Usage Documentation

For PzPHP 1.0.X

Table of Contents

Introduction

PzPHP was built to be easy to use, and very easy to get started with. This usage guide will help you use PzPHP's various modules. It will also touch on many Pz Library functionality that is easily available through PzPHP.

Throughout this guide you will be shown clear code examples, as well as tips that will reveal seemingly hidden PzPHP functionality. That makes this guide a useful development tool during your first project using PzPHP.

The PzPHP team makes sure that this usage guide stays up-to-date with the most recent stable PzPHP version.

Requirements

PzPHP requires a standard PHP 5.3 installation to function properly. As of version 1.0.2, PzPHP functions just as well on PHP 5.4. Future versions of PHP will also be supported.

Keep in mind that it is up to you to install extensions for specific PHP functionalities (like mysqli).

The Pz Library

While you read through this guide, you will notice how we use PzPHP and the Pz Library interchangeably. Although this may seem as an inconsistency; the truth is that PzPHP owes almost all of its abilities to the Pz Library. PzPHP's role is to abstract away from Pz Library's low-level methods, allowing developers to more easily build their applications.

The Pz Library is a collection of useful methods and features developed over the years by Kevork Aghazarian and Fayez Awad. It is still updated today and new features that are introduced to the Pz Library eventually pop-up in PzPHP.

Configuration

PzPHP works out of the box. It does not require any configuring for it to work in your setup. However, thanks to the Pz Library, you do have a good amount of control over how some of its internal features work.

Before we get into the specific configuration options, we will cover how PzPHP deals with configuration files, including configuring for different environments (development, production, etc…).

PzPHP Configuration

The config.php file holds all of PzPHP's configuration constants. Constants are prefixed with either "PZPHP_" or "PZ_". These prefixes signify exactly where these settings are used. Generally speaking, these values do not need to be modified. However, they do allow you to enable (or disable) different features of PzPHP, such as the Pz Debugger (see the "Debugger" chapter).

Custom Configuration

By default, there is an empty my_config.php file where you can put all of your custom configuration values. Both this file and config.php are included in bootstrap.php (see the "Bootstrap" chapter). If you do not plan to use this file, comment-out its inclusion in bootstrap.php.

Configuring For Different Environments

PzPHP makes it really easy for you to separate different environment configurations. By default, config.php, and my_config.php are used when in production mode. You set the environment mode in the .htaccess file on line 2 (PZPHP_ENVIRONMENT).

When this value is set to anything other than “production”, PzPHP will look for configuration files suffixed with the value used for “PZPHP_ENVIRONMENT” in the .htaccess file. For example, if your .htaccess file was setup like so:

```
###ENVIRONMENT VARIABLES
SetEnv PZPHP_ENVIRONMENT dev
```

Then PzPHP will attempt to load config_dev.php, and my_config_dev.php.

It is as simple as that to use different configuration files for different environments. When you deploy to production (or any other environment), all you have to do is change the PZPHP_ENVIRONMENT variable in your .htaccess file.

Full Configuration Overview

For the purposes of this guide, we will cover all of the configuration options in config.php. This review will be comprehensive, so you do not have to spend too much time right now learning every available configuration option. However, once you are more comfortable with using the framework, you can come back to this section to learn more on how to tune PzPHP more to your needs.

PZPHP_X_DIR
All constants that begin with “PZPHP_” and end with “_DIR” define the default directory structures that PzPHP is using. Generally speaking, you should not change these values; however if you know what you are doing, feel free to set new values for your particular setup.

PZPHP_CACHE_MODE_X

All constants beginning with “PZPHP_CACHE_MODE_” identify a different method of caching (see the Caching chapter). One of these values needs to be assigned to the “PZPHP_CACHING_MODE” constant. By default, “PZPHP_CACHE_MODE_NO_CACHING” is assigned to that constant.

PZPHP_DATABASE_X

All constants beginning with “PZPHP_DATABASE_” identify a different method of accessing a database (see the Database chapter). One of these values needs to be assigned to the “PZPHP_DATABASE_MODE” constant. By default, “PZPHP_DATABASE_MYSQLI” is assigned to that constant.

PZ_SECURITY_HASH_TABLE

The Pz Library uses a custom encryption algorithm for its one-way and two-way encryption method. It takes from the traditional Caesar encryption method, and attempts to take-away from its traditional short-comings. See the Security chapter to find out how to customize your particular PzPHP installation, to make encryption even harder to crack.

PZ_SECURITY_SALT

The Pz Library by default uses a salt when encrypting strings. You may override this global salt using this constant (see more in the Security chapter).

PZ_SECURITY_POISON_CONSTRAINTS

The Pz Library employs an extra layer of encryption strength by poisoning hashed or encrypted strings at predefined locations. See the Security chapter to see how you can customize poisoning for your particular PzPHP installation.

PZ_SECURITY_REHASH_DEPTH

The Pz Library re-hashes a string several times to prevent traditional rainbow cracking techniques. You can set your own custom rehashing depth here (any value above 0 is recommended, higher values are better).

PZ_SETTING_DB_CONNECT_RETRY_ATTEMPTS

Upon PzPHP coming across a failed connection attempt to your database, it will try to reconnect before giving up completely. You can disable this feature, or increase the retry attempts by changing this value.

PZ_SETTING_DB_CONNECT_RETRY_DELAY

This setting tells PzPHP how long it should wait between database connection retry attempts.

PZ_SETTING_DB_AUTO_CONNECT_SERVER
If this setting is set to true, database connections are attempted automatically once a server has been added. This goes against PzPHP's goals of being as lightweight as possible. However, sometimes you may need this type of behavior.

PZ_SETTING_DB_AUTO_ASSIGN_ACTIVE_SERVER
Since PzPHP allows you to add multiple database servers, as well as communicate with multiple database servers at the same time; it sets one active database server that is used by default when using database methods (see the Database chapter). When this setting is set to true, when you add a server, it will automatically be set as the active one.
PZ_SETTING_DB_WRITE_RETRY_FIRST_INTERVAL_DELAY
PzPHP automatically handles deadlock detection when using MySQL. It will retry an insert when a deadlock is detected. There are two stages to deadlock detection in PzPHP (see the Database chapter). This setting defines (in milliseconds) the delay between retries in the first stage of retry attempts.

PZ_SETTING_DB_WRITE_RETRY_SECOND_INTERVAL_DELAY
PzPHP automatically handles deadlock detection when using MySQL. It will retry an insert when a deadlock is detected. There are two stages to deadlock detection in PzPHP (see the Database chapter). This setting defines (in milliseconds) the delay between retries in the second stage of retry attempts.

PZ_SETTING_DB_WRITE_RETRY_FIRST_INTERVAL_RETRIES
This setting dictates how many retry attempts PzPHP will make in the first stage of deadlock detection (see the Database chapter).

PZ_SETTING_DB_WRITE_RETRY_SECOND_INTERVAL_RETRIES
This setting dictates how many retry attempts PzPHP will make in the second stage of deadlock detection (see the Database chapter).

PZ_SETTING_CACHE_CONNECT_RETRY_ATTEMPTS
Upon PzPHP coming across a failed connection attempt to your cache server, it will try to reconnect before giving up completely. You can disable this feature, or increase the retry attempts by changing this value.

PZ_SETTING_CACHE_CONNECT_RETRY_DELAY
This tells PzPHP how long it should wait between failed connection retry attempts.

PZ_SETTING_CACHE_AUTO_CONNECT_SERVER
If this setting is set to true, cache server connections are attempted automatically once a server has been added. This goes against PzPHP's goals of being as lightweight as possible. However, sometimes you may need this type of behavior.

PZ_SETTING_CACHE_AUTO_ASSIGN_ACTIVE_SERVER
Since PzPHP allows you to add multiple cache servers, as well as communicate with multiple cache servers at the same time; it sets one active cache server that is used by default when using cache methods (see the Cache chapter). When this setting is set to true, when you add a server, it will automatically be set as the active one.

PZ_SETTING_OUTPUT_COMPRESSION
PzPHP has the ability to shave off a small to significant amount of bytes from your final response by compressing the output (removing extra whitespace, moving everything to one line, etc…). This is achieved using a combination of output buffering, as well as a custom algorithm that scans every byte of response code/text. This compression method is very fast, and has little to no impact on total memory usage, and execution time.

PZ_SETTING_OUTPUT_BUFFERING
When set to true, PzPHP automatically calls a blank ob_start() at the beginning of the script's execution.

PZ_SETTING_DOMAIN_PROTECTION
This enables or disables domain protection. For more information, see the Security chapter.

PZ_SETTING_DOMAIN_ALLOWED_DOMAINS
One or more allowed domains can be set for this value. For more information, see the Security chapter.

PZ_SETTING_DOMAIN_TARGET_DOMAIN
The target domain for the domain protection feature. For more information, see the Security chapter.

PZ_SETTING_DEBUG_MODE
When set to true, the Pz Debugger will be activated, and will begin to monitor script events. For more information, see the Debugging chapter.

PZ_SETTING_DEBUG_ERROR_LOGGING
When set to true, PzPHP will begin logging mysql, pdo, memcache, etc… errors to error logs in the LOGS directory.

PZ_SETTING_DEBUG_DISPLAY_BAR
If debug mode has been activated, you can turn on the Pz Debug Bar, that will give you access to various statistics on the current script that executed (when viewing in your browser). For more information, see the Debugging chapter.

PZ_SETTING_DEBUG_DB_USER
If you plan on logging debugger information to a database, you will need to provide the database user here.

PZ_SETTING_DEBUG_DB_PASSWORD
If you plan on logging debugger information to a database, you will need to provide the database password here.

PZ_SETTING_DEBUG_DB_NAME
If you plan on logging debugger information to a database, you will need to provide the database name here.

PZ_SETTING_DEBUG_DB_HOST
If you plan on logging debugger information to a database, you will need to provide the database host here.

PZ_SETTING_DEBUG_DB_PORT
If you plan on logging debugger information to a database, you will need to provide the database port here.

PZ_SETTING_DEBUG_DB_LOG
If you plan on logging debugger information to a database, set this value to true.

PZ_SETTING_DEBUG_LOG_FILE_AUTO_ROTATE
If you have error logging enabled, set this to true if you want PzPHP to automatically rotate log files.

PZ_SETTING_DEBUG_DELETE_LOG_FILES_AFTER_X_DAYS
If auto rotate has been enabled, set this value to the number of days old an error log file needs to be before PzPHP deletes it.

PZ_SETTING_DEBUG_X_LOG_ERRORS
All constants that begin with “PZ_SETTING_DEBUG_” and end with “_LOG_ERRORS” define if a particular error log should be enabled.

PZ_SETTING_DEBUG_X_ERROR_LOG_FILE_NAME
All constants that begin with “PZ_SETTING_DEBUG_” and end with _ERROR_LOG_FILE_NAME” define the name of particular error log files.

PZ_SETTING_DEBUG_PHP_DISPLAY_ERRORS
If set to true, PHP errors will be displayed inline in your response (should always be turned off for production environments).

Bootstrap

Basic Usage

The bootstrap.php file is found in the root of the PzPHP installation directory. This script loads the required configuration files (based on environment), and initializes the Pz Class Autoloader, and the PzPHP Core itself.

You can use this script in two ways. The first way is to use it as an include for all of your application's pages/scripts. This will give you access to the PzPHP Core, and thus, everything you need t get started with PzPHP.

The second way to use the bootstrap.php file is as a gateway for all requests to your application. You would configure your .htaccess file to route all requests to the bootstrap.php file. Then, you would handle the requests there, and internally redirect to the proper script/controller.

This gives you a lot of flexibility in comparison to traditional frameworks which always require a class to be the entry point to your application (at least when employing the MVC pattern).

Accessing PzPHP

The variable in which the PzPHP Core object is stored is called $_PZPHP. It is passed to all registered modules, and can be accessed globally from any of your classes/functions.

PzPHP Core

The PzPHP_Core class is your starting point when using PzPHP. You never directly interact with any underlying module, or library class. PzPHP_Core gives you access to everything you need, which includes all official modules, as well as direct access to the Pz Library Core (for more fine-grained access).

The PzPHP_Core class also offers several other useful methods that you can use in your application.

Register a Module

PzPHP allows you to register your own modules to work with the rest of the PzPHP framework. Building a module for the PzPHP framework comes with a lot of bonuses. For one, all well-formed modules have instant access to the PzPHP_Core class (see the Modules chapter for more information). This means your module can leverage any of the other functionality PzPHP has to offer, without any kind of hack-ish workarounds. Your module also gets loaded on demand, like every other module, saving you memory and execution time wherever possible.

To register a module with PzPHP, call the registerModule() method from PzPHP_Core:

```
$_PZPHP->registerModule('Your_Module_Class_Name');
```

The method takes in one parameter, which is the full class name of your module. It will return true or false depending on if the registering process was successful or not. This only returns false if you have already registered the module. Remember: registering a module does not instantiate the object. It simply tells PzPHP that this module exists and may be used during this scripts execution.

If you want to access a module, it is as simple as calling the module() method:

```
$_PZPHP->module('Your_Module_Class_Name')->someMethod();
```

That's it! That is all you need to do to register a custom module with PzPHP. Take note that the registerModule() method does not actually require you to pass it a custom PzPHP module. You can register any kind of class you would like. The functionality stays the same.

Once you register a module, you cannot unregister it. However there is no real reason why you would want to unregister a module. If you find yourself in the need of this functionality, you may want to reconsider your approach to using PzPHP.

Add/Remove/Change/Get Variables

PzPHP_Core also allows you to register variables of any kind with PzPHP. The benefits of registering a variable with PzPHP_Core are that it can be accessed via any module or class with access to PzPHP_Core. This means you can easily share variables amongst classes and methods. Think of this as a more socialist way of using dependency injection.

Here is an example of how you can use this kind of functionality (forgive us for using a semi-procedural approach for this example!):

```php
$_PZPHP->registerVariable('age', 17);
$_PZPHP->registerVariable('ageLimit', 18);

function isOfDrinkingAge(PzPHP_Core $pzphpcore)
{
    if($pzphpcore->getVariable('age') >= $pzphpcore->getVariable('ageLimit'))
    {
        return true;
    }
    else
    {
        //dice roll to see if you will lie
        if(mt_rand(1,2) === 1)
        {
            lieAboutAge($pzphpcore);

            return isOfDrinkingAge($pzphpcore);
        }
        else
        {
            return false;
        }
    }
}

function lieAboutAge(PzPHP_Core $pzphpcore)
{
    $pzphpcore->changeVariable('age', 22);
}
```

```php
//this will return true if the dice roll is 1 or age is 18 or above
if(isOfDrinkingAge($_PZPHP))
{
    echo 'Get in!';
}
else
{
    echo 'Get out!';
}

//remove our variables
$_PZPHP->unregisterVariable('age', 'ageLimit');
```

Again, it is as simple as that to register and use variables with PzPHP!

Accessing Official Modules

The PzPHP_Core allows quick and easy access to official modules that are packaged with PzPHP.

Each of these modules have their own chapter in this guide, however here are some quick examples of how you can access them:

```php
//not really a module, but you have direct access to the Pz Library Core
$_PZPHP->pz()->getSetting('some_setting');

//easy access to the Database module
$_PZPHP->db()->select('SELECT * FROM my table');

//easy access to the Cache module
$_PZPHP->cache()->cacheMethod();

//easy access to the Localize module
$_PZPHP->localize()->setLocale('en_us');

//easy access to the Security module
$_PZPHP->security()->twoWayDecrypt('asf43dfasdfasf');
```

Modules

PzPHP ships with a small bundle of modules for you to use and interact with. These modules allow you to leverage the power of the Pz Library. All official modules are easily accessible via their own methods in the PzPHP Core.

Sticking to PzPHP's goals of being performance-centric, modules are not loaded into memory until you explicitly attempt to use one of their methods. PzPHP takes care of all of it for you. You never have to worry about a module being available or not. You can worry about more important things, while PzPHP takes care of the foundation of your application.

Of course, you are not locked in to this convention. PzPHP wants you to help you develop faster. So it tries its best to impose the least amount of restrictions possible when using its core functionality.

For example, if you are using mysqli, you can access the most common features via the Database module:

```
$_PZPHP->db()->insert("INSERT INTO my table () VALUES ()");
$_PZPHP->db()->affectedRows();
$_PZPHP->db()->insertId();
```

However, if you wanted to use a specific method not covered by the Database module, or you prefer using the mysqli object directly, you could just as easily transform the code above to the following:

```
$mysqli = $_PZPHP->db()->returnActiveServerObject()->returnMysqliObj();
$mysqli->query("INSERT INTO my table () VALUES ()");
$mysqli->affected_rows;
$mysqli->insert_id;
```

Of course, you will need knowledge in how the Pz Library works to more easily maneuver around its low-level functionality. However a respectable IDE will help you greatly.

Creating Your Own Module

To create a class or set of classes considered real PzPHP modules, all you need to do is first extend your class with the PzPHP_Wrapper. And then register your module with PzPHP in your bootstrap.php file. Below are examples of this:

```php
class Module_MyModule extends PzPHP_Wrapper
{
    public function myMethod()
    {
        return $this;
    }
}
```

```php
$_PZPHP->registerModule('Module_MyModule');
```

Note that you can name your class anything you want.

Once you register your module, you can access it with one method:

```php
$_PZPHP->module('Module_MyModule')->myMethod();
```

PzPHP_Wrapper

The first thing we should look at is the PzPHP_Wrapper that you are extending from. There are two reasons for why you would want to extend from the PzPHP Wrapper.

First off, it gives you instant access to the PzPHP core. No hassle, no fuss:

```
class Module_MyModule extends PzPHP_Wrapper
{
    public function myMethod()
    {
        return $this->pzphp()->getVariable('some_variable');
    }
}
```

In the above example, we use $this->pzphp() to return the PzPHP_Core instance. We then access its getVariable() method. This example shows you how easy it is to share variables amongst all modules, without having to use “global”, or traditional dependency injection.

The second reason it is useful to extend from the PzPHP Wrapper is, because you can overload its init() method. As you already know, PzPHP uses an on-demand system to load classes only when necessary. This means you do not have direct access to a class’ __construct() method (at least if you are registering it as a module).

When PzPHP loads a module on-demand, it will check to see if it has a public init() method available. If it does, it will execute it, and pass it the PzPHP core as its first parameter.

This allows you to overload the init() method, basically converting it into a __construct() alternative.

```php
class Module_MyModule extends PzPHP_Wrapper
{
    private $_someFlag = false;

    public function init(PzPHP_Core $PzPHPCore)
    {
        parent::init($PzPHPCore);

        $this->_someFlag = $this->pzphp()->getVariable('some_flag');
    }

    public function myMethod()
    {
        return $this->pzphp()->getVariable('some_variable');
    }
}
```

In the above example, we are overloading the init() method from the PzPHP_Wrapper class. We make sure we first call the parent logic, and pass the $PzPHPCore variable. After that, we can setup whatever we need to for our class (if necessary).

This also shows you another way you can use the get and set variable methods in the PzPHP_Core.

That's it! You just saw the simple steps to starting your own PzPHP module. Remember, you do not have to do any of the above to register a "module" with PzPHP. However, as always, there are benefits to using the PzPHP_Wrapper, if you want to really integrate with what the Pz Library, and PzPHP have to offer.

PzPHP Db Module

The Db (Database) module is meant to give you easy access to the most commonly used methods when interacting with a database. This is where PzPHP tries to make things really simple for you. Using this Db module, you could start your application off using a database such as SQLITE, and when going to production switch to MYSQL without any code/logic change in your application.

This is done using the "PZPHP_DATABASE_MODE" constant in config.php. You can set whatever database you want PzPHP to connect to and use, sit back, and see how your application automatically converts to the new database (assuming your queries are compatible with the new database.

The only other thing you need to make sure is correct, is the login credentials (of course).

How to Connect to a Database

If you remember reading from previous chapters, PzPHP wants to make your application as lightweight as possible. This holds true to database connections as well. When you add server credential information to PzPHP, it is stored and referred to only when your first query is executed. This means if a particular script in your application runs and does not do anything to the database; you do not have to worry about an unnecessary connection. At the same time, PzPHP also handles disconnecting from all active database connections.

Of course, you have the ability to manually connect and disconnect whenever you want, but PzPHP handles that stuff for you by default.

```
$dbId = $_PZPHP->db()->addServer('username', 'password',' dbname', 'localhost', 3306);
```

The above example registers one set of database credentials with PzPHP. The returned value is an identifier for this set of credentials. This highlights another feature of the Db Module; PzPHP allows you to connect to and use as many different databases as you need to. It manages this by assigning an Id every time you call the addServer() method.

If you have not changed the default value of "PZ_SETTING_DB_AUTO_ASSIGN_ACTIVE_SERVER" in config.php, then by default, the id returned by addServer() is set to the active database server id.

This means all other query related methods in the Db Module will use this server's credentials (or connection object if a connection has already been established).

Below is an example to clarify all of this:

```php
#returns id 1, and assigns it as active
$dbId = $_PZPHP->db()->addServer('username', 'password',' dbname', 'localhost', 3306);

#returns id 2, and assigns it as active
$dbId2 = $_PZPHP->db()->addServer('username2', 'password2',' dbname2', 'some.host', 3306);

#connects to database server #2 ($dbId2) as it is the current active one.
#once the connection completes, the query is executed.
$result = $_PZPHP->db()->select("SELECT * FROM a table");

#lets say for one query only, we need the first database. We can override the active id like this
$result = $_PZPHP->db()->select("SELECT * FROM a table", $dbId);
#this queried the database located at localhost

#lets say we want the rest of this script to use the first database
#instead of passing its ID to every call, we can change the current active id like so
$_PZPHP->db()->setActiveServerId($dbId);

#this query will now be sent to the first database ($dbId)
$result = $_PZPHP->db()->select("SELECT * FROM a table");
```

Methods in the Db module conform to this methodology.

PDO Support

The Db module has basic PDO support. Almost nothing in previous examples changes when using PDO. There are however some additional parameters that are available in the addServer() method which are PDO specific. Some other methods in the Db module also require an extra parameter when using PDO. The examples below illustrate these differences.

```
$driverOptionsArray = array(
    PDO::MYSQL_ATTR_READ_DEFAULT_FILE => '/etc/my.cnf'
);

$server = ''; //if the mysql pdo driver needed a server value, you would set it here.
$protocol = ''; //if the mysql pdo driver needed a protocol value, you would set it here.
$socket  = ''; //if the mysql pdo driver needed a socket value, you would set it here.

$dbId = $_PZPHP->db()->addServer('username', 'password',' dbname', 'localhost', 3306, $driverOptionsArray, $server, $protocol,
$socket);
```

When you set the “PZPHP_DATABASE_MODE” constant to a PDO value, the addServer() method will route to the PDO extension in the Pz Library. The Pz Library handles the generation of the DSN string based on the information given. You can also pass PDO driver options in an array.

CRUD Queries and Queries in General

The Db Module contains methods for the most common queries. At its current version, these methods may look redundant, as they all call the same method in the Pz Library. However, it is recommended that you still use them appropriately, since in future versions of PzPHP, they may change to better suit the type of query they are expecting.

These methods are: select(), insert(), delete(), update(), set(), optimize(), analyze(), check().

If you need to run a query that does not have its own method, you can do so by first getting the database object (varies depending on if you are using mysqli, mysql, pdo, etc…), and then call the respective query method.

Query Sanitization

The Db Module also allows you to sanitize variables before you place them into your query. Sanitization is crucial in any application, and PzPHP offers you some neat and easy to use ways to clean user input.

You have two methods in the Db Module: sanitizeNumeric() and sanitizeNonNumeric().

If your variable needs to be a number (integer, or float), then you would use the sanitizeNumeric() method. It expects the first argument to be your variable, and has two optional arguments that define decimal places, and as well as the server id (necessary when using mysqli or mysql).

If your variable needs to be a string, then you would use the sanitizeNonNumeric() method. It expects the first argument to be your variable, and has two optional arguments that define the type of cleaning you want PzPHP to execute on your string, as well as the server id (necessary when using mysqli or mysql).

Both methods also accept arrays of values. If you pass either method an array, the method will recursively go through each element in the array, and sanitize it. Multi-dimensional arrays are fully supported.

PzPHP Cache Module

The Cache module is meant to give you easy access to the most commonly used methods when interacting with a caching system. This is where PzPHP tries to make things really simple for you. Using this Cache module, you could start your application off using a caching system such as APC, and when going to production switch to Memcache without any code/logic change in your application.

This is done using the "PZPHP_CACHING_MODE" constant in config.php. You can set whatever caching system you want PzPHP to connect to and use, sit back, and see how your application automatically converts to the new caching system.

The only other thing you need to make sure is correct, is the login credentials (if any, of course).

How to Connect to a Cache Server

Please note that this section is only relevant if you are using a caching system that requires connection/logging in.

If you remember reading from previous chapters, PzPHP wants to make your application as lightweight as possible. This holds true to cache server connections as well. When you add server credential information to PzPHP, it is stored and referred to only when your first read/write/delete is executed. This means if a particular script in your application runs and does not do anything with your cache server; you do not have to worry about an unnecessary connection. At the same time, PzPHP also handles disconnecting from all active cache server connections.

Of course, you have the ability to manually connect and disconnect whenever you want, but PzPHP handles that stuff for you by default.

```
$cacheId = $_PZPHP->cache()->addServer('127.0.0.1', 11211);
```

The above example registers one set of cache server credentials with PzPHP. The returned value is an identifier for this set of credentials. This highlights another feature of the Cache Module; PzPHP allows you to connect to and use as many different cache servers as you need to. It manages this by assigning an Id every time you call the addServer() method.

If you have not changed the default value of "PZ_SETTING_CACHE_AUTO_ASSIGN_ACTIVE_SERVER" in config.php, then by default, the id returned by addServer() is set to the active cache server id.
This means all other read/write/delete related methods in the Cache Module will use this server's credentials (or connection object if a connection has already been established).

Below is an example to clarify all of this:

```php
#returns id 1, and assigns it as active
$cacheId = $_PZPHP->cache()->addServer('127.0.0.1', 11211);

#returns id 2, and assigns it as active
$cacheId2 = $_PZPHP->cache()->addServer('127.0.0.1', 11212);

#connects to cache server #2 ($cacheId2) as it is the current active one.
#once the connection completes, the read is executed.
$result = $_PZPHP->cache()->read_csl('myVarName');

#Let's say we wanted to access a different cache server for just one query, we can override it like so:
$result = $_PZPHP->cache()->read_csl('myVarName', $cacheId);
#this queried cache server located at port 11211

#Let's say we wanted to use the first cache server for the rest of the script
#instead of passing the ID every time, we can change the current active ID like so
$_PZPHP->cache()->setActiveServerId($cacheId);

#this query will now be sent to the first cache server ($cacheId)
$result = $_PZPHP->cache()->read_csl('myVarName');
```

Methods in the Cache module conform to this methodology. For a clearer explanation of the read/write/delete methods, go to the next section.

Read/Write/Delete Methods

If you have not noticed already, read/write/delete methods in the Cache module are suffixed with seemingly random letter. Of course, these letters are not random at all, and carry important meaning in terms of how you code your application's logic.

PzPHP brings you native atomicity in all of its cache methods. This is important, as most caching drivers in PHP do not support atomic environments. Those that do support it in some form do not make it easy. PzPHP has done all of the heavy lifting for you with the Cache module.

The Cache module has 6 methods for reading, writing, and deleting; two for each specific type of action. In this section, we will go through all 6 methods, giving you examples of how you

would use them, and describe how atomicity comes in to play. You do not have to use the atomic methods in the Cache module. They are an optional feature set if you need atomicity in your application.

read_il

The read_il() method is your basic read function for your chosen cache (APC, Memcache, etc…). If you do not require atomicity, then this is the method you should always be using. It will read from your ache using the specified variable name, without checking for a lock on that variable first. This method will also skip the step where it locks the variable.

read_csl

The read_csl() method is the function you will be using often if you require atomicity in your application. When you execute this method, it will first check for a lock on the passed variable name (your key). If a lock is present, the method will wait until the lock is released. Once the lock is released, the method will set a lock, and then retrieve the value (if it exists). Locks should be deleted by you in your application (PzPHP makes this really easy, take a look at the write_dl() method), though there is a default expiry time for locks being set. You can change this default by modifying the "PZ_SETTING_CACHE_LOCK_EXPIRE_TIME" constant in the config.php file.

PzPHP makes sure this entire process is atomic. So even in a case where a particular key is being slammed by different scripts, PzPHP makes sure only one procedure at a time is run on the value. Not only that, but PzPHP will ensure that the value retrieved at the microsecond the lock is removed, is the most up-to-date.

write_ddl

The write_ddl() method is your basic write function for your chosen cache. If you are not using the atomic methods of PzPHP, then you can use this function to save your variable. This function will first attempt to replace an existing key before it tries to create a new one (this is valid for caching systems that support this kind of functionality).

write_dl

The write_dl() method is the function you will be using often if you require atomicity in your application. This function works just like the write_ddl() method, except that this function will also delete a lock on a key if the lock exists. You will understand how this is important later on in the next section.

delete_nlc

The delete_nlc() method is your basic delete function for your chosen cache. If you are not using the atomic methods of PzPHP, then you can use this function to delete keys in your cache.

delete_lc

The delete_lc() method is the function you will be using often if you require atomicity in your application. This function works like the delete_nlc() method, except this function will check to see if the key is locked before trying to delete it. If a lock is present, the function will wait until the lock is removed before it continues with its deletion.

In the next section, we will explore how to use PzPHP's atomic cache methods to make for a more stable application!

Atomic Operations

The concept behind an atomic procedure using your caching system is as follows:

- Read from the cache checking for a lock first, and waiting for a lock to lift if it is present
- Once the lock is lifted (if it was present in the first place), place a lock and then retrieve the value stored in the cache
- Manipulate the value as you see fit in your logic
- Write the value back to the cache, and delete the lock

These 4 steps are made easy for you when using PzPHP. Below is an example of such a transaction:

```
#this will fetch the value form the cache, first checking for a lock
#if a lock exists on this key, the method will wait until the lock expires, and then retrive the value
#a new lock is placed on the key once the previous lock (if any) is released
$myValueFromCache = $_PZPHP->cache()->read_csl('myStoredValue');

#we will pretend our stored value was an array
$myValueFromCache[] = 'a new string stored in this array';

#this method will write the new value to the cache, and then delete the lock
$_PZPHP->cache()->write_dl('myStoredValue', $myValueFromCache);


#lets say we now want to delete this key
#we can use this method to delete the key, so long as there is no lock on it
#if a lock is found, the method will wait until the lock is released before proceding with the delete
$_PZPHP->cache()->delete_lc('myStoredValue');
```

The above example shows how easy it is to deal with a cache value in an atomic way.

Please note that the Local Cache caching system does not support atomic transactions for obvious reasons. Therefore, using atomic specific cache methods in PzPHP will act like non-atomic cache methods. This helps keep your application caching system agonstic.

Caching Systems

Below is a list of caching systems supported by PzPHP.

Local Cache
Local Cache is just that…it exists for the life of the current script execution, and does not carry over to any other instance. The point of using Local Cache is a temporary solution if your development setup does not support the official caching system you will be using (like APC or Memcache).

Shared Memory
The Shared Memory Cache uses PHP's Shmop functions to emulate a more established caching system. Although we do not recommend using this type of caching system in production; there are instances where using a Shared Memory Cache is the ideal, or only solution for your application. PzPHP handles everything behind the scenes, including calculating the size of memory blocks to create for your variables, etc… Unlike the Local Cache, Shared Memory Cache does propagate to other instances of your application.

APC
APC is now a built-in caching system in the PHP language. If you are using PHP 5.3+, then you most likely have APC installed by default. It is not the most performant caching system, but it does the job well for most low-medium traffic websites.

Memcache & Memcached
PzPHP also supports both Memcache and Memcached modules for PHP. The Memcache caching system is the most popular caching system on the web today. It is used by many large-scale websites, including Facebook. If performance is what you are after, then Memcache is the caching system you should be using. You need to make sure your web server has Memcache installed, before using the Memcache functions in PzPHP.

The Memcache module was the first installment of the caching system in PHP. The Memcached module is the more up-to-date version, and you should be using it if possible.

PzPHP Security Module

The Security Module seems simple at first glance. It is a small collection of methods that wrap around methods in the Pz Library. What makes the Security Module so powerful is that it leverages the unique hashing algorithms inside the Pz Library. The following sections will describe how the hashing and encryption methods in the Pz Library work, and how easy it is for you to incorporate them into your application using PzPHP.

Pz_Crypt

Pz_Crypt has powerful one-way and two-way hashing/encryption methods. They allow for a completely unique way to protect data. Generally speaking, md5 or sha1 algorithms are used with most applications. Although they generally do the job, they are very common, and crackers have found ways to crack the hashes these algorithms produce.

There are best practices for using md5 or sha1. Pz_Crypt takes these practices to a new level. Not only does it use pre-set salts, multi-level re-hashing, and the option to add an additional salt. It also provides a custom strong Caesar Encryption algorithm, in conjunction with hash poisoning.

Pz's Caesar Encryption
The Pz Library's Caesar Encryption is stronger than most implementations due to two major factors: it is based on 238 characters (whereas traditional implementations are based on a small set), and it can be uniquely generated for every installation of PzPHP.

What this means is that even if the default Caesar encryption is cracked; you can easily generate your own Caesar encryption for your particular installation of PzPHP. This functionality is built-in to the Pz Library. You just need to call one method to get a new encryption set. In the following section, we will go over how you can customize Pz_Crypt to make it absolutely unique for your application.

Note that Caesar Encryption is used when you are using two-way encryption only.

Pz's Hash Poisoning
Pz_Crypt also has the ability to poison hashes is produces. This makes it nearly impossible for someone to reverse a one-way or two-way encryption without knowing where the poisning is taking place. And since poisoning happens in several locations in a hashed string, this level of security will turn-off most decryption/reversal attempts.

The Pz Library has preset rules for poisoning, but they can be regenerated for your particular application, making them unique. This ability is covered in the following section.

Customize Pz_Crypt for Your Application

There are 4 aspects you can customize in Pz_Crypt to make it unique for your application. If security is a very high priority for your application, then it is recommended that you customize all 4 aspects of Pz_Crypt before going to production.

You have the ability to regenerate the Caesar Encryption that Pz_Crypt employs, allowing you to use a unique two-way encryption.

You also have the ability to change the base salt, and hashing depth Pz_Crypt uses for its one-way encryption method.

Finally, you can regenerate the poison constraints Pz_Crypt uses for both one-way and two-way encryption methods.

You can store these customized values in your config.php file. This file has 4 constants that reflect the 4 aspects of Pz_Crypt you can customize:

- PZ_SECURITY_HASH_TABLE
- PZ_SECURITY_SALT
- PZ_SECURITY_POISON_CONSTRAINTS
- PZ_SECURITY_REHASH_DEPTH

Pz_Crypt has 4 regeneration methods which each generate unique values for the aforementioned aspects. You only need to run each method once, and copy its output into your config.php file.

Below are examples for each method:

```php
#The following is how you set a custom Caesar table
#You will copy the returned value in PZ_SECURITY_HASH_TABLE constant in config.php
echo serialize($_PZPHP->pz()->pzSecurity()->regeneratePzCryptHash());

#The following is how you set a custom poison constraints
#You will copy the returned value in PZ_SECURITY_POISON_CONSTRAINTS constant in config.php
echo serialize($_PZPHP->pz()->pzSecurity()->regeneratePzCryptPoisonConstraints());

#The following is how you set a new base salt for Pz_Crypt's one-way hashing method
#You will copy the returned value in PZ_SECURITY_SALT constant in config.php
echo $_PZPHP->pz()->pzSecurity()->regeneratePzCryptSalt();

#To set your own rehash depth, simply put an int value for the PZ_SECURITY_REHASH_DEPTH constant
```

Keep in mind that the above procedure should be a one-time thing (and it is optional).

How to Protect Your Data

Now that you know what Pz_Crypt has to offer, and how to make its security features more unique for your application. This section will show you examples of how to actually utilize one-way and two-way hashing/encryption in your application using PzPHP.

Below are some examples:

```php
$myPassword = '123456!';
$hashedAndPoisoned = $_PZPHP->security()->oneWayEncrypt($myPassword);
#the above will produce a 44 character long hash that will look something like this:
# cc01733488a97ab978a473f59763fa039402e3941926
#using the default poison constraints, this method will always produce a 44 character long hash
```

```php
#you can add your own custom salt, as well as other options by overriding the secuirty module's oneWayEncrypt method
$myPassword = '123456!';
$hashedAndPoisoned = $_PZPHP->pz()->pzSecurity()->encrypt(
    $myPassword,
    array(
        Pz_Security::ONE_WAY,
        Pz_Security::STRICT
    ),
    array(
        Pz_Security::UNIQUE_SALT => 'some_salt_you_choose'
    )
);
#let us examine the above
# Pz_Security::ONE_WAY tells the encrypt() method to produce an irreversable one-way hash
# Pz_Security::STRICT tells the encrypt() method to produce a valid length string for the chosen hashing algorithm
# Pz_Security::UNIQUE_SALT will be added to the $myPassword string before it is hashed
#The above will produce a 32 character long hash that will look something lile this:
# b7d8c61e9c41b374bfb396d40ff4813d
```

```php
#you can compare input to a hashed value like so:
$matched = $_PZPHP->security()->oneWayHashComparison('123456789', $hashedAndPoisoned);
#the above will return false. Note that PzPHP handles de-poisoning for you.
```

```php
$secret = 'This is a secret I will need to know later.';
$reversableCaesarEncryptionWithPoisoning = $_PZPHP->security()->twoWayEncrypt($secret);
#the above will generate a strong encryption for the $secret value. It will look something like this:
# K4f0efÅ¡effceÅ¡1boÅ¡f2n92c04Å¡GÅ¡6eff3ddÅ¡7225Å¡4acÅ¡m7a6Å¡do429,
#the length of this encryption will depend on the length of your $secret variable
#to decrypt this encryption:
$secretRevealed = $_PZPHP->security()->twoWayDecrypt($reversableCaesarEncryptionWithPoisoning);
#the above will reveal the original value: This is a secret I will need to know later.

#you can compare input to an encryption value like so:
$matched = $_PZPHP->security()->twoWayHashComparison($secret, $reversableCaesarEncryptionWithPoisoning);
#the above will return true. Note that PzPHP handles de-poisoning for you.
```

PzPHP Locale Module

The Locale Module allows you to support multiple languages in your application. It is easy to setup and even easier to use.

Setting-up Locale Files

To set up a locale file, first you must create a directory in the root of PzPHP called “Resources”. Within that directory, create another called “translations”. This directory is configured in the PZPHP_TRANSLATIONS_DIR constant. You can change this value to anything you choose.

Once the directory structure is created, place a new PHP file inside the “translations” directory. Call it en.php.

Inside en.php, create an empty array called $translations. PzPHP expects you to provide key/value pairs in this array for use with the translate() method.

Using Internationalization in Your Application

Next, you need to define which languages your application will support. You should do this in your bootstrap.php file.

```
   #the following two lines will tell PzPHP you support english and french.
   #the first value for each method needs to be the long form language identifier
   #the second value for each method needs to be the short form language identifier
   $_PZPHP->locale()->addLanguage('en_us', 'en');
   $_PZPHP->locale()->addLanguage('fr_ca', 'fr');

   #from this point forth, whenever you provide PzPHP with a language identifier, you can provide either long or short
form
   #PzPHP will convert your input as it needs to. You will see this in the following examples.

   #right now, you should set the default language
   $_PZPHP->locale()->setLocale('en');

   #this also works the same way
   $_PZPHP->locale()->setLocale('en_us');

   #so does this
   $_PZPHP->locale()->setLocale('eN_Us');

   #you now need to tell PzPHP to load the translation file (en.php)
   $_PZPHP->locale()->loadTranslationSet();
```

```
#assuming you have populated the $translations array in your en.php file, you can display the values like so:
echo $_PZPHP->locale()->translate('my.homepage.header.text');
```

```php
#this means your $translations array in en.php looks like this:
$translations = array(
    'my.homepage.header.text' => 'Welcome to my website!'
);
```

```php
#you can also put placeholders in your strings for dynamic data
echo $_PZPHP->locale()->translate('dynamic.welcome.text', array('name' => 'David'));
```

```php
#this means your $translations array in en.php looks like this:
$translations = array(
    'my.homepage.header.text' => 'Welcome to my website!',
    'dynamic.welcome.text' => 'Hi %name%! Enjoy my website!'
);
```

The above code blocks demonstrate how easy it is to use PzPHP's Locale Module! There is nothing left to it.

Class Naming

Assuming you will be using Pz Library's autoloader, you will need to make sure you name your classes appropriately.

Pz Library has adopted the Pear Naming Conventions for classes. This means underscores _ denote a sub directory up until the last piece of text in your class name. For example a class called My_ClassName_Is_Really_Long would be located in the following directory structure: My/ClassName/Is/Really/Long.php

This Was Just the Beginning

This document was a brief introduction to using PzPHP. It contains enough examples and explanation to allow you to start building your application. Using any respectable IDE, you will further discover hidden powers in PzPHP's modules and their methods.

Goodluck and happy coding! ☺