Foundation

Overview

The goal of this document is to give a high-level overview of how the framework works, with more details on some of the major pieces.

Application Structure

The Root Directory

The default application structure of a hub is intended to provide a clean separation of a hub's content, configuration, extensions, and everything else that makes a hub unique from the core framework.

```
/hubzero
.. /administrator
.. /api
.. /app
.. /core
.. muse
.. index.php
.. htaccess.txt
.. robots.txt
```

The App Directory

The brain, or uniqueness, of a hub lives in the app directory. All (non-core) extensions installed, templates, cache files, uploaded content, and configurations will reside in this directory.

When developing extensions for a hub, the constant PATH_APP should be used for any paths relating to directories or files within the app directory. This is shorter and allows for the potential renaming of the directory while keeping the hub functioning smoothly.

The app directory contains a number of sub-directories used by the hub for managing extensions and files. Most of these directories will initially be empty.

bootstrap
: The bootstrap folder contains a few files that bootstrap the framework and configure available services.

cache
: The cache directory is used for storing generated content. Nothing within is vital but, rather, is used for dramatically improving site performance. The directory is further sub-divided by application type: admin, site, api, cli.

components
: The components directory is where 3rd-party and custom made components will reside.

config
: The config directory, as the name implies, contains all of the hub's configuration files.

logs

modules
plugins
templates
tmp

The Core Directory

If the app directory is the brain, the core directory is the skeleton, muscles, and heart of a hub, containing the framework and numerous pre-installed extensions.

As with the app directory, a global constant of PATH_CORE representing the file path is available.

Admin & API

The administrator and api directories are carry-overs from prior versions of the hub framework and marked for deprecation in a future version fo the framework. Do not place any files or folders within these two directories.

administrator
: The Administrator application, also known as the Back-end, Admin Panel or Control Panel, is the interface where administrators and other site officials with appropriate privileges can manipulate the appearance, enable/disable installed extensions, or manage users and content.

api
: Every hub comes with an API for accessing data from the various components and extensions in a light-weight, speedy manner. This directory contains the entry point to the API and can be accessed by visiting http://{yourhub}.org/api

Request Lifecycle

The entry point for all requests to an application is the index.php file. For /administrator and /api, this is the only file within those directories! All requests are directed to this file by the web server configuration. The index.php file doesn't contain much code. Rather, it is simply a starting point for loading the rest of the framework.

```
ROOT/administrator/index.php
```

```
ROOT/api/index.php
ROOT/index.php
```

The file itself is rather short and simple. Within index.php, a number of constants and paths are established, the file autoloader is included, and the core application bootstrap, which initializes the application / service container, is included.

The application serves as the central location that all requests flow through. Part of the instantiation process includes registering an array of bootstrappers that will be run before the request is executed. These bootstrappers configure error handling, logging, detect the application environment, and perform other tasks that need to be done before the request is actually handled.

All requests must then pass through a list of middleware, each of which processes the request and builds a response.

Entry Point

For /administrator, /api, and /, all incoming calls are routed to the index.php file within those directories.

```
ROOT/administrator/index.php
ROOT/api/index.php
ROOT/index.php
```

The file itself is rather short and simple. Within index.php, a number of constants and paths are established, the file autoloader in included, and the core application bootstrap which initializes the application is included. Finally, run is called on the application.

```
Incoming call
    -> index.php
        // Define constants for paths to the ROOT, /app, and /core dir
ectories
        -> include 'core/bootstrap/paths.php'

        // Include the file autoloader
        -> include 'core/bootstrap/autoload.php'

        // Include the application
        -> include 'core/bootstrap/start.php'

        // Run the application
        -> $app->run()
```

As noted, the initialization of the application, registering of services, and a number of other setup processes are contained within core/bootstrap/start.php. Next, we'll take a closer look at what happens in that file.

Application Initialization (core/bootstrap/start.php)

First and foremost, we set the strictest error reporting options, and also turn off PHP's error reporting, since all errors will be handled by the framework and we don't want any output leaking back to the user.

```
error_reporting(-1);
ini_set('display_errors', 0);
```

Next, we create a new application instance which serves as the "glue" for all the parts of a hub, and is the IoC container for the system binding all of the various parts.

```
$app = new HubzeroBaseApplication;
```

From there we try to automatically detect the client type being called (administrator, api, site, cli, etc). This will determine the set of services, facades, etc. that get loaded further on in the application lifecycle. Note that we detect the client and assign it to a $client variable, which we'll use later.

```
$client = $app->detectClient(array(

 'administrator' => 'administrator',
 'api'           => 'api',
 'cli'           => 'cli',
 'install'       => 'install',
 'files'         => 'files',

))->name;
```

The next step may look strange, but we actually want to bind the app into itself in case we need to Facade test an application. This will allow us to resolve the "app" key out of this container for the app's facade.

```
$app['app'] = $app;
```

Next up, the app's configuration is loaded. The configuration repository is used to lazily load in the options for this application from the configuration files (/app/config/*). The files are easily separated by their concerns so they do not become really crowded.

```
/* Note that we pass in the client type. This is because configuration
 options can potentially be overridden per client type. */
$app['config'] = new HubzeroConfigRepository($client);

// [!] Some legacy support here for old Joomla-defined constants
if (!defined('JDEBUG'))   define('JDEBUG',   $app['config']->get('debu
g'));
if (!defined('JPROFILE')) define('JPROFILE', $app['config']->get('debu
g') || $app['config']->get('profile'));
```

Register all of the core pieces of the framework including session, caching, and more. First, we'll load the core bootstrap list of services and then we'll give the app a chance to modify that list.

```
// Bootstrap path: core/bootstrap/client/services.php
$providers = PATH_CORE . DS . 'bootstrap' . DS . $client .  DS . 'serv
ices.php';
$services  = file_exists($providers) ? require $providers : array();

// Alternate bootstrap path following PSR-4 conventions: core/bootstra
p/Client/services.php
$providers = PATH_CORE . DS . 'bootstrap' . DS . ucfirst($client) .  D
S . 'services.php';
$services  = file_exists($providers) ? array_merge($services, require
$providers) : $services;
```

```
// App bootstrap path: app/bootstrap/client/services.php
$providers = PATH_APP . DS . 'bootstrap' . DS . $client .  DS . 'servi
ces.php';
$services  = file_exists($providers) ? array_merge($services, require
$providers) : $services;

foreach ($services as $service)
{
 $app->register($service);
}
```

The alias loader is responsible for lazy loading the class aliases setup for the application. First, we'll load the core bootstrap list of aliases and then, as with services, we'll give the app a chance to modify that list.

```
// Bootstrap path: core/bootstrap/client/aliases.php
$facades = PATH_CORE . DS . 'bootstrap' . DS . $client .  DS . 'aliase
s.php';
$aliases = file_exists($facades) ? require $facades : array();

// Alternate bootstrap path following PSR-4 conventions: core/bootstra
p/Client/aliases.php
$facades = PATH_CORE . DS . 'bootstrap' . DS . ucfirst($client) .  DS
. 'aliases.php';
$aliases = file_exists($facades) ? array_merge($aliases, require $faca
des) : $aliases;

// App bootstrap path: app/bootstrap/client/aliases.php
$facades = PATH_APP . DS . 'bootstrap' . DS . $client .  DS . 'aliases
.php';
$aliases = file_exists($facades) ? array_merge($aliases, require $faca
des) : $aliases;

$app->registerFacades($aliases);
```

Finally, this script returns the application instance. The instance is given to the calling script so we can separate the building of the instances from the actual running of the application and sending responses.

```
return $app;
```

Constants

System Constants

These constants are defined for use in the CMS and extensions:

DS	Directory separator. "/"
PATH_ROOT	The path to the current installation.
PATH_CORE	The path to the core framework of the CMS.
PATH_APP	The path to the app directory. This is where all a hub's data, custom code, and uploads will reside.

Note: These paths are the absolute paths of these locations within the file system, NOT the path used in a URL.

Service Providers

Overview

Service providers are the central place of application bootstrapping. All of a hub's core services are bootstrapped via service providers.

Every application or "client" type, such as "administrator" or "api", has their own list of services. These are all of the service provider classes that will be loaded for your application. Providers are laze loaded, meaning they will not be loaded on every request, but only when the services they provide are actually needed.

Standard Provider

Service providers must extend the HubzeroBaseServiceProvider class and are required to define at least one method: register(). Aside from the register method, a boot method may also be defined, which allows for a little setup or processing that may need to occur after all services have been registered.

The Register Method

```
<?php

namespace AppProviders;

use HubzeroBaseServiceProvider;

class FooServiceProvider extends ServiceProvider
{
    /**
     * Register services in the container.
     *
     * @return void
     */
    public function register()
    {
        $this->app['foo'] = function ($app)
        {
            return new Foo();
        });
    }
}
```

The Boot Method

The boot method is called after all other service providers have been registered, giving it access to all other services that have been registered by the framework.

```
<?php

namespace AppProviders;

use HubzeroBaseServiceProvider;

class FooServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
       $this->app['foo']->bar();
    }
}
```

Middleware Provider

A Middleware provider is an extended service provider with a handle method. Rather than extending HubzeroBaseServiceProvider, these providers extend HubzeroBaseMiddleware. While they can register services, they are not required to do so. Instead, they handle (i.e., modify) the incoming request and outgoing response.

The Handle Method

The handle method is called after the application has been booted and accepts a HubzeroHttpRequest object as the only argument.

```
<?php

namespace AppProviders;

use HubzeroBaseServiceProvider;
```

```
class FooServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function handle(Request $request)
    {
       // Forcefully set the 'foo' var to 'bar'
       $request->setVar('foo', 'bar');

       return $this->next($request);
    }
}
```

Facades

Overview

Facades serve as "static proxies" to underlying classes in the service container. This provides flexibility over traditional static methods with the benefit of terser syntax.

Use

In the context of a hub, a facade is a class that provides access to an object from the container. For this to work, all facades extend the base HubzeroFacadesFacade class.

A facade class only needs to implement a single method: getAccessor, which defines what to resolve from the container. The base Facade class makes use of the __callStatic() magic-method to defer calls from the facade to the resolved object.

Below is the facade for the Filesystem wherein the getAccessor() method returns the string 'filesystem', which is the key that the Filesystem service is registered with on the application.

```
class Filesystem extends Facade
{
    /**
     * Get the registered name.
     *
     * @return  string
     */
    protected static function getAccessor()
    {
        return 'filesystem';
    }
}
```

In the example below, a call is made to Filesystem to check that a file exists. Looking quickly at the code, one might assume that the static method exists() is being called on the Filesystem class:

```
<?php

namespace ComponentsBlogSiteControllers;

use HubzeroComponentSiteController;
use Filesystem;
use App;

class Media extends SiteController
{
    public function downloadTask()
    {
        //...

        if ( ! Filesystem::exists($file))
        {
            App::abort(404, 'File not found');
        }

        //...
    }
}
```

This facade serves as a proxy to accessing the underlying implementation of the HubzeroFilesystemFilesystem interface. So, when any static method on the facade is referenced, the application resolves the binding from the service container and runs the requested method against that object. In short, any calls made using the facade will be passed to the underlying instance of the filesystem service.

Class Reference

Below is a list of every facade, its underlying class, and the service container binding key where applicable.

Global (all client types)	Facade	Class	Service Key	Client
	App	HubzeroBaseApplication	app	all
	Auth	HubzeroAuthManager	auth	admin, s
	Cache	HubzeroCacheManager	cache	admin, s
	Component	HubzeroComponentLoader	component	admin, s
	Config	HubzeroConfigRepository	config	all
	Date	HubzeroUtilityDate		all
	Document	HubzeroDocumentManager	document	admin, s
	Event	HubzeroEventsDispatc	dispatcher	all

Facade	Class	Service Key	Client	
		her		
	Filesystem	HubzeroFilesystemFile system	filesystem	all
	Html	HubzeroHtmlBuilder	html.builder	admin, s
	Lang	HubzeroLanguageTran slator	language	all
	Log	HubzeroLogWriter	log.debug	all
	Module	HubzeroModuleLoader	module	admin, s
	Notify	HubzeroNotificationHan dler	notification	admin, s
	Pathway	HubzeroPathwayTrail	pathway	site
	Plugin	HubzeroPluginLoader	plugin	all
	Request	HubzeroHttpRequest	request	all
	Response	HubzeroHttpResponse	response	all
	Router	HubzeroRoutingRouter	router	all
	Session	HubzeroSessionManag er	session	admin, s
	Toolbar	HubzeroHtmlToolbar	toolbar	admin
	Submenu	HubzeroHtmlToolbar	submenu	admin
	User	HubzeroUserManager	user	all

Extensions

Overview

HUBzero CMS is already a rich featured content management system but if you're building a hub and you need extra features which aren't available by default, you can easily extend it with extensions. There are five types of extensions: Components, Modules, Plugins, Templates, and Languages. Each of these extensions handle specific functionality.

Components

The largest and most complex of the extension types, a component is in fact a separate application. A component is a relatively self-contained portion of code with its own functionality, its own database tables and its own presentation. Examples of components are a forum, a blog, a wiki, a photo gallery, etc. One could easily imagine all of these as separate applications or stand-alone systems. A component will be shown in the main part of the website and only one component will be shown. A menu is then in fact nothing more then a switch between different components.

Modules

Modules are extensions which present certain pieces or smaller chunks of information on the site. It is not uncommon to have a number of modules on each web page. A module differs from a component in that it doesn't make sense as a standalone application; Rather, it will just present information or add a functionality to an existing application. Common examples would include displaying the latest blog post on the home page or a search box to be present throughout the site. This is a small piece of re-usable HTML that can be placed anywhere desired and in different locations on a template-by-template basis. This allows one site to have the module in the top left of their template, for instance, and another site to have it in the right side-bar.

Plugins

Plugins serve a variety of purposes. As modules enhance the presentation of the final output of the Web site, plugins enhance the data and can also provide additional, installable functionality. Plugins enable you to execute code in response to certain events, either core events or custom events that are triggered from your own code. This is a powerful way of extending the basic functionality.

Templates

A template is a series of files within the Joomla! CMS that control the presentation of the content. The template is not a website; it's also not considered a complete website design. The template is the basic foundation design for viewing your website. To produce the effect of a "complete" website, the template works hand-in-hand with the content stored in the database.

Each hub comes with default templates for both the administrator area and the front-end site.

- **administrator** - kameleon
- **site** - kimera

Languages

Probably the most basic extensions are languages. Languages can be packaged in two ways, either as a core package or as an extension package. In essence, these files consist key/value pairs, these pairs provide the translation of static text strings which are assigned within the source code. These language packs will affect both the front and administrator side. Note: these language packs also include an XML meta file which describes the language and font information to use for PDF content generation.

Conclusion

If the difference between the three types of extensions is still not completely clear, then it is advisable to go to the admin pages of your installation and check the components menu, the module manager and the plugin manager. A hub comes with a number of core components, modules and plugins. By checking what they're doing, the difference between the three types of building blocks should become clear.