Seite eingesetzt; weiterhin wird node.js bereits auch von Thrift unterstützt. Leider sind momentan node.js und JavaScript für Browser unabhängig voneinander implementiert - der einzige gemeinsame Code ist innerhalb des Thrift-Compilers. Um das Testing in diesem Punkt zu perfektionieren, wäre ein JavaScript-Dependency-Framework wie beispielsweise require.js ein vielversprechender Ansatz, um beide JavaScript-Implementierungen zu vereinen und eine stärkere Modularisierung des JavaScript-Quellcodes zu forcieren.

Links

[HttpComponents] Apache HttpComponents http://hc.apache.org/
[jQuery] http://jquery.com/
[JSLint] http://www.jslint.com/
[JSLint4Java] http://code.google.com/p/jslint4java/
[phantomJS] http://www.phantomjs.org/
[QUnit] http://docs.jquery.com/QUnit
[Require.JS] http://requirejs.org/
[Thrift] Apache Thrift, http://thrift.apache.org/
[ThriftWhitePaper] M. Slee, A. Agarwal, M. Kwiatkowski, Thrift: Scalable Cross-Language Services Implementation, 2007, http://thrift.apache.org/static/thrift-20070401.pdf
[Xvfb] Mehr Infos zum Xvfb X-Server, http://www.wikipedia.org/wiki/Xvfb

Weiterführende Information

http://thrift.apache.org/

Roger Meier arbeitet im Hauptsitz der Siemens-Division Building Technologies Division in der Schweiz. Er beschäftigt sich mit den Bereichen Embedded-Linux-Plattformen, Webtechnologien und Open-Source-Software.
E-Mail: r.meier@siemens.com

Michael C. Jaeger ist bei Siemens Corporate Technology in München-Perlach. Dort arbeitet er im Bereich System Architecture & Platforms; Schwerpunkte sind Softwarearchitekturen, Patterns und Verteilte Systeme. Zuvor studierte er Technische Informatik an der TU Berlin.
E-Mail: michael.c.jaeger@siemens.com