Pines Development

Contents

Preface

How This Book is Organized

This book is divided into three parts, each dealing with a different aspect of Pines development. Each part builds on the information and instruction given in the previous part.

Part I: The Pines Framework

Part I focuses on getting acquainted with the Pines framework. It provides information about Pines, how it works, and what is provided by Pines. I discuss how to get Pines from the development repositories. Pines Core, the system on which the framework is built, is discussed in depth. System services, which provide additional functionality for developers, are explained in detail.

Part II: Application Development

Part III: Distribution

Conventions Used in This Book

Part I

The Pines Framework

Chapter 1

Pines

1.1 What is Pines?

Pines is a MVC based PHP framework, designed to help produce complex web applications rapidly. It is a full featured application framework, which provides features not only unique to Pines as a framework, but never before seen in PHP. Pines is designed with the developer in mind. This is reflected in the history of Pines.

Pines' goal as a framework is to provide the developer with a set of tools, which simplify the development of complex applications and allow administrators to easily and quickly set up complex, secure installations. One of the key aspects considered in Pines development is customization. Pines has a long history of being customized for all sorts of use, and that history has brought Pines some important tools. The SciActive maintained configurator component, com_configure, is a perfect example of Pines' obsession with customization, allowing the administrator to provide different configuration based on users, groups, and even dependency sets.

Pines strives to also provide this level of customization to its developers. Many of the features available in Pines are provided through system services, which can be implemented in many different ways. For example, the entity manager, used to provide data storage and retrieval, can use any number of different backends. Because Pines defines a standard way of interacting with these system services, developers can design their components once, and have it work with any implementation of these system services. Developers can also extend components to add features not available by default. For example, the official user manager adds access control features to the entity manager, which by default, has no considerations for users or groups. Amazingly, this extension to the entity manager is accomplished without altering any of its code. The user manager uses method hooking to alter the functionality of the entity manager. This level of flexibility allows developers to create complex, scalable applications with ease.

1.2 The History of Pines

Pines was started in the Fall of 2008 by myself, Hunter Perrin. While taking my first PHP class, along with a French I class, I developed a small PHP application to allow the collaborative editing of a play script. The application was used for my French class, then stowed away. Several months after that, I started a new job, working as a web designer. There, they requested me to find an application capable of sending newsletters to their clients. I dug up the old application, and adapted it to send emails. This incarnation was called Dandelion, and released under the GNU Affero GPL.

After a year there, I took a new position. They had several new ideas for web applications, which I slowly started building using Dandelion. When the project started to really resemble a framework, not just an application, we renamed it to Pines. Problem after problem would arise, and solution after solution would be built. All atop the Pines framework. Every feature in Pines has come from the real need of a business environment. Pines has seen many redesigns, and it barely resembles the early application.

Now Pines is a solid, well tested, and well documented framework, suitable for building high demand enterprise applications.

1.3 Implications of Pines' License, The GNU AGPL

Pines is, and always will be, free and open source software. This means that anyone can freely redistribute Pines, obtain the source code for study or modification, and create derived works (under the same license). It is released under the GNU AGPL, which places certain requirements on use of Pines software.

If you modify any of Pines' source code (including the SciActive maintained components), and use this modified code to run a server, which users can access, your modified code must be freely available to these users. This does not mean that you must release any custom components you create, just any modifications to Pines' core or official components.

You are free to make modifications and not release them to the public, but any users which access the system on which the modified code runs must have access to the code.

If you would like to change Pines' behavior without modifying its code, you can use the hook system to accomplish almost any needed change. Because a component can be licensed under any license, you do not necessarily have to distribute this source code.

1.4 Features That Make Pines Unique

Pines has many features which are either vastly superior to their counterparts in other frameworks, or have never been seen before in PHP. This is a list of just a few features that make Pines stand out in the ever expanding field of PHP frameworks.

- The Package Manager

  The package manager resembles the package managers common in Linux. It handles dependencies, retrieves updates, checks signed code, and much more. This isn't necessarily uncommon in a PHP framework, but Pines' package manager is 100% PHP code. That includes everything. The package downloading, dependency checker, code signing, even the package format is 100% PHP.

- The Slim Archive Format

  The Slim archive format is the heart of the package manager. Using just PHP, Slim is able to rival other archive formats. Slim archives can even be built to self extract.

- The Hook System

  Similar hook systems have been attempted by other frameworks, but Pines' hook system is unparalleled. It allows the developer to extend or even change any method in a component. This level of flexibility allows developers to collaborate and build like never before.

- The Entity Manager

  The entity manager is an object-relational mapper, which provides developers with a very simple and very flexible way of saving, retrieving, and manipulating data. With a simple, yet powerful querying system, you can make a complex, portable application without ever writing a single SQL query. Complex data relations are easy to handle and easy to query.

Pines is full of features which make coding applications fast and easy, yet it's flexible enough to fit almost any application. Let's get started!

Chapter 2

Getting Started

2.1 Setting Up an Environment

The first and most important thing you need to develop for Pines is a web server. Pines is designed for Apache on Linux, but it will work fine on most setups. If you are planning on using the entity manager, which is the easiest way to store and manipulate data in Pines, you will also most likely need a database server. MySQL works well with Pines, and is officially supported by SciActive. I recommend using an IDE, such as NetBeans or Eclipse. Both have great PHP support. Once you have Pines set up and you are ready to make your own components/templates, you can use symlinks to test them with different Pines setups.

2.2 Getting Pines

There are several sites which host Pines, and Pines related software.

- http://sourceforge.net/projects/pines/

  The Pines project page on SourceForge. This is where almost everything about Pines can be found.

- http://pines.sourceforge.net/

  Pines' main website.

- http://pines.sourceforge.net/pines-docs/

  Pines API Documentation. This can be very helpful to have handy while you are learning Pines development. Before using Pines' advanced features, I recommend reading through the related API documentation to ensure you have a firm understanding of how Pines' features work and how to use them.

- http://www.sciactive.com/

  Pines' main developer's website.

- http://www.pinesplaza.com/

  The official Pines Repository. Pines components and templates are hosted and distributed here. You can get a free account and distribute your software through Pines Plaza.

Pines, and its related projects are available in Mercurial repositories from SourceForge.

- http://pines.hg.sourceforge.net:8000/hgroot/pines/pines *or* http://pines.hg.sourceforge.net:8000/hgroot/pines/core

  Pines Core repository. Contains the core system files.

- http://pines.hg.sourceforge.net:8000/hgroot/pines/components

  Pines Components repository. Contains official components and templates.

- http://pines.hg.sourceforge.net:8000/hgroot/pines/pform
  Pines Form.
- http://pines.hg.sourceforge.net:8000/hgroot/pines/pgrid
  Pines Grid.
- http://pines.hg.sourceforge.net:8000/hgroot/pines/pnotify
  Pines Notify.
- http://pines.hg.sourceforge.net:8000/hgroot/pines/ptags
  Pines Tags.
- http://pines.hg.sourceforge.net:8000/hgroot/pines/tools
  Build tools, documentation, logo sources, and other various tools.
- http://pines.hg.sourceforge.net:8000/hgroot/pines/cash-drawer
  Pines POS Cash Drawer Firefox Extension

To get a local copy of Pines for testing or development, clone the Mercurial repositories.

```
mkdir pines
hg clone http://pines.hg.sourceforge.net:8000/hgroot/pines/pines pines

mkdir components
hg clone http://pines.hg.sourceforge.net:8000/hgroot/pines/components components
```

Then create links to the components/templates under the correct directories.

```
cd pines/components/
ln -s ../../components/com_* ./

cd ../templates/
ln -s ../../components/tpl_* ./
```

You will need to unlink conflicting components, such as extra entity managers. If you plan to use MySQL for example, you will need to delete the links to com_pgentity and com_pgsql.

```
cd pines/components/
unlink com_pgentity
unlink com_pgsql
```

To update your local repository and get the latest code, pull changes and update.

```
cd pines/
hg pull
hg up

cd ../components/
hg pull
hg up
```

I recommend using a Mercurial GUI to make using the repository easier. I use TortoiseHG (http://tortoisehg.org/).

2.3 Coding Style

Note
This coding style is the style used by Pines Core and the SciActive maintained components/templates. If you are submitting a patch, please keep it written in this style. In your own components/templates you are free to use whatever style you like, but you must conform to the naming guidelines.

2.3.1 Indentation

Indentation is done with one tab character. The examples in this book use four space characters, simply because the program used to write it does not allow tabs. Cases in a switch are indented. Arguments of a function call are indented once, with the exception of entity manager queries, which should be indented twice. This is in order to accurately and quickly find and understand entity manager queries in a function. In each selector of an entity manager query, the selector type should be placed on the line with the opening "array(".

Example 2.1: Indentation

```
if ($var) {
    // Something
}

if ($var)
    do_something();

switch ($var) {
    case 'barbecue':
        echo 'what?';
        break;
    case 'something':
    default:
        echo 'that\'s normal';
        break;
}

// Typical function call.
$pines->something(
    $arg1,
    $arg2
);

// Entity manager query.
$pines->entity_manager->get_entities(
        array('class' => entity),
        array('&',
            'tag' => array('some', 'tags')
        ),
        array('!&',
            'data' => array('something', true)
        )
    );
```

2.3.2 Brackets and Parentheses

Opening brackets are placed on the same line. Closing brackets are placed on a new line. If you break parentheses, indent the contained lines. The while in a do while loop is placed on the same line as the closing bracket. In PHP, one line if and else blocks can either use brackets or not. In JavaScript (in order to reduce file size) brackets are not recommended in this case.

Example 2.2: Brackets and Parentheses

```
if ($var) {
    one_thing();
    another_thing();
```

```
}

if ($var)
    one_thing();

// This is fine.
if ($var)
    one_thing();
else
    another_thing();
// Also fine.
if ($var) {
    one_thing();
} else {
    another_thing();
}

if ($var) {
    one_thing();
    another_thing();
} elseif ($othervar) {
    // Something else
} else {
    // Something different
}

function something($arg1, $arg2 = null) {
    // Something
}

if (
        $athing &&
        $anotherthing &&
        (
            $thisthing ||
            $thatthing
        )
    )
    do_something();
```

2.3.3 Naming

Names in Pines always use lowercase, and words are separated by underscores.

Component names start with "com_" and don't contain any underscores after that. Component names can only contain letters, numbers, and the underscore. A component's class shares its name with the component. Other than a few special classes, like `user` and `entity`, that are provided by services, classes are prefixed with their component's name. E.g. `com_example_widget`.

Templates are the same, except "tpl_" is used instead of "com_".

Action names only contain letters, numbers, and underscores. Actions which the user sees (i.e. Not JSON only actions) do not contain underscores.

View names can contain any valid filename character, but try to only use letters, numbers, underscores, and dashes.

Entity tags cannot contain commas. The only tag that begins with "com_" should be the name of the component that uses it.

Components' functions should be prefixed with their component's name, followed by two underscores, then the name. E.g. `com_example__do_something()`. This is only for functions, not for methods.

2.3.4 Commenting

Pines uses both C89 (/*...*/) and C99 (//...) style comments. Comments should have a space after the comment mark. Code which is commented out should not. C89 style is preferred for longer comments.

Example 2.3: Comments

```
// This is a short comment.

/*
 * This is a long comment which is
 * too long to fit on one line, so
 * it's been commented using C89
 * style comments.
 */

// However if it's not too long,
// then this is also acceptable.

// Don't use a leading space when commenting out code.
//$pines->com_example->something();
```

Pines uses phpDocumentor (http://www.phpdoc.org) to generate API documentation. Files, functions, classes, interfaces, properties, methods, constants, and requires/includes that are part of Pines Core or SciActive's components should be commented using phpDoc style comment blocks using the package "Pines". Each component and template is a subpackage of "Pines".

2.4 Building Pines

To build a custom release of Pines, you will need to clone the "tools" repository. There, you will find the `build-tools/buildrelease.php` file. Point your browser to this script to use the release builder. The directory you enter should be in the directory above the "tools" repository clone.

In the build tools, you will also find scripts to package Pines' jQuery plugins and to generate an icon component's CSS.

Chapter 3

Pines Core

3.1 Structure

3.1.1 File Structure

Pines has a strict and logical file structure. The system and individual components have a similar file structure. Some less important files and folders have been left out of the following list for brevity.

- `components` - Contains all currently installed components.
- `media` - Contains uploaded and installed media files.
  - `images` - Contains uploaded images.
  - `logos` - Contains logos, including Pines' default logo.
- `system` - Contains the Pines Core code.
  - `actions` - Contains actions always available in Pines.
  - `classes` - Contains the core classes used by Pines.
  - `includes` - Contains Pines JavaScript files.
  - `init` - Contains the system init scripts which build the Pines environment and control execution of scripts.
  - `views` - Contains views used by Pines Core.
  - `defaults.php` - The default configuration settings for Pines Core.
  - `i01common.php` - An init script which is run with the component init scripts. It sets up several shortcut functions and other useful functions.
  - `info.php` - The Pines Core information file. Contains information such as version number and core abilities.
  - `menu.json` - The main menu file. This sets up the main menu and its submenus.
  - `offline.php` - The file used to build the offline page.
  - `template_error.php` - The file used to inform of an error with the currently selected template.
- `templates` - Contains all currently installed templates.
- `index.php` - The main controller script. This sets up some constants, then immediately executes the system init scripts.

3.1.2 Object Structure

Pines is structured almost entirely around the Pines object, `$pines`. The variable is created in the global scope by the system init script, `i30load_system.php`. To use the variable in any function or method, declare it using the global keyword.

Example 3.1: Using `$pines` in a function or method.

```
function example() {
    global $pines;
    $pines->something();
}
```

Several core services are loaded as variables of `$pines`. See the Core Services section for an explanation of these services.

- `$pines->info`
- `$pines->config`
- `$pines->hook`
- `$pines->depend`
- `$pines->menu`
- `$pines->page`

`$pines` also contains some useful variables.

- `$pines->components`
  An array of the enabled components.
- `$pines->all_components`
  An array of all installed components.
- `$pines->services`
  An array of the provided system services.
- `$pines->request_component`
  The requested component. This is passed as the request variable "option".
- `$pines->request_action`
  The requested action. This is passed as the request variable "action".
- `$pines->component`
  The currently running or most recently called component/option.
- `$pines->action`
  The currently running or most recently called action.

The following variables are not loaded until after the component init scripts are run.

- `$pines->current_template`
  The name of the current template.
- `$pines->template`
  The current template object.

3.2 Pines Object

As well as containing services and variables, the Pines object (`$pines`) also includes several useful methods.

3.2.1 Running an Action

To run an action, use the `action` method. Actions are run automatically when requested in the URL by the client. The "option" request variable determines the component to which the action belongs, and the "action" request variable determines which action to run. The action is run by the system init script `i60action.php`.

Sometimes, actions need to be run manually in code. For example, the default action (`default.php` in a component's `action` directory) can be used to direct the user to a common action by calling `action`.

When an action is run, `action` will look in the component's `action` directory for a file with the same name as the desired action (with .php appended). If the desired component is "system", Pines will instead look in the system's `action` directory. If no action was specified, `action` will look for a `default.php` file. If no component was specified, `action` will use the default component specified in Pines' configuration. Once the correct file is found, `action` will run the file. The `$pines` variable is already included when the file is run, so there is no need to include it with the global keyword.

Note
Since actions are run inside a function, they are not in the global scope. This means actions don't have to clean up variables and need not worry about naming collisions, unlike init scripts, which will be discussed later.

If no action file was found for the given arguments, `action` will return the string "error_404". Otherwise, it will return the value returned by the action file. If `i60action.php` receives "error_404" when it runs the requested action, it will attach a module with the system's "error_404" view into the page's "content" position and send a 404 HTTP status code. This means your action can return the string "error_404" if the requested information is not found and a 404 error will be sent to the client.

Example 3.2: Manually Running an Action

```
// Run a specific action.
$pines->action('com_example', 'widgets/list');

// Run a component's default action.
$pines->action('com_example');

// Run the default component's default action.
$pines->action();
```

3.2.2 Accessing Sessions

Sessions in Pines are handled exclusively through the `session` method. Using this method, you can access an existing session for reading or writing, and close or destroy it. Using this method to open a session for reading allows asynchronous requests to Pines to respond quickly, without blocking.

If you've never experienced blocking in PHP sessions, it occurs when a script has a session open for writing, and another script (or the same one in a different request) tries to open the same session. The second script will pause execution until the first script either closes the session or exits.

The `session` method takes one argument, the access type requested. It can be one of the following:

- read

  Open the session for reading only. This is the default.

- write

  Open the session for reading and writing.

- close

  Close a session that was opened for writing. The session remains open for reading.

- destroy

  Unset and destroy a session.

If you open a session for writing, you should always close it once you don't need write access anymore.

Example 3.3: Reading Session Data

```
// Open the session for reading.
$pines->session('read');

// Also works.
//$pines->session();

// Now the session variable is full of the session data.
echo $_SESSION['messages'];
```

Example 3.4: Writing Session Data

```
// Open the session for writing.
$pines->session('write');
// Now use the session variable normally.
$_SESSION['messages'] = '';

foreach ($messages as $cur_message) {
    if (empty($cur_message)) {
        // If you exit out of the script, remember to close the session.
        $pines->session('close');
        pines_error('Broken message encountered.');
        return;
    }
    $_SESSION['messages'] .= " $cur_message";
}

// Now that we're done, close the session.
$pines->session('close');
pines_notice('Messages saved.');
```

3.2.3 Redirecting the User

The Pines object includes `redirect`, a method to redirect users to a different URL. Using this method ensures that any messages and errors that are queued to be displayed to the user will be displayed when the user reaches the destination URL, assuming that URL is also handled by the same Pines installation. The HTTP status code returned to the client can be changed and defaults to 303 See Other.

Example 3.5: Redirecting the User

```
// Notices and errors will be saved.
pines_notice('You have been redirected here.');
$pines->redirect(pines_url('com_example', 'widgets/list'));

// Redirect to the homepage.
$pines->redirect(pines_url());

// Use a permanent redirection code.
$pines->redirect(pines_url('com_example', 'widgets/list'), 301);
```

3.2.4 Formatting Output

There are several types of content that need to be formatted correctly before being output to the user. The Pines object provides functions to allow easy formatting of this data.

3.2.4.1 Formatting Content

To format content, use `format_content`. By itself, `format_content` does nothing. Its purpose is to provide a way for components to alter content before it is shown to the client. By using this method to format your component's content before outputting it, you allow other components to use the hooking system to provide special alterations to your content. This is meant to provide things like string replacements, inline modules, HTML cleansers, etc. This does not mean you should always run content through this method. Certain types of content, such as page text, blog post text, product descriptions, etc are appropriate for content formatting. However, content such as user comments, forum posts, etc may allow an unprivileged user to use dangerous services if run through the formatter. A good idea may be to use a configuration option to allow certain content to be altered by other components. Generally only user provided content should ever be formatted. Content like your component's forms should most likely never be formatted.

Example 3.6: Formatting Content for Output

```
<div>
    <?php echo $pines->format_content($post->content); ?>
</div>
```

As mentioned, components can hook this function to provide special alterations to content. See the Hook section in Core Services for information about hooks.

Example 3.7: Using a Hook to Alter Content

```
// Replace the word "Pines" with "Barbecue".
function com_example__replace(&$arguments) {
    $arguments[0] = str_replace('Pines', 'Barbecue', $arguments[0]);
}

$pines->hook->add_callback('$pines->format_content', -10, 'com_example__replace');
```

3.2.4.2 Formatting a Date / Time

To format a date or time using a timestamp, use `format_date`. The current user's timezone is automatically loaded by the user manager and used for calculations. You can also pass `format_date` a timezone to use. When using a custom format, any format recognized by the `format` method of the `DateTime` class can be used. `format_date` supports several format types.

Table 3.1: Date Format Types

Type	Format Code	Description
full_sort	`Y-m-d H:i T`	Date and time, big endian and 24 hour format so it is sortable.
full_long	`l, F j, Y g:i A T`	Date and time, long format.
full_med	`j M Y g:i A T`	Date and time, medium format.
full_short	`n/d/Y g:i A T`	Date and time, short format.
date_sort	`Y-m-d`	Only the date, big endian so it is sortable.
date_long	`l, F j, Y`	Only the date, long format.
date_med	`j M Y`	Only the date, medium format.
date_short	`n/d/Y`	Only the date, short format.
time_sort	`H:i T`	Only the time, 24 hour format so it is sortable.
time_long	`g:i:s A T`	Only the time, long format.
time_med	`g:i:s A`	Only the time, medium format.
time_short	`g:i A`	Only the time, short format.
custom	(Contents of `$format`)	Use whatever is passed in `$format`.

Example 3.8: Formatting a Date

```
<div>
    <?php echo $pines->format_date($timestamp, 'custom', 'l jS \of F Y h:i:s A', ' ↩
        America/Los_Angeles'); ?>
</div>
```

3.2.4.3 Formatting a Phone Number

To format a phone number, use `format_phone`. It uses US phone number format. E.g. "(800) 555-1234 x56".

Example 3.9: Formatting a Phone Number

```
<div>
    <?php echo $pines->format_phone('1800555123456'); ?>
</div>
```

3.2.5 Checking an IP Address

There are three methods of checking whether an IP address is on a given network in the Pines object.

3.2.5.1 Using CIDR Notation

You can use `check_ip_cidr` to check an IP address using the CIDR notation of a network.

Example 3.10: Checking an IP Using CIDR Notation

```
$good = $pines->check_ip_cidr('192.168.0.5', '192.168/24'); // Returns true.

$bad = $pines->check_ip_cidr('192.168.1.5', '192.168/24'); // Returns false.
```

3.2.5.2 Using an IP Range

You can use `check_ip_range` to check an IP address using an IP range.

Example 3.11: Checking an IP Using an IP Range

```
$good = $pines->check_ip_range('192.168.0.5', '192.168.0.0', '192.168.0.255'); //  ↩
    Returns true.

$bad = $pines->check_ip_range('192.168.1.5', '192.168.0.0', '192.168.0.255'); // Returns ↩
     false.
```

3.2.5.3 Using a Subnet Mask

You can use `check_ip_subnet` to check an IP address using the subnet mask of a network.

Example 3.12: Checking an IP Using a Subnet Mask

```
$good = $pines->check_ip_subnet('192.168.0.5', '192.168.0.0', '255.255.255.0'); //  ↩
    Returns true.

$bad = $pines->check_ip_subnet('192.168.1.5', '192.168.0.0', '255.255.255.0'); //  ↩
    Returns false.
```

3.3 Core Services

3.3.1 Config (`config` class)

The config service loads configuration for both Pines and any components/templates. The configuration for Pines is always loaded, and components' configuration is loaded the first time it is accessed during each script run. Config will start by loading the `defaults.php` file (if it exists) in the component's directory. The return value from this file is an array which is used to load configuration variables. Each array entry should be an associative array containing the following entries:

Table 3.2: Config Array Entries

Key	Required	Description
name	Yes	The configuration option's name. This is used to access it from code.
cname	Yes	A canonical name. This is displayed to the user when setting the option.
description	Yes	A description of the configuration option.
value	Yes	The default value of the configuration option.
options	No	An array of the possible values of the option.
peruser	No	A boolean determining whether the option can be set on a per user basis.

Tip
Calling a function in a `defaults.php` file will use extra resources when the configuration variable for the component is first accessed on each request, even if the configuration has been set to a non-default value. To speed up your component, avoid calling functions in `defaults.php`.

Config will then load the `config.php` file (if it exists) in the component's directory. This file is generated by the configurator service and stores user specified configuration. The file is structured the same, except that each entry only contains "name" and "value".

The configuration is built using these files, and placed into an object named after the component on the first attempted access of the object. Pines' configuration options are placed in the config object itself.

Example 3.13: Accessing Configuration for Pines and Components

```
// Access Pines' "system_name" configuration option:
echo $pines->config->system_name;

// Access a component's "website_name" configuration option:
echo $pines->config->com_example->website_name;

// Access a template's "tagline" configuration option:
echo $pines->config->tpl_green->tagline;
```

Caution
Configuration options are not cleaned or checked (except when an options array is used). When you are inserting them into HTML, they should be escaped with `htmlspecialchars()`.

3.3.2 Depend (depend class)

The depend service manages and runs dependency checkers. It includes several dependency checkers by default.

Table 3.3: Built In Dependency Checkers

Name	Description
ability	Check that the current user has certain abilities.
action	Check the currently running or originally requested action.
class	Check whether classes exist.
clientip	Check the client's IP address.
component	Check whether components/templates are installed, enabled, and are certain versions.
function	Check whether functions exist.
host	Check the requested server hostname. Uses the hostname provided by the client in the HTTP request.
option	Check the currently running or originally requested component.
php	Check PHP's version.
pines	Check Pine's version.
service	Check the available system services.

To check a dependency, use the `check()` method. It will return true if the value passes the check, or false if it does not.

Example 3.14: Using the Dependency Checker

```
if (!$pines->depend->check('component', 'com_example>=2.0|com_sample')) {
    pines_notice('This feature is only available when you have com_example 2.0 or ↩
        greater, or com_sample.');
```

```
    return;
}
```

Adding a new dependency checker is easy. Add a new entry to the checkers variable, using the name of the checker as the key, and a callback to the checker function as the value. The callback function should return true if the check passes and false if it does not. This should usually be done using an init script. That will ensure the checker is available by the time an action is run.

Example 3.15: Adding a New Checker

```
function com_example__check_something($value) {
    return ($value == 'barbecue');
}

$pines->depend->checkers['something'] = 'com_example__check_something';

// Call the new checker like this:
// $pines->depend->check('something', 'a value'); // Would return false.
```

Caution
Since unprivileged users are often allowed to run custom dependency checkers, it is important to not allow disclosure of sensitive data using a checker. For example, an unprotected file checker could be used to find files on the host machine.

Caution
Be careful that a checker does not recursively call itself, which could result in an infinite loop.

All the built in dependency checkers use `simple_parse()` to understand simple logic. The syntax is simple, including and "&", or "|", not "!", and grouping using parentheses "()". Here's an example of how to use `simple_parse()` in the example checker.

Example 3.16: Using the Simple Parser

```
function com_example__check_something($value) {
    global $pines;
    if (
            strpos($value, '&') !== false ||
            strpos($value, '|') !== false ||
            strpos($value, '!') !== false ||
            strpos($value, '(') !== false ||
            strpos($value, ')') !== false
        )
        return $pines->depend->simple_parse($value, 'com_example__check_something');
    return ($value == 'barbecue');
}

$pines->depend->checkers['something'] = 'com_example__check_something';

// Call the new checker like this:
// $pines->depend->check('something', 'a value|barbecue'); // Would return true.
```

If you'd like to include your checker in your component's class, it is better to consider the resources used by including a callback directly to your component's method. Component classes are loaded as soon as the object is first accessed. Instead of using a callback to the method, you can use a shortcut function to save Pines from loading your component's class during each script run.

Example 3.17: Using a Shortcut Function to a Method

```
// Using an extra function to call the real checker keeps the com_example
// class from being loaded on each script run.
function com_example__check_something($value) {
    global $pines;
    return $pines->com_example->something($value);
}

$pines->depend->checkers['something'] = 'com_example__check_something';
```

Note
Many things use the dependency checker, including menu entries, the package manager, conditional groups, and conditional configuration just to name a few.

3.3.3 Hook (hook class)

The hook service provides method hooking for the entire `$pines` object and most classes. Method hooking allows a developer to intercept function calls and alter arguments, alter return values, and change the actual function being called.

3.3.3.1 How Hook Works

Hook uses a complex technique to override an object and allow all its public methods to be hooked. When `hook_object()` is called and passed an object, it begins by using PHP's reflection API to analyze the object and build a new object. The class used to build the new object is the `hook_override__NAMEHERE_` class. Hook will create a new class based on this class by replacing "_NAMEHERE_" with the name of the class. Each method on the original object is recreated in the new class. The replacement methods, when called will run the callbacks associated with that hook and the original method on the original object. Once the new class is complete, it is loaded, and an object is created using it. The new object stores the original object, and replaces the variable holding that object. The new object now resides in place of the original object.

When a variable is requested from the object, the new object will pass the request directly to the original object. When the object is requested as a string, invoked, or cloned it will also request it from the original object. When the object is cloned, it will hook the clone as well.

Caution
Calls from within a method of the original class to another method of the class will not be intercepted. This includes all calls to private methods. Static methods cannot be hooked either.

Caution
Functions which return or check the class name of the object will use the class name of the new object, and could therefore return unwanted results. For example, a hooked user object would return false with `is_a($user, 'user')`. You can check for the class `hook_override_user` or check `hook_override` just to see if it is hooked.

3.3.3.2 How to Use Hooks

To hook an object, so calls to its methods can be intercepted, use `hook_object()`. The prefix will be used when setting up callbacks for the hooks.

Example 3.18: Hooking an Object

```
$widget = new com_example_widget;
$pines->hook->hook_object($widget, 'com_example_widget->', false);
```

Note
`$pines`, along with all objects within it (except the hook, depend, config, and info services) are automatically hooked. Most components are designed to automatically hook their classes using the `factory()` static method.

To set up a callback, use `add_callback()`. A callback is called either before a method runs or after. The callback is passed an array of arguments or return value which it can freely manipulate. If the callback runs before the method and sets the arguments array to false (or causes an error), the method will not be run. Callbacks before a method are passed the arguments given when the method was called, while callbacks after a method are passed the return value (in an array) of that method.

The callback can receive up to 5 arguments, in this order:

1. `&$arguments`

   An array of either arguments or a return value.

2. `$name`

   The name of the hook.

3. `&$object`

   The object on which the hook caught a method call.

4. `&$function`

   A callback for the method call which was caught. Altering this will cause a different function/method to run.

5. `&$data`

   An array in which callbacks can store data to communicate with later callbacks.

A hook is the name of whatever method it should catch. A hook can also have an arbitrary name, but be wary that it may already exist and it may result in your callback being falsely called. In order to reduce the chance of this, always use a plus sign (+) and your component's name to begin arbitrary hook names. E.g. "+com_games_player_bonus".

If the hook is called explicitly, callbacks defined to run before the hook will run immediately followed by callbacks defined to run after.

A negative `$order` value means the callback will be run before the method, while a positive value means it will be run after. The smaller the order number, the sooner the callback will be run. You can think of the order value as a timeline of callbacks, zero (0) being the actual method being hooked.

Additional identical callbacks can be added in order to have a callback called multiple times for one hook.

The hook "all" is a pseudo hook which will run regardless of what was actually caught. Callbacks attached to the "all" hook will run before callbacks attached to the actual hook.

Example 3.19: Adding a Callback to a Hook

```
function com_example__callback() {
    // Do something.
}

// Run the callback before something() is called on com_example.
// Note the single quotes. Double quotes would be parsed by PHP and not work.
$pines->hook->add_callback('$pines->com_example->something', -10, 'com_example__callback ↩
    ');

// Run the callback before delete() is called on com_example_widget.
$pines->hook->add_callback('com_example_widget->delete', -1, 'com_example__callback');

// Run the callback after every method call is caught.
$pines->hook->add_callback('all', 10, 'com_example__callback');
```

To manually run callbacks (without an originating function call), use `run_callbacks()`.

Example 3.20: Manually Run Callbacks

```
$pines->hook->run_callbacks('+com_games_player_bonus');
```

3.3.4 Info (`info` class)

The info service provides information about Pines and installed components/templates. When Info is loaded, the `info.php` file in the `system` folder is retrieved. This file is used to fill variables in the info object. Similarly, when a component's variable is accessed, Info will look for an info.php file in the component's directory. It will use this file to create an object for that component. The `template` variable will contain the info of the current template. The info file should return an associative array with the following entries:

Table 3.4: Info Array Entries

Key	Required	Description
name	Yes	The component's name. This is the name displayed to the user.
author	Yes	The component's author.
version	Yes	The component's version.
license	No	The URL or name of the license under which the component is released.
website	No	The URL of the component's website.
services	No	An array of the names of services the component provides.
short_description	No	A short description of the component.
description	No	A description of the component.
depend	No	An associative array of dependencies which the component requires. The keys are the names of the dependency checker to use.
conflict	No	An associative array of dependencies which the component conflicts with. The keys are the names of the dependency checker to use.
recommend	No	An associative array of dependencies which the component recommends. The keys are the names of the dependency checker to use.
abilities	No	An array of abilities used by the component. Each array has three entries, a name, a title, and a description, in that order.

Example 3.21: Example Info File

```
defined('P_RUN') or die('Direct access prohibited');

return array(
    'name' => 'System Configurator',
    'author' => 'SciActive',
    'version' => '1.0.0',
    'license' => 'http://www.gnu.org/licenses/agpl-3.0.html',
    'website' => 'http://www.sciactive.com',
    'services' => array('configurator'),
    'short_description' => 'Manages system configuration',
    'description' => 'Allows you to edit your system\'s configuration and the
       configuration of any installed components.',
    'depend' => array(
        'pines' => '<2',
        'component' => 'com_jquery&com_ptags&com_pgrid&com_pform'
    ),
    'recommend' => array(
        'service' => 'entity_manager&user_manager'
    ),
    'abilities' => array(
        array('edit', 'Edit Configuration', 'Let the user change (and see) configuration
            settings.'),
        array('editperuser', 'Edit Per User Configuration', 'Let the user change (and
           see) per user/group configuration settings.'),
        array('view', 'View Configuration', 'Let the user see current configuration
           settings.'),
        array('viewperuser', 'View Per User Configuration', 'Let the user see current
           per user/group configuration settings.')
    ),
);
```

Example 3.22: Accessing Info for Pines and Components

```
// Access Pines' "license" info entry:
echo $pines->info->license;

// Access a component's "description" info entry:
echo $pines->info->com_example->description;

// Access a template's "version" info entry:
echo $pines->info->tpl_green->version;
```

3.3.5 Menu (menu class)

The menu service builds and renders menus. In Pines, menus are built using arrays, and these arrays are usually stored in JSON files. Menu will begin by loading the `menu.json` file in the `system` directory. This will set up the main menu and its submenus. It will then load any `menu.json` files found in all components' directories. The JSON structure of a menu file is an array of objects (because JSON doesn't support associative arrays) which are each placed into the `menu_arrays` variable in the menu object. Each entry is either a top level menu, or a menu entry. Top level menus require a position in which to place them, as well as several other values:

Table 3.5: Top Level Menu Array Entries

Key	Required	Description
path	Yes	The name of the menus path.

Table 3.5: (continued)

Key	Required	Description
text	Yes	The text to title the menu's module.
position	Yes	The page position in which to place the menu.
sort	No	Boolean value of whether the menu short be sorted alphabetically.
depend	No	An associative array of dependencies required to show the menu.

Regular menu entries have these values:

Table 3.6: Regular Menu Array Entries

Key	Required	Description
path	Yes	The path of this menu entry. Paths are hierarchical like file paths, using forward slashes as separators.
text	Yes	The text to show in the menu entry.
sort	No	Boolean value of whether the menu short be sorted alphabetically.
href	No	A URL, or an array of arguments to pass to `pines_url()`, which the entry will link to.
target	No	The HTML target of the link. Defaults to "_self".
onclick	No	JavaScript which will run when the entry is clicked.
depend	No	An associative array of dependencies required to show the entry.

An extra dependency called "children" can be used on menu entries in order to show the menu only if it has visible children.

To add a menu entry using code, you can insert it into the `menu_arrays` variable.

Example 3.23: Adding a Menu Entry Through Code

```
// Using pines_url().
$pines->menu->menu_arrays[] = array(
    'path' => 'main_menu/example',
    'text' => 'Example',
    'href' => array('com_example', 'something', array('id' => $some_id)) // These are ↩
       used as arguments for pines_url().
);

// Using a URL. This is placed under the above entry. It only appears when the user is ↩
   logged in. It opens in a new window.
$pines->menu->menu_arrays[] = array(
    'path' => 'main_menu/example/google',
    'text' => 'Google',
    'href' => 'http://google.com',
    'target' => '_blank',
    'depend' => array(
        'ability' => ''
    )
);
```

The system init script i80menus.php is responsible for loading the JSON files, by calling `add_json_file()` for the system menu file, then each component's menu file. This means that menu entries are not loaded until after all init/kill scripts and requested actions have run. This means that if you need to alter a component's menu entries (as opposed to just adding submenus and entries), you must hook Menu's `add_json_file()` or `render()` method.

Once all menu entries have been loaded, `render()` will remove entries when their dependencies are not met. It will convert the entries into multi-dimensional arrays for each menu using their path values. It then passes these arrays to the `menu()` method of the current template and places the return value into a module in the menu's requested position.

Example 3.24: Example menu.json File

```
[
    {
        "path": "main_menu/other/example",
        "text": "Example",
        "depend": {
            "children": true
        }
    },
        {
            "path": "main_menu/other/example/widgets",
            "text": "Widgets",
            "href": ["com_example","widget/list"],
            "depend": {
                "ability": "com_example/listwidgets"
            }
        },
        {
            "path": "main_menu/other/example/newwidget",
            "text": "New Widget",
            "href": ["com_example","widget/edit"],
            "depend": {
                "ability": "com_example/newwidget"
            }
        }
]
```

3.3.6 Page (page class)

The page service manages modules, notices, and errors, and renders content to be output to the client by the template. Modules, which are objects that use views to generate content, are attached to Page in specific positions. A template's info object will have a list of positions the template supports. Page uses PHP's output buffering system to retrieve the output from modules and templates. Page also controls the title of the page.

Page has several methods for manipulating the page title.

- `title`

  Append test to the title.

- `title_set`

  Replace the title.

- `title_pre`

  Prepend text to the title.

- `get_title`

  Retrieve the title.

Page has several methods for displaying notices and errors to the user. It is ultimately the responsibility of the template to display the messages to the user.

- `notice`
  Add a notice to be displayed to the user.
- `get_notice`
  Get the array of notices.
- `error`
  Add an error to be displayed to the user.
- `get_error`
  Get the array of errors.

Though module attachments are usually handled by the module when it is created, you can manually attach a module to Page. The second argument of `attach_module()` is the position in which to place the module. You can also manually detach the module. If the module is attached to the same position twice, it must likewise be removed twice.

Example 3.25: Manually Attaching and Detaching a Module

```
$module = new module;

// Attach the module.
$pines->page->attach_module($module, 'content');

// Detach the module.
$pines->page->detach_module($module, 'content');
```

The template will need to place the module content in the appropriate place on the page. To do this, templates use `render_modules()`, providing a position. If needed, the template can also provide a model, which will be used to format the content as necessary.

Example 3.26: Rendering the Modules in a Position Using a Model

```
<div id="modules_left">
    <?php echo $pines->page->render_modules('left', 'module_sidebar'); ?>
</div>
```

Under special circumstances, you may need to only output certain data to the client. For example, when you are returning JSON data to an AJAX request. To do this, set the `override` variable to true, then pass the content to `override_doc()`.

Example 3.27: Overriding the Page

```
// Turn override on.
$pines->page->override = true;

// Output JSON data.
$pines->page->override_doc(json_encode($var));

// If you need to retrieve the data:
$data = $pines->page->get_override_doc();
```

The page is rendered when the system init script `i90render.php` calls `render()`. If the page is overridden, the override document is returned. Otherwise, each module is rendered individually, then the template's `template.php` file is run. Because each module is rendered before the page, a module can cause another module to be attached to the page. However, that module will ultimately be rendered during the page's rendering. Therefore, if that module then causes another module to be attached in a position that has already been rendered, the third module will not be output to the user.

Note
In order to allow it to be hooked, `render()` only returns the page output. The init script then echoes it to the user.

3.4 Core Functions

Pines Core includes several functions. Most of these functions are just shortcuts to common methods of the Pines object and system services.

Table 3.7: Core Functions

Function	Description
`clean_checkbox`	Clean the name of a checkbox to use in an HTML form.
`clean_filename`	Clean a filename, so it doesn't refer to any parent directories.
`format_content`	Shortcut to `$pines->format_content()`.
`format_date`	Shortcut to `$pines->format_date()`.
`format_phone`	Shortcut to `$pines->format_phone()`.
`gatekeeper`	Shortcut to `$pines->user_manager->gatekeeper()`. If no user manager is installed, `gatekeeper` returns true.
`is_clean_filename`	Check whether a filename refers to any parent directories.
`pines_action`	Shortcut to `$pines->action()`.
`pines_depend`	Shortcut to `$pines->depend->check()`.
`pines_error`	Shortcut to `$pines->page->error()`.
`pines_log`	Shortcut to `$pines->log_manager->log()`. If no log manager is installed, `pines_log` returns true.
`pines_notice`	Shortcut to `$pines->page->notice()`.
`pines_redirect`	Shortcut to `$pines->redirect()`.
`pines_url`	Shortcut to `$pines->template->url()`. If there is no current template, `pines_url` returns null.
`punt_user`	Shortcut to `$pines->user_manager->punt_user()`. If no user manager is installed `punt_user` redirects the user to the homepage.

3.5 Core Classes

3.5.1 Component

The `component` class is a class which all components' main classes extend.

3.5.2 Template

The `template` class is a class which all templates' main classes extend. This class itself is fairly empty, consisting mostly of basic methods and variables just to satisfy the interface's requirements. It does, however, include the standard implementation of the `url` method. This is what is called when using the shortcut function `pines_url` (unless the current template implements it).

The `url` method accepts up to four arguments, in this order:

1. `$component`

   The name of the component that the URL should request.

2. `$action`

   The name of the action that the URL should request.

3. `$params`

   An associative array of parameters to place in the URL's query string.

4. `$full_location`

   Whether to use the full location, instead of the relative location. For links in email, RSS feeds, and similar contexts, a full location should be used.

If `url` is called with no arguments, it will return the relative URL of the index page.

3.5.3 Module

The `module` class is used to generate content using a view and place it into a page. Modules allow Pines to pass data from the logic portion of a component to the presentation portion. This allows the business logic to remain the same no matter which type of page is being constructed (XHTML, RSS, etc.). Modules select the proper view based on the format provided by the current template.

Modules have several variables. However, the position and order variables must be set by using the `attach` method or the constructor.

- `muid`

  A unique ID available to views to allow the use of IDs in HTML. Generated with `mt_rand`.

- `title`

  The module's title. This is usually set by the view.

- `note`

  The module's note. This is usually set by the view.

- `classes`

  A list of additional classes to be added to the module. This applies only to HTML modules.

- `content`

  The modules content. This is almost always, but not necessarily, filled by the output of a view.

- `component`

  The component which contains the view.

- `view`

  The view from which the content will be retrieved.

- `position`

  The position on the page in which the module is placed.

- `order`

  The order in which the module is placed in the position.

- `show_title`

  Whether or not the title and note of the module should be displayed.

When creating a module, the component, view, position, and order can be specified in the constructor. Position and order are optional. The order is not guaranteed and will be ignored if it is already taken.

Example 3.28: Creating a Module

```
// Add the widget form from com_example to the content part of the page.
// Place in order as #2, so #0, and #1 come before it.
$module = new module('com_example', 'widget/form', 'content', 2);
```

You can also manually attach a module once it is created.

Example 3.29: Manually Attach an Existing Module

```
$module = new module('com_example', 'widget/form');
if (isset($position))
    $module->attach($position);
else
    $module->attach('content');
```

You can use the module to generate content without putting it into the page.

Example 3.30: Using a Module to Render Content

```
$module = new module('com_example', 'widget/email');
$email_body = $module->render();
```

Caution
Even though the module is not added to the page in this example, it still uses the template's format to find the view.

Pines provides a blank view, which you can use if you want to add content to the module manually.

Example 3.31: Manually Creating the Module Content

```
// Of course, this example message would be much more suited to use pines_notice() or ↩
    pines_error(),
// but this is merely an example of what you could do.
$module = new module('system', 'null', 'content');
$module->title = 'Result';
$module->content('The process '.($success ? 'finished successfully.' : 'encountered an ↩
    error.'));

// And we can retrieve the content to log it.
pines_log($module->get_content());
```

For more information about creating and using views, please see the Views section.

3.6 Order of Execution

Pines is a very complex framework, and has many different interacting parts. Understanding how these parts are loaded and when they become available to use is important in developing successful Pines software. If you are reading this book sequentially, you may be unfamiliar with some of the terms used in this section. Since this section is so important for Pines development, I recommend you come back to this section after reading Part II, and read it again to reinforce your understanding of the order in which Pines executes various parts of the system.

Pines begins execution by a request to the `index.php` file.

- Defines constants.
  - `P_EXEC_TIME` - The microtime when the script began executing.
  - `P_RUN` - Used to determine that Pines was executed properly through `index.php`.
  - `P_BASE_PATH` - The base path of the Pines installation.
  - `P_INDEX` - The name of the index file. (Almost always `index.php`.)
  - `P_SCRIPT_TIMING` - Whether to use the script timing system. Change this value to help check for bottlenecks in your code.
- Scans the `system/init/` directory for the system init scripts, and executes them in order.
  1. `i00timing.php`
     - Loads script timing functions. These functions will use the JavaScript console.log function to display timing information for various portions of the script run.
  2. `i10functions.php`
     - Loads various functions used specifically by Pines Core.
  3. `i20interfaces.php`
     - Defines interfaces for system services and classes.
  4. `i30load_system.php`
     - Strips slashes from request variables. This ensures that request data always appears as if Magic Quotes are turned off.
     - Loads system classes from `system/classes/`.
     - Loads `$pines`.
       * Loads config service.
         · Loads system configuration.
         · Determines full and relative location using client provided hostname, if they are not already set by the config.
         · Sets `$pines->config->location` to static location, or relative location if static is not set.
       * Loads info service.
         · Loads system info.
       * Loads hook service.
       * Loads depend service.
         · Loads default dependency checkers.
       * Loads menu service.
       * Loads page service.
       * Checks system config.
         · Determines current template from config setting or request variable and sets it as the template service.
         · Gets timezone from config and passes to `date_default_timezone_set`.
         · If offline mode is turned on, loads `system/offline.php`, which ends execution here.
         · If the current template's class file is missing, loads `system/template_error.php`, which ends execution here.
         · Adds current template's class file to the list of class files.
       * Fills the lists of components and templates.

    * Finds components' class files and adds them to the list of class files.
    * Determines requested component and action, using URL rewriting if it is enabled.
5. `i40init_system.php`
   - Loads the class autoloader function. This function allows all component classes to be auto loaded the first time they are used.
   - Hooks the `$pines` object.
   - Starts a session.
   - Displays any pending notices and errors found in the session and removes them from the session. This is part of the redirection system.
6. `i50init_components.php`
   - Scans all components' init directories for init scripts.
   - Adds the `system/i01common.php` file to the list.
   - Sorts them by filename.
   - Executes each init script in order. Some important events are noted below.
     * i00_ : Common cleaning functions can be overridden here. However, shortcut functions (like `gatekeeper`) are not available yet.
     * i01_ : Common functions are being defined here.
     * i10_ : Components which provide system services should set themselves as the system service here.
     * i11_ : The user manager should fill the session with user data here.
     * i12_ and i13_ : Any component which may log the user out (such as a timer) should do it here.
     * i14_ : Conditional and per user/group config should be loaded here.
     * i15_ : Any component which may change the requested component/action should do it here.
   - Checks system config again in case config has changed. (Repeats same steps as `$pines` does when loading.)
7. `i60action.php`
   - Runs `action`, using the requested component and action.
   - If the string "error_404" is returned, issues a 404 HTTP status header and loads a module with "system" as component, "error_404" as view, and "content" as position.
8. `i70kill_components.php`
   - Scans all components' init directories for kill scripts.
   - Sorts them by filename.
   - Executes each kill script in order.
9. `i80menus.php`
   - Loads the menu file `system/menu.json`.
   - Loads each component's `menu.json` file consecutively.
   - Renders and attaches the menu.
     * Calculates each menu entry's dependencies and builds a multidimensional array of menu items whose dependencies are met.
     * Cleans each menu, removing any entry whose "children" dependency isn't met. Also removes some unnecessary data.
     * Creates and attaches a module for each menu, the content of which is filled with the response of the `menu` method of the current template.
10. `i90render.php`
    - Calls the `render` method of the page.
      * Returns the override document if override is set to true. (Does not do the steps immediately below.)
      * Iterates through each position, calling `render` on all the modules in that position.
      * Runs the `template.php` file of the current template.
    - Prints the response.

As mentioned before, this section is very important for Pines development. Knowing when certain features will be available for use allows you to plan accordingly. Pines is very flexible, but it is also very strictly structured. If you know Pines' execution path, debugging will be much easier. Remember to come back to this section if you are currently unfamiliar with the features discussed here.

Chapter 4

System Services

System services provide many features, and allow these features to be implemented in many different ways. System services are not provided by Pines Core. They are, however, well defined and must conform to certain guidelines to ensure that any component designed to use them will work with any implementation.

4.1 System Services

4.1.1 Configurator

The configurator service allows the user and other components to edit/save configuration and enable/disable components. In Pines, a component is disabled by prepending a period (.) to its directory's name.

The configurator itself has the following methods:

- `disable_component`

  Disables a component. This is accomplished by prepending a dot to its directory's name. Accepts the name of the component as the only argument.

- `enable_component`

  Enables a component. This is accomplished by removing the dot from the beginning of its directory's name. Accepts the name of the component as the only argument.

- `list_components`

  Creates and attaches a module which lists configurable components. It also returns the created module.

Example 4.1: Enabling and Disabling Components

```
// Enable the component com_good.
$pines->configurator->enable_component('com_good');

// Disable the component com_bad.
$pines->configurator->disable_component('com_bad');
```

4.1.1.1 Configurator Component

The configurator also provides a class for retrieving and manipulating a component's configuration. This class is the `configurator_component` class. It provides the static method `factory`, which accepts the component's name as the only argument and returns a new instance of `configurator_component` for the provided component. The class constructor also accepts this argument.

The component configurator class provides these methods:

- `get_full_config_array`

  Get a full config array. (With defaults replaced.)

- `is_configurable`

  Check if a component is configurable.

- `is_disabled`

  Check if a component is disabled.

- `print_form`

  Print a form to edit the configuration.

- `print_view`

  Print a view of the configuration.

- `save_config`

  Write the configuration to the config file.

- `set_config`

  Set the current config by providing an array of key => values. Any value not provided will be set back to default.

Example 4.2: Changing a Component's Configuration

```
// Load the component's configuration.
$conf = configurator_component::factory('com_example');
// Set the new configuration.
$conf->set_config(array(
    'host' => $host,
    'user' => $user,
    'password' => $password,
    'database' => $database
));
// Save the new configuration to disk.
$conf->save_config();
```

When the configuration is saved, the configurator creates a `config.php` file in the component's directory (if it does not already exist), and saves a script to the file, which returns the configuration values as an array of associative arrays. Each associative array contains an entry called "name" and an entry called "value". The config service uses this file when building the component's configuration, replacing values from `defaults.php` with the values from this file.

Example 4.3: Typical config.php File

```php
<?php
defined('P_RUN') or die('Direct access prohibited');
return array (
  0 =>
  array (
    'name' => 'string',
    'value' => 'dsgsdfgfg',
  ),
  1 =>
  array (
    'name' => 'multi_float',
    'value' =>
    array (
      0 => 9.33,
      1 => 8.4,
    ),
  ),
);
?>
```

4.1.2 Editor

The editor service provides a more friendly HTML editor for the user. When editing some sort of content, which is to be displayed in a page or email, a textarea is usually not the best solution. The editor service provides a method which allows the system administrator to choose between multiple HTML editors to provide for the system's users.

Once the editor is loaded, it will transform any textareas with the class "peditor" or "peditor-simple" into HTML editors. If it provides a simplified editor, textareas with the class "peditor-simple" will use this editor instead.

The editor has one method, `load`, which loads the editor. This method should be called from a view, to ensure it is only called when outputting HTML.

Example 4.4: Using the Editor Service in a View

```php
<?php
/**
 * Prints editors.
 *
 * @license http://www.gnu.org/licenses/agpl-3.0.html
 * @author Hunter Perrin <hunter@sciactive.com>
 * @copyright SciActive.com
 * @link http://sciactive.com/
 */
defined('P_RUN') or die('Direct access prohibited');
$this->title = 'Editors';
// Load the editor.
$pines->editor->load();
?>
<div>Regular Editor</div>
<div><textarea rows="3" cols="35" class="peditor"></textarea></div>
<br />
<div>Simple Editor</div>
<div><textarea rows="3" cols="35" class="peditor-simple"></textarea></div>
```

4.1.3 Entity Manager

Go brew some coffee or drink an energy drink, cause this is without a doubt, the most complex and important part of Pines! Understanding the entity manager lets you build complex relationships between data as easily as setting the value of a property.

4.1.3.1 Introduction

The entity manager service provides database abstraction for Pines. Data in Pines is based on objects called entities. An entity is an object which can hold any type of data available in PHP, including other entities. All entity classes (and therefore all entities) inherit the `entity` class.

Components can freely access the database if they wish, however it is strongly discouraged, as this prevents portability and extensibility. Using entities to store data has several benefits for both developers and site operators.

- Site operators can choose which database backend to use, based on their own needs.
- Moving from one database to another, backing up and restoring a database, and mirroring databases are much easier through an entity manager.
- Data storage is much easier and faster to code.
- Data querying is much easier and faster to code.
- A component can store data in another component's entities and extend its functionality.
- The user manager provides access control for all entities automatically.

Entities are not strictly structured, so data of any type can be added and saved just by assigning a variable on the entity and calling `save()`. This makes data manipulation in Pines very easy.

All entities are given a globally unique identifier (GUID), which is an integer. No entities in the same Pines installation will ever have the same GUID. The entity manager also provides UIDs, which can be used to number certain types of entities (or practically anything else). UIDs can be used to provide a more visibly pleasing identifier for entities.

Entities are organized with tags. For example, if a component, com_blog, wanted to store posts using an entity, it may add the tags "com_blog" and "post" to all new instances. This allows components to select and differentiate their entities from other components' entities.

An entity's class is usually named by appending to their component's name. In the previous example, the class would probably be called `com_blog_post`. When selecting entities with the entity manager, the class used to retrieve them must be specified, or the `entity` class will be used. If an entity is referenced in another entity's variable, the class is saved along with the GUID. When this variable is accessed later, the entity manager will retrieve the referenced entity using the saved class.

Warning
When changing an entity's class name, any entities referencing it must be resaved after setting the reference again using the new class name.

4.1.3.2 Creating Entities

To create an entity, call the `factory` static method of an entity's class. The `entity` class takes a list of tags as arguments. However, if an author chooses, the `factory` method of an inheriting class can take any other arguments. It will usually take only one argument, a GUID. It would return the entity with the GUID or, if it didn't exist or no GUID was provided, a new entity. The `factory` method is used instead of the new keyword to provide entities a chance to set up hooking on their objects.

Example 4.5: Creating an Entity

```
// Using the entity class.
$new_entity = entity::factory();

// Using a custom class.
$blog_post = com_blog_post::factory();
```

Much like blogs in many blogging systems, entities are organized using tags. Entity tags are not supposed to be added by the user, as this could allow them to create or manipulate users, groups, configurations, etc. It may seem inviting to use the entity tags to store user provided data, but don't! Though not strictly necessary, it is **highly recommended** to give every entity your component creates a tag identical to your component's name, such as 'com_xmlparser'. You don't want to accidentally get another component's entities.

Be very cautious when saving an entity in another entity's variable. If the referenced entity is newly created and does not have a GUID, the entity manager will not be able to retrieve it later. Always save the referenced entity first.

Example 4.6: Saving a Referenced Entity the Wrong Way

```
$entity = entity::factory();
$entity->foo = entity::factory();

$entity->save(); // foo hasn't been saved yet.
$entity->foo->bar = 'It works!';
$entity->foo->save();

$guid = $entity->guid;
unset($entity);
$entity = $pines->entity_manager->get_entity(array(), array('&', 'guid' => $guid));

isset($entity->foo->guid); // False
echo $entity->foo->bar; // Outputs nothing.
```

Example 4.7: Saving a Referenced Entity the Right Way

```
$entity = entity::factory();
$entity->foo = entity::factory();

$entity->foo->bar = 'It works!';
$entity->foo->save();
$entity->save(); // now foo has been saved.

$guid = $entity->guid;
unset($entity);
$entity = $pines->entity_manager->get_entity(array(), array('&', 'guid' => $guid));

isset($entity->foo->guid); // True
echo $entity->foo->bar; // Outputs 'It works!'.
```

Since the reference entity's class name is stored in the reference on the entity's first save and used to retrieve the reference entity using the same class, if you change the class name in an update, you need to reassign the reference entity and save to storage.

When an entity is loaded, it does not request its referenced entities from the entity manager. This is done the first time the variable/array is accessed. The referenced entity is then stored in a cache, so if it is altered elsewhere, then accessed again through the variable, the changes will **not** be there. Therefore, you should take great care when accessing entities from multiple variables. If you might be using a referenced entity again later in the code execution (after some other processing occurs), it's recommended to call `clear_cache`.

4.1.3.3 Entities

As mentioned before, entities are organized using tags. To add, remove, and check tags, the methods `add_tag`, `remove_tag`, and `has_tag` are used, respectively. Each takes any number of tags or an array of tags as arguments.

Example 4.8: Manipulating Entity Tags

```
$entity = entity::factory();

$entity->add_tag('com_foobar', 'foo', 'bar');
$entity->has_tag('foo'); // True

$entity->remove_tag('foo', 'bar');
$entity->has_tag('foo'); // False
```

To clear the cache of referenced entities, so that the next time one is accessed it will be pulled from the database, use the `clear_cache` method.

Example 4.9: Clearing the Reference Entity Cache

```
$entity = entity::factory();
$entity->foo = entity::factory();
$entity->foo->bar = 'Old value.';
$entity->foo->save();
$entity->save();

$entity = $pines->entity_manager->get_entity(array(), array('&', 'guid' => $entity->guid ↩
    ));

$inst_of_foo = $pines->entity_manager->get_entity(array(), array('&', 'guid' => $entity ↩
    ->foo->guid));
$inst_of_foo->bar = 'New value.';
$inst_of_foo->save();

echo $entity->foo->bar; // Outputs 'Old value.'
$entity->clear_cache();
echo $entity->foo->bar; // Outputs 'New value.'
```

Much like clearing the entity cache, you may need to refresh the entity's own data. Use the `refresh` method for this.

Example 4.10: Refreshing an Entity's Data

```
$entity = entity::factory();
$entity->foo = 'Old value.';
$entity->save();

$inst_of_ent = $pines->entity_manager->get_entity(array(), array('&', 'guid' => $entity ↩
    ->guid));
$inst_of_ent->foo = 'New value.';
$inst_of_ent->save();

echo $entity->foo; // Outputs 'Old value.'
$entity->refresh();
echo $entity->foo; // Outputs 'New value.'
```

As you've already seen in the examples, to save an entity, use the `save` method. Likewise, to delete the entity, use the `delete` method. You can also call the `save_entity`, `delete_entity`, and `delete_entity_by_id` methods of the entity manager itself. Entities use these methods themselves. This allows hooking of all entity saves and deletes.

Example 4.11: Different Ways of Saving and Deleting Entities

```
$entity = entity::factory();

// Save the entity.
$entity->save();
// or
$pines->entity_manager->save_entity($entity);

// Delete the entity.
$entity->delete();
// or
$pines->entity_manager->delete_entity($entity);
// or
$pines->entity_manager->delete_entity_by_id($entity->guid);
```

Several methods are provided by entities to find and compare them with other data. Entities cannot simply be checked using the == operator. It will almost always fail because of things like entity caching and sleeping references.

- `is`

  Perform a less strict comparison of two entities. To return true, the entity and the object passed must meet the following criteria:
  They must be entities.
  They must have equal GUIDs. (Or both can have no GUID.)
  If they have no GUIDs, their data must be equal.

- `equals`

  Perform a more strict comparison of two entities. To return true, the entity and the object passed must meet the following criteria:
  They must be entities.
  They must have equal GUIDs. (Or both can have no GUID.)
  They must be instances of the same class.
  Their data must be equal.

- `in_array`

  Check whether the entity is in an array. Takes two arguments, the array and a boolean `$strict`. If `$strict` is false, the function uses `is` to compare, and if it's true, the function uses `equals`.

- `array_search`

  Search an array for the entity and return the corresponding key. Takes two arguments, the array and a boolean `$strict`. If `$strict` is false, the function uses `is` to compare, and if it's true, the function uses `equals`.

Example 4.12: Comparing Entities

```
$entity = entity::factory();

$entity->is($entity); // True


$array = array('foo' => null, 'bar' => $entity);

echo $entity->array_search($array); // Outputs 'bar'
```

There are three functions to sort entities. Each one uses a different method of sorting.

- `hsort`

  Sort an array of entities hierarchically by a specified property's value.

- `psort`

  Sort an array of entities by parent and a specified property's value. If they have the same parent, their own value is used.

- `sort`

  Sort an array of entities by a specified property's value.

Example 4.13: Sorting Entities

```
$cats = $pines->entity_manager->get_entities(
        array('class' => com_foobar_category),
        array('&',
            'tag' => array('com_foobar', 'category')
        )
    );

$case_sensitive = false;
$reverse = false;

// Sort categories hierarchically by their name.
$pines->entity_manager->hsort($cats, 'name', 'parent', $case_sensitive, $reverse);

// Sort categories by their parent's name.
$pines->entity_manager->psort($cats, 'name', 'parent', $case_sensitive, $reverse);

// Sort categories by their name.
$pines->entity_manager->sort($cats, 'name', $case_sensitive, $reverse);
```

4.1.3.4 Entity Querying

The real power behind the entity manager is the entity querying system. After all, what's the use of having complex data and relationships if you can't find it?

The simplest way to get an entity is usually its `factory` static method. Almost all entities provided by SciActive maintained components use a `factory` that takes a GUID as an argument. However, the author of an entity's class can use whatever they like as arguments. For example, the `user` class takes either a GUID or a username. Usually the method will return a brand new entity if the queried entity is not found.

Tip
If you're using a user provided value as the GUID, casting it to an int will ensure it is evaluated as a GUID. Also, non-numeric values will be evaluated as 0, resulting in a new entity, since no entity has GUID 0. Then you can determine if it was found by checking that `guid` is set.

Example 4.14: Getting Entities Using the Factory Method

```
// Most entities use a GUID.
$baz = com_foobar_baz::factory((int) $_REQUEST['id']);

if (!isset($baz->guid)) {
    pines_notice('The specified baz cannot be found!');
    return;
}

// When selecting a user, you can use a GUID or a username.
$cron_user = user::factory('cron');

if (!isset($cron_user->guid)) {
```

```
    pines_notice('Can\'t find the cron user! Have you created one yet?');
    return;
}

// If the script made it to this point, both $baz and $cron_user were found!
pines_notice('Hooray! I found the baz you wanted and the cron user!');
```

The really powerful way of querying entities is the entity manager's `get_entities` and `get_entity` methods.

The first argument to these functions is an options array. It is an associative array of key value pairs signifying options. The options array can be empty, in which case the `entity` class will be used for returned entities. The following is a list of options which all entity managers support.

Table 4.1: Entity Querying Options

Option	Type	Default	Description
class	string	"entity"	The class to create each entity with. It must have a `factory` static method that returns a new instance.
limit	int	null	The limit of entities to be returned. Not needed when using `get_entity`, as it always returns only one.
offset	int	0	The offset from the oldest (or newest if reversed) matching entity to start retrieving.
reverse	bool	false	If true, entities will be retrieved from newest to oldest. Therefore, offset will be from the newest entity.
skip_ac	bool	false	If true, the user manager will not filter returned entities according to access controls.

Note
When using skip_ac, any referenced entities, when accessed, will also be retrieved using skip_ac.

A component or entity manager implementation is free to add extra options, but it should prepend the component's name to the name of the option. For example, if your component, com_entityversions, wants to add the option "date", it should use "com_entityversions_date".

Every argument following the options array is a selector. A selector is an associative array of criteria. An entity must pass each selector's criteria to be returned. The first member of the array (the value at index 0) is the type of selector, which determines how the criteria are matched against an entity's data.

Table 4.2: Entity Selector Types

Type	Name	Description
&	And	All values in the selector must be true.
|	Or	At least one value in the selector must be true.
!&	Not And	All values in the selector must be false.
!|	Not Or	At least one value in the selector must be false.

The following members of the array are the criteria of the selector. They must be in the form `$selector['name'] = $value`, or `$selector['name'] = array($value1, $value2,...)`.

Table 4.3: Entity Selector Criteria

Name	Value	Condition	Example Selector	Matching Entity
guid	A GUID.	True if the entity's GUID is equal.	`array('&', 'guid' => 12)`	`$entity->guid = 12;`
tag	A tag.	True if the entity has the tag.	`array('&', 'tag' => 'com_foobar')`	`$entity->add_tag('com_foobar');`
isset	A name.	True if the named variable exists and is not null.	`array('&', 'isset' => 'foo')`	`$entity->foo = 0;`
data	An array with a name, then value.	True if the named variable is defined and equal.	`array('&', 'data' => array('foo', false))`	`$entity->foo = 0;`
strict	An array with a name, then value.	True if the named variable is defined and identical.	`array('&', 'strict' => array('foo', 0))`	`$entity->foo = 0;`
array	An array with a name, then value.	True if the named variable is an array containing the value. Uses in_array().	`array('&', 'array' => array('foo', 'bar'))`	`$entity->foo = array('bar', 'baz');`
match	An array with a name, then regular expression.	True if the named variable matches. Uses preg_match().	`array('&', 'match' => array('foo', '/bar/'))`	`$entity->foo = 'foobarbaz';`
gt	An array with a name, then value.	True if the named variable is greater than the value.	`array('&', 'gt' => array('foo', 5))`	`$entity->foo = 6;`
gte	An array with a name, then value.	True if the named variable is greater than or equal to the value.	`array('&', 'gte' => array('foo', 6))`	`$entity->foo = 6;`
lt	An array with a name, then value.	True if the named variable is less than the value.	`array('&', 'lt' => array('foo', 7))`	`$entity->foo = 6;`
lte	An array with a name, then value.	True if the named variable is less than or equal to the value.	`array('&', 'lte' => array('foo', 6))`	`$entity->foo = 6;`
ref	An array with a name, then either a entity, or a GUID.	True if the named variable is the entity or an array containing the entity.	`array('&', 'ref' => array('foo', 12))`	`$entity->foo = com_foobar_baz::factory(12);`

So putting it all together, you can specify any of the options, and any number of selectors to find the exact entities you want.

Example 4.15: Getting Entities by Querying

```
// Get all entities using the entity class.
$entities = $pines->entity_manager->get_entities();

// Get only the first entity.
$entity = $pines->entity_manager->get_entity();

// Get the last entity.
$entity = $pines->entity_manager->get_entity(array('reverse' => true));

// Get all baz entities, using the com_foobar_baz class.
$entities = $pines->entity_manager->get_entities(
        array('class' => com_foobar_baz),
        array('&',
            'tag' => array('com_foobar', 'baz')
        )
    );

// Get the five newest baz entities.
$entities = $pines->entity_manager->get_entities(
        array('class' => com_foobar_baz, 'reverse' => true, 'limit' => 5),
        array('&',
            'tag' => array('com_foobar', 'baz')
        )
    );

// Get baz entities with names.
$entities = $pines->entity_manager->get_entities(
        array('class' => com_foobar_baz),
        array('&',
            'tag' => array('com_foobar', 'baz'),
            'isset' => 'name'
        )
    );

// Get baz entities with either first names or last names.
$entities = $pines->entity_manager->get_entities(
        array('class' => com_foobar_baz),
        array('&',
            'tag' => array('com_foobar', 'baz')
        ),
        array('|',
            'isset' => array('first_name', 'last_name')
        )
    );

// Get baz entities with names, who either make not greater than 8 dollars pay or are ↩
   under 22.
$entities = $pines->entity_manager->get_entities(
        array('class' => com_foobar_baz),
        array('&',
            'tag' => array('com_foobar', 'baz'),
            'isset' => 'name'
        ),
        array('!|', // at least one must be false
            'gte' => array('age', 22),
            'gt' => array('pay', 8)
        )
    );

// Get baz entities named Clark, James, Chris, Christopher, Jake, or Jacob.
$entities = $pines->entity_manager->get_entities(
```

```
        array('class' => com_foobar_baz),
        array('&',
            'tag' => array('com_foobar', 'baz')
        ),
        array('|',
            'strict' => array(
                array('name', 'Clark'),
                array('name', 'James')
            ),
            'match' => array(
                array('name', '/Chris(topher)?/'),
                array('name', '/Ja(ke|cob)/')
            )
        )
    );

// Get all baz entities, regardless of whether the logged in user has access to them.
$entities = $pines->entity_manager->get_entities(
        array('class' => com_foobar_baz, 'skip_ac' => true),
        array('&',
            'tag' => array('com_foobar', 'baz')
        )
    );
```

4.1.3.5 Exporting and Importing Entities

The entity manager has the following methods for exporting and importing entities:

- `export`
- `export_print`
- `import`

4.1.3.6 UIDs

The entity manager has the following methods for handling UIDs:

- `delete_uid`
- `get_uid`
- `new_uid`
- `rename_uid`
- `set_uid`

4.1.4 Icons

4.1.5 Log Manager

4.1.6 Template

4.1.7 Uploader

4.1.8 User Manager

4.1.8.1 User

4.1.8.2 Group

4.2 Creating a Service

Creating a system service for Pines can be a complex process, and certain rules must be followed to ensure it works with all components. All system services have an interface, which their main class must implement. Some services have additional classes which must be provided. These classes also have interfaces which must be implemented.

Interfaces are defined in the `system/init/i20interfaces.php` file.

In order to implement a service, one of the component's classes (usually its main class) must extend the `component` class and implement the appropriate interface for the service.

A component which implements a service must inform Pines that it is providing the service. To do this, the component uses an init script with the number 10. In the init script, it sets the service's variable under `$pines` to a string containing the class name. For example, if a component, com_xmlentity, is providing the entity manager service, it would have something like the following in an init script such as `i10set.php`.

Example 4.16: Setting a Component as a System Service in an Init Script

```
<?php
/**
 * Set the system entity manager.
 *
 * @license http://www.gnu.org/licenses/agpl-3.0.html
 * @author Hunter Perrin <hunter@sciactive.com>
 * @copyright SciActive.com
 * @link http://sciactive.com/
 */
defined('P_RUN') or die('Direct access prohibited');

/**
 * The entity manager.
 * @global com_xmlentity $pines->entity_manager
 */
$pines->entity_manager = 'com_xmlentity';

?>
```

The first time this variable is accessed, Pines will recognize it as a system service and load a new instance of the class into the variable. A component can implement any number of services.

The one exception to these practices is the template service. Templates do not have to use an init script, but can only provide the template service. They also must use their main class, which only needs to extend the `template` class. The `template` class already implements `template_interface`.

Chapter 5

Pines JavaScript Object

Part II

Application Development

Chapter 6

Components

6.1 Component Design

6.2 Views

Chapter 7

Templates

7.1 Template Design

Part III

Distribution

Chapter 8

Packaging

Chapter 9

Pines Plaza