QSE-ASE-08
Tenez
Helma Einführung

Philipp Naderer

21. Oktober 2009

Inhaltsverzeichnis

Kapitel 1

Grundbegriffe

1.1 Was muss man downloaden?

Download vom aktuellen Helma 1.7 Release Candidate geht unter folgender URL: http://adele.helma.org/download/helma/1.7.0-rc3/

Warum setzen wir schon Helma 1.7 ein? Mit Helma 1.7 kommen Unit Tests ins Helma Framework und die sind zwingend für unser Projekt erforderlich. Außerdem steht Helma 1.7 kurz vor der Veröffentlichung und bringt viele Verbesserungen zur Release 1.6.

1.2 Installation

Einfach das ZIP extrahieren und an einem Platz im Filesystem ablegen. Danach kann man in der start.bat bzw. start.sh die Ports festlegen, mit denen Helma gestartet wird. Ratsam: Port 8080 für HTTP. Danach das Startskript ausführen und schon läuft Helma.

Jetzt müssen wir die Management Konsole aktivieren. Unter der URL http://localhost:8080/manage/makekey lässt sich ein User + Passwort generieren. Da wir nur lokal arbeiten, empfiehlt sich admin mit Passwort admin. Die so generierte Zeile muss man am Schluss in den server.properties eintragen.

Wenn man die Zeile eingefügt hat, kann man unter http://localhost:8080/manage/main auf die Adminkonsole zugreifen. Links sieht man alle aktuellen Applikationen, die von Helma gefunden werden. Man kann sie so einfach neu starten oder mit "flush"den Helma-Cache entleeren. Nach einem Flush holt sich Helma alle Objekte frisch aus der Datenbank.

Nun läuft Helma und man kann eine erste Hello World Applikation erstellen.

1.3 Hello World

Im /apps-Verzeichnis von Helma legen wir eine neue Applikation an, in dem wir einfach einen neuen Ordner mit dem Namen der Applikation erstellen. In unserem Fall einfach "helloworld". Wenn man jetzt in die Management Konsole schaut, sieht man danach schon die Applikation in der Übersicht.

Es gibt nun 2 Wege zum Starten: In der Management Konsole, wobei dieser Weg kaum benutzt wird. Oder wir tragen die Applikation in die Datei apps.properties ein. Der Mountpoint legt fest, unter welcher URL die Applikation eingehängt wird. In unserem Fall http://localhost:8080/hello/ - ruft man diese URL zum jetzigen Zeitpunkt auf, sollte Helma noch einen Error liefern, da wir ja noch keine Ausgabe programmiert haben. Das Repository legt fest, wo der Code zu finden ist, den Helma zur Applikation laden soll. Mit Hilfe der Nummer kann man eine Reihung festlegen, also z.B. Code aus unterschiedlichen Verzeichnissen laden. Das wird später im Projekt eine wichtige Rolle spielen, denn so kann man das Datenmodell und die Business Logic in ein Verzeichnis /base ablegen und z.B. das Frontend in ein Verzeichnis /frontend und beide als Code Repository angeben. Also z.B. tenez.repository.0 = ./apps/tenez/base und tenez.repository.1 = ./apps/tenez/frontend!

Listing 1.1: apps.properties Eintrag für HelloWorld

```
1 # Hello World
2 helloworld
3 helloworld.mountpoint = /hello
4 helloworld.repository.0 = ./apps/helloworld
```

Sobald diese Zeilen in die apps.properties eingefügt sind, ist die Applikation gestartet. Um sie zu beenden bzw. zu deaktiveren, kommentiert man einfach die erste Zeile mit # aus. Helma bekommt diese Änderung relativ rasch – meist nur eine Sekunde – mit und stoppt die Applikation.

Nun erstellen wir ein Verzeichnis Root in unserer Applikation, also /apps/helloworld/Root. Nun haben wir das Wurzelklasse für die Applikation angelegt und können hier Actions anlegen, die dann aufgerufen werden können. Die main_action ist dabei wie eine index.html-Datei, sie ist also die Default-Action für einen bestimmten Pfad.

Wir führen folgenden Dateinamens-Konvention ein: Actions werden immer die Datei PROTOYPE.actions.js geschrieben, Macros ist in PROTOTYPE.macros.js und Business Logic in PROTOTYPE.js Unsere main_action muss also in die Datei apps/helloworld/Root/Root.actions.js abgelegt werden.

Listing 1.2: Root.actions.js

```
1 var main_action = function() {
2    res.write("Hello World!");
3
```

```
4    if (req.params["name"]) {
5       res.write("Your name is: " + req.params["name"]);
6    }
7 }
```

Unter http://localhost:8080/hello/ wird nun "Hello World!äusgegeben. Geben wir zusätzlich einen GET- oder POST-Parameter name an, wird dieser zusätzlich ausgegeben.

1.4 JavaScript und Objekte

JavaScript heißt eigentlich ECMAScript und ist auch unter diesem Namen standardisiert. Es ist schlanke, dynamisch typisierte, objektorientierte, Skriptsprache. Es gibt keine Klassen, jedoch basieren Objekte auf Prototypen. Aus Wikipedia: "Prototypenbasierte Programmierung, auch als klassenlose Objektorientierung bekannt, ist eine Form der Objektorientierten Programmierung, die auf das Sprachelement der Klasse verzichtet. Objekte werden durch das Clonen bereits existierender Objekte erzeugt. Beim Clonen werden alle Eigenschaften (Attribute und Methoden) vom Prototyp-Objekt geerbt und durch eigene Eigenschaften erweitert."

Daher: Alles, was dem Prototype eines Objekts hinzugefügt wird, wird quasi der Klasse hinzugefügt. Dies ist vergleichbar mit statischen Methoden bzw. statischen Variablen in Java.

Listing 1.3: Prototype in der Praxis

```
1  function Person(name) {
2     this.name = name;
3     // lastAdded bzw. whoAmI sind fuer alle von aus
4     // Person erzeugten die gleichen Referenzen!
5     // Vergleichbar mit static in Java
6     Person.prototype.lastAdded = name;
7     Person.prototype.whoAmI = function() { return this.name; };
8  }
9
10 var rudi = new Person("Rudi");
11 var otto = new Person("Otto");
12
13 otto.name = "Otto der Grosse";
14
15 console.log(otto.whoAmI());
16 console.log(otto.lastAdded);
17 console.log(rudi.lastAdded);
18 console.log(rudi.whoAmI());
19
20 /* Ausgabe im Firebug */
21 Otto der Grosse
22 Otto
23 Otto
24 Rudi
```

Versionsprotokoll

v1.0	23.10.2009	Philipp Naderer	Erste Version vom Tutorial