hawthorn CHAT

Connector development

© 2009 Samuel Marshall

Contents

Introduction......3
- Audience......3
- Systems......3
- Tasks......3
- Languages......4
- Examples......4
- Contributions......4

Adding configuration options......5
- Server addresses......5
- Magic number......5

Serving client-side files......6
- Using original files......6
- Using modified versions......6
- Performance considerations......6

Libraries......7
- Library documentation......7
- Authorisation keys......7

Translating user information......8
- User name......8
- Permissions......8
- Display names......8
- Extra data......8

Displaying recent messages......9
- Options......9
- Raw code......9
- PHP......11

Adding links to chat......12
- Raw code......12
- JSP......12

Creating a re-acquire script......14
- Raw code......14
- JSP......15
- PHP......15

Adding links to statistics......16
- Raw code......16
- JSP......16
- PHP......16

Language translations......17
- Raw code......17
- JSP / PHP......18

Including extra user data......19
- Provide extra data......19
- CSS styles......21

Introduction

This document describes how to write a connector that links a host Web system to the Hawthorn chat server.

- If you haven't read the Hawthorn system manual, read that first. This document doesn't explain the basic concepts or how Hawthorn works.

Audience

You must be a developer capable of writing code within your host system, in whatever server-side language it uses. You may also need some JavaScript skills.

Systems

The host system:

- Provides authentication facilities.

Your connector, as part of the host system:

- Creates links and includes JavaScript code (provided as part of the Hawthorn distribution) that can be used to chat.

The Hawthorn server:

- Responds to connections initiated by the JavaScript code in order to provide the chat service.

Tasks

Writing a connector involves the following tasks:

- Provide configuration options to store the Hawthorn server addresses and magic number.
- Make the Hawthorn JavaScript and related client-side files accessible to users.
- Work out how your user information translates to the Hawthorn system.
- Add Hawthorn information (recent messages in a channel) to pages in the host system.
- Add links to open Hawthorn chat.
- Create a re-acquire script through which users can request a new key if they chat for long enough that their key expires.
- Add links to Hawthorn statistics.
- Optionally provide alternative language translations for Hawthorn text.
- Optionally customise Hawthorn to include extra data and facilities.

Languages

This document explains how to achieve the tasks:

- Using JSP.
- Using PHP.
- By adding the appropriate JavaScript code, regardless of language.

Examples

Please look in the `connectors/` folder of the distribution for examples. You may want to base your code on the existing examples.

Looking at an example alongside reading this guide might help you to understand the process.

Contributions

If you write a new connector for a publicly-available system, please consider releasing the code to the public by contributing it to the Hawthorn distribution. See `contributions.txt` for more information on contributing.

Adding configuration options

Your connector needs to store configuration information within the host system:

- Server addresses.
- The magic number.

The connector might have additional configuration options that control how the connector works, but those are not covered here.

Server addresses

Hawthorn servers have standard HTTP addresses. For example, a Hawthorn server might be:

```
http://123.123.123.123:13370/
```

- The URL can be a numeric IP like the example, or a named address.
- The URL does not include `hawthorn/`, even though most requests Hawthorn serves do start with that prefix. The prefix is added by JavaScript code.

A Hawthorn system might have one server address or several. These linked servers form a load-balanced system; you need to provide all the addresses to JavaScript so that it can switch between them automatically if a server fails. Consequently, your configuration settings need to allow for multiple servers.

Independent systems

For systems with high usage, you might want to run several independent Hawthorn systems: several servers, or pairs of servers, that are not linked to each. Each channel would be assigned to a particular system. This allows for infinite scalability.

The supplied connectors do not support this situation. If you wanted to support it, you would need to allow configuration to include multiple sets of servers.

Magic number

The magic number is normally a 32-digit hexadecimal number. An example might be:

```
23d70acbe28943b3548e500e297afb16
```

Your configuration options need to allow users to set this number, but you have to take some care:

- This number **must be kept totally secure**.

In particular, you need to ensure that this number is never sent to user Web browsers, and that only trusted system administrators have access to see it. You might also need to consider where the data is stored, and make sure access permissions to the relevant filesystem or database area are suitably restricted.

Serving client-side files

Users of your system need to be able to access the files that make up Hawthorn's client-side implementation. You need to put these files somewhere so that they can be served to user browsers.

The files are:

- `hawthorn.js` – full JavaScript implementation.
- `hawthorn.css` – layout information for the popup window.
- `popup.html` – default implementation of the popup window.

You can find these files inside `connectors/htmlexample` in the distribution.

Using original files

To begin with, it is probably easiest to use these files directly. Place all three files in the same folder that is available to the Web. Ensure that all logged-in users can access these files. (It doesn't do any harm if other people can too.)

Using modified versions

We don't recommend modifying the `.js` or `.css` files[1], although this is possible. However, you will need to modify `popup.html` (perhaps in the process making it into a dynamic file such as `popup.jsp` or `popup.php`) in order to achieve the following:

- Customised display during chat, such as user pictures.
- Non-English language support.

Make sure that your equivalent of `popup.html` still references the correct location for the other two files.

Performance considerations

The JavaScript file may be included on many pages of your host system, and is unlikely to change frequently. To improve performance for your server and for users, configure the server so that it sends an Expires header (for example 1 week in the future) when delivering the file. Reduce this setting when an upgrade is imminent, or change the name of the new file.

You may also wish to minimise the JavaScript file using available third-party utilities.

The CSS and `popup.html` are only requested when a user actually starts a Hawthorn chat, so are less critical.

1 Modifying these files makes it harder to update them if a new Hawthorn version is released.

Libraries

Libraries for JSP and PHP users are provided. See the `readme.txt` file for more details about installing each one.

These libraries make it easier to integrate Hawthorn, but you don't have to use them if you don't want to. This guide also covers 'raw' implementation where you do all the work yourself.

Library documentation

JSP

The JSP library is fully documented in the tag library descriptor XML file, which may be available automatically in certain tools. Otherwise, you can look at the file directly; it's included in the distribution as `src/com/leafdigital/hawthorn/jsp/taglib.tld`.

This file lists all tags and all parameters. Refer to it to obtain detailed information that isn't covered in this guide.

If you need to see the source for the tags, the Java source is in the same folder.

PHP

The PHP library is documented with PHPdoc comments in `hawthorn.php`.

The documentation covers all functions and all parameters. Refer to it to obtain detailed information that isn't covered in this guide.

The full source of the PHP library is also in this file.

Authorisation keys

If you use a library, authorisation keys are calculated for you. However, if you need to calculate your own authentication key, here's how:

1. Build up a string consisting of the following: channel name, user name, extra data, permissions, key time, and the server magic number. Each element is separated by a line feed `\n`. (There's no line feed at the end.)
2. Use a standard SHA-1 implementation to generate an SHA-1 hash of that string.
3. Convert the hash into a 32-digit hexadecimal number. The letters `a-f` must be in lower case, and the hash must always have 32 digits (in other words, if it starts with one or more 0 characters, make sure these are included).

If in doubt, refer to the existing code that generates these keys. This is in the Java class `com.leafdigital.hawthorn.util.Auth`, or in the PHP library `hawthorn.php`.

Translating user information

Your host system holds information about users. Some of this information needs to be passed to Hawthorn. You need to make decisions about how to do so.

User name

Hawthorn needs a unique user name which identifies every user. User names are limited to letters, numbers, and _; you can of course use _ to escape other characters if required.

Decide how your system's user names will translate to Hawthorn.

- User names are kept secret; unless users have `m` permission, they only ever get to see hashed versions of user names.
- If you have multiple possible unique identifiers, pick one. Bear in mind that this identifier will be used in Hawthorn logs.
- In the PHP library, there are functions that automatically escape arbitrary characters into Hawthorn user names.

Guest access

Some systems allow access to guest users without a login. Hawthorn is not really suitable for providing chat facilities to these users.

If you need to allow these users to chat, you might like to try turning their IP address into a user name, or generating a random user name and storing it in a cookie. But it will be difficult to restrict access if any such user causes problems.

Permissions

You will need to decide Hawthorn permissions for each user on each channel. Normal users need `rw` permissions, while those who can ban other users have `rwm`. The system will also need to decide who should and shouldn't be allowed to chat on a channel at all.

Decide how your system's permission/roles system corresponds to Hawthorn's.

Display names

Each user has a display name which is shown to other users. The only character restrictions are that this must not contain control characters or the " symbol.

Extra data

You can add optional data for each user; this might for example be the URL of a user picture. When initially implementing a connector, just leave this blank.

Displaying recent messages

In some existing pages of the host system, your connector will probably want to display information about a chat channel so that users can decide whether or not to join it.

Hawthorn provides a way to do this via JavaScript. You need to ensure that the Hawthorn JavaScript is included on the page, and then make an appropriate call to have it filled.

Options

Options for display include:

- How many recent messages to display (maximum). Can be 0.
- Maximum age of displayed messages.
- How many user display names to display (maximum). Can be 0.

There aren't any options for styling. Hawthorn uses specific classes; add CSS within the host system that displays those how you like.

Raw code

Your page needs to:

1. Include the Hawthorn JavaScript (at start or end of page).
2. Call the `hawthorn.init()` method to set up the server addresses from your configuration.
3. Add one or more `<div>` tags, each with an `id` attribute, at the point(s) where you want to display messages.
4. Call `hawthorn.handleRecent()` once for each `<div>`, with a long list of parameters that define the Hawthorn channel and the current user as well as the `id` of the `<div>`.

This will be achieved automatically by the provided PHP and JSP libraries (see below), but you can also do it manually in any other environment.

If working without the libraries, please see the documentation and code in `hawthorn.js` for the two methods mentioned. It may also be helpful to refer to the library source code to see how they do it, or to look at the resulting HTML output from the example connectors in your browser's View Source feature.

JSP

Your page needs to include the Hawthorn tag library, initialise it, and then use the `<hawthorn:recent>` tag.

Include the tag library

Make sure you've followed the instructions from `lib/jsp/readme.txt` to install the tag library in your application. Then include it within your page using this declaration:

```
<%@ taglib uri="http://www.leafdigital.com/tld/hawthorn"
 prefix="hawthorn" %>
```

Initialise the Hawthorn tags

A single tag, used once on the page, sets up shared Hawthorn parameters. You should put this near the start of the page before calling other Hawthorn tags.

```
<hawthorn:init magicNumber="" user="" displayName="" extra=""
permissions="" jsUrl="" popupUrl="" reAcquireUrl="">
 <server>http://123.123.123.123:13370/</server>
</hawthorn:init>
```

- All the user parameters need to be filled in with the relevant details for the current logged-in user.
- The magic number and the servers should come from your system configuration. (You can list one or more `<server>` tags.)
- The three URL parameters should point to `hawthorn.js`, `popup.html`, and a re-acquire script. (The re-acquire URL is only used when opening a chat window and will be covered later. For now you can put in anything there.)

Display recent messages

The `<hawthorn:recent>` tag inserts recent messages at the current point of the page. You should put this wherever you want the chat information to be inserted.

```
<hawthorn:recent channel=""/>
```

- The channel name should be set to the channel you want to display. (This will also give the defined user a key granting access to this channel for one hour.)
- The tag has various parameters (listed in the tag library descriptor).
- By setting these parameters you can control the maximum number of messages or names shown, and four pieces of text: the headings for message and name lists, the 'loading chat information, please wait' message, and the message that appears if JavaScript is not available.

If there are no recent messages or names present in the chat channel, this tag won't display anything visible to users.

PHP

Your page needs to include the Hawthorn library, instantiate a `hawthorn` object, and then print out the results of calling its `recent()` method.

Include the library

The `hawthorn.php` file should be available to your script. If it's in the same folder, you could include it as follows:

```
require_once(dirname(__FILE__) . '/hawthorn.php');
```

Instantiate a Hawthorn object

The PHP library uses a class called `hawthorn` to handle shared user parameters and other details. You need to create an instance of this before you can do anything else with the library.

```
$hawthorn = new hawthorn($magicnumber, $servers, $user, $displayName,
 $extra, $permissions, $jsUrl, $popupUrl, $reacquireUrl);
```

- All the user parameters need to be filled in with the relevant details for the current logged-in user.
- The magic number and the servers should come from your system configuration. `$servers` must be an array of strings containing the server URLs.
- The three URL parameters should point to `hawthorn.js`, `popup.html`, and a re-acquire script. (The re-acquire URL is only used when opening a chat window and will be covered later. For now you can put in anything there.)

Display recent messages

The `recent()` function returns HTML code (also containing JavaScript) which you should print out at the point where you want the messages and names to be displayed.

```
print $hawthorn->recent($channel);
```

- Set the channel name to whatever channel you want to display. (This will also give the defined user a key granting access to this channel for one hour.)
- The function has various parameters (listed in the library PHPdoc).
- By setting these parameters you can control the maximum number of messages or names shown, and four pieces of text: the headings for message and name lists, the 'loading chat information, please wait' message, and the message that appears if JavaScript is not available.

If there are no recent messages or names present in the chat channel, this function won't display anything visible to users.

Adding links to chat

In some pages of the host system, your connector needs to add links to Hawthorn chat channels. Users can click on these links to launch the Hawthorn popup window and chat in the channel.

Raw code

Your page needs to:

1. Include the Hawthorn JavaScript (at start or end of page).
2. Call the `hawthorn.init()` method to set up the server addresses from your configuration.
3. Add a link which the user will click on to chat.
4. Make this link call the JavaScript method `hawthorn.openPopup()` when clicked.

This will be achieved automatically by the provided PHP and JSP libraries (see below), but you can also do it manually in any other environment.

If working without the libraries, please see the documentation and code in `hawthorn.js` for the two methods mentioned. It may also be helpful to refer to the library source code to see how they do it, or to look at the resulting HTML output from the example connectors in your browser’s View Source feature.

JSP

Your page must include the tag library and call the `<hawthorn:init>` tag as described on page 10. It should then use the `<hawthorn:linkToChat>` tag wherever you want a link to appear.

```
<hawthorn:linkToChat channel="" title="">
Chat now!
</hawthorn:linkToChat>
```

- The `channel` parameter needs to contain the Hawthorn channel name. The `title` parameter contains the visible channel name that will display to users in the title of the popup window.
- You can use the optional parameters `icon` and `iconAlt` if you want to place an icon after the link which indicates that the link opens in a new window.

PHP

Your page must include the library and instantiate the hawthorn object as described on page 11. It can then print the results of the `linkToChat()` function:

```
print $hawthorn->linkToChat($channel, $title, 'Chat now!');
```

- The `$channel` parameter needs to contain the Hawthorn channel name. The `$title` parameter contains the visible channel name that will display to users in the title of the popup window.
- The third parameter contains the text (optionally HTML tags) that will be placed within the link tag.
- You can use the optional parameters `$icon` and `$iconAlt` if you want to place an icon after the link which indicates that the link opens in a new window.

Creating a re-acquire script

Hawthorn keys expire after the time set in the `keyTime` value. By default this is one hour from the time the key was granted. If a user chats for longer than this time, they need to acquire a new authorisation key by contacting a *re-acquire* script. The re-acquire script is provided by the connector. It checks that the user still has authorisation from the host system, and provides a new key.

The standard Hawthorn popup attempts to re-acquire a key 5 minutes before the time expires.

Because the re-acquire script is called directly from JavaScript, it cannot present a user interface. Consequently it's important that the user's session is maintained. If host application sessions expire after 30 minutes, the key expiry time should also be set to 30 minutes. After 25 minutes, the re-acquire script will be called, ensuring that the user's session is maintained.

Raw code

If you write the re-acquire script from scratch, it must:

1. Check that the user is authenticated in the host system.
2. Generate a new key for the user, set to expire after the desired key expiry time. The key should be generated for the current user and for the Hawthorn `channel` passed as a parameter to the re-acquire script.
3. Return JavaScript code that confirms the new key, passing the id parameter that was passed to the script.

When calling `hawthorn.openPopup()` to create the popup window, you pass in the URL of the re-acquire script.

Re-acquire parameters

The following details are passed to the re-acquire script as GET parameters:

- `id` – an arbitrary ID which should be used in the response JavaScript.
- `channel` – the Hawthorn channel name.
- `user`, `extra`, `displayname`, `permissions` – other user details. These may not be necessary as the connector might determine them based on session information. If used, they should be checked to ensure that the user has access to that identity.

Response

The response should be sent with MIME content type `text/javascript; charset=UTF-8`. Its text should be one of the following:

- `hawthorn.reAcquireComplete('`*[ID parameter]*`','`*[New key]*`','`*[Key time]*`');`
- `hawthorn.reAcquireDeny('`*[ID parameter]*`','`*[Error text]*`');`

JSP

When writing a re-acquire script, ensure that the page is sent as JavaScript:

```
<%@ page contentType="text/javascript; charset=UTF-8" %>
```

Check via session information that the user has permission to access the requested channel. Determine or verify their user name, display name, and permissions. If this all matches, call `<hawthorn:init>` as before. Then use the following tag:

```
<hawthorn:reAcquireAllow id="" channel=""/>
```

The channel and id values should be those that were given as parameters to the script.

If the user cannot be authenticated or does not match, instead deny the request:

```
<hawthorn:reAcquireDeny id="" error=""/>
```

PHP

The Hawthorn library does most of the work for you. Instantiate a Hawthorn object as usual, then check via session information that the user has permission to access the requested channel. Determine or verify their user name, display name, and permissions.

If everything matches, make the following call:

```
$hawthorn->reAcquireAllow($id, $channel);
```

- Unlike other Hawthorn methods, this does not return a result which you then print. The result is output directly.
- This function automatically sets the correct MIME type header.

If there is an error, deny the request:

```
$hawthorn->reAcquireDeny($id, $error);
```

Adding links to statistics

You should provide a way for system administrators to access the statistics pages for the Hawthorn server(s). Clicking one of these links will take the user directly to the statistics page hosted on a Hawthorn server.

Raw code

Your page needs to:

1. Loop around all servers.
2. For each server, display a link to the statistics page.
3. This link should point to the following URL, but with all the parameters filled in:

[*server URL*]`hawthorn/html/statistics?channel=!system&user= &displayname=&extra=&permissions=&keytime=&key=`

JSP

Your page must include the tag library and call the `<hawthorn:init>` tag as described on page 10. It should then use the `<hawthorn:linkToStatistics>` tag wherever you want the statistics links to appear.

```
<hawthorn:linkToStatistics/>
```

PHP

Your page must include the library and instantiate the `hawthorn` object as described on page 11. It can then print the results of the `linkToStatistics()` function:

```
print $hawthorn->linkToStatistics();
```

Language translations

By default, Hawthorn is in English. If you want to add support for other languages, you must do so in the connector. This allows Hawthorn to be presented in different languages depending on user, channel, or global settings that the host system holds.

Raw code

Recent messages

When displaying recent messages, two language strings are used. Both should be added as properties to the `details` object that you pass to `handleRecent()`.

String	English value and usage
`details.recentText`	'Recent messages' Heading for the recent messages list.
`details.namesText`	'People in chat' Heading for the list of currently-present names.

Popup

The `popup.html` or equivalent file constructs a `HawthornPopup` object, referred to here as popup. Between constructing the object and calling its `init()` method, you can set language strings.

String	English value and usage
`popup.strJoined`	' joined the chat' Appears after a user's name when they join chat.
`popup.strLeft`	' left the chat' Appears after a user's name when they leave chat.
`popup.strBanned`	' banned user: ' When one user bans another, the name of the user banning appears before this text, and the name of the user being banned appears after it.
`popup.strError`	'A system error occurred' When an error occurs, this text appears above the error message. The error message, which comes from the server, is always in English and cannot currently be translated.

`popup.strConfirmBan`	‘Are you sure you want to ban $1?\n\nBanning means they will not be able to chat here, or watch this chat, for the next 4 hours.’ When a moderator clicks the Ban button, this confirmation prompt checks that they really want to go ahead with the ban. $1 will be replaced by the target user’s name.
`popup.strCloseChat`	‘Close chat’ Label of the close button on the popup window.
`popup.strIntro`	‘To chat, type messages in the textbox and press Return to send.’ These instructions are displayed at the top of the popup window.
`popup.strBanUser`	‘Ban user’ Label of the ban button on the popup window.

JSP / PHP

Recent messages

In JSP and PHP the two strings above are provided as optional parameters to the `<recent>` tag or `recent()` function. Two additional parameters are available:

String	**English value and usage**
`loadingText`	‘(Loading chat information, please wait...)’ Initial content of the `<div>` tag that will be replaced with information about the channel.
`noScriptText`	‘(Chat features are not available because JavaScript is disabled.)’ Text of a `<noscript>` tag which is added alongside the `<div>`.

Popup

The procedure for configuring text in the popup window is the same in JSP or PHP as it is for raw implementation. The library doesn’t provide any way to make this easier. (It’s pretty easy already!)

Including extra user data

In addition to the display name, Hawthorn lets you include extra user data which will be sent along with the user's name. This extra data can be processed inside the JavaScript popup window to provide additional facilities.

The most obvious use of this data is to provide a user picture / avatar that displays alongside the user's messages.

Extra data must be consistent within a chat session. If it changes, a user's key may become invalid. They will need to close the session and open another one.

In order to provide this customisation within your connector, you need to follow these steps:

1. Provide the extra data when making Hawthorn requests.
2. Add JavaScript handling in the popup that does something useful with the extra data.
3. Add extra CSS styles in the popup if required to display the new facilities.

Provide extra data

If extra data is used, it must be included with every Hawthorn request. Along with the user name, display name, permissions, and key time, it is used in calculating the permission cache.

Raw code

- When generating authorisation keys, your server-side code must include the extra data (see page 7).
- The Hawthorn JavaScript methods all contain an `extra` parameter. Your server-side code must include the right information in this parameter when generating the code that calls these methods.

JSP / PHP

The built-in libraries support extra data. In the `<init>` tag or `init()` function, make sure to pass in the correct extra data.

Use extra data in popup

There are several places to hook into the HawthornPopup class – see hawthorn.js for documentation. The easiest is to redefine the addMessageExtra() function, which by default does nothing. You can redefine this function inside your popup.html or equivalent code, in between creating the HawthornPopup object and calling its init() method.

The addMessageExtra() function is called after a new <div> has been added for a new message (join, leave, say, or ban). It is given parameters indicating the type of the message, the extra data, and the <div> element. You can use these to add extra HTML elements to the DOM tree.

When working out what you want to do to the DOM, it might help to use a tool like the FireBug extension for Firefox so that you can see what Hawthorn already puts there.

Example

This example assumes that the extra data consists of an image URL and nothing else. It creates a <div> with class userpic, places this at the beginning of the new entry, and puts inside it an image with source URL from the extra data. Because the image is purely decorative and the user's real name will appear immediately afterward, it sets the alt text to an empty string.

At the end of the entry it adds another <div> with a style that clears floats. This works with the CSS (see below) to ensure that entries don't overlap when the text is shorter than the image.

If the current message is a LEAVE, it makes the image partially transparent (may not do anything on Internet Explorer) to signify that the user is departing.

```
popup.addMessageExtra = function(type, extra, entry)
{
  var div = document.createElement('div');
  div.className = 'userpic';
  var img = document.createElement('img');
  div.appendChild(img);
  img.src = extra;
  img.alt = '';
  if(type == 'LEAVE')
  {
    img.style.opacity='0.3';
  }
  entry.insertBefore(div, entry.firstChild);
  div = document.createElement('div');
  div.style.clear = 'left';
  entry.appendChild(div);
};
```

CSS styles

You can add extra CSS styles to the `popup.html` or equivalent. Adding styles here, rather than modifying `hawthorn.css`, means you can safely update `hawthorn.css` if there is a new version supplied with a later Hawthorn system update.

Example

Here are the extra styles required to go with the JavaScript above.

```
<style type="text/css">
.entry .message
{
  margin-left:65px;
  text-indent:-25px;
}
.entry .userpic
{
  float:left;
}
</style>
```