Helma Documentation

Table of Contents

Helma Documentation

Michael Platzer

<michael.platzer@knallgrau.at>

Michael Jakl

Table of Contents

Introduction

What is Helma?

Helma is a scriptable Java Web Application Server.

Basically the web application market can be divided into two segments. On the one hand, there are the server side (interpreted) scripting languages (like Perl and PHP), which are being supported and used by thousands of independent developers ever since the early days of the dynamic web. On the other hand, there are the web application servers (like JBoss, Tomcat, IBM Websphere), generally developed, supported and sold by large companies, requiring the in−depth knowledge of strictly−typed, compiled languages as Java or C#. With

Helma being a "scriptable Application Server", it tries to bring together the benefits of both worlds.

Helma is a mature piece of software, which has been in production use for large, high−traffic news & community sites since years.

Helma is mainly being developed and maintained by Hannes Wallnoefer, a developer that is already well−known in the open−source community for providing the Java implementation of XML−RPC, which by now became part of the Apache foundation.

Why Helma?

So, what are the reasons to choose Helma in favour of another application platform?

1. Applications are (generally) being scripted in JavaScript (resp. EcmaScript), a standardized, well−known, well−established, easy−to−learn scripting language.
2. Any available Java−library can be made accessible for the scripting environment.
3. Helma forces an object−orientated application design.
4. Helma provides a transparent database−mapping, freeing the developer from hassling around, and polluting the code with SQL. As a side−effect, the relational database becomes substitutable at any point by any other JDBC−accessible database. Helma can even work without a relational database, by falling back to a file−based XML−database.
5. Helma is Java. Scripted applications are compiled to Java, and server−side Java has proven to be quite fast and reliable, at least with the more recent Runtime Environments.
6. Helma provides a smart caching mechanism, which minimizes the number of submitted select−queries to the database, by keeping recently−used objects in memory. This simple mechanism allows Helma to serve high−traffic sites, without developers even start thinking about performance tuning.
7. Compared to other web application servers Helma is a light−weight (~1,5MB download), has a rather small memory footprint, and is fast on start−up.
8. Helma is embeddable. It can be used to write a web frontend to existing Java applications.
9. If one Helma−server proves to be not sufficient for the request load, more servers can be easily clustered together. Again without requiring developers to rewrite their applications.
10. Scripted code is compiled to Java instantaneously, i.e. as soon as the modified source file is saved to disk. No need for developers to "compile and run" the whole application, before they can see their modifications in action.
11. Helma provides a smart templating mechanism, which forces a strict separation of content, logic and presentation.
12. Helma comes along with numerous extensions, which are usually required for developing web applications. These allow developers to easily

    a. send emails
    b. and write text−files
    c. and manipulate images
    d. a XML−RPC−client and −server
    e. access databases
    f. to a FTP−server
    g. generate Flash−Files
    h. much more.
13. Helma provides transparent session management, i.e. developers are freed from the task to assign requests to sessions.

14. Developers can easily generate clear, intuitive, easy−to−remember, search−engine−friendly URLs.
15. A scheduling mechanism is integrated into Helma, which allows the automated execution of scripts at any given time.
16. Helma is open−source, and comes with a liberal BSD style license. Its source−code can be studied, modified (and also distributed) by anybody who wishes to. No license costs are being charged, neither for private, nor for commercial use. All the advantages of open−source software apply to Helma. See the License itself: http://helma.org/download/license/.

What is Helma really?

The technology−affine reader might wonder by now, what Helma really is. The Helma distribution consists of multiple Java libraries, most of them provided by third−party, all having their own specific task.

These libraries are:

helma.jar
: The core of Helma, which brings together all of the following libraries and is in charge of Application Management, Code loading, Request Path Resolution, Session Management, DB mapping, Caching, Templating, Scheduling, and more. Take a look at the JavaDocs (or the source code) to get an in−depth view of Helma.

rhino.jar
: The JavaScript to Java compiler, which is actively developed by the Mozilla foundation. Read more about Rhino at http://www.mozilla.org/rhino.

jetty.jar
: The Jetty Server, which is also integrated with JBoss for example, provides a powerful and stable web server, as well as the AJP13−connection to integrate Helma with (nearly) any other web server (via the mod_jk module provided by the Apache foundation). See http://jetty.mortbay.org.

jimi.jar
: An image−manipulation library provided by Sun.

apache−dom.jar
: Providing a XML− and HTML−DOM−Parser.

mail.jar
: Sun's official JavaMail library.

crimson.jar
: a XML−SAX−Parser.

netcomponents.jar
: Providing (among other things) a Java FTP−client.

servlet.jar
: Providing the Java Servlet Api.

xmlrpc.jar
: Implementation of a XML−RPC−client and server.

commons−fileupload.jar
: File Upload Library. See http://jakarta.apache.org/commons/fileupload/.

activation.jar

commons−logging.jar

User's Guide

Setting up Helma

Obtaining Helma

Helma can be obtained in various forms from http://www.helma.org. Recommended for beginners are the latest stable Helma−packages (either as .zip or .tgz), which also provide a number of demo applications, to demonstrate the power of Helma. Simply download that package, and unzip it anywhere on your disk.

As an alternative it is possible to download the Helma−source−package, and compile Helma on your own. Experienced users, who want to test the latest developments, might want to download a nightly snapshot of Helma (http://adele.helma.org/download/helma/nightly/), or even fetch Helma (source) from the CVS repository (http://helma.org/development/cvs/howto/).

Prerequisites

The only prerequisite to get Helma up and running is a Java Runtime Environment version 1.3 or higher. In order to use Helma's new image manipulation code a version 1.4.1 or higher is required. Sun's JRE 1.4.2 or 1.5 are recommended for production use anyways.

In order to generate JavaDocs or to compile Helma from source, a JDK is required.

General File Structure of Helma

After download and unzipping the Helma−package, you will find the following files and directories:

`apps`
: The default directory for Helma to look for application source code.

`lib`
: In this directory all standard Helma−libraries are stored, and loaded at startup.

`lib/ext`
: A directory to place additional Java−libraries, which should be available in the scripting environment. This is also the put the JDBC−drivers.

`licenses`
: A directory containing licenses of third−party−libraries, that are packaged together with Helma.

`scripts`
: This directory contains useful scripts to run Helma as a service.

`static`
: The default directory for Helma to look for static content.

`launcher.jar`
: A helper Java−library, which is actually started by the start−scripts, and which takes care of the loading of all the other libraries.

`apps.properties`
: This file defines the applications to be started and also contains certain properties for these applications.

`server.properties`
: This file can contain server−wide settings.

`hop.bat`
: A start−script for Windows−systems.

`hop.sh`
: A start−script for Unix−based−systems.

The following directories will be created after starting Helma for the first time:

`db`
: The default directory for Helma to keep its file−based XML−database.

`log`
: The default directory for Helma to store its log−files.

If you downloaded the Helma−source−package, you will find the following additional directories:

`build`
: Contains the Java build−tool ant, and build−settings to compile Helma from sources, which is done by calling '***[HelmaHome]/build/build.sh jar***' on Linux, resp. ' ***[HelmaHome]/build/build.bat jar***' on Windows. You will probably need to set the variable JAVA_HOME within that script to the location of your installed JDK for this to work

`classes`
: Contains the compiled Java−classes

`docs`
: Contains the generated JavaDocs.

`src`
: Contains the source code for Helma, i.e. the helma.jar.

Starting / Stopping Helma (Development Setup)

Helma can be started by executing hop.bat on Windows, respectively hop.sh on Unix−based systems. If java is not present at the command line, or JAVA_HOME is not set as a system variable, the start−script needs to be edited accordingly, i.e. the JAVA_HOME needs to be set to the java installation directory.

Helma will log its output by default to several files in the log−directory. For a development setup it is useful to have all output being directed to the standard output, i.e. the console. This is achieved by adding the line 'logDir = console' to the file `[HelmaHome]/server.properties`. Helma needs to be restarted for this change to be effective.

The default setting for Helma is to start a web server port listening on port 8080, which means that you should be able to access Helma after startup at http://localhost:8080/ and/or at http://127.0.0.1:8080/.

If you open the start−script in a text−editor, you will see several startup−options that can be specified for Helma:

HTTP_PORT
: If this variable is specified Helma will start a web server listening on that port.

XMLRPC_PORT
: If this variable is specified Helma will start a XML−RPC−server listening on that port. Just needs to be set, if you want to make certain functions callable via XML−RPC.

AJP13_PORT
: If this variable is specified Helma will start a AJP13−listener on that port, which is just required to connect Helma to another Web Server via mod_jk. Port 8009 is the standard port for AJP13.

RMI_PORT
: If this variable is specified Helma will start a RMI−listener on that port.

JAVA_HOME
: As mentioned before, this can/should point to the base directory of the Java installation directory to be used. The start script will look for the directory 'bin' and a file named 'java' (resp. 'java.exe') in that

directory.

JAVA_OPTIONS
: Additional options that need to be passed to the Java Runtime Environment.

HOP_HOME
: The home directory of Helma (where the libraries need to reside in a subdirectory named 'lib') can be set to something different than the directory of the start script.

OPTIONS
: It is possible to set the location of the file server.properties to something different than the default, by specifying the '−f' option. Call 'java jar launcher.jar −−help' for a complete list of Helma startup options.

It is possible to run multiple Helma instances on the same machine. You only need to take care that these instances all use separate ports, otherwise Helma will exit immediately with a "port is already in use" error. The same error will also appear if one of the ports is already in use by another application (e.g. port 8080 is also used by Tomcat by default). You will either have to change the conflicting port in your start−script, or close the other application first.

Stopping Helma is simply done by exiting the Helma process, which usually can be done by pressing Ctrl−C. Restarting Helma is a matter of stopping and starting Helma manually.

Also make sure that the user, under which Helma is run, has the necessary read−privileges for the lib− and apps−directory, and write−privileges for the log− and the db−directory.

Server Setup

As it has been demonstrated in the previous section, it is quite simple to get Helma up and running. To deploy Helma for production use (on a server), a number of additional tasks usually need to be performed.

Helma as Service on Linux

It is advisable to install Helma as a service on Linux, and have Helma being started and stopped automatically when the server is started and stopped.

This is achieved by copying the file `[HelmaHome]/scripts/helma.conf` to `/etc/helma.conf`, and `[HelmaHome]/scripts/helma` to `/etc/init.d/helma` (or to wherever the Linux−Distribution keeps its service−scripts). Make sure that the latter is actually executable (i.e. '***chmod u+x /etc/init.d/helma***'), and that the settings in `/etc/helma.conf` are correctly configured (see that file for further documentation). Now it is possible to start/stop Helma via the command ' ***helma start***', resp. ' ***helma stop***'. Additional actions are ' ***helma restart***', and ' ***helma reload***'.

By copying the java−binary to something like 'javahop' (and adapt the JAVA_BIN−setting in `/etc/helma.conf`), it is possible to watch this process separately from other possible Java−processes that are active on that server.

It is also possible to run multiple Helma instances as separate services. You just need to copy the script−files to `/etc/helma2.conf` and `/etc/init.d/helma2` and adapt the HELMA_CONFIG−setting in `/etc/init.d/helma2` accordingly.

Depending on your Linux−Distribution you can automatically instruct your system to start and stop Helma when the server is being rebooted. On a Debian−System this is for example done by executing the command 'update−rc.d helma defaults'.

Important: If you decide to start any port (web, xml−rpc,..) below the number 1024, which are considered as 'privileged ports' under Linux, then you have to run Helma as the root user, otherwise these ports won't be accessible.

Helma as Service on Windows

In order to install Helma as a Service on Windows please refer to http://adele.helma.org/download/helma/contrib/hannes/helmaservice/.

Java Startup Options

If you want to use Helma's image processing capabilities on a Linux−Server without an X−Server (which is generally the case), you need to specify '−Djava.awt.headless=true' as a JAVA_OPTION.

Depending on the Java Runtime Environment it is also possible to specify a number of startup options in order to tweak performance. Probably most important is the assignment of the maximum (and minimum) memory size to be used by Helma, which is done by '−Xmx', resp. '−Xms'. Helma's powerful object caching mechanism can become quite memory−hungry, if cache size is set high. If you run Helma on a multi−processor−system you also might want to specify the garbage collector to something different than the default. '−XX:+UseParallelGC' might prove very effective in such a case.

Apache Setup (via mod_jk)

See the online−documentation for now: http://helma.org/docs/howtos/mod_jk/

FIXME

Include section on mod_ssl !

Tomcat Setup

See the online−documentation for now: http://helma.org/docs/howtos/lamtha/

FIXME

modGcj Setup

FIXME ??

Setting up an Application

apps.properties

[HelmaHome]/apps.properties defines active applications.

Assuming that Helma is run by the class helma.main.Server (which is the case if Helma is started via one of the start scripts, and is not embedded for example into Tomcat), the list of active applications is defined by the file `[HelmaHome]/apps.properties`. Each line in that file, that is neither a comment (i.e. starts with a '#') nor contains a period '.', corresponds to an active application. For each such defined application, it is possible to specify further (optional) parameters.

So, in order to start a new application, you just need to add a new line to apps.properties, containing the application name. This may also be done while Helma is running. Helma detects changes to its property−files automatically, and applies them immediately. In order to stop a running application, the line containing the application name needs to be commented out, or removed. The defined parameters can be left untouched.

```
# The next line tells Helma to start/run an
# application named 'appname'
appname

# The following lines list the available
# optional parameters, together with their
# default settings.
appname.appdir = apps/appname
appname.repository.0 = apps/appname
appname.repository.0.implementation = \
  helma.framework.repository.FileRepository
appname.dbdir = db/appname
appname.mountpoint = /appname
appname.static
appname.staticMountpoint
appname.staticHome = index.html,index.htm
appname.staticIndex = false
appname.protectedStatic
appname.uploadLimit = 4096
appname.uploadSoftfail = false
appname.cookieDomain
appname.sessionCookieDomain = HopSession
appname.protectedSessionCookie = true
```

appdir

Defines the location of the source code repository of that application. Can be either relative (to [HelmaHome]), or absolute. The default location is `[HelmaHome]/apps/[appname]`. We will refer to this directory as [AppDir] in this document.

repository.X, repository.X.implementation

Alternatively to specifying a single code repository, it is possible to define multiple repositories. See the online documentation for details: http://helma.org/development/rfc/77606/ (FIXME).

dbdir

Defines the location of the internal XML−based database used by Helma for that application. Even if all HopObjects are mapped to a relational database, this directory is needed for certain files. Can be either relative (to [HelmaHome]), or absolute. The default location is `[HelmaHome]/db/[appname]`. We will refer to this directory as [DbDir] in this document.

mountpoint

This setting tells Helma which web requests should be handled by that application. By default this is /appname, i.e. all requests to http://localhost:8080/appname (assuming the HTTP_PORT 8080 is enabled) are handled by that application. It is also possible to set this value just to '/', so that the name of the application is not contained in the URL anymore.

Hint: For Apache/mod_jk/mod_rewrite/Helma−Setups it is common practice to mount applications at /helma/appname, so that just a single configuration line is required to tell mod_jk what kind of requests should be forwarded to Helma (i.e. those starting with /helma).

static

Helma, resp. Jetty can also serve static content. By default this is disabled, but can be enabled by specifying 'static'. This setting expects the absolute or relative (with respect to `[HelmaHome]`) path to a directory, from which static content should be served.

staticMountpoint

If the serving of static content is enabled, the URL for static content starts with the mountpoint of the application with '/static' being appended to that (e.g. http://localhost:8080/appname/static). By specifying this setting, it is possible to change that mountpoint.

staticHome

Analog to Apache's DirectoryIndex−directive it is possible to tell Jetty which file to serve, if a static directory is requested. This can be a comma−separated list of multiple filenames. The default value is 'index.html,index.htm'.

staticIndex

By setting this to 'true' directory−listings are returned by Jetty. Similar to Apache's mod_autoindex. The default behaviour is to not return such listings.

protectedStatic

Helma includes methods to serve static content from the scripting environment (see res.forward). That way it is possible to have some application code being processed (for example to check for access−rights), before actually serving that content. In order to enable this feature, the protectedStatic−setting has to be set to the location of a directory (either absolute, or relative to [HelmaHome]) from which that content is served.

uploadLimit

Specifies the maximum size for uploaded files in Kilobyte. The default is 1024, i.e. 1MB.

uploadSoftfail

Determines how an upload, that exceeds the uploadLimit should be handled. If this is explicitly set to "true", then Helma will not generate an immediate error response, and just sets the property helma_upload_error in the req.data object, which indicates that an error has occured. Otherwise a non−customizable HTTP error with status code 413 will be thrown.

cookieDomain

By default all cookies that are sent to the Client with res.setCookie are assigned to the same domain as the request went to. If cookieDomain is set, and cookieDomain is a substring of the requested domain, cookies are assigned to that cookieDomain. This has the advantage, that the same cookies can be shared across all sub−domains, by setting the cookieDomain to something like '.domain.tld'.

sessionCookieName

By default Helma assigns requests to sessions according to a session−cookie named 'HopSession'. By specifying the setting sessionCookieName, it is possible to tell Helma to use another name for that cookie.

protectedSessionCookie

By default Helma binds the generated session cookie to the first three octets of the client's ip−address. This mechanism offers some protection against possible cross−site−scripting attacks (http://en.wikipedia.org/wiki/Cross_site_scripting), but on the other hand can make Helma session's mechanism unusable for some users (e.g. users with a round−robin proxy setup). This mechanism can be disabled by setting this property to 'false'.

As soon as the application is started, Helma will look for the `[AppDir]`, and checks whether it already exists and whether it contains at least one file. If this is not the case, Helma will create four directories: `Global`, `Root`, `HopObject` and `User`. Similar, the `[DbDir]` will be created, if it does not exist yet.

General File Structure of an Application

[AppDir] is the main code repository.

The `[AppDir]` of a Helma application can look something like this:

```
[AppDir]/app.properties
[AppDir]/db.properties
```

```
[AppDir]/Global/
[AppDir]/Root/
[AppDir]/HopObject/
[AppDir]/User/
[AppDir]/PrototypeNameABC/
...
[AppDir]/PrototypeNameXYZ/
[AppDir]/someExtraAppCode.zip
[AppDir]/someExtraJavaLibrary.jar
```

Each directory that is contained in `[AppDir]`, with the exception of `Global`, is interpreted as a scriptable prototype. These prototypes are enriched with methods/functions by adding (arbitrarily named) files with the file−extension '.js', with skins/templates by adding files with file−extension '.skin', and can be mapped to a database table via a file named `type.properties`.

A typical prototype directory might therefore look something like this:

```
[AppDir]/PrototypeNameABC/type.properties
[AppDir]/PrototypeNameABC/actions.js
[AppDir]/PrototypeNameABC/macros.js
[AppDir]/PrototypeNameABC/functions01.js
[AppDir]/PrototypeNameABC/functions02.js
[AppDir]/PrototypeNameABC/template01.skin
[AppDir]/PrototypeNameABC/template02.skin
[AppDir]/PrototypeNameABC/template03.skin
```

Additionally you might find files in existing applications ending with '.hsp', which are deprecated template−files, and files ending with '.hac', representing web−accessible methods. Since web−accessible methods are just like ordinary methods appended with '_action', it is now common practice to have them being defined in '.js' just like ordinary methods.

If `[AppDir]` contains files ending with '.zip', they are assumed to be additional application code. Helma treats these files, as if they were unzipped in `[AppDir]`, whereas zipped code and templates have a lower precedence than unzipped code.

If `[AppDir]` contains files ending with '.jar', these are assumed to be extra Java libraries that are being loaded in the runtime environment, and are therefore available for scripting. Helma needs to be restarted if such libraries are added or updated.

All other files and subdirectories are being ignored by Helma.

app.properties

[AppDir]/app.properties is the deployment descriptor for an application.

This file can contain settings of various special parameters used by Helma. The general form for all key/value pairs follows this syntax:

```
keyNameABC = SomeStringValue
# This line here just demonstrate how comments are being added
```

Examples for such parameters would be the setting of the request timeout, the maximum number of Java threads to be available at a time, or whether the logging of SQL should be enabled. See the reference section (FIXME LINK) for a complete list of internal parameters.

Additionally this file can also contain any number of custom key/value pairs, that can be accessed from within the scripting environment. That value could then be accessed from the application code via 'app.properties.keyNameABC' (with the key being case−insensitive). Purpose of this mechanism is, to have all deployment−relevant settings being stored in one common file, so that (theoretically) this would be the only file to be modified in order to deploy an application.

The file '[HelmaHome]/server.properties' follows the same syntax, and a similar purpose. It can be seen as the deployment descriptor for a server. All defined key/value pairs are also available in all applications, but can be overridden by `[AppDir]/app.properties` for each one. Generally it is a good advice to keep all relevant custom−settings just in app.properties. Also see the reference (FIXME LINK) section for server.properties.

db.properties

[AppDir]/db.properties defines connections to relational databases.

This file can contain any number of relational data sources. These sources can be used for Helma's DB mapping mechanism, which maps prototypes to database tables, or to submit SQL statements directly to a database.

For each defined data source, the arguments 'url', 'driver', 'user' and 'password' need to be set. Any further arguments are also passed along, when a new connection is established. The name of the source can be arbitrarily chosen.

```
sourceName.url = jdbc:mysql://ServerName/DatabaseName
sourceName.driver = NameOfJavaClassToBeUsedAsDriver
sourceName.user = UserName
sourceName.password = Password
sourceName.XYZ = ABC
```

Examples of data sources for a MySQL, resp. an Oracle database:

```
mysqlDB.url      = jdbc:mysql://localhost/db_twoday
mysqlDB.driver   = com.mysql.jdbc.Driver
mysqlDB.user     = helma
mysqlDB.password = secret

oracleDB.url      = jdbc:oracle://db.domain.com/oracle
oracleDB.driver   = oracle.jdbc.driver.OracleDriver
oracleDB.user     = helma
oracleDB.password = secret
```

LINK: reference/db.properties

class.properties

[AppDir]/class.properties defines alternative underlying classes for scripted prototypes.

FIXME

See http://helma.org/docs/properties/class.properties/

Scripting

Helma provides an interface (namely the Java interface helma.scripting.ScriptingEngine) through which theoretically any Java scripting language can be used for writing application code in Helma. Currently only JavaScript (Rhino) has been implemented for Helma. Other possible candidates for such integration would be Jacl, Jython, JRuby, JudoScript, Pnuts and Groovy.

JavaScript

So, in order to develop Helma applications a thorough knowledge and understanding of the JavaScript language is necessary. The better you learn the language from the beginning on, the smoother your experience with Helma will be.

The Rhino version currently shipped with Helma implements JavaScript 1.5, which confirms to the 3rd edition of the ECMA−262 ECMAScript standard, downloadable from http://www.ecma−international.org/publications/standards/Ecma−262.htm. An extensive, but still readable guide can be found at http://www.croczilla.com/~alex/reference/javascript_guide/index.html, the according reference at http://www.croczilla.com/~alex/reference/javascript_ref/index.html. A much more comprehensive and visually enhanced overview on the syntax is offered at http://javascript−reference.info/. But be aware that often JavaScript references do not just document the language itself, but also the so called host objects common to client−side JavaScript (i.e. 'windows', 'document', etc.). Helma is basically JavaScript 1.5 enhanced with a number of host objects typical for server−side web development. A main part of this documentation is devoted to the description of these additional objects.

Some dummy JavaScript code:

```
// This a comment

/* This is a
   multi−line
   comment */

var m;

var i = 0;
i = i + 3;

var str = "This is a String.";
str += " And another one appendend.";
str += new String(" And another.");
str = str.substring(str.indexOf("."), str.length − 3);

var bool = true;
var isTrue = bool ? true : false;

var arr = new Array();
arr[0] = "value01";
arr.push("value02");
var str = arr.join(" & ");
arr.push([1,2,3,4]);
var len = arr.length;

var obj = new Object();
obj.key01 = "value01";
obj["key02"] = str;
obj.key03 = {name: "product", price: 12.3};
```

```
obj.key04 = arr;

var date = new Date();
var str = date.format("dd.MM.yyyy");

var j = 0;
for (var k=0; k<10; k++) {
   if (k%2 == 0) continue;
   j += k;
}

try {
   someUndefinedFunction();
} catch (err) {
   return "Error occurred!";
}
```

Rhino / Direct Scripting of Java

Rhino extends the JavaScript language with a couple of functions, as for example Array.prototype.toSource(), uneval(Object) and seal(Object) (see http://www.mozilla.org/rhino/rhino15R5.html for details).

But more importantly, Rhino provides full access to any kind of Java class that has been loaded into the Runtime Environment. A feature that gives Helma developers (nearly) the full power of Java at their fingertips, and let's them take advantage of available Java libraries within their application.

Assuming that your Java classes are already loaded into Helma (which is for example done by dropping your jared−library into `[HelmaHome]/lib/ext` or to `[AppDir]`, and restarting Helma), you can call its static methods, access constants, instantiate new objects, and so forth.

The following example demonstrates the use of the freely available GeoIP Java API to look up country codes for ip−addresses. See http://www.maxmind.com/app/java for more information on this package.

```
var ls = new Packages.com.maxmind.geoip.LookupService("modGeoIP.dat",
           Packages.com.maxmind.geoip.LookupService.GEOIP_MEMORY_CACHE);
var code = ls.getCountry("81.223.46.226").getCode();
```

As you can see, Java classes are made accessible through a top−level variable named Packages. The basic java classes (starting with '***java***.') do not require this prefix, but are directly accessible. See the following example:

```
var hash = new java.util.Hashtable();
hash.put("key01", "a string");
hash.put("key02", new Date());
```

Note that Rhino takes care of all the necessary type conversion.

Make sure that (at least at some point) you read through the 'Scripting Java' tutorial located on the Rhino website: http://www.mozilla.org/rhino/ScriptingJava.html, which also explains how to instantiate Java arrays, implement interfaces and call overloaded methods.

Helma's Host Objects

As it has been mentioned before, Helma extends Rhino with a number of host objects, providing extra functionality useful for web scripting. These are:

req
: Represents the HTTP request, that was sent by the client.

res
: Represents the HTTP response, that is sent back to the client.

session
: A session object, that is assigned to the current request.

app
: Represents the application as a whole.

Additionally the following two variables are available in the scripting environment:

root
: The root of the object model hierarchy.

path
: An array of objects that are contained in the request path.

Helma Basics

Before we will walk through Helma's functionality in detail, we give a quick, very compressed overview over some of its basic mechanism, and introduce a couple of Helma specific terms.

Helma applications generally follow an object orientated approach. That is, developers define their prototypes (classes), together with properties and methods. Instances of these prototypes are called HopObjects, which are basically just extended JavaScript objects. These prototypes are usually mapped on a table of a relational database, each entry (row) of that table representing a HopObject.

Prototypes are being defined by adding new directories to the `[AppDir]` (directory name = prototype name), their functions by adding (arbitrarily named) js−files containing function definitions into these directories. Properties only need to be declared within `[AppDir]/PrototypeNameABC/type.properties` explicitly, if they are going to be mapped on a relational database. Then a property corresponds to specific column in that database table.

In addition to this basic mapping, it is also possible to build object models and define relations between prototypes. A prototype property can be a reference to another prototype, a prototype can contain collections/lists of other prototypes, and it is possible (and highly recommended) to define an object hierarchy for these prototypes.

The Root prototype is a special prototype that is always present and immediately persistent as soon as the application is started. It represents the root of the object model hierarchy.

The DB mapping and object modelling are defined in files named '`type.properties`', which need to reside in the according prototype directories. Besides these mappings, Helma developers (generally) do not have to mess around with SQL. Any creating, retrieving, modifying and removing of HopObjects happen within the scripting scope, with Helma taking care of the building and execution of all the necessary SQL statements.

Besides having prototype specific methods, which all require the existence of a HopObject (i.e. no static methods can be defined), it is possible to define methods in a global context, which can be directly called from anywhere in the application code.

There are four ways of how a function call can be initiated:

- a web request via HTTP
- a XmlRpc request
- an internal function call by the scheduler
- an external function call via helma.main.launcher.Commandline

with the first type naturally being the most common one for a web application.

If a browser/client makes a web request to an application, Helma tries to resolve the request path from left to right, walking down the object hierarchy (starting with root) and trying to find an according web accessible function. A prototype function is web accessible if and only if its function name ends with '_action'. This means that a request to http://localhost:8080/appname/main is handled by the function main_action of the prototype Root. A request to http://localhost:8080/appname/latestStories/437/show (i.e. the request path handled by the application is '/latestStories/437/show') will have Helma fetch the root object, fetch a collection named 'latestStories' attached to the root (if defined), fetch a HopObject (supposedly of a prototype named 'Story') with ID 437 (if exists), and call function show_action (if defined) on that HopObject. That action will subsequently write to the response buffer, which is then send back to the client as soon as the function returns. Any parameter that is sent together with the request, no matter whether that is a URL parameter (?key=value), or an element of a submitted POST−form or a Cookie can be read via the Object req.data.

Aside from writing directly to the response, via the function res.write, Helma provides a powerful and clean rendering framework, consisting of so called skins and macros. Skins are basically just parts of the response, usually being HTML code, mixed with (tightly controlled) calls of certain functions, the macros. These skins are either defined in a global context, or for a specific prototype. By default skins are expected to be file−based, stored with the file−extension '.skin' in the according prototype directory, but can also be fetched from a database or any other data source. A macro is simply a global or prototype function with its name ending with '_macro'. These macros are being replaced with some other string/text, usually in dependence on certain conditions, when that skin is being rendered.

Actions

Any defined prototype function with its name ending to '_action' becomes automatically web accessible to any HTTP client (which is generally a webbrowser), and is referred to as 'action' in this documentation. The following example defines two such actions:

```
### Root/functions.js ###

function main_action() {
  res.write("The main action of the Root object.");
}

function hello_action() {
  res.write("Hello");
}
```

By requesting the URL http://localhost:8080/appname/main the first defined action is being executed. A http://localhost:8080/appname/hello would call the second action. Actions that are named 'main' are also taken as the default action, if no particular action is being specified in the request path. Trailing slashes in the request path are ignored. Therefore, http://localhost:8080/appname, http://localhost:8080/appname/, http://localhost:8080/appname/main and http://localhost:8080/appname/main/ all result in the same action

call.

Note, that function names, and therefore also action names are case−sensitive, i.e. http://localhost:8080/appname/main and http://localhost:8080/appname/Main are not resulting in the same action call.

Since the dot ('.') is not allowed to be part of a function name in JavaScript, an underscore needs to be used in order to define an action containing a dot. The root action 'style_css_action' would therefore be accessible as '`/appname/style.css`', as well as '`/appname/style_css`'. Being able to specify dots in action names is important, since there are still browser in use, that try to guess the content type of a response from the extension of the requested 'file'.

Helma provides a special function named onRequest(), which can be defined for each prototype. This function will be automatically called by Helma before any action is actually processed for a prototype. Therefore that function is a good place to implement functionality that the developer wants to make sure to be always called, like checks for access privileges.

Example:

```
 ### Root/functions.js ###

function main_action() {
  res.write(" and Goodbye. ");
}

function onRequest() {
  res.write("Hello");
}
```

Calling http://localhost:8080/appname/main will result in the string 'Hello and Goodbye' being returned.

Requests

Let's now have a look at a typical HTTP request being submitted by a common browser to Helma:

```
GET /appname/main HTTP/1.1
Host: localhost:8080
User−Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en−US; rv:1.7.6) Gecko/20050317 Firefox/1.0.
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png
Accept−Language: en−us,en;q=0.5
Accept−Encoding: gzip,deflate
Accept−Charset: ISO−8859−1,utf−8;q=0.7,*;q=0.7
keep−alive: 300
Connection: keep−alive
Cookie: HopSession=127.0.0.zsy455yhwnbe
If−None−Match: "O+nl/8BoDSesnGNp7ASjjw=="
```

The first word of the first line contains the HTTP method, which is generally GET. Also quite common is the HTTP method POST, which can be used for submitting forms. Helma's standard actions handle all incoming GET, POST and HEAD requests, with the latter just returning the response header without the actual body. In case that a developer wants to differentiate between these methods, she can use req.isGet(), req.isPost() or req.method to detect the actual method of the current request.

But the HTTP protocol is not limited to just these methods (see http://www.w3.org/Protocols/rfc2616/rfc2616−sec9.html). An implementation of a REST API (http://en.wikipedia.org/wiki/Representational_State_Transfer) for example, generally requires the handling of DELETE and PUT requests. WebDAV (http://en.wikipedia.org/wiki/Webdav) even adds 7 more additional methods to the protocol. Helma provides a flexible way to handle all kind of possible methods separately. The developer can define special actions with their function names ending with '_action_methodname', i.e. '_action_delete', '_action_propfind' and so forth. A DELETE request to /appname/hello will initiate the execution of an action named 'hello_action_delete'.

As can be seen in listed sample request, all kind of other information is being passed along to the server besides the used HTTP method. This data is exposed to Helma application code through the request object 'req'. See the reference section for a detailed list of available methods and fields. Probably most important is the Object req.data, that allows access to any form data, url parameter or cookie value. A special case is the uploading of a file, which is being discussed later on. Additionally the class javax.servlet.http.HttpServletRequest is directly exposed via req.servletRequest.

Responses

A response is sent back to the client, as soon as the action returns. This can either happen by making an explicit return−call in the action, or by calling res.abort() or res.redirect(), or happens if an error occurs, that is not being caught.

Now, let's investigate such a returned HTTP response in detail:

```
HTTP/1.1 200 OK
Date: Mon, 04 Apr 2005 22:02:20 GMT
Server: Jetty/4.2.22 (Windows XP/5.1 x86 java/1.5.0_02)
Last−Modified: Tue, 23 Oct 2001 09:54:46 GMT
ETag: "67987−2095−3bd53e66"
Accept−Ranges: bytes
Content−Length: 8341
Content−Type: text/html

<!DOCTYPE HTML PUBLIC "−//W3C//DTD HTML 3.2 Final//EN">
<HTML>
...
</HTML>
```

This is the content of a typical, successful response, consisting of a response header (containing some meta information) and the body (containing the actual HTML to be rendered). The outgoing response is exposed to the scripting environment via the response object 'res'. Again, see the reference for a complete list of available methods and fields. The response object offers a string−buffer, that the developer can write to. This can either be done by directly calling the function res.write() or by using renderSkin(), which renders a specific skin to the response. Helma also provides a way to write back binary content, which is done through the methods res.forward(), resp. res.writeBinary(). Most of the common response header fields can be set via the according fields of the response object. Examples are res.contentType, res.charset, res.status, and so forth.

The response status code of a successfully processed request is 200. A complete documentation of officially defined status codes can be found here: http://www.w3.org/Protocols/rfc2616/rfc2616−sec10.html. Two other common status codes are discussed in the following:

Status 303 represents a 'See other'−response, which is used to send the browser to some other URL, a task which is very common for web applications. In Helma such a redirect is triggered with the method

res.redirect(url), which expects a URL string as its single argument. As soon as Helma encounters such a redirect statement, processing stops and the 303 response, together with the new URL is immediately sent back.

```
### Root/functions.js ###

function hello_action() {
  res.redirect("http://localhost:8080/appname/ciao");
}

function ciao_action() {
  res.write("Ciao!");
}
```

In the above example a request to http://localhost:8080/appname/hello would redirect the browser immediately to http://localhost:8080/appname/ciao.

Status 404 represents a 'Notfound'−response. Helma returns a 404 if there is no such defined action for a particular request. It is possible to catch such 404 by specifying a special notfound−action, which should be called as a fallback in such a case. This mechanism can be used to display a nicer formatted version of the 'Notfound'−response.

```
### Root/functions.js ###

function notfound_action() {
  res.write("Sorry, but nothing was found on this server.");
}
```

A request to http://localhost:8080/appname/foo will actually have the above notfound_action being called. It should be noted, that it is even possible to define separate notfound_actions for each prototype, and it is also possible to change the name of that fallback action via the property 'notfound' in app.properties.

In case that an error occurs during the processing of an action, Helma will catch that error and return a status code 200 with the error message contained as its response body. Similar to the notfound−action, it is also possible to define a special fallback action, which should be called in such a case. This is again useful for delivering a nicely rendered error message to the client, or alternatively to set the status code to some other value. That special function is named 'error_action', and can again be implemented for each prototype, and can be renamed by a property 'error' in app.properties.

A special error that might be thrown is the Request−Timeout. By default Helma throws that error if the processing of a request takes longer than 30 seconds. Since there are applications, which might be very computing−intense, this timeout−threshold can be changed through the setting 'requestTimeout' in app.properties.

In case that the provided methods of the response object are not sufficient, res.servletResponse exposes the class javax.servlet.http.HttpServletResponse directly to the scripting environment.

Global

Aside from the functions defined for specific prototypes, it is also possible to define functions and macros (but not actions) in a global context. This is done by adding js−files to the directory `[AppDir]/Global`. The following example demonstrates the calling of a root and a global function from within a root action.

```
### Global/functions.js ###

function foo() {
  return "a global function";
}

### Root/functions.js ###

function foo() {
  return "a root function";
}

function main_action() {
  res.write(root.foo() + " and " + foo());
}
```

A request to http://localhost:8080/appname/main will display the string 'a root function and a global function'. Note, that opposed to prototypes it does not make sense to define actions in a global context, since these can never be accessed via a request path.

Handling Form Data and File Uploads

This section on the handling of user input comes at a rather early point in this User's Guide, since the following sample applications require the user to interact with the application through forms (as it is the nature with most web applications). Knowing how to access such information is therefore essential for understanding these latter examples.

As has already been mentioned earlier, all submitted form (and cookie) data are available as properties of the object 'req.data'. Submitted information always comes in the form of a key/value pair with the key−identifier being the name of the input field. The submitted string of a text input field with the name 'surname', will therefore be accessible through req.data.surname, resp. req.data["surname"]. The method req.get("someKey") can also be used to access that value, but is considered deprecated.

Put the following code into an application directory named 'formdemo', and start up that application by adding an according line to `[HelmaHome]/apps.properties`.

```
### Root/forms.js ###

function getForm_action() {
   if (req.data.submit01 || req.data.submit02) {
      res.write("F01: " + req.data.field01 + "<br>");
      res.write("F02: " + req.data.field02 + "<br>");
      res.write("F03: " + req.data.field03 + "<br>");
      res.write("F04: " + req.data.field04 + "<br>");
      res.write("F05: " + req.data["field05_array"].join("/") + "<br>");
      res.write("S01: " + req.data.submit01 + "<br>");
      res.write("S02: " + req.data.submit02 + "<br>");
   }
   this.renderSkin("getForm");
}

function postForm_action() {
   if (req.data.submit) {
      res.write("F01: " + encode(req.data.field01) + "<br>");
   }
   this.renderSkin("postForm");
}
```

```
function uploadForm_action() {
   if (req.data.submit) {
      res.write("F01: " + req.data.field01 + "<br>");
      var mimepart = req.data.field01;
      res.write("Name: " + mimepart.getName() + "<br />");
      res.write("Size: " + mimepart.getContentLength() + "<br />");
      res.write("Type: " + mimepart.getContentType() + "<br />");
   }
   this.renderSkin("uploadForm");
}


### Root/getForm.skin ###

<form method="GET" action="/formdemo/getForm">
  <input type="text" name="field01">
  <input type="hidden" name="field02" value="a hidden value">
  <br />
  <select name="field03">
    <option value="1">1</option>
    <option value="2">2</option>
  </select>
  <br />
  <input type="radio" name="field04" value="a">a
  <input type="radio" name="field04" value="b">b
  <br />
  <select name="field05" size="3" multiple>
    <option value="x">x</option>
    <option value="y">y</option>
    <option value="z">z</option>
  </select>
  <br />
  <input type="submit" name="submit01" value="Go GET it! (01)">
  <input type="submit" name="submit02" value="Go GET it! (02)">
</form>

### Root/postForm.skin ###

<form method="POST" action="/formdemo/postForm">
  <textarea name="field01"></textarea>
  <input type="submit" name="submit" value="Go POST it!">
</form>

### Root/uploadForm.skin ###

<form method="POST" action="/formdemo/uploadForm" enctype="multipart/form-data">
  <input type="file" name="field01" accept="image/*">
  <input type="submit" name="submit" value="Go UPLOAD it!">
</form>
```

The code defines three (Root-)actions, and accordingly three (Root-)skins, which are being rendered from within these actions. A request to http://localhost:8080/formdemo/getForm will render a simple form with text input fields, radio buttons, dropdowns and two submit-buttons. In `Root/getForm.skin` we specified the used method to be 'GET', and defined that same action as the action (i.e. the target) of that form. If the user clicks on one of the submit buttons, the browser will send the filled out form back to the getForm-action, which now renders each of the submitted values at the top of the page. If you take a look at the address bar of your browser, you will see that your browser actually submitted the information to the server by appending key/value-pairs at the end of the URL.

Generally it is common practice to use the POST−method instead of GET to submit data. In case that you have a textarea (which allows multi−line text strings to be inserted), or want to upload files, you actually must use the POST−form. The browser will then send the submitted information not as part of the URL, but appends it to the request body. Nevertheless the req.data−object provides a transparent way to access that information, no matter what method has been used. See http://localhost:8080/formdemo/postForm for an example. After submitting the form, you will remain on the same action, but the submitted text will now be displayed at the top of the page. Try hitting the refresh−button in your browser, and you will be prompted whether the browser should send the previously submitted information again to the server. Therefore it is common practice to perform a redirect, after a form has been processed, so that a browser loses its memory and that a unintentional re−submission of data is avoided. Note that the only reason why we call the global method 'encode' in our example.is being called on the submitted string is to transform the linebreaks of the textarea into actual HTML linebreaks, i.e. br−tags.

http://localhost:8080/formdemo/uploadForm demonstrates how to upload a file to the application. Firstly, you need to use the POST−method, secondly you need to set the enctype−attribute of the opening form−tag to "multipart/form−data", and thirdly you need to include (at least) one form−field of type "file". That way, the browser knows how to correctly submit binary data to the server. The according property in req.data now does not contain a string, but a mimepart−object, which can be further processed (usually written to disk). See the documentation on the mimepart−object for a full list of available methods.

As a final note on HTML forms two special cases should be pointed out. Firstly, the form type "checkbox" does not send any data at all, if it is not checked. Therefore it is not possible to detect whether a checkbox has not been checked, or whether it has not been present in the form at all. A way out of this dilemma is to always include a second (hidden) field input, that indicates the presence of that particular checkbox in the form. Secondly, a submitted multi−select input form will contain several key/value−pairs with identical keys. Accessing the according property in req.data will only return of the submitted values, but not all of them. Helma provides an additional req.data−property appended with the key−name being appended with '_array', that returns an array of all selected options. See the getForm−example above.

Cookies

Besides prompting the user for data input through web forms, it is possible to store and retrieve information on the client side through the means of so called cookies, without the interaction of the user. All common browsers support this technology, whereas it is possible that cookies are not accepted due to strict security settings. We distinguish between session−cookies, which are removed from disk as soon as all browser windows are being closed, and permanent cookies. The latter can for example be used to track a client's action over several sessions, or to store user credentials, so that the user does not have to login each time she visits the web application.

```
### Root/cookies.js ###

function demoCookie_action() {
   var cnt = req.data.counter;
   if (cnt == null) cnt = 0;
   res.write("Your access counter is at " + cnt);
   res.setCookie("counter", cnt + 1);
}
```

The code example above will count the number of requests to http://localhost:8080/appname/demoCookie, and won't lose track even if several days/weeks are between each of the requests. Cookies are automatically created on the client−side (if the client accepts cookies, which is generally the case), if it did not exist before. Changing the value of a cookie is simply done by re−setting that cookie. Removing a cookie can be done by

setting its value to an empty string.

Since form data as well as cookie data are both available as properties req.data, and can not be distinguished there, it is sometimes necessary to use req.servletRequest.getCookies() in order to make sure that a certain information has actually been submitted as a cookie.

Prototypes

FIXME

```
IGNORE THE FOLLOWING NOTES FOR NOW, AND JUST TAKE A LOOK AT THE SAMPLE CODE !!

Prototypes
    persisting; fetching; removing
    folders
    fetching objects via ID
    Actions: changes are just effective if no errors occur (res.abort, res.commit)
    constructor()
    inheritance − HopObject
    HopObject
        .add
        .remove
        .get
        HopObject.getById()

Explain everything before we actually start showing some sample code; we use internal xml−based d

Explain what we want to do in the sample app: Create a single prototype, create/persist, fetch, m

Create folder named 'Person';

Create minimalistic rendering framework, i.e. a simple global main−skin with res.data.body;

Root/main_action: loops through all existing persons, and builds string for res.data.body with id

Root/createPerson_action: builds string with simple form to insert name, dateOfBirth and size wit

Insert link to deletePerson−action with personId as url−parameter in the Root/main_action−list

Root/deletePerson_action expects personId to be passed as url−string; writes out error, if no per

Explain: how to fetch a persistent object, and how to delete it;

Insert link to Person/edit−action; contains a small form to change the name of that person; redir

Explain: modification of existing objects;

Explain constructor(), HopObject−inheritance

calls list.skin;lists all existing person

Create root−action with small form to create new Person with certain 'name', and make that persis

create prototype 'person'

hopobject−functions; HopObject−inheritance

properties id, name, dateofbirth, height

create Root/main_action that creates and lists all Persons
```

```
person-skin, person-macro, person-function

person-action; _parent ?? person-url?

href_macro ??
```

demoPrototypes.zip

```
### HopObject/functions.js ###

function info_action() {
   res.write("HELLO " + this._prototype.toUpperCase() + " " + this._id);
}


### Root/functions.js ###

function main_action() {
   // list all Persons
   for (var i=0; i<root.count(); i++) {
      var p = root.get(i);
      res.write("<li>");
      res.write(p.name + " ");
      if (p.dateofBirth) res.write(p.dateOfBirth.format("dd.MM.yyyy") + " ");
      res.write("<a href=" + this.href("deletePerson") +
         "?personId=" + p._id + ">delete this Person</a> ");
      res.write("<a href=" + p.href("edit") + ">edit this Person</a>");
   }
   res.write("<br />");
   res.write("<a href=" + this.href("createPerson") + ">create new Person</a>");
}


function createPerson_action() {
   if (req.data.send) {
      // instantiate a new object of type 'Person'
      var p = new Person();
      // set the property 'name' of the new Person
      p.name = req.data.name;
      // set a integer value
      p.bodySize = 180;
      // set the date of birth
      p.dateOfBirth = new Date(1978, 4, 24);
      // store the new object persistently
      root.add(p);
      // now redirect the client to the list of all Persons
      res.redirect(this.href("main"));
   }
   this.renderSkin("createPerson");
}


function deletePerson_action() {
   // if personId is passed in URL, and such a Person exists...
   if (req.data.personId && root.get(req.data.personId)) {
      // ... we delete that Person
      var p = root.get(req.data.personId);
      p.remove();
   }
   // now redirect the client to the list of all Persons
   res.redirect(this.href("main"));
```

```
}


### Root/createPerson.skin ###

<form action="">
  <input type="text" name="name">
  <input type="submit" name="send" value="create new Person">
</form>


### Person/functions.js ###

function edit_action() {
   // if the form has been submitted...
   if (req.data.send) {
      // ... change the name of that person
      this.name = req.data.name;
      // ... and redirect to the root main action
      res.redirect(root.href("main"));
   }
   this.renderSkin("edit");
}


function constructor() {
   // a constructor can be optionally defined for each
   // prototype in order to have programm code being
   // executed whenever a new Object is instantiated
   ;
}


### Person/edit.skin ###

<form action="">
  <input type="text" name="name">
  <input type="submit" name="send" value="change name">
</form>
```

DB Mapping

FIXME

```
IGNORE THE FOLLOWING NOTES FOR NOW, AND JUST TAKE A LOOK AT THE SAMPLE CODE !!

DB Mapping
  JDBC driver
  MySQL
  needs ID
  recall advantages: no necessity for sql
  Caching Mechanism
    HopObject.cache
    clearCache
    invalidate
    http://helma.org/stories/77405/
  logSQL
  basic mapping
    _db
    _table
    _id
```

```
    _idgen (for oracle)
  Simple Property Mappings
    prop.readonly
    prop.private

Make sure that the SQL-Statements in mysql.sql are executed before the app is started.

demoSimpleMapping equals demoPrototypes with the difference, that objects are now stored in a rel

Turn on logSQL and watch what happens. Call app.clearCache and see what happens.
```

Theoretically any JDBC accessible database can be used: MySQL, hsqldb (a pure Java database), Oracle, MS SQL Server, and so forth. As an alternative to a relational database, Helma's XML database can be used. This file-based database is located at `[HelmaHome]/db/appname`, each file in that directory corresponding to a HopObject. The advantage is, that no explicit mapping is required and the developer does not have to worry about database design up front. On the other hand, this XML-database admittedly can not compete with industry-proven relational databases as MySQL or Oracle regarding reliability and scalability. Furthermore some of Helma's object modelling features are just supported for relational databases. For these reasons we will focus in this documentation on an object model, that is mapped on such a database.

demoSimpleMapping.zip

```
### mysql.sql ###

CREATE DATABASE demoSimpleMapping;
USE demoSimpleMapping;

GRANT ALL ON demoSimpleMapping.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE person (
  person_id MEDIUMINT(10) NOT NULL,
  person_name TINYTEXT,
  person_height TINYINT unsigned,
  person_dateofbirth DATETIME,
  PRIMARY KEY (person_id)
);


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoSimpleMapping
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret


### Root/functions.js ###

function main_action() {
   // list all Persons
   for (var i=0; i<root.count(); i++) {
      var p = root.get(i);
      res.write("<li>");
      res.write(p.name + " ");
      if (p.dateofBirth) res.write(p.dateOfBirth.format("dd.MM.yyyy") + " ");
      res.write("<a href=" + this.href("deletePerson") +
         "?personId=" + p._id + ">delete this Person </a> ");
      res.write("<a href=" + p.href("edit") + ">edit this Person</a>");
   }
   res.write("<br /><a href=" + this.href("createPerson") + ">create new Person</a>");
```

```
}

function createPerson_action() {
   if (req.data.send) {
      // instantiate a new object of type 'Person'
      var p = new Person();
      // set the property 'name' of the new Person
      p.name = req.data.name;
      // set a integer value
      p.height = 180;
      // set the date of birth
      p.dateOfBirth = new Date(1978, 4, 24);
      // store the new object persistently
      root.add(p);
      // now redirect the client to the list of all Persons
      res.redirect(this.href("main"));
   }
   this.renderSkin("createPerson");
}

function deletePerson_action() {
   // if personId is passed in URL, and such a Person exists...
   if (req.data.personId && root.get(req.data.personId)) {
      // ... we delete that Person
      var p = root.get(req.data.personId);
      p.remove();
   }
   // now redirect the client to the list of all Persons
   res.redirect(this.href("main"));
}


### Root/createPerson.skin ###

<form action="">
  <input type="text" name="name">
  <input type="submit" name="send" value="create new Person">
</form>


### Person/type.properties ###

_db = jad
_table = person

_id = person_id

name = person_name
height = person_height
dateOfBirth = person_dateofbirth


### Person/functions.js ###

function edit_action() {
   // if the form has been submitted...
   if (req.data.send) {
      // change the name of that person
      this.name = req.data.name;
      // redirect to the root main action
      res.redirect(root.href("main"));
   }
```

```
   this.renderSkin("edit");
}


### Person/edit.skin ###

<form action="">
  <input type="text" name="name">
  <input type="submit" name="send" value="change name">
</form>
```

Object Modelling

FIXME

```
IGNORE THE FOLLOWING NOTES FOR NOW, AND JUST TAKE A LOOK AT THE SAMPLE CODE !!

Object Modelling
  Object References
    ref.local
    ref.foreign
    ref.local.X
    ref.foreign.X
    ref.logicalOperator
  Collections
    type.properties
      _children
      coll.local
      coll.foreign
      coll.loadmode
      coll.cachemode
      coll.order
      coll.filter
      coll.filter.additionalTables
      coll.hints
      coll.maxSize
      coll.accessname
    .count()
    .add()
    .removeChild()
    .list()
    .invalidate()
    .contains()
  Grouped Collections
    coll.group
    coll.group.order
    coll.group.prototype
  mountpoint
  _parent
```

Inheritance

FIXME

```
IGNORE THE FOLLOWING NOTES FOR NOW, AND JUST TAKE A LOOK AT THE SAMPLE CODE !!
```

Request Path Resolution

FIXME

```
IGNORE THE FOLLOWING NOTES FOR NOW, AND JUST TAKE A LOOK AT THE SAMPLE CODE !!

Request Path resolution / URL space
  notfound
  error
  baseuri
  _parent for URL generation
  .href()
  hrefSkin, hrefFunction, hrefRootPrototype
```

demoObjectModelling.zip: collections, parent, object−reference

```
### mysql.sql ###

CREATE DATABASE demoObjectModelling;
USE demoObjectModelling;

GRANT ALL ON demoObjectModelling.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_person (
  person_id MEDIUMINT(10) NOT NULL,
  person_name TINYTEXT,
  person_height TINYINT unsigned,
  person_dateofbirth DATETIME,
  person_org_id MEDIUMINT(10) unsigned
  PRIMARY KEY (person_id)
);

CREATE TABLE tb_organisation (
   org_id MEDIUMINT(10) unsigned NOT NULL,
   org_name TINYTEXT,
   org_country TINYTEXT,
   PRIMARY KEY (org_id)
);

CREATE INDEX idx_pers_org ON tb_person (person_org_id);

INSERT INTO tb_organisation values (1, "knallgrau", "at");
INSERT INTO tb_organisation values (2, "helma", "at");
INSERT INTO tb_organisation values (3, "amazon", "us");
INSERT INTO tb_organisation values (4, "ebay", "us");
INSERT INTO tb_organisation values (5, "yahoo", "us");

INSERT INTO tb_person values (1, "michi", 180, "1978−05−24", 1);
INSERT INTO tb_person values (2, "matthias", 179, "1976−06−22", 1);
INSERT INTO tb_person values (3, "dieter", null, "1978−06−13", 1);
INSERT INTO tb_person values (4, "hannes", null, null, 2);
INSERT INTO tb_person values (5, "weirdo", 185, null, null);


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoObjectModelling
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret
```

```
### Root/type.properties ###

_children = collection(Organisation)
_children.accessname = org_name

tallPeople = collection(Person)
tallPeople.order = person_height desc
tallPeople.filter = person_height IS NOT NULL
# tallPeople.maxsize = 5

allPeople = collection(Person)


### Root/functions.js ###

function main_action() {
   // list all Organisation and their Persons
   for (var i=0; i<root.count(); i++) {
      var o = root.get(i);
      res.write("<h3>" + o.name + " (" + o.country + ")</h3>");
      for (var j=0; j<o.count(); j++) {
         var p = o.get(j);
         res.write("<li>");
         res.write(p.name + " " + (j+1) + " of " + p.organisation.count());
         res.write(" <a href=" + p.href("info") + ">info</a>");
      }
   }
   res.write("<br /><a href=" + root.href("tallPeople") + ">tall People</a>");
}


function tallPeople_action() {
   // list the tallest people
   for (var i=0; i<root.tallPeople.count(); i++) {
      var p = root.tallPeople.get(i);
      res.write("<li>" + p.name + " " + p.height + " " +
          p.href("info") + " " + root.get("knallgrau").contains(p));
   }
}

### Organisation/type.properties ###

_db = jad
_table = tb_organisation

_id = org_id
_parent = root

_children = collection(Person)
_children.local = org_id
_children.foreign = person_org_id
_children.accessname = person_name
_children.order = person_name

name = org_name
country = org_country


### Person/type.properties ###
```

```
_db = jad
_table = tb_person

_id = person_id
_parent = organisation, root.allPeople

name = person_name
height = person_height
dateOfBirth = person_dateofbirth

organisation = object(Organisation)
organisation.local = person_org_id
organisation.foreign = org_id
```

demoGrouping.zip

```
### mysql.sql ###

CREATE DATABASE demoGrouping;
USE demoGrouping;

GRANT ALL ON demoGrouping.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_person (
  person_id MEDIUMINT(10) NOT NULL,
  person_name TINYTEXT,
  person_height TINYINT unsigned,
  person_dateofbirth DATETIME,
  person_org_id MEDIUMINT(10) unsigned
  PRIMARY KEY (person_id)
);


CREATE TABLE tb_organisation (
   org_id MEDIUMINT(10) unsigned NOT NULL,
   org_name TINYTEXT,
   org_country TINYTEXT,
   PRIMARY KEY (org_id)
);

CREATE INDEX idx_pers_org ON tb_person (person_org_id);

INSERT INTO tb_organisation values (1, "knallgrau", "at");
INSERT INTO tb_organisation values (2, "helma", "at");
INSERT INTO tb_organisation values (3, "amazon", "us");
INSERT INTO tb_organisation values (4, "ebay", "us");
INSERT INTO tb_organisation values (5, "yahoo", "us");

INSERT INTO tb_person values (1, "michi", 180, "1978-05-24", 1);
INSERT INTO tb_person values (2, "matthias", 179, "1976-06-22", 1);
INSERT INTO tb_person values (3, "dieter", null, "1978-06-13", 1);
INSERT INTO tb_person values (4, "hannes", null, null, 2);
INSERT INTO tb_person values (5, "weirdo", 185, null, null);


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoGrouping
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret
```

```
### Root/type.properties ###

_children = collection(Organisation)
_children.accessname = org_name

countries = collection(Organisation)
countries.group = org_country
countries.group.prototype = Country
countries.group.order = org_country desc


### Root/functions.js ###

function main_action() {
   // list all Countries and their Organisations
   for (var i=0; i<root.countries.count(); i++) {
      var c = root.countries.get(i);
      res.write("<h3>" + c.getName() + " <a href=" + c.href("info") + ">info</a></h3>");
      res.write("");
      for (var j=0; j<c.count(); j++) {
         var o = c.get(j);
         res.write("<li>");
         res.write(o.name + " " + o.count());
         res.write(" <a href=" + o.href("info") + ">info</a>");
      }
   }
}


### Organisation/type.properties ###

_db = jad
_table = tb_organisation

_id = org_id
_parent = root

_children = collection(Person)
_children.local = org_id
_children.foreign = person_org_id
_children.accessname = person_name
_children.order = person_name

name = org_name
country = org_country


### Person/type.properties ###

_db = jad
_table = tb_person

_id = person_id
_parent = organisation, root.allPeople

name = person_name
height = person_height
dateOfBirth = person_dateofbirth

organisation = object(Organisation)
organisation.local = person_org_id
```

```
organisation.foreign = org_id


### Country/functions.js ###

function getName() {
   if (this.groupname == "at")
      return "Austria";
   else if (this.groupname == "us")
      return "USA";
}
```

demoMNRelation.zip

```
### mysql.sql ###

CREATE DATABASE demoMNRelation;
USE demoMNRelation;

GRANT ALL ON demoMNRelation.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_person (
  person_id MEDIUMINT(10) NOT NULL,
  person_name TINYTEXT,
  PRIMARY KEY (person_id)
);

CREATE TABLE tb_relationship (
   rel_id MEDIUMINT(10) unsigned NOT NULL,
   rel_person01_id MEDIUMINT(10) unsigned,
   rel_person02_id MEDIUMINT(10) unsigned,
   PRIMARY KEY (rel_id)
);

CREATE INDEX idx_rel_pers01 ON tb_relationship (rel_person01_id);
CREATE INDEX idx_rel_pers02 ON tb_relationship (rel_person02_id);

INSERT INTO tb_person values (1, "michi");
INSERT INTO tb_person values (2, "matthias");
INSERT INTO tb_person values (3, "dieter");
INSERT INTO tb_person values (4, "hannes");

INSERT INTO tb_relationship values (1, 1, 2);
INSERT INTO tb_relationship values (2, 2, 1);
INSERT INTO tb_relationship values (3, 1, 3);
INSERT INTO tb_relationship values (4, 1, 4);
INSERT INTO tb_relationship values (5, 4, 3);


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoMNRelation
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret


### Root/type.properties ###

allPeople = collection(Person)
allPeople.accessname = person_name
```

```
### Root/functions.js ###

function main_action() {
   // list all People and their friends
   for (var i=0; i<root.allPeople.count(); i++) {
      var p = root.allPeople.get(i);
      res.write("<h3>" + p.name + "</h3>");
      res.write(p.relations.count() + " Friend(s)");
      for (var j=0; j<p.relations.count(); j++) {
         var rel = p.relations.get(j);
         var f = rel.person02;
         res.write(" " + f.name);
      }
      res.write("<br />Friend of " + p.backrelations.count() + " Person(s)");
      for (var j=0; j<p.backrelations.count(); j++) {
         var rel = p.backrelations.get(j);
         var f = rel.person01;
         res.write(" " + f.name);
      }
   }

   var michi = root.allPeople.get("michi");
   var matthias = root.allPeople.get("matthias");
   var dieter = root.allPeople.get("dieter");
   res.write("<hr><b>Is matthias a friend of michi?</b> ");
   res.write(michi.isFriendOf(matthias) ? "yes" : "no");
   res.write("<hr><b>Is matthias a friend of dieter?</b> ");
   res.write(dieter.isFriendOf(matthias) ? "yes" : "no");
}



### Person/type.properties ###

_db = jad
_table = tb_person

_id = person_id
_parent = root.allPeople

name = person_name

relations = collection(Relationship)
relations.local = person_id
relations.foreign = rel_person01_id
relations.accessname = rel_person02_id

backrelations = collection(Relationship)
backrelations.local = person_id
backrelations.foreign = rel_person02_id
backrelations.accessname = rel_person01_id


### Person/functions.js ###

function isFriendOf(p) {
   if (!p) return false;
   // determine whether we have a relation to that person
   /*
   // the clumsy way
   for (var i=0; i<this.relations.count(); i++) {
```

```
      if (this.relations.get(i).person02 == p) return true;
   }
   return false;
   */
   // the smart way
   if (this.relations.get(p._id+"") != null)
      return true;
   else
      return false;
}


### Relationship/type.properties ###

_db = jad
_table = tb_relationship

_id = rel_id
_parent = person01.relations

person01 = object(Person)
person01.local = rel_person01_id
person01.foreign = person_id

person02 = object(Person)
person02.local = rel_person02_id
person02.foreign = person_id
```

demoMountpoint.zip

```
### mysql.sql ###

CREATE DATABASE demoMountpoint;
USE demoMountpoint;

GRANT ALL ON demoMountpoint.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_person (
  person_id MEDIUMINT(10) NOT NULL,
  person_name TINYTEXT,
  PRIMARY KEY (person_id)
);

INSERT INTO tb_person values (1, "michi");
INSERT INTO tb_person values (2, "matthias");
INSERT INTO tb_person values (3, "dieter");
INSERT INTO tb_person values (4, "hannes");


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoMountpoint
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret


### Root/type.properties ###

allPeople = collection(Person)

manage = mountpoint(SysMgr)
```

```
### Root/functions.js ###

function main_action() {
   res.writeln("HELLO WORLD!");
   res.writeln("<a href=" + root.manage.href("main") + ">SysMgr</a>");
   for (var i=0; i<root.allPeople.count(); i++) {
      var p = root.allPeople.get(i);
      res.writeln("<a href=" + p.manage.href("main") + ">SysMgr of " + p.name + "</a>");
   }
}


### Person/type.properties ###

_db = jad
_table = tb_person

_id = person_id
_parent = root.allPeople

name = person_name

manage = mountpoint(SysMgr)


### SysMgr/type.properties ###

_children = collection(Person)


### SysMgr/functions.js ###

function main_action() {
   res.writeln("TOP SECRET STUFF");
   res.writeln("this: " + this);
   res.writeln("parent: " + this._parent);
   res.writeln("children: " + this.count());
}
```

Rendering Framework

FIXME

```
IGNORE THE FOLLOWING NOTES FOR NOW, AND JUST TAKE A LOOK AT THE SAMPLE CODE !!

Rendering Framework
  Skins
    file-based
    res.skinpath
    db-based
    skinmanager
    .skin
    renderSkin
    renderSkinAsString
    createSkin()
    res.data
    allowMacro()
    containsMacro()
  Macros
```

```
    prefix, suffix, encoding, default
    macro handlers
      global
      default handlers
        response
        request
        session
        param
      path/object hierarchy handler
      res.handlers
    all properties as macro (but not password)
    res.push, res.pop
    res.write or return !
  string manipulation
    format
    encodeXml
    encode
  Helma render framework tutorial
    url: http://helma.org/docs/devguide/framework/
    Basics: The Helma Request Cycle
    Separating Presentation from Logic
    Actions, Skins and Macros
    Enter the Prototypes
    The render framework in action
    Macro Handlers
    Macro Attributes
    Manipulating Macro Handlers with res.handlers
    Advanced Skins: Skin Paths
    Appendix: typical skinset setup with relational db mapping
```

http://helma.org/docs/devguide/framework/

demoRenderingSkinHsp.zip

```
### mysql.sql ###

CREATE DATABASE demoRenderingSkinHsp;
USE demoRenderingSkinHsp;

GRANT ALL ON demoRenderingSkinHsp.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_person (
  person_id MEDIUMINT(10) NOT NULL,
  person_name TINYTEXT,
  PRIMARY KEY (person_id)
);

INSERT INTO tb_person values (1, "michi");
INSERT INTO tb_person values (2, "matthias");
INSERT INTO tb_person values (3, "dieter");
INSERT INTO tb_person values (4, "hannes");


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoRenderingSkinHsp
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret
```

```
### Global/functions.js ###

function aGlobalMacro_macro() {
   res.write("a global macro");
   return;
}


### Global/main.skin ###

<html>
  <head>
  </head>
  <body style="background-color:yellow">
    <% response.body %>
  </body>
</html>


### HopObject/functions.js ###

function href_macro(param) {
   var action = param.action ? param.action : "";
   res.write(this.href(action));
   return;
}

function skin_macro(param) {
   this.renderSkin(param.name);
   return;
}


### Root/type.properties ###

persons = collection(Person)


### Root/functions.js ###

function main_action() {

   // write directly to response, but we 'pop' that string
   // out of the response-buffer
   res.push();
   this.renderSkin("listAllPersons");
   var str = res.pop();

   // render hsp-template and string to response, but we 'pop' that string
   // out of the response-buffer
   res.push();
   this.createPersonTemplate();
   this.renderSkin("createPerson");
   str += res.pop();

   // render hsp-template as string
   str += this.createPersonTemplate_as_string();
   // render skins as string
   str += this.renderSkinAsString("createPerson");

   res.data.body = str;
   // actually render a skin directly to the response
```

```
   renderSkin("main");
   return;
}


function dummy_macro(param) {
   var until = param.until ? parseInt(param.until, 10) : 5;
   for (var i=0; i<until; i++) {
      res.write(i + " ");
   }
   return;
}


function listAllPersons_macro(param) {
   for (var i=0; i<root.persons.count(); i++) {
      var p = root.persons.get(i);
      p.renderSkin("listitem");
   }
   return;
}


### Root/createPerson.skin ###

<form action="<% this.href action="createPerson" %>" method="POST">
  <input type="text" name="name">
  <input type="submit" name="send">
  <% this.dummy until="10" %>
  <% this.skin name="hello" %>
  <% aGlobalMacro %>
</form>


### Root/createPersonTemplate.hsp ###

<form action="<%= this.href("createPerson") %>" method="POST">
  <input type="text" name="name">
  <input type="submit" name="send">
  <% var until = 10;
  for (var i=0; i<until; i++) res.write(i + " "); %>
</form>


### Root/hello.skin ###

Hello!


### Root/listAllPersons.skin ###

<h2>List of all Persons</h2>
<% this.listAllPersons prefix="<ul>" suffix="</ul>" default="no persons" %>


### Person/type.properties ###

_db = jad
_table = tb_person

_id = person_id
_parent = root.persons
```

```
name = person_name


### Person/functions.js ###

function getInfo() {
   return "just a person (" + this.name + ")";
}


### Person/listitem.skin ###

<li>Name: <% this.name %></li>
```

demoRenderingMacroHandlers.zip

```
### mysql.sql ###

CREATE DATABASE demoRenderingMacroHandlers;
USE demoRenderingMacroHandlers;

GRANT ALL ON demoRenderingMacroHandlers.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_person (
  person_id MEDIUMINT(10) NOT NULL,
  person_name TINYTEXT,
  person_org_id MEDIUMINT(10) unsigned,
  PRIMARY KEY (person_id)
);


CREATE TABLE tb_organisation (
   org_id MEDIUMINT(10) unsigned NOT NULL,
   org_name TINYTEXT,
   org_country TINYTEXT,
   PRIMARY KEY (org_id)
);

CREATE INDEX idx_pers_org ON tb_person (person_org_id);

INSERT INTO tb_organisation values (1, "knallgrau", "at");
INSERT INTO tb_organisation values (2, "helma", "at");
INSERT INTO tb_organisation values (3, "amazon", "us");
INSERT INTO tb_organisation values (4, "ebay", "us");
INSERT INTO tb_organisation values (5, "yahoo", "us");

INSERT INTO tb_person values (1, "michi", 1);
INSERT INTO tb_person values (2, "matthias", 1);
INSERT INTO tb_person values (3, "dieter", 1);
INSERT INTO tb_person values (4, "hannes", 2);


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoRenderingMacroHandlers
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret


### Global/main.skin ###
```

```
<html>
  <head>
  </head>
  <body style="background-color:yellow">
    <% response.body %>

    <hr />Macro Handlers Demonstration:<br />
      <li>response.key01: <% response.key01 %></li>
      <li>request.key01: <% request.key01 %></li>
      <li>session.key01: <% session.key01 %></li>
      <li>param.key01: <% param.key01 %></li>
  </body>
</html>


### HopObject/functions.js ###

function href_macro(param) {
   var action = param.action ? param.action : "";
   res.write(this.href(action));
   return;
}

function skin_macro(param) {
   this.renderSkin(param.name);
   return;
}


### Root/type.properties ###

_children = collection(Organisation)
_children.accessname = org_name

persons = collection(Person)

manage = mountpoint(SysMgr)


### Root/functions.js ###

function main_action() {

   res.handlers.SystemManager = root.manage;

   res.push();
   res.write("<h2>List of all Persons</h2>");
   for (var i=0; i<root.persons.count(); i++) {
      var p = root.persons.get(i);
      p.renderSkin("info");
   }

   // create a skin from a string
   var str = "<hr />Root Href: <% this.href %>";
   var sk = createSkin(str);
      // sk.allowMacro();
      // sk.containsMacro();
   this.renderSkin(sk);

   res.data.body = res.pop();
```

```
   // macro handler demo
   res.data.key01 = "value01";
   // set req.data.key01 by appending '?key01=value01' to the URL
   session.data.key01 = "value01";
   var obj = new Object();
      obj.key01 = "value01";
      obj.key02 = "value02";
   renderSkin("main", obj);
   return;
}


### Organisation/type.properties ###

_db = jad
_table = tb_organisation

_id = org_id
_parent = root

name = org_name

persons = collection(Person)
persons.local = org_id
persons.foreign = person_org_id

allPersons = collection(Person)


### Person/type.properties ###

_db = jad
_table = tb_person

_id = person_id
_parent = org.persons

name = person_name

org = object(organisation)
org.local = person_org_id
org.foreign = org_id


### Person/functions.js ###

function main_action() {

   res.handlers.SystemManager = root.manage;

   res.data.body = this.renderSkinAsString("info");
   renderSkin("main");
}


### Person/info.skin ###

<li>Name: <a href="<% this.href %>"><% this.name %></a> <% Organisation.name %> – <% SystemManage
```

demoRenderingSkinsets.zip

```
### mysql.sql ###
```

```
CREATE DATABASE demoRenderingSkinsets;
USE demoRenderingSkinsets;

GRANT ALL ON demoRenderingSkinsets.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_person (
  person_id MEDIUMINT(10) NOT NULL,
  person_name TINYTEXT,
  person_org_id MEDIUMINT(10) unsigned,
  PRIMARY KEY (person_id)
);

INSERT INTO tb_person values (1, "michi", 1);
INSERT INTO tb_person values (2, "matthias", 1);
INSERT INTO tb_person values (3, "dieter", 1);
INSERT INTO tb_person values (4, "hannes", 2);


CREATE TABLE tb_skinset (
   skinset_id MEDIUMINT(10) unsigned NOT NULL,
   PRIMARY KEY (skinset_id)
);

CREATE TABLE tb_skin (
   skin_id MEDIUMINT(10) unsigned NOT NULL,
   skin_proto TINYTEXT,
   skin_name TINYTEXT,
   skin_skin TEXT,
   skin_set_id MEDIUMINT(10) unsigned NOT NULL,
   PRIMARY KEY (skin_id)
);

INSERT INTO tb_skinset values (1);
INSERT INTO tb_skinset values (2);
INSERT INTO tb_skinset values (3);

INSERT INTO tb_skin values (1, "Global", "main", "<body bgcolor=green><% response.body %></body>"
INSERT INTO tb_skin values (2, "Global", "main", "<body bgcolor=blue><% response.body %></body>",


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoRenderingSkinsets
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret


### Global/main.skin ###

<html>
  <head>
  </head>
  <body bgcolor="yellow">
    <% response.body %>
  </body>
</html>


### HopObject/functions.js ###
```

```
function href_macro(param) {
   var action = param.action ? param.action : "";
   res.write(this.href(action));
   return;
}

function skin_macro(param) {
   this.renderSkin(param.name);
   return;
}


### layouts/gray/Global/main.skin ###

<body bgcolor="gray"><% response.body %></body>


### layouts/orange/Global/main.skin ###

<body bgcolor="orange"><% response.body %></body>


### Root/type.properties ###

persons = collection(Person)

skinsets = collection(Skinset)


### Root/functions.js ###

function main_action() {

   if (req.data.skinsetId) {
      // we define the lookup-path for the skins
      // the file-based skins residing in the code-directly are always used as the last fallback
      res.skinpath = new Array(root.skinsets.getById(req.data.skinsetId).skinmanager, app.dir + "
   }
   res.write("<li><a href=" + this.href() + "?skinsetId=1>green</a>");
   res.write("<li><a href=" + this.href() + "?skinsetId=2>blue</a>");
   res.write("<li><a href=" + this.href() + "?skinsetId=3>default</a>");

   res.push();
   res.write("<h2>List of all Persons</h2>");
   for (var i=0; i<root.persons.count(); i++) {
      var p = root.persons.get(i);
      p.renderSkin("info");
   }

   res.data.body = res.pop();

   renderSkin("main");
   return;
}


### Skinset/type.properties ###

_db = jad
_table = tb_skinset

_id = skinset_id
```

```
_parent = root.skinsets

skinmanager = collection(Skin)
skinmanager.local = skinset_id
skinmanager.foreign = skin_set_id
skinmanager.group = skin_proto
skinmanager.accessname = skin_name


### Skin/type.properties ###

_db = jad
_table = tb_skin

_id = skin_id
_parent = skinset.skinmanager
_name = skin_name

skinset = object(Skinset)
skinset.local = skin_id
skinset.foreign = skin_set_id

proto = skin_proto
name = skin_name
skin = skin_skin


### Person/type.properties ###

_db = jad
_table = tb_person

_id = person_id
_parent = root.persons

name = person_name


### Person/info.skin ###

<li>Name: <a href="<% this.href %>"><% this.name %></a></li>
```

Session & User Management

FIXME

```
IGNORE THE FOLLOWING NOTES FOR NOW, AND JUST TAKE A LOOK AT THE SAMPLE CODE !!

Session & User Management
  User
    _name
    password
    app.registerUser(name, pwd)
  session
    HopSession
      what if cookies are disabled
    session.data
    session.user
    session.login(name, pwd)
    session.login(User)
    session.logout()
```

```
    sessionTimeout
  onLogout()
```

demoSessionUser.zip

```
### mysql.sql ###

CREATE DATABASE demoSessionUser;
USE demoSessionUser;

GRANT ALL ON demoSessionUser.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_user (
  user_id MEDIUMINT(10) NOT NULL,
  user_name TINYTEXT,
  user_password TINYTEXT,
  PRIMARY KEY (user_id)
);

INSERT INTO tb_user values (1, "michi", "ihcim");
INSERT INTO tb_user values (2, "matthias", "saihttam");
INSERT INTO tb_user values (3, "dieter", "reteid");
INSERT INTO tb_user values (4, "hannes", "sennah");


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoSessionUser
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret


### Root/type.properties ###

users = collection(User)
users.accessname = user_name


### Root/functions.js ###

function main_action() {
   // first we try to auto-login the user
   if (req.data.autoLoginName && root.users.get(req.data.autoLoginName)) {
      var usr = root.users.get(req.data.autoLoginName);
      var hash = Packages.helma.util.MD5Encoder.encode(usr.name + usr.password);
      if (hash == req.data.autoLoginHash) session.login(usr);
   }

   // display link to login, resp logout
   if (session.user != null) {
      res.writeln("Hi " + session.user.name + "!<br />");
      res.writeln("<a href=" + root.href("logout") + ">Logout</a>");
   } else {
      res.writeln("<a href=" + root.href("login") + ">Login</a>");
      res.writeln("<a href=" + root.href("register") + ">Register</a>");
   }

   // list all Users
   res.write("<hr>");
   res.write("<h2>Users</h2>");
   for (var i=0; i<root.users.count(); i++) {
```

```
      res.write("<li>" + root.users.get(i).name);
   }
   return;
}

function login_action() {
   if (req.data.login) {
      var name = req.data.username;
      var pass = req.data.password;
      var usr = root.users.get(name);
      if (usr && usr.password == pass) {
         session.login(usr);
         if (req.data.remember) {
            var hash = Packages.helma.util.MD5Encoder.encode(name + pass);
            res.setCookie("autoLoginName", name, 30);
            res.setCookie("autoLoginHash", hash, 30);
         }
         res.redirect(root.href());
      } else {
         res.write("Login failed!");
      }
   }
   this.renderSkin("login");
}

function logout_action() {
   session.logout();
   res.setCookie("autoLoginName", "");
   res.setCookie("autoLoginHash", "");
   res.redirect(root.href());
}

function register_action() {
   if (req.data.login) {
      var name = req.data.username;
      var pass = req.data.password;
      if (root.users.get(name) == null && pass) {
         var usr = new User();
         usr.name = name;
         usr.password = pass;
         root.users.add(usr);
         session.login(usr);
         res.redirect(root.href());
      } else {
         res.write("Registration failed!");
      }
   }
   this.renderSkin("register");
}


### Root/login.skin ###

<form action="<% this.href action="login" %>" method="POST">
  name: <input type="text" name="username"><br />
  pass: <input type="password" name="password"><br />
  remember me? <input type="checkbox" name="remember" value="1"><br />
  <input type="submit" name="login" value="Login!">
</form>


### Root/register.skin ###
```

```
<form action="<% this.href action="register" %>" method="POST">
  name: <input type="text" name="username"><br />
  pass: <input type="password" name="password"><br />
  <input type="submit" name="login" value="Register!">
</form>


### HopObject/functions.js ###

function href_macro(param) {
   var action = param.action ? param.action : "";
   res.write(this.href(action));
   return;
}

function skin_macro(param) {
   this.renderSkin(param.name);
   return;
}


### User/type.properties ###

_db = jad
_table = tb_user

_id = user_id
_parent = root.users

name = user_name
password = user_password
```

Extensions

Direct DB Connection

FIXME

File Extension

FIXME

FTP Extension

FIXME

See http://helma.org/docs/reference/ftp/ for now.

Image Extension

FIXME

See http://helma.org/docs/reference/image/ for now.

Mail Extension

FIXME

mail.charset, ISO−8859−15

mail.host

smtp

See http://helma.org/docs/reference/mail/ for now.

XmlObject Extension

FIXME

separator

_mode

See http://helma.org/docs/reference/xml/ for now.

XmlRpc Client

FIXME

See http://helma.org/docs/reference/xmlrpc/ for now.

XmlRpc Server

FIXME

See http://helma.org/docs/reference/xmlrpc/ for now.

3rd party Extensions

e.g. Flash FIXME

app.properties.extensions

demoExtensions.zip

```
### mysql.sql ###

CREATE DATABASE demoSessionUser;
USE demoSessionUser;

GRANT ALL ON demoSessionUser.* TO helma@localhost IDENTIFIED BY 'secret';

CREATE TABLE tb_user (
  user_id MEDIUMINT(10) NOT NULL,
  user_name TINYTEXT,
  user_password TINYTEXT,
```

```
  PRIMARY KEY (user_id)
);

INSERT INTO tb_user values (1, "michi", "ihcim");
INSERT INTO tb_user values (2, "matthias", "saihttam");
INSERT INTO tb_user values (3, "dieter", "reteid");
INSERT INTO tb_user values (4, "hannes", "sennah");


### app.properties ###

XmlRpcAccess = api.demoXmlRpcServer
XmlRpcHandlerName = *


### db.properties ###

jad.url      = jdbc:mysql://localhost/demoSessionUser
jad.driver   = com.mysql.jdbc.Driver
jad.user     = helma
jad.password = secret


### static/foto.jpg ###
-> put any king of image here


### Root/type.properties ###

api = mountpoint(Api)


### Api/functions.js ###

function demoXmlRpcServer() {
   return "HELLO WORLD!";
}


### Root/functions.js ###

function main_action() {
   return;
}


// make sure that you start Helma with enabled XML-RPC-Port at 8081
// and that you have the following two lines in app.properties:
//   XmlRpcAccess = api.demoXmlRpcServer
//   XmlRpcHandlerName = *
function demoXmlRpcClient_action() {
   var client = new Remote("http://betty.userland.com/RPC2");
   var i = 12;
   var obj = client.examples.getStateName(i);
   if (!obj.error) {
      res.writeln(i + ": " + obj.result);
   } else {
      res.debug(obj.error);
   }
   res.writeln("-----------------------");
   var client = new Remote("http://localhost:8081/");
   var obj = client.api.demoXmlRpcServer(i);
```

```
   if (!obj.error) {
      res.writeln(obj.result);
   } else {
      res.debug(obj.error);
   }
}


function demoDB_action() {
   var dbc = getDBConnection("jad");
   var sql = "SELECT * from tb_user";
   var rows = dbc.executeRetrieval(sql);
   if (rows) {
      while (rows.next()) {
         var userId = rows.getColumnItem("user_id");
         res.writeln(userId + ": " + rows.getColumnItem("user_name"));
      }
      rows.release();
      var sql = "INSERT into tb_user values (" + (userId+1) + ", \"joe" + (userId+1) + "\", \"sec
      var i = dbc.executeCommand(sql);
      if (i > 0) {
         res.writeln("Inserted new User " + i);
      } else {
         res.debug(dbc.getLastError());
      }
   } else {
      res.debug(dbc.getLastError());
   }

   dbc.release();
   return;
}


function demoFtp_action() {
   // FIXME
   return;
}


function demoXml_action() {
   res.contentType = "text/plain";
   var obj = new HopObject();
      obj.name = "michi";
      obj.height = 180;
   res.write(uneval(obj));
   res.write("\n-----------------\n");

   var str = Xml.writeToString(obj)
   res.write(str);
   res.write("\n-----------------\n");

   var obj2 = Xml.readFromString(str);
   obj2.height++;
   res.write(uneval(obj2));
   res.write("\n-----------------\n");
}


function demoImage_action() {
   var staticDir = new File(app.dir, "static");
   var img = new Image(staticDir.getAbsolutePath() + "/foto.jpg");
```

```
   var w = img.getWidth();
   var h = img.getHeight();
   res.writeln("Image Original: " + w + "x" + h);
   img.resize(parseInt(w/2), parseInt(h/2));
   img.saveAs(staticDir + "/foto_small.jpg");
   img.dispose();
   res.write("<img src=/static/justademo/foto_small.jpg>");
}


function demoMail_action() {
   var m = new Mail();
   m.addTo("michi@knallgrau.at", "michi");
   m.setFrom("bill@microsoft.com", "Bill Gates");
   m.setSubject("MS Office for free $$");
   m.addText("Wow! This is so incredible.");
   var f = new java.io.File(app.dir, "test.txt");
   if (f.exists()) m.addPart(f);
   m.send();
   if (m.status == 0) {
      res.write("Mail OK");
   } else {
      res.write("Error occurred! Make sure that you set smtp in your app.properties correctly!" +
   }
   return;
}


function demoFile_action() {
   var dir = new File(app.dir);
   var files = dir.list();
   for (var i in files) {
      var f = new File(dir, files[i]);
      if (f.isFile()) {
         res.writeln("File " + f.getName());
      } else {
         res.writeln("Dir " + f.getName());
      }
   }
   res.write("<hr>");
   var staticDir = new File(dir, "static");
   if (staticDir.exists() == false) staticDir.mkdir();
   var f = new File(staticDir + "/test.txt");
   res.writeln("File:" + f.getAbsolutePath());
   if (f.exists()) {
      res.writeln("Read and remove " + f.getName());
      res.write(format(f.readAll()));
      f.remove();
   } else {
      f.open();
      var nr = 10;
      for (var i=0; i<nr; i++) {
         f.writeln("Line " + i);
      }
      f.close();
      res.writeln("Wrote " + nr + " lines to " + f.getName());
   }
   return;
}
```

Scheduler

schedulerInterval

cronJobs

FIXME

http://helma.org/development/rfc/cronjobs/

demoScheduler.zip

```
### app.properties ###

# schedulerInterval = 5

cron.sthg.function = doSomething

# cron.sthg.year = *
# cron.sthg.month = 3,4,5
# cron.sthg.day = 1-7,10-20
# cron.sthg.weekday = monday,friday
# cron.sthg.hour = 1-6
# cron.sthg.minute = 0,5,10,15,20,25,30

# cron.sthg.timeout = 60000


### Global/functions.js ###

function doSomething() {
   app.log("doSomething now actually does something!!!");
   return;
}
```

Logging

FIXME

See app.log() [http://helma.org/stories/48012/] for now.

AspectJS

FIXME

Performance Tuning

FIXME

Clustering Helma

A cluster setup for Helma applications can be implemented at the web−node−level as well as at the db−node−level.

At the web−node−level several Helma instances (running on different machines) need to be able to communicate with each other. This can be acchieved by using the HelmaSwarm−extension. For further details please refer to the documentation coming along with HelmaSwarm. In order to distribute the incoming requests to these Helma instances either a load balancer (hardware or software based) can be installed, or load balancing on the DNS−level can be configured (see http://hacks.oreilly.com/pub/h/79), with the latter being a much cheaper, but less flexible solution.

At the db−node−level the clustering depends of course on the database system in use. If you use MySQL, then you can implement a Master/Slave setup on the db−side, and have the DB−requests being distributed on these instances via the ReplicationDriver that is part of MySQL's Connector/J (version 3.1.11 or higher; see http://dev.mysql.com/doc/mysql/en/cj−replication−connection.html).

```
### db.properties ###
antville.url      = jdbc:mysql://master,slave1,slave2,slave3/test
antville.driver   = com.mysql.jdbc.ReplicationDriver
antville.user     = user
antville.password = pass
antville.autoReconnect = true
antville.roundRobinLoadBalance = true
```

Calling Helma from Commandline

Besides having Helma run as a server, it is also possible to run a single script from command line. Helma will start up, execute the script and terminate immediately. A possible usage scenario would be for example to perform certain update scripts from command line, if an installation of an application should be upgraded. Another one is to generate the HelmaDocs via the manage−application without actually having to start that application.

Usage:

java −cp launcher.jar helma.main.launcher.Commandline [options] [appname].[function] [argument−list]

Example:

java −cp launcher.jar helma.main.launcher.Commandline twoday.m.stories.1.dummy mi mo

In this example the function dummy() on the Story with the Id 1 is called, with the arguments 'mi' and 'mo' being passed to that function.

helmaLib

FIXME

antvilleLib

FIXME

Reference

Helma Core

Global Functions

This sections documents methods available in a global context. For further details also see the JavaDocs for helma.scripting.rhino.GlobalObject.

authenticate(String user, String pwd) returns Boolean

Authenticates a user against a standard Unix password file. Returns true if the provided credentials match an according entry in the files `[AppDir]/passwd` or `[HelmaHome]/passwd`. The stored passwords in these files must be either encrypted with the unix crypt algorithm, or with the MD5 algorithm. The Apache web server provides the utility tool '***htpasswd***' to generate such password files.

createSkin(String str) returns SkinObject

Creates a SkinObject from the passed String. The returned object can be passed to the global functions renderSkin, resp. renderSkinAsString. Also see the reference on the SkinObject.

encode(String str) returns String

Performs the following string manipulations:

- All line breaks (i.e. carriage returns and line feeds) are replaced with
 tags.
- All special characters (i.e. non ASCII) are being replaced with their equivalent HTML entity.
- Existing markup tags are being encoded.

and returns the modified string.

encodeForm(String str) returns String

Performs the following string manipulations:

- All special characters (i.e. non ASCII) are being replaced with their equivalent HTML entity.
- Existing markup tags are being encoded.

and returns the modified string.

encodeXml(String str) returns String

Performs the following string manipulations:

- All special characters (i.e. non ASCII) are being replaced with their equivalent HTML entity.
- Existing tags, single and double quotes, as well as ampersands are being encoded.
- Some invalid XML characters below '0x20' are removed.

and returns the modified string.

format(String str) returns String

Performs the following string manipulations:

- All line breaks (i.e. carriage returns and line feeds) are replaced with
 tags, with the exception of line breaks that follow certain block tags (e.g. table, div, h1, ..).
- All special characters (i.e. non ASCII) are being replaced with their equivalent HTML entity.

and returns the modified string.

formatParagraphs(String str) returns String

Performs the following string manipulations:

- All line breaks (i.e. carriage returns and line feeds) are replaced with
 tags, with the exception of line breaks that follow certain block tags (e.g. table, div, h1, ..).
- Paragraphs are detected, and surrounded with <p> tags. Two following line breaks are considered to indicate a paragraph.

- All special characters (i.e. non ASCII) are being replaced with their equivalent HTML entity.

and returns the modified string.

getDBConnection(String dbsource) returns DatabaseObject

Connects to a relational database, and returns a DatabaseObject representing that connection. The passed string must match one of data sources being defined in `[AppDir]/db.properties`, or an error will be thrown. Also see the reference on the DatabaseObject. (FIXME LINK)

getHtmlDocument(Object src) returns org.apache.html.dom.HTMLDocumentImpl

Tries to parse an object to a XML DOM tree using Xerces' HTML parser (nekohtml). The argument must be either a URL, a piece of XML, an InputStream or a Reader. Returns an instance of org.apache.html.dom.HTMLDocumentImpl. See the JavaDocs for that class for further details (LINK: −> JavaDocs; LINK: other recommended HTML parsers)

CHECK with current Online Docs

getProperty(String propname, String defvalue) returns String

Returns any property defined in `[AppDir]/app.properties`, resp. [HelmaHome]/server.properties that matches the passed property name. This lookup is case−insensitive. Through the optional second parameter it is possible to define a default value that is being returned, in case that that property has not been set.

getURL(String url, Object option) returns MimePart

Retrieves a file/document from the passed URL as a MimePart Obejct, and therefore functions as a minimalist version of a HttpClient. The optional second parameter can either contain a (last−modified) date object, or an eTag string, which will be passed along with the Http request to the specified URL.

Also see the reference on the MimePart Object.

getXmlDocument(Object src) returns Object

Tries to parse an object to a XML DOM tree using the Crimson' Parser. The argument must be either a URL, a piece of XML, an InputStream or a Reader. Returns an instance of org.apache.crimson.tree.XmlDocument. See the JavaDocs for that class for further details (LINK: −> JavaDocs; LINK: other recommended XML parsers)

CHECK with current Online Docs

renderSkin(SkinObject skin, Object param)

Renders the passed SkinObject to the response buffer.

The properties of the optional parameter object are accessible within the skin through the 'param' macro handler.

renderSkin(String skinname, Object param)

Renders a global skin matching the passed name to the response buffer.

The properties of the optional parameter object are accessible within the skin through the 'param' macro handler.

renderSkinAsString(SkinObject skin, Object param) returns String

Returns the result of the rendered SkinObject.

The properties of the optional parameter object are accessible within the skin through the 'param' macro handler.

renderSkinAsString(String skinname, Object param) returns String

Returns the result of a rendered global skin matching the passed name to the response buffer.

The properties of the optional parameter object are accessible within the skin through the 'param' macro handler.

seal(Object obj)

Seals an object, and prevents any further modifications of that object. If any property is tried to be modified after it has been sealed, an error is thrown.

stripTags(String str) returns String

Removes any markup tags contained in the passed string, and returns the modified string.

write(String str)

Writes a string to java.lang.System.out, i.e. to the console. Useful for debugging purposes.

writeln(String str)

Writes a string together with a line break to java.lang.System.out, i.e. to the console. Useful for debugging purposes.

Application Object 'app'

Within the scripting environment Helma provides a host object representing the application itself. This object is always present, and does not need to be instantiated.

For further details also see the JavaDocs for helma.framework.core. ApplicationBean. Since that class is a JavaBean all get− and set−methods are also directly available as properties of that object.

helma.framework.core.Application app.__app__

This property contains a reference to Helma's application class, representing the currently running application, and offers some additional public methods. See Helma's JavaDocs on helma.framework.core.Application for more information.

app.getActiveThreads() returns Int

Returns the number of currently active threads (=request evaluators).

app.getActiveUsers() returns User[]

Returns an array of Users, that are currently logged in.

app.addCronJob(String functionName)

Adds a global function to the list of CronJobs, that are being called periodically. If the property 'schedulerInterval' has not been set otherwise in app.properties, the function will be called every 60 seconds.

app.addCronJob(String functionName, String year, String month, String day, String weekday, String hour, String minute)

Adds a global function to the list of CronJobs, that are being called periodically. By passing along further arguments it is possible to define at what times that function should be called. The same syntax ('*', '1,10,15', '1−5',..) as for Unix' crontab file can be used. Note, that if the property 'schedulerInterval' has been set in app.properties below 60, the function will be called several times in the minute, that it is supposed to run.

app.getAppDir() returns String

Returns the absolute path to the application directory. For multiple repositories this is either, the directory specified as 'appdir' in apps.properties, or the first FileRepository occurring in the repository list.

app.getCacheUsage() returns Int

Returns the number of currently cached objects for the current application.

app.getCronJobs() returns Object

Returns a JavaScript object with the function name as property names and the helma.util.CronJob instance as property values.

app.clearCache()

Removes all objects from the object cache for the current application. By calling this method it is possible to force Helma to fetch all objects fresh from the database.

app.countSessions() returns Int

Returns the number of currently active sessions.

app.createSession(String sessionID) returns SessionObject

Creates a SessionObject with the passed sessionID as its unique identifier. If such a session with that ID already exists, it will return that session.

Object app.data

This property offers the possibility to store arbitrary data on an application wide level, and is available as long as the application is running. Note, that this can just be used as a temporary storage (i.e. as a 'cache'), since this data is not stored persistently within Helma, and is lost when the application is restarted.

app.debug(String str)

Writes a string to the eventLog−file, if debug is set to true in app.properties.

app.debug(String logname, String str)

Writes a string to a logfile named 'logname', if debug is set to true in app.properties.

app.getDir() returns String

Returns the absolute path to the application directory. For multiple repositories this is either, the directory specified as 'appdir' in apps.properties, or the first FileRepository occurring in the repository list.

app.getErrorCount() returns Int

Returns the number of errors that have occurred since the application has been started.

app.getFreeThreads() returns Int

Returns the number of currently free threads (i.e. request evaluators). This is equivalent to app.getMaxThreads() minus app.getActiveThreads().

app.getSession(String sessionID) returns SessionObject

Returns a SessionObject identified through the passed sessionID, if such a session exists.

app.getSessions() returns SessionObject[]

Returns an array of all currently active sessions, represented as SessionObjects.

app.getSessionsForUser(String username) returns SessionObject[]

DEPRECATED

Returns an array of all currently active sessions, which have been associated with a certain user, that is identified through the passed username. Returns an empty array if no such User can be found.

app.getSessionsForUser(User usr) returns SessionObject[]

Returns an array of all currently active sessions, which have been associated with the passed User. Returns an empty array if no User is passed.

app.getSkinfiles() returns Map

Returns a Map that allows access to all defined file−based skins. The map contains for each prototype one entry (for prototypes containing also uppercase letters, also an entry with the lowercased prototype name is contained). This entry contains for each skin file residing in that prototype directory, an entry with the name of the file and the content/source of that file as its value.

app.getUser(String username) returns User

DEPRECATED

Returns a User identified through the passed username. The prototype User must have a username defined through the '_name'−property in type.properties for this to work.

app.log(String str)

Writes a string to the eventLog−file.

app.log(String logname, String str)

Writes a string to a logfile named 'logname'.

Int app.maxThreads
: This property contains the maximum number of threads (=request evaluators) that are being created by Helma to handle incoming requests. This property is read− and write−able.

Map app.modules
: This property offers a dedicated place to store module−related data on an application wide level. Note, that this can just be used as a temporary storage (i.e. as a 'cache'), since this data is not stored persistently within Helma, and is lost when the application is restarted.

app.getName() returns String
: Returns the name of the current application, i.e. the name used in apps.properties.

Map app.properties
: This property offers access to each defined key/value pair defined in either app.properties or in server.properties. This property is read−only.

app.getRegisteredUsers() returns User[]
: DEPRECATED

  Returns an array of all created Users.

app.registerUser(String username, String password)
: DEPRECATED

  Creates a new HopObject of prototype User, stores it persistently, and associates the current session with that User. The prototype User must have a username defined through the '_name'−property in `type.properties`, and must have a property named 'password'.

app.removeCronJob(String functionName)
: Removes a CronJob, identified through the passed function name, from the list of CronJobs.

app.getRequestCount() returns Int
: Returns the number of web requests that occurred since the application has been started.

app.getServerDir() returns String
: Returns the absolute path to the home directory of this Helma installation. If Helma is run in embedded mode, this will be equal to the application directory.

app.getUpSince() returns Date
: Returns the timestamp of when that application has been started.

app.getXmlrpcCount() returns Int
: Returns the number of XmlRpc requests that occurred since the application has been started.

Hop Object

A HopObject is an extended JavaScript object, providing the concept of so called collections, i.e. containers of other HopObjects. Each instantiated prototype in Helma is a scripted HopObject, with additional HopObjects for each of its defined collections and mountpoints.

HopObjects can be in transient state or are persistently mapped on a database. For further details see the according section in the User's Guide.

http://helma.org/docs/reference/hopobject/

see JavaDocs for helma.scripting.rhino.HopObject

HopObject.getById(Int id, String prototypename) return HopObject
: Fetches a HopObject of a certain prototype through its ID and its prototype name. The prototype name can either be passed as a second argument, or as an alternative, the function can also be called on the prototype itself with a single argument (e.g. Story.getById(123)).

In case of multiple prototypes being mapped on the same table (which is for instance the case with inherited prototypes) Helma will not check whether the prototype of the fetched object actually matches the specified prototype.

Note, that this refers to the static method 'getById', not to be mixed up with the method getById called on a specific HopObject.

obj.add(HopObject child) returns Boolean

Adds a HopObject to a collection. If the HopObject, on which that function is called on, is persistent, then the added HopObject will also be made persistent. Returns true in case of success.

obj.addAt(Int index, HopObject child) returns Boolean

Adds a HopObject to a collection at a certain position, and shifts the index of all succeeding objects in that collection by one. Index positions start with 0. Just makes sense for HopObjects, that are not mapped on a relational DB, since the sort order of the collection would in such a case be defined by type.properties, resp. by the database itself. Returns true in case of success.

HopObject obj.cache

This property contains a transient HopObject, which can be used for caching purposes. Information stored in that property will get lost as soon as the application is restarted, or that particular HopObject gets kicked out of Helma's ObjectCache, or if the method clearCache is being called.

obj.clearCache()

Removes all information stored in the property obj.cache. Just calling 'obj.cache = null' is not possible, since the property itself can not be set.

obj.contains(HopObject obj) returns Int

Returns the position index of the passed HopObject in that collection (with 0 being the first element in the collection). If the passed HopObject is not contained in that collection −1 will be returned.

obj.count() returns Int

Returns the size of that collection. This method is equivalent to obj.size().

obj.get(Int index) returns HopObject

Fetches a HopObject from a collection through its index position. If the passed index is too large, then 'null' will be returned. If the passed index is negative, a java.lang.ArrayIndexOutOfBoundsException will be thrown.

obj.get(String str) returns HopObject

Fetches a HopObject from a collection through its accessname, which needs to be specified for that collection in the according type.properties, and which needs to be a unique identifier in that collection.

This method just works for persistent HopObjects. If no specific accessname is defined in type.properties, the id will be taken as accessname.

obj.getById(String str) returns HopObject

FIXME

obj.href(String action) returns String

FIXME

obj.invalidate() returns Boolean

FIXME

obj.invalidate(String childId) returns Boolean

FIXME

obj.list() returns HopObject[]

Returns the contained HopObjects in that collection as an array, with the sort order being preserved. Note, that this function should not be called on very large collections, since on the one hand Helma needs to fetch all contained HopObjects, and on the other hand JavaScript is not well optimized for handling large arrays.

obj.list(Int start, Int length) returns HopObject[]

Returns all contained HopObjects, starting from a certain index position until another certain index position, as an array. Both arguments need to be specified.

obj.persist() returns Int

FIXME

obj.prefetchChildren(Int startIndex, Int length)

FIXME

obj.remove() returns Boolean

FIXME

obj.removeChild(HopObject child) returns Boolean

FIXME

obj.renderSkin(SkinObject skin, Object param) returns Boolean

FIXME

obj.renderSkin(String skinname, Object param) returns Boolean

FIXME

obj.renderSkinAsString(SkinObject skin, Object param) returns String

FIXME

obj.renderSkinAsString(String skinname, Object param) returns String

FIXME

obj.set(String key, Object value) returns Boolean

FIXME

obj.size() returns Int

Returns the size of that collection. This method is equivalent to obj.size().

MimePart

see JavaDocs for helma.util.MimePart

new Packages.helma.util.MimePart(String name, byte[] content, String contentType)

FIXME

part.contentLength returns Int

FIXME

part.contentType returns String

FIXME

part.eTag returns String

FIXME

part.getContent() returns byte[]

FIXME

part.getText() returns String

FIXME

part.lastModified returns Date

FIXME

part.name returns String

FIXME

part.writeToFile(String dir) returns String

FIXME

part.writeToFile(String dir, String fname) returns String

FIXME

onXYZ() Event Functions

onCodeUpdate()
: FIXME

onLogout()
: FIXME

onRequest()
: FIXME

onStart()
: FIXME

Request Object 'req'

http://helma.org/docs/reference/request/

see JavaDocs for helma.framework.RequestBean

String req.action
: Returns the name of the requested action (without the suffix '_action). This property is read−only.

Object req.data
: Returns an object containing all passed request parameters as named properties, no matter whether these have been submitted as URL−parameters, as part of a HTTP POST request, or as cookie values.

  If more than one value is submitted for one parameter name, the values are stored inside an array called req.data.paramname_array, that is the _array suffix is added to the parameter name. req.data.paramname itself will just contain a single value.

  Uploaded files are available in req.data as a MimePart object.

  Additionally Helma sets the following HTTP environment variables if they are available:

  authorization
  : Equivalent to the variable 'authorization' sent in the request header.

  http_browser
  : Name and version of the client browser. Equivalent to the variable 'User−Agent' sent in the request header.

  http_host
  : Host name to which that request was sent to. Equivalent to the variable 'Host' sent in the request header.

  http_language
  : Equivalent to the variable 'Accept−Language' sent in the request header.

  http_language
  : Equivalent to the variable 'Accept−Language' sent in the request header.

  http_remotehost
  : IP−Address of the client machine. Equivalent to a getRemoteAddr() call on the servletRequest.

  http_referer
  : URL of the page the user came from. Equivalent to the variable 'Referer' sent in the request header.

  All properties of req.data are read− and write−able.

req.isGet() returns Boolean

Returns true if the current request is a HTTP GET request, false otherwise.

req.isPost() returns Boolean

Returns true if the current request is a HTTP POST request, false otherwise.

req.getMethod() returns String

Returns the HTTP method (in uppercase letters) of the current request, which is usually 'GET' or 'POST'. For non−web−requests this function will return on of the following: 'XMLRPC', 'EXTERNAL' or 'INTERNAL'.

req.getPassword() returns String

Returns the according decrypted password, if the request contains user credentials sent with 'basic authentication scheme' method, typically as a result of a previously returned 401 HTTP response. (LINK: http://en.wikipedia.org/wiki/Basic_authentication_scheme)

String req.path

Returns the path of the current request, relative to the mountpoint of the current application, and without a preceding slash. This property is read−only.

req.getRuntime() returns Int

Returns the amount of time that has elapsed since the start of the processing of the current request, measured in milliseconds.

req.getServletRequest() returns javax.servlet.http.HttpServletRequest

Returns an instance of the Java HttpServletRequest Class corresponding to the current request, which allows full access to the methods of that class.

req.username

Returns the according decrypted username, if the request contains user credentials sent with 'basic authentication scheme' method, typically as a result of a previously returned 401 HTTP response. (LINK: http://en.wikipedia.org/wiki/Basic_authentication_scheme)

Response Object 'res'

http://helma.org/docs/reference/response/

see JavaDocs for helma.framework.ResponseBean

res.abort()

FIXME

res.cache returns Boolean

FIXME

res.charset

FIXME

res.commit()

FIXME

res.contentType returns String

FIXME

res.data

FIXME

res.debug(String)

FIXME

res.dependsOn(String)

FIXME

res.digest()

FIXME

res.encode(String)

FIXME

res.encodeXml(String)
: FIXME

res.error
: FIXME

res.etag
: FIXME

res.format(String)
: FIXME

res.forward(String url)
: FIXME

res.handlers
: FIXME

res.lastModified returns Date
: FIXME

res.message
: FIXME

res.meta
: FIXME

res.pop() returns String
: FIXME

res.push()
: FIXME

res.realm
: FIXME

res.redirect(String)
: FIXME

res.reset()
: FIXME

res.servletResponse returns javax.servlet.http.HttpServletResponse
: FIXME

res.setCookie(String key, String value, Int days, String path, String domain)
: FIXME

res.skinpath
: FIXME

res.status
: FIXME

res.write(String str)
: FIXME

res.writeBinary(ByteArray bytes)
: FIXME

res.writeln(String str)
: FIXME

SessionObejct 'session'

Each web request is associated with a SessionObject representing a 'user session'. Helma recognises requests being made from the same client within the same session through a session cookie named 'HopSession'. If no such cookie is sent with the request, Helma will set that a cookie with a random hash with the next response.

Within the scripting environment 'session' always represent the current session of the user, who initiated the web request. Furthermore it is also possible to fetch active sessions of other clients through the method

app.getSessions(), and to artificially create SessionObjects through app.createSession().

For further details also see the JavaDocs for helma.framework.core.SessionBean. Since that class is a JavaBean all get− and set−methods are also directly available as properties of that object.

String session._id
: Contains the unique identifier of the current session, which is equivalent to the value stored in the HopSession−cookie on the client side. This property is read−only.

String session.cookie
: Contains the unique identifier of the current session, which is equivalent to the value stored in the HopSession−cookie on the client side. This property is read−only.

HopObject session.data
: This property of the SessionObject offers the possibility to store arbitrary data within the current user session. Note, that this can just be used as a temporary storage, since sessions are not stored persistently within Helma, and are generally lost when the application is restarted.

Date session.lastActive
: Contains the timestamp of the last web request, that has been submitted by that client. This property is read−only, but can be set to the current time through session.touch().

Date session.lastModified
: Contains the timestamp of when the associated client started the session, or logged in, or logged out the last time, whichever happened most recently. This property is read− and write−able.

String session.message
: ??? −> HANNES

Date session.onSince
: Contains the timestamp of when the client started the session. This property is read−only.

User session.user
: Contains a reference to the UserObject associated with this session. This property is null if the client has not been logged in yet, or has already been logged out. Checking this for being unequal to null is the usual way to check whether a client is logged in or not. This property is read−only, but can be set through the method session.login(User usr).

session.lastActive() returns Date
: Returns the timestamp of the last web request, that has been submitted by that client.

session.login(User usr)
: Associates the passed User to that session, i.e. logs the client in as that User. The property user of the session object will refer to the User.

session.login(String username, String password) returns Boolean
: DEPRECATED

  Fetches a User via the passed username (which is defined through _name in User/type.properties, and which needs to be unique), checks whether the passed password matches the password of that User (the password property actually needs to be named 'password' in User/type.properties), and in case of a match, actually associates that session with that User. The property user of the session object will then refer to the User.

  If either such a User is not found, or the password does not match the method will return false.

session.logout()
: Logs out the current session, i.e. removes the reference to a User if one exists. Additionally the global function onLogout will be called.

session.onSince() returns Date
: Returns the timestamp of when the client started the session.

session.touch()

This will set the lastActive property to the current timestamp, and can be used to artificially avoid a session timeout.

Skin Object

http://helma.org/docs/reference/skin/

see JavaDocs for helma.framework.core.Skin

sk.allowMacro(String macroname)
FIXME
sk.containsMacro(String macroname) returns Boolean
FIXME
sk.getSource() returns String
FIXME

Tutorials

Hello World

This is the most basic version of a tutorial. It will guide you to writing your own 'Hello World'−application.

If you haven't setup Helma yet, then go ahead and download a recent Helma package from http://helma.org/download/, unzip it anywhere on your disk and start Helma by double−clicking on the ***hop.bat***, resp. ***hop.sh*** residing in the unzipped directory (to which we will refer as `[HelmaHome]` from now on).

Now, open the file `[HelmaHome]/apps.properties` with your favourite text editor, and add the following line:

```
helloworld
```

As soon as you save your changes to disk, Helma will automatically detect these changes, will create a directory named `[HelmaHome]/apps/helloworld` (with several subdirectories) and will activate this application.

Now change to the directory `[HelmaHome]/apps/helloworld/Root` and save the following JavaScript−code as a new text−file named 'functions.js' (again with your preferred editor).

```
function main_action() {
  res.write("Hello World!");
}
```

And that's already it. We just created a new web accessible function named 'main' for the Root object. So, go ahead, open up http://localhost:8080/helloworld/main with your web browser of choice, and you will see the "Hello World" message being displayed to you.

Some explanations:

- The Root object is always there in Helma, so we didn't had to create it explicitly.

- By having our defined function end with '_action' we told Helma, that this particular function should be callable for browsers.
- The single line of code in that function simply appends the string "Hello World" to the response.

Address book

TODO. See http://helma.org/docs/tutorial/ for the moment.

Mail Client

TODO. See the source for the time being.