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;
```