rDeveloping With the Horde Application Framework

Chuck Hagenbuch

The Horde Application Framework is a set of PHP classes, scripts, standards, and conventions that provide a foundation for building web applications in PHP.

Application frameworks can provide a wide range of services. Horde provides a lot of libraries and blocks from which you can build your code; it doesn't specify how you need to store, authenticate, or keep track of your users. The various systems are very flexible and don't make assumptions about each other; you can authenticate users against your IMAP server, store their preferences in an LDAP server, and store permissions in a SQL database with no problems. Or you can store everything in a unified SQL database – the backends are abstracted; use the APIs, and everything works.

What the Framework provides

The generalized Horde libraries have grown out of more specific needs. Therefore, everything provided in the framework is a proven, tested solution to a problem and is in use in at least one real-world application.

These pieces include: authentication, browser detection, data caching, help systems, data import and export, multiple user identities, language detection and independence, MIME, permissions, preferences, application configuration and inter-application method calls, encryption, data encapsulation, text handling, and token generation.

Instead of attempting to overview everything, this paper will go into a bit of the software engineering used in the framework. It then goes in depth into the Registry system, and touches on a few other areas of wide interest.

Design Patterns

The Framework makes use of a number of design patterns in the classes that are provided. Design patterns are ways of describing a problem and a proven solution to that problem, along with any consequences of that solution and a name so that designers can share their vocabulary. Design patterns prevent designers from having to solve the same problem over and over again and pass knowledge on from experienced designers.

One of the simplest and most common patterns is the *Factory* pattern. Horde uses the Factory pattern in all of the systems that have a backend of some sort – Preferences, Authentication, etc. The Factory pattern specifies that you defer instantiation of a class to a method of the class, and so gain abstraction in what kind of subclass you get back. Here is a concrete example. The following line of code specifically requests that we want an FTP Authentication object:

```
$auth = &Auth::factory(‘ftp’);
```

If we wanted an SQL-based Authentication object instead, all we would need to do would be to change the argument to the factory method:

```
$auth = &Auth::factory(‘sql’);
```

So if we moved the type argument to a variable in a configuration file, we wouldn’t even have to worry in the code about what kind of Authentication object we are using:

```
$conf[‘auth’][‘driver’] = ‘imap’;
…
$auth = &Auth::factory($conf[‘auth’][‘driver’]);
```

PHP Note: The ‘&’ character before the Auth::factory() method name specifies that we want a reference back from the method. Otherwise, PHP would return the object by value, and in doing so would make an extra copy of it. Using ‘&’ saves PHP from doing that extra work.

Another extremely simple but useful pattern is the *Singleton* pattern. If an object should only be instantiated once, then that object is a great candidate to be a Singleton. In languages with private methods, the constructor of the Singleton is made private, and a static method is provided to return the object and instantiate it if necessary. In PHP, we pretend, but singletons are still useful. Here is an example:

```
$registry = &Registry::singleton();
```

PHP Note: The use of '&' to obtain a reference is even more important with Singletons. If you leave it off, then every time you call the Singleton method you will be getting back a copy of the object – not the same object. This can cause problems that are extremely difficult to track down. Here is an example of how to implement a Singleton in PHP:

```
// The function must be declared with a & before the
// function name to ensure that it returns a reference.
function &singleton()
{
    // Static variables persist beyond method calls. If
    // set once, it will retain that value in
    // subsequent calls to the method.
    static $registry;

    // Only instantiate the object if this is the first
    // time the method has been called.
    if (!isset($registry)) {
        $registry = new Registry();
    }

    // The function has been declared to return a
    // reference; a & would be meaningless here.
    return $registry;
}
```

The Horde Registry

The Registry is perhaps the most powerful and complicated tool provided by the Horde Framework. No doubt some view it as unfortunately named, but it is a “registry” – it is where applications using the framework “register” their names, path information, and APIs.

Configuration is the simpler half of the Registry’s functionality. In the registry configuration file, each Horde application specifies where it lives in the webroot, where it lives on the filesystem, its icon, its name, whether or not users who are not logged in to the framework should be allowed to access it, and whether or not it should show up in the Horde menubar. Optionally, graphics and templates locations can be specified if they differ from the defaults, as can the domain and path that should be used for cookies. Here is an example configuration section for Nag, a task manager:

```
$this->applications['nag'] = array(
    'fileroot'      => dirname(__FILE__) . '/../nag',
    'webroot'       => '/horde/nag',
     // This defaults to ‘webroot’/graphics:
    'graphics'      => '/special/graphics/nag',
     // This defaults to ‘fileroot’/templates:
    'templates'     => '/special/templates/nag',
    'icon'          => '/horde/nag/graphics/nag.gif',
    'name'          => _("Tasks"),
    'allow_guests' => true,
    'show'          => true
);
```

PHP Note: The `__FILE__` constant contains the full filesystem path of the source file in which it is used. Unlike the server variable $PHP_SELF, which always refers to the script being executed, the __FILE__ constant reflects included files, so you can tell which include file you are inside. The array above lives in “`horde/config/registry.php`”, so `dirname(__FILE__)` will chop “`registry.php`” off of that path, giving us horde’s configuration directory. Adding “`/../nag/`” to that gives us “`horde/nag`”, a reasonable default path.

Putting all path information into the registry means that there is only one place to look to find out where an application’s support files, graphics, and other pieces live. This is especially important for inter-application procedure calls and application management, which are the other purposes of the registry.

The application management built in to the Registry follows a stack metaphor. When you start using an application, you “push” it onto the application stack, and when you are done, you “pop” it off. These functions take care of a number of things, including loading configuration files, translations, and preferences.

In order to provide inter-application calls, the Registry class provides two basic methods: call() and link(). These perform a lot of behind-the-scenes magic, but they stick to some well-defined interfaces and make a lot of functionality *very* easy to implement. Call() is a way of getting a result back from a different Horde application than the one you are currently in – for instance, getting a task list to display alongside events in a calendar program, or searching for an email address in a contact manager from within an email program. Link() generates links to pieces of other applications without having to hardcode that application’s path information.

The purpose of all of this is abstraction and ease of development. It may seem like there is an awful lot of overhead involved just to search for email addresses, but the gains are worth it: applications only need to implement one of the interfaces (“contacts/search”, for example) and then they can be dropped in, completely on the fly, merely by editing a configuration file.

Here is a detailed explanation of what happens when you use Registry::call():

1. All Horde applications should get a reference to the global $registry variable as one of the first things they do. This is done by these two lines of code:
   ```
   require_once HORDE_BASE . ‘/lib/Registry.php’;
   $registry = &Registry::singleton();
   ```

2. To call a method in another application, you may first want to see if an application provides that method. You can check if the Registry has an implementation available of a specific method by using the hasMethod() call:
   ```
   if ($registry->hasMethod(‘tasks/list’)) { … }
   ```
3. Once you’ve determined that the method is available, all you have to do is call it:
   ```
   $tasks = $registry->call(‘tasks/list’);
   ```

That’s it. The Registry takes care of loading whatever application is registered as providing the ‘tasks/list’ implementation (using Registry::pushApp()), calls the function in that application which lists tasks, returns the results, and pops the application back off the application stack, returning you to the context you were in before the call (using Registry::popApp()).

Registering a method handler takes just a few steps:

1. Define the method implementation in the registry configuration file:
   ```
   $this->services['nag']['tasks']['list'] = array(
       'file'     => '%application%/lib/api.php',
       'function' => 'nagListTasks',
       'args'     => array('sortby', 'sortdir'),
       'type'     => 'array'
   );
   ```
   This entry means that: a file named “api.php” in Nag’s library directory contains a function named “nagListTasks”, which takes two arguments – “sortby” and “sortdir” – and returns an array.
2. Register that implementation of the method as the default one:
   ```
   $this->registry['tasks']['list'] = 'nag';
   ```
   This tells the Registry that Nag is the application that should be used whenever someone makes a “tasks/list” call, without asking for a specific application to handle it.

The implementation of the nagListTasks() function is now completely up to you. It must take the two arguments, and return an array as advertised, but other than that it is independent. It is *loosely coupled* to any application using it. The implementation of "tasks/list" may change completely, but any application using the "tasks/list" method will continue to function as long as the arguments and return type are not changed.

Session Management

Horde uses PHP4's built-in session management functions. Various parts of the framework, such as the Auth and Prefs systems, rely on session storage. However, you are free to drop in another session backend using PHP4's session handler options; Horde in no way relies on the file-based session backend.

Horde also provides a number of methods – notably Horde::url() and Horde::applicationUrl() for propagating session IDs via GET strings if necessary. These methods will tack the session id on to the URI passed in if cookies aren't enabled, and leave it off if they are.

User Preferences

The Horde preferences system is a very mature implementation that handles backend driver abstraction and caching. It also allows fine-grained control over individual preferences, allowing the site administrator to lock individual preferences to forced defaults and share preferences between multiple applications.

The available backends are SQL, LDAP, and session –based backend session drivers. The SQL driver is the most widely used and actively developed, so it is the recommended driver. However, there is no reason new backends couldn't be easily written; the API to implement is fairly small.

Because of the integrated caching mechanism, the Horde preferences system is very efficient. If a user goes through an entire session without changing any of his or her preferences, the preferences system is smart enough not to perform any write operations. In addition, all reads after the initial read are performed against the preferences cache, which is stored in the user's session. Note that because the cache is stored in the session, normal session reading and writing overhead must be taken into account.

Horde also provides completely automatic generation of the GUI for changing preferences. This means that adding preferences to an application, complete with the UI, can really be a matter of about 10 minutes to write the preferences configuration file, drop in the UI code, and maybe a little bit more if there are any special cases to handle.

Conclusion

The Horde Application Framework provides a solid collection of code on which to base web applications. Some of the largest gains in using Horde are in pieces such as the Registry, which allows your application to use pieces of other applications seamlessly, and the Preferences system, which can be dropped in to an existing application in no time at all. And the engineering behind these systems ensures that they will scale and adapt gracefully as the framework grows and evolves.