Server-side JavaScript with the Helma application server

EASY READER

The powerful Helma application server brings new powers to the JavaScript language. We'll show you how to use Helma to build a simple RSS reader. **BY STEPHAN SCHMIDT**

JavaScript began as a dynamic scripting language for the browser. The JavaScript language is prototype-based with syntax reminiscent of C or Java. Because JavaScript includes higher order closures and functions, it is referred to as a functional language, as are the languages Lisp and OCaml. JavaScript became popular because of its use in browsers, but it eventually gained a bad reputation for language and library incompatibility. Thanks to standardization through the vendor-neutral ECMAScript standard, as well as new browser libraries such as Yahoo YUI, Mochikit, or Mootools, the picture is starting to change, and JavaScript is gradually becoming a mature programming environment.

Now that JavaScript is coming of age, developers are starting to notice possibilities for JavaScript on the server. Netscape once used JavaScript server-side under the moniker LiveWire, but the idea failed to gain widespread support. Now JavaScript is starting to attract a bigger crowd because of its use in AJAX and frameworks such as Ruby on Rails. This growing crowd of developers has helped JavaScript achieve the critical mass necessary for its breakthrough into mainstream development.

JavaScript on the Server

Helma [1] is a JavaScript-based application server that has been in use for many years. In this article, I investigate Helma by showing how you can use it to build a simple RSS reader.

Web applications for Helma can be written in JavaScript, and in Java, if needed. As an application framework, Helma includes everything you need – from an automatic database interface to a web framework. The project uses its own BSD-style license, known as the Helma License, which permits distribution and modification.

The Helma core is written in Java, as is the related JavaScript implementation, Mozilla Rhino [2]. Rhino transparently builds Java from JavaScript applications. Helma was developed and released in 2001 by Hannes Wallnoefer, who works

Installation

To install the current Helma version 1.5.3, you need Sun Java version 1.4 or newer. After you dowload Helma [3], the installation is straightforward: unpack the zipped tarball and start the service by executing the *start.sh* shell script in the Helma directory. You may need to modify the *JAVA_HOME* environmental variable, which points to the installed JRE/JDK. You should now be able to access the Helma "Welcome" application (Figure 2) supplied with the server at *http://localhost:8080*.

Approaches to deploying Helma on the Internet are various, such as using an Apache proxy or the Apache *mod_jk* module. The integrated web server is powerful enough to support direct Helma deployment on the Internet for many applications.

Listing 1: apps.properties

```
01 rssreader
02 rssreader.mountpoint = /
03 rssreader.repository.0 = apps/
   rssreader/code/
04 rssreader.repository.1 =
   modules/helmaTools.zip
05 rssreader.static = apps/
   rssreader/static
06 rssreader.staticMountpoint =
   /static
07 rssreader.staticHome = index.
   html,default.html
08 rssreader.staticIndex = true
```

for the Austrian broadcasting corporation ORF. Wallnoefer maintains the project to this day. The Helma framework can look back on years of successful deployment on high-volume websites.

Best of Both Worlds

Because the server is implemented in Java, Helma has some advantages over languages such as PHP, Python, and Ruby. Helma can use any Java library, giving developers a vast gallery of application components. Although the performance of a Helma application is excellent, parts of the application can be converted to Java for better performance.

In contrast to Ruby on Rails, for example, Helma is an application container that manages one or multiple Helma applications. You can start and stop the applications using a web-based management interface. In fact, the web interface can do much more – introspection gives developers the ability to view the documented API for the application at any time and to inspect the current objects on the active server (Figure 1). This feature is useful for maintenance and troubleshooting of Helma applications.

A scripting language removes the need to compile an application. Like other scripting languages, you can modify the code and then reload the page in your browser to see the changes. This speeds up the development process, especially in the case of larger projects that require longer build and deploy times.

As is evidenced by its deployment with ORF and other high-volume websites, Helma scales well. This scalability is due to simple but effective caching, as well as running the server on state-of-the-art virtual machines for Java.

Sample: RSS Reader

To explore various Helma components and their interactions, I'll look at a sample application that reads multiple web-based RSS feeds, lists them in an overview, and displays individual feeds.

The first step is to configure the application in Helma. To do this, you need to modify *apps.properties* in your Helma directory. Listing 1 shows what the file looks like. The file starts with the name of the application, *rssreader* in this example. Mountpoint describes the URL for accessing the Helma application on the server, which is / in this case. The repositories point to the JavaScript source code, and the *static* directive configures static pages for the application.

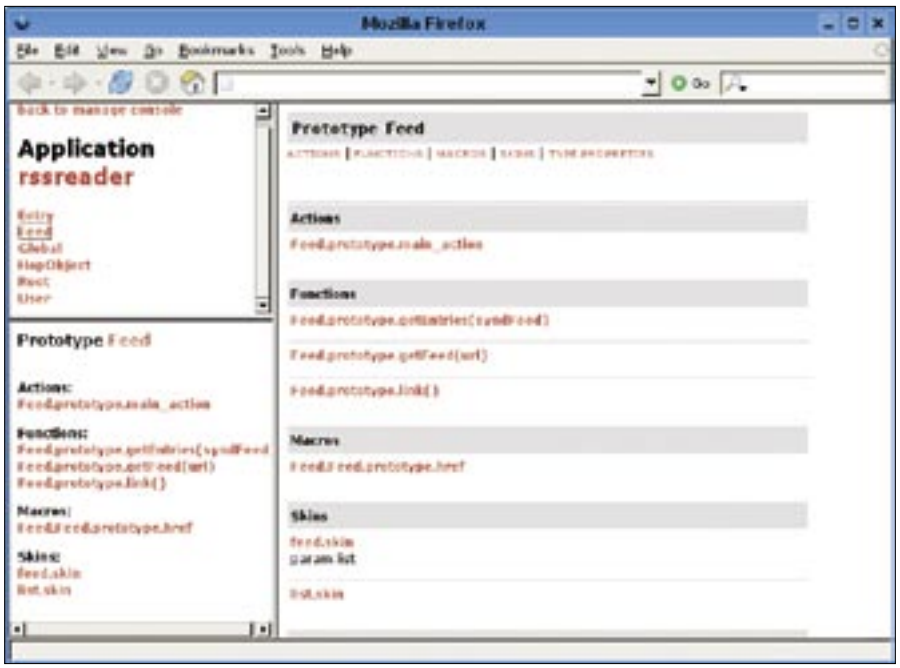

Figure 1: Helma's web-based management interface showing the API for an application in a Javadoc-style view.

You can now write a working "Hello World" application. To do so, create the subdirectory *rssreader/code/Root* below the *apps* directory. Then create a *Root.js* file in the *apps/rssreader/code/Root* directory with the following contents:

```
Root.prototype.main_action = ➥
function() {
  res.write ➥
  ("<h1>Hello World!</h1>");
}
```

When you surf to *http://localhost:8080/*, your browser should output a *Hello world!* text message. The RSS reader stores the configuration with the RSS URLs in a database. Additionally, the sample program uses the free Java library, Rome, which can read and write various versions of RSS and Atom feeds.

It is easy for Helma to access any Java library. To add accessories of this kind, copy the required *.jar* files to your Helma *lib/ext* subdirectory. For example, download Rome [4] and unpack the compressed archive and copy *rome.jar*. Rome also requires *jdom.jar*, which is available from jdom.org [5].

Follow the same approach to install a JDBC database driver. For the sample application, you need a MySQL database, which Helma does not include. A driver is available [6] and you need only to copy the *.jar* file. When launched, Helma will automatically detect the libraries and bind them to the Java classpath (Figure 3).

Now bind the database. Helma introduced transparent mapping of objects to database tables before this became a popular technique in other development environments.

To allow Helma to find the database, you need a *db.properties* file below the Helma directory. The file for my RSS reader looks like this:

```
rssreader.url = jdbc:mysql://
Hostname/rssreader
rssreader.driver= ➥
org.gjt.mm.mysql.Driver
rssreader.user=Username
rssreader.password=Password
```

Listing 2: type.properties

```
01 _db = rssreader
02 _table = Feed
03 _parent = root
04 _id = id
05
06 url = url
07 title = title
08 description = description
```

The data source name is *rssreader*. I will need to refer to this data source later to map objects to the database. This example uses MySQL as its database and specifies the database host, the correct driver class, the database username, and the password. You can test the database connection using the SQL shell in Helma's web-based management interface.

In Helma version 1.5.3, you had to set up the database manually. Automatic database generation is under discussion for the imminent Helma 2.0 in the context of rapid prototyping. If you don't want to do without it, take a look at Rabbit, a Helma application that automatically sets up database tables and Helma objects. In the case of the RSS reader, I just need a single table for the feeds:

```
CREATE TABLE Feed (
  id MEDIUMINT(10) NOT NULL,
  url  TINYTEXT,
  title  TINYTEXT,
  description  TINYTEXT,
  PRIMARY KEY (id)
);
```

The next few paragraphs describe some programming details for the RSS reader application. Working application code is available from the *Linux Magazine* download page [7].

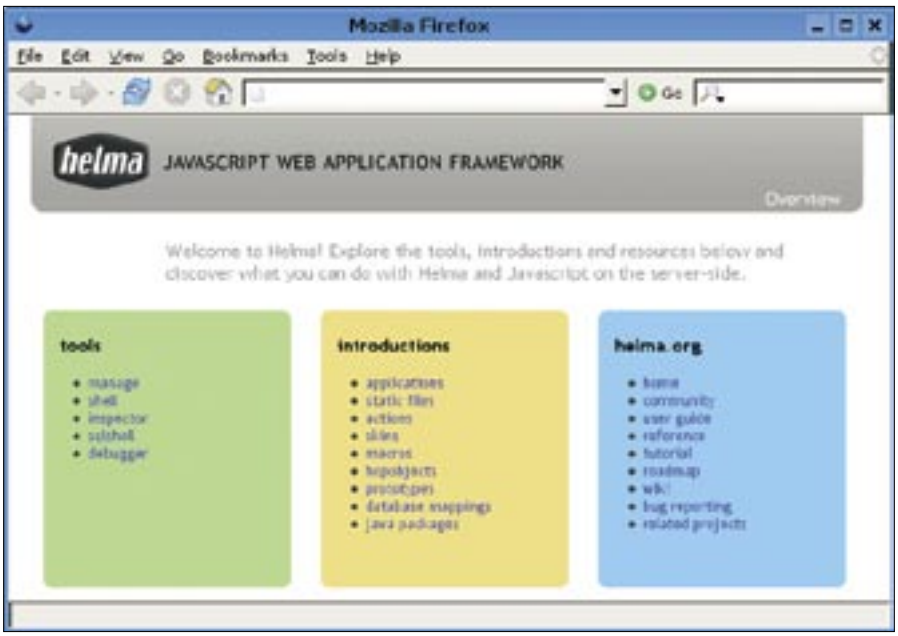

Figure 2: The Helma "Welcome" application greets you when you first access the server.

Feed Object

The RSS reader needs a feed object. In Helma, these objects are known as HOP (Helma Object Publisher) objects. The feed object is mapped to the database using a *type.properties* file (Listing 2). Attributes starting with an underline are internal Helma attributes. They should be self-explanatory with the possible exception of *_parent*.

The parent attribute defines the parent object for feed – in this case, it is the

Listing 3: Feed.js

```
01 Feed.prototype.main_action = function() {
02   var syndFeed = this.getFeed(this.url);
03
04   if (this.title != syndFeed.title) {
05     this.title = syndFeed.title;
06   }
07   if (this.description != syndFeed.description) {
08     this.description = syndFeed.description;
09   }
10
11   var entries = this.getEntries(syndFeed);
12   var list = "";
13   for (var i = 0; i < entries.length; i++) {
14     list += entries[i].renderSkinAsString("entry"
   );
15   }
16
17   this.renderSkin("feed", { list: list });
18 }
19
20 Feed.prototype.getEntries = function(syndFeed) {
21   var result = [];
22   var entries = syndFeed.getEntries();
23   for (var i = 0; i < entries.size(); i++) {
24     result[i] =  new Entry(entries.get(i));
25   }
26   return result;
27 }
28
29 Feed.prototype.getFeed = function(url) {
30   var Url = Packages.java.net.URL;
31   var SyndFeed = Packages.com.sun.syndication.
   feed.synd.SyndFeed;
32   var SyndFeedInput = Packages.com.sun.
   syndication.io.SyndFeedInput;
33   var XmlReader = Packages.com.sun.syndication.
   io.XmlReader;
34
35   var input = new SyndFeedInput();
36   var syndFeed = input.build(new XmlReader(new
   Url(url)));
37   return syndFeed;
38 }
39
40 Feed.prototype.href_macro = function() {
41   return this.href();
42 }
```

root of the Helma application. You can organize the HOP objects in a tree and then access them by means of hierarchical URLs.

In Helma, the functionality of an application lies in its HOP objects, macros, and actions. The RSS reader needs an action that reads and displays the feed for a URL on the Internet (Listing 3). The action is described in the *Feed.js* file.

When the URL for the object is called, an action titled *main_action* is triggered. Calling a URL of, say, *http://localhost:8080/1/* looks for an object with an ID of *1* below the root; in this feed example, this is the first database entry. The entry is mapped to the feed object, which is then loaded and made available to the main method as *this*. The method fetches the feed using the *getFeed* method, which uses the *Package* object to access the Java classes in the Rome library. For the sake of convenience, all the Java feed entries are encapsulated in JavaScript. *Entry.js*, which passes the Java object to the constructor and stores the attributes, handles this step.

Although Helma can access Java, use of native JavaScript objects is simpler. The *renderSkinAsString* method converts the entries to HTML. The method uses HTML formattings from *entry.skin* (Listing 4).

Before this happens, the application saves the title and description of the current feed in the *Feed* HOP object.

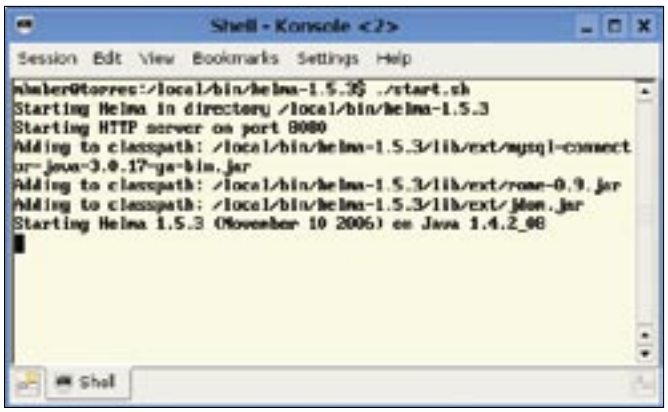

Figure 3: Helma automatically binds new Java libraries to the classpath.

Reuse Code

The JavaScript code I have discussed thus far, with its actions and skins, runs inside the Helma server. If you intend to use code for output that is visible to the client, with HTML for its output, you need to rewrite the code. The Model View Controller (MVC) approach is useful for abstracting the output. MVC separates the output into a model comprising an adapter, a controller, and a view of the data. The list of all feeds was implemented using a model of this kind (see Listing 5).

A *TableRenderer* uses the model to access individual feeds. To allow this to happen, the *TableModel* has methods for accessing cells and lines. The renderer walks through the cells and columns and

Listing 4: entry.skin

```
01 <p><a href="<% feed.href %>">
02   <% feed.title suffix=@,"
   %></a>
03   <% feed.description %>
04   ( <% feed.url %> )</p>
```

converts the data into HTML output. The model is implemented in *HopTableModel*, which converts cell access into access for a list of HOP objects. The same object, *TableRenderer*, could just as easily run client-side with a model that maps cell access to JSON to output feeds. JSON is a JavaScript data representation standard that serves a purpose similar to XML; developers use JSON to exchange data between AJAX applications and other programs.

If you abstract your applications, you can use much of the same code for the presentation logic on the server and the client. If you are using the same presentation logic, business logic can gradually be swapped out onto the client, dynamically and on demand. This approach reduces the effect of network latency and relieves the server load. An application designed like this scales more easily.

Helma has much more to offer. In addition to application management and introspection tools, Helma includes additional applications that speed up the development process. A shell gives developers JavaScript-based access to the server, and the SQL shell supports database access. The debugger makes it easier for developers to find errors in JavaScript applications.

Looking Ahead

The new 2.0 version of Helma is scheduled for release in 2007. The Helma 2.0 project is a complete rewrite. The developers intend to keep the strengths of the 1.0 line and take insights from many years of deployment into consideration. New scripting, extended skins and templates, and the capabilities of newer versions of the Rhino JavaScript implementation are the main focuses of Helma 2.0.

The Helma Community has not been twiddling its thumbs: Jala [8] is a new library for application development in Helma. The Jala library, which is based on work by ORF, was released recently as open source. It supports convenient extensions and wrappers to integrate key Java libraries for localization, Captchas, or RSS. Jala is characterized by excellent code quality and good documentation. Where is the trend headed? The breakthrough of mobile objects and mobile code is an exciting prospect for JavaScript. As mentioned earlier, Helma and JavaScript support the use of the same code both browser- and server-side.

Things heat up when you dynamically pass JavaScript objects to the browser, which lets developers modify applications at run time. This means more flexible, dynamic, and interactive web applications. JavaScript will continue to conquer the browser.

Whereas JavaScript projects have historically implemented the graphical presentation of an application, the next step will be to offload business logic onto the browser. More powerful browser-side JavaScript engines will contribute to this trend. And don't underestimate the importance of an open source contribution to Mozilla by Adobe: Tamarin, a just-in-time compiler for JavaScript with a virtual machine.

JavaScript has been hiding its light under a bushel for too long; the time has come for application developers to start taking JavaScript seriously. ■

Listing 5: Model View Controller

```
01 var TableRenderer = function(model) {
02   this.model = model;
03
04   this.render = function() {
05     var str = "";
06     str += '<table border="1"><thead>';
07     for (var column = 0; column < model.columns; column++) {
08       str += "<th>"+ model.getColumnName(column) + "</th>";
09     }
10     str += "</thead><tbody>\n";
11     for (var row = 0; row < model.rows; row++) {
12       str += '<tr>\n';
13       for (var column = 0; column < model.columns; column++) {
14         str += "<td>" + model.get(column, row) + "</td>\n";
15       }
16       str += "</tr>\n";
17     }
18     str += "</tbody></table>\n";
19     return str;
20   }
21 }
```

INFO

[1] Helma project homepage: *http://www.helma.org/*

[2] Mozilla Rhino: *http://www.mozilla.org/rhino/*

[3] Helma downloads: *http://helma.org/download/*

[4] Java RSS library, Rome: *https://rome.dev.java.net/*

[5] JDOM: *http://www.jdom.org/downloads/index.html*

[6] JDBC driver for MySQL: *http://www.mysql.com/downloads/api-jdbc-stable.html*

[7] Listings online: *http://www.linux-magazine.com/Magazine/Downloads/84*

[8] Jala project with useful libraries for Helma: *https://opensvn.csie.org/traccgi/jala/wiki/Download*

THE AUTHOR

Stephan Schmidt has been a software developer for more than 20 years, working as project manager and head of development of various mid-sized enterprises. Today, Stephan is a developer and project manager with Fraunhofer FIRST. His major focuses are Java and various scripting languages from Ruby to JavaScript.