

CodeIgniter4 User Guide

- [Welcome to CodeIgniter4](#)
 - [Welcome to CodeIgniter4](#)
 - [Server Requirements](#)
 - [Credits](#)
 - [PSR Compliance](#)

Getting Started

- [Installation](#)
 - [Manual Installation](#)
 - [Composer Installation](#)
 - [Git Installation](#)
 - [Running Your App](#)
 - [Upgrading From a Previous Version](#)
 - [Troubleshooting](#)
 - [CodeIgniter Repositories](#)
- [Tutorial](#)
 - [Static pages](#)
 - [News section](#)
 - [Create news items](#)
 - [Conclusion](#)

Overview & General Topics

- [CodeIgniter4 Overview](#)
 - [Application Structure](#)
 - [Models, Views, and Controllers](#)
 - [Autoloading Files](#)
 - [Services](#)
 - [Working With HTTP Requests](#)

- [Security Guidelines](#)
- [General Topics](#)
 - [Working With Configuration Files](#)
 - [CodeIgniter URLs](#)
 - [Helper Functions](#)
 - [Global Functions and Constants](#)
 - [Logging Information](#)
 - [Error Handling](#)
 - [Web Page Caching](#)
 - [Code Modules](#)
 - [Managing your Applications](#)
 - [Handling Multiple Environments](#)

Request Handling

- [Controllers and Routing](#)
 - [Controllers](#)
 - [URI Routing](#)
 - [Controller Filters](#)
 - [HTTP Messages](#)
 - [Request Class](#)
 - [IncomingRequest Class](#)
 - [Content Negotiation](#)
 - [HTTP Method Spoofing](#)
- [Building Responses](#)
 - [Views](#)
 - [View Cells](#)
 - [View Renderer](#)
 - [View Layouts](#)
 - [View Parser](#)
 - [HTTP Responses](#)
 - [API Response Trait](#)
 - [Localization](#)
 - [Alternate PHP Syntax for View Files](#)

Handling Databases

- [Working With Databases](#)
 - [Quick Start: Usage Examples](#)
 - [Database Configuration](#)
 - [Connecting to a Database](#)
 - [Running Queries](#)
 - [Generating Query Results](#)
 - [Query Helper Functions](#)
 - [Query Builder Class](#)
 - [Transactions](#)
 - [Getting MetaData](#)
 - [Custom Function Calls](#)
 - [Database Events](#)
 - [Database Utilities](#)
- [Modeling Data](#)
 - [Using CodeIgniter's Model](#)
 - [Using Entity Classes](#)
- [Managing Databases](#)
 - [Database Manipulation with Database Forge](#)
 - [Database Migrations](#)
 - [Database Seeding](#)

Libraries & Helpers

- [Library Reference](#)
 - [Caching Driver](#)
 - [CURLRequest Class](#)
 - [Working with Files](#)
 - [Honeypot Class](#)
 - [Image Manipulation Class](#)
 - [Pagination](#)
 - [Security Class](#)
 - [Session Library](#)

- [Throttler](#)
- [Dates and Times](#)
- [Typography](#)
- [Working with Uploaded Files](#)
- [Working with URIs](#)
- [User Agent Class](#)
- [Validation](#)
- [Helpers](#)
 - [Array Helper](#)
 - [Cookie Helper](#)
 - [Date Helper](#)
 - [Filesystem Helper](#)
 - [Form Helper](#)
 - [HTML Helper](#)
 - [Inflector Helper](#)
 - [Number Helper](#)
 - [Security Helper](#)
 - [Text Helper](#)
 - [URL Helper](#)
 - [XML Helper](#)

Advanced Topics

- [Testing](#)
 - [Getting Started](#)
 - [Database](#)
 - [Controller Testing](#)
 - [HTTP Testing](#)
 - [Benchmarking](#)
 - [Debugging Your Application](#)
- [Command Line Usage](#)
 - [Running via the Command Line](#)
 - [Custom CLI Commands](#)
 - [CLI Library](#)
 - [CLIRequest Class](#)

- [Extending CodeIgniter](#)
 - [Creating Core System Classes](#)
 - [Events](#)
 - [Contributing to CodeIgniter](#)

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Welcome to CodeIgniter4

CodeIgniter is an Application Development Framework - a toolkit - for people who build web sites using PHP. Its goal is to enable you to develop projects much faster than you could if you were writing code from scratch, by providing a rich set of libraries for commonly needed tasks, as well as a simple interface and logical structure to access these libraries. CodeIgniter lets you creatively focus on your project by minimizing the amount of code needed for a given task.

Where possible, CodeIgniter has been kept as flexible as possible, allowing you to work in the way you want, not being forced into working any certain way. The framework can have core parts easily extended or completely replaced to make the system work the way you need it to. In short, CodeIgniter is the malleable framework that tries to provide the tools you need while staying out of the way.

Who is CodeIgniter For?

CodeIgniter is right for you if:

- You want a framework with a small footprint.
- You need exceptional performance.
- You want a framework that requires nearly zero configuration.
- You want a framework that does not require you to use the command line.
- You want a framework that does not require you to adhere to restrictive coding rules.
- You are not interested in large-scale monolithic libraries like PEAR.
- You do not want to be forced to learn a templating language (although a template parser is optionally available if you desire one).
- You eschew complexity, favoring simple solutions.
- You need clear, thorough documentation.

Server Requirements

[PHP](#) [<http://php.net/>] version 7.2 or newer is required, with the [*intl* extension](#) [<http://php.net/manual/en/intl.requirements.php>] installed.

The following PHP extensions should be enabled on your server: `php-json`, `php-mbstring`, `php-mysqlnd`, `php-xml`

In order to use the [CURLRequest](#), you will need [libcurl](#) [<http://php.net/manual/en/curl.requirements.php>] installed.

A database is required for most web application programming. Currently supported databases are:

- MySQL (5.1+) via the *MySQLi* driver
- PostgreSQL via the *Postgre* driver
- SQLite3 via the *SQLite3* driver

Not all of the drivers have been converted/rewritten for CodeIgniter4. The list below shows the outstanding ones.

- MySQL (5.1+) via the *pdo* driver
- Oracle via the *oci8* and *pdo* drivers
- PostgreSQL via the *pdo* driver
- MS SQL via the *mssql*, *sqlsrv* (version 2005 and above only) and *pdo* drivers
- SQLite via the *sqlite* (version 2) and *pdo* drivers
- CUBRID via the *cubrid* and *pdo* drivers
- Interbase/Firebird via the *ibase* and *pdo* drivers
- ODBC via the *odbc* and *pdo* drivers (you should know that ODBC is actually an abstraction layer)

Credits

CodeIgniter was originally developed by [EllisLab](https://ellislab.com/). The framework was written for performance in the real world, with many of the original class libraries, helpers, and sub-systems borrowed from the code-base of [ExpressionEngine](https://ellislab.com/expressionengine). It was, for years, developed and maintained by EllisLab, the ExpressionEngine Development Team and a group of community members called the Reactor Team.

In 2014, CodeIgniter was acquired by the [British Columbia Institute of Technology](http://www.bcit.ca/) and was then officially announced as a community-maintained project.

Bleeding edge development is spearheaded by the handpicked contributors of the CodeIgniter Council.

PSR Compliance

The [PHP-FIG](http://www.php-fig.org/) [http://www.php-fig.org/] was created in 2009 to help make code more interoperable between frameworks by ratifying Interfaces, style guides, and more that members were free to implement or not. While CodeIgniter is not a member of the FIG, we are compatible with a number of their proposals. This guide is meant to list the status of our compliance with the various accepted, and some draft, proposals.

PSR-1: Basic Coding Standard

This recommendation covers basic class, method, and file-naming standards. Our [style guide](#)

[https://github.com/codeigniter4/CodeIgniter4/blob/develop/contributing/styleguide.rst] meets PSR-1 and adds its own requirements on top of it.

PSR-2: Coding Style Guide

This PSR was fairly controversial when it first came out. CodeIgniter meets many of the recommendations within, but does not, and will not, meet all of them.

PSR-3: Logger Interface

CodeIgniter's [Logger](#) implements all of the interfaces provided by this PSR.

PSR-4: Autoloading Standard

This PSR provides a method for organizing file and namespaces to allow for a standard method of autoloading classes. Our [Autoloader](#) meets the PSR-4 recommendations.

PSR-6: Caching Interface

CodeIgniter will not be trying to meet this PSR, as we believe it oversteps its needs. The newly proposed [SimpleCache Interfaces](https://github.com/dragonis/fig-standards/blob/psr-simplecache/proposed/simplecache.md) [https://github.com/dragonis/fig-standards/blob/psr-simplecache/proposed/simplecache.md] do look like something we would consider.

PSR-7: HTTP Message Interface

This PSR standardizes a way of representing the HTTP interactions. While many of the concepts became part of our HTTP layer, CodeIgniter does not strive for compatibility with this recommendation.

—

If you find any places that we claim to meet a PSR but have failed to execute it correctly, please let us know and we will get it fixed, or submit a pull request with the required changes.

Installation

CodeIgniter4 can be installed in a number of different ways: manually, using [Composer](https://getcomposer.org/) [https://getcomposer.org/], or using [Git](https://git-scm.com/) [https://git-scm.com/]. Which is right for you?

- If you would like the simple “download & go” install that CodeIgniter3 is known for, choose the manual installation.
- If you plan to add third party packages to your project, we recommend the Composer installation.
- If you are thinking of contributing to the framework, then the Git installation is right for you.

- [Manual Installation](#)
- [Composer Installation](#)
- [Git Installation](#)
- [Running Your App](#)
- [Upgrading From a Previous Version](#)
 - [Upgrading from 3.x to 4.x](#)
- [Troubleshooting](#)
- [CodeIgniter Repositories](#)

However you choose to install and run CodeIgniter4, the [user guide](https://codeigniter4.github.io/userguide/) [https://codeigniter4.github.io/userguide/] is accessible online.

Note

Before using CodeIgniter 4, make sure that your server meets the [requirements](#), in particular the PHP version and the PHP extensions that are needed. You may find that you have to uncomment the `php.ini` “extension” lines to enable “curl” and “intl”, for instance.

Manual Installation

The [CodeIgniter 4 framework](https://github.com/codeigniter4/framework) repository holds the released versions of the framework. It is intended for developers who do not wish to use Composer.

Develop your app inside the `app` folder, and the `public` folder will be your public-facing document root. Do not change anything inside the `system` folder!

Note: This is the installation technique closest to that described for [CodeIgniter 3](https://www.codeigniter.com/user_guide/installation/index.html).

Installation

Download the [latest version](https://github.com/CodeIgniter4/framework/releases/latest), and extract it to become your project root.

Setup

None

Upgrading

Download a new copy of the framework, and then follow the upgrade instructions in the release notice or changelog to merge that with your project.

Typically, you replace the `system` folder, and check designated `app/Config` folders for affected changes.

Pros

Download and run

Cons

You are responsible for merge conflicts when updating

Structure

Folders in your project after setup: app, public, system, writable

Translations Installation

If you want to take advantage of the system message translations, they can be added to your project in a similar fashion.

Download the [latest version of them](https://github.com/codeigniter4/translations/releases/latest)

[<https://github.com/codeigniter4/translations/releases/latest>]. Extract the downloaded zip, and copy the Language folder contents in it to your PROJECT_ROOT/app/Languages folder.

This would need to be repeated to incorporate any updates to the translations.

Composer Installation

- [App Starter](#)
- [Dev Starter](#)
- [Adding CodeIgniter4 to an Existing Project](#)
- [Translations Installation](#)

Composer can be used in several ways to install CodeIgniter4 on your system.

The first two techniques describe creating a skeleton project using CodeIgniter4, that you would then use as the base for a new webapp. The third technique described below lets you add CodeIgniter4 to an existing webapp,

Note: if you are using a Git repository to store your code, or for collaboration with others, then the vendor folder would normally be “git ignored”. In such a case, you will need to do a composer update when you clone the repository to a new system.

[App Starter](#)

The [CodeIgniter 4 app starter](#) [https://github.com/codeigniter4/appstarter] repository holds a skeleton application, with a composer dependency on the latest released version of the framework.

This installation technique would suit a developer who wishes to start a new CodeIgniter4 based project.

Installation

In the folder above your project root:

```
composer create-project codeigniter4/appstarter -s beta
```

Setup

The command above will create an “appstarter” folder. Feel free to rename that for your project.

Upgrading

Whenever there is a new release, then from the command line in your project root:

```
composer update
```

Read the upgrade instructions, and check designated app/Config folders for affected changes.

Pros

Simple installation; easy to update

Cons

You still need to check for app/Config changes after updating

Structure

Folders in your project after setup:

- app, public, writable
- vendor/codeigniter4/framework/system
- vendor/codeigniter4/codeigniter4/app & public (compare with yours after updating)

[Dev Starter](#)

Installation

The [CodeIgniter 4 dev starter](https://github.com/codeigniter4/devstarter) repository holds a skeleton application, just like the appstarter above, but with a composer dependency on the develop branch (unreleased) of the framework. It can be composer-installed as described here.

This installation technique would suit a developer who wishes to start a new CodeIgniter4 based project, and who is willing to live with the latest unreleased changes, which may be unstable.

The [development user guide](https://codeigniter4.github.io/CodeIgniter4/) is accessible online. Note that this differs from the released user guide, and will pertain to the develop branch explicitly.

In the folder above your project root:

```
composer create-project codeigniter4/devstarter -s dev
```

Setup

The command above will create an “devstarter” folder. Feel free to rename that for your project.

Upgrading

`composer update` whenever you are ready for the latest changes.

Check the changelog to see if any recent changes affect your app, bearing in mind that the most recent changes may not have made it into the changelog!

Pros

Simple installation; easy to update; bleeding edge version

Cons

This is not guaranteed to be stable; the onus is on you to upgrade. You still need to check for app/Config changes after updating.

Structure

Folders in your project after setup:

- app, public, writable
- vendor/codeigniter4/codeigniter4/system
- vendor/codeigniter4/codeigniter4/app & public (compare with yours after updating)

Adding CodeIgniter4 to an Existing Project

The same [CodeIgniter 4 framework](https://github.com/codeigniter4/framework) [https://github.com/codeigniter4/framework] repository described in “Manual Installation” can also be added to an existing project using Composer.

Develop your app inside the app folder, and the public folder will be your document root.

In your project root:

```
composer require codeigniter4/framework @beta
```

Setup

Copy the app, public and writable folders from vendor/codeigniter4/framework to your project root

Copy the env, phpunit.xml.dist and spark files, from vendor/codeigniter4/framework to your project root

You will have to adjust paths to refer to vendor/codeigniter/framework``, - the \$systemDirectory variable in app/Config/Paths.php

Upgrading

Whenever there is a new release, then from the command line in your project root:

`composer update`

Read the upgrade instructions, and check designated app/Config folders for affected changes.

Pros

Relatively simple installation; easy to update

Cons

You still need to check for app/Config changes after updating

Structure

Folders in your project after setup:

- app, public, writable
- vendor/codeigniter4/framework/system

Translations Installation

If you want to take advantage of the system message translations, they can be added to your project in a similar fashion.

From the command line inside your project root:

```
composer require codeigniter4/translations @beta
```

These will be updated along with the framework whenever you do a `composer update`.

Git Installation

This would *not* be suitable for app development, but *is* suitable for contributing to the framework.

Installation

Install the latest version of the codebase by

- forking the [codebase](https://github.com/codeigniter4/CodeIgniter4) [https://github.com/codeigniter4/CodeIgniter4] to your github account
- cloning **your** forked repository locally

Setup

The command above will create a “CodeIgniter4” folder. Feel free to rename that as you see fit.

You will want to setup a remote repository alias, so you can synchronize your repository with the main one:

```
git remote add upstream https://github.com/codeigniter4/CodeIgnit
```

Copy the provided env file to .env, and use that for your git-ignored configuration settings,

Copy the provided phpunit.xml.dist to phpunit.xml and tailor it as needed, if you want custom unit testing for the framework.

Upgrading

Update your code anytime:

```
git checkout develop
git pull upstream develop
git push origin develop
```

Merge conflicts may arise when you pull from “upstream”. You will need to resolve them locally.

Pros

- You have the latest version of the codebase (unreleased)
- You can propose contributions to the framework, by creating a feature branch and submitting a pull request for it to the main repo
- a pre-commit hook is installed for your repo, that binds it to the coding-standard we use

Cons

You need to resolve merge conflicts when you synch with the repo.

You would not use this technique for app development.

Structure

Folders in your project after setup:

- app, public, system, tests, user_guide_src, writable

Translations Installation

If you wish to contribute to the system message translations, then fork and clone the [translations repository](https://github.com/codeigniter4/translations) [https://github.com/codeigniter4/translations] separately from the codebase. These are two independent repositories!

Coding Standards Installation

This is bound and installed automatically as part of the codebase installation.

If you wish to use it inside your project too, composer require
codeigniter4/translations @beta

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01,
2019. Created using [Sphinx](#) 1.4.5.

Running Your App

- [Initial Configuration & Setup](#)
- [Local Development Server](#)
- [Hosting with Apache](#)
- [Hosting with Vagrant](#)

A CodeIgniter 4 app can be run in a number of different ways: hosted on a web server, using virtualization, or using CodeIgniter’s command line tool for testing. This section addresses how to use each technique, and explains some of the pros and cons of them.

If you’re new to CodeIgniter, please read the [Getting Started](#) section of the User Guide to begin learning how to build dynamic PHP applications. Enjoy!

[Initial Configuration & Setup](#)

1. Open the **app/Config/App.php** file with a text editor and set your base URL. If you need more flexibility, the baseURL may be set within the .env file as **app.baseURL="http://example.com"**.
2. If you intend to use a database, open the **app/Config/Database.php** file with a text editor and set your database settings. Alternately, these could be set in your .env file.

One additional measure to take in production environments is to disable PHP error reporting and any other development-only functionality. In CodeIgniter, this can be done by setting the `ENVIRONMENT` constant, which is more fully described on the [environments page](#). By default, the application will run using the “production” environment. To take advantage of the debugging tools provided, you should set the environment to “develop”.

Local Development Server

CodeIgniter 4 comes with a local development server, leveraging PHP's built-in web server with CodeIgniter routing. You can use the serve script to launch it, with the following command line in the main directory:

```
php spark serve
```

This will launch the server and you can now view your application in your browser at <http://localhost:8080>.

Note

The built-in development server should only be used on local development machines. It should NEVER be used on a production server.

If you need to run the site on a host other than simply localhost, you'll first need to add the host to your hosts file. The exact location of the file varies in each of the main operating systems, though all unix-type systems (include OS X) will typically keep the file at **/etc/hosts**.

The local development server can be customized with three command line options:

- You can use the `--host` CLI option to specify a different host to run the application at:

```
php spark serve --host=example.dev
```

- By default, the server runs on port 8080 but you might have more than one site running, or already have another application using that port. You can use the `--port` CLI option to specify a different one:

```
php spark serve --port=8081
```

- You can also specify a specific version of PHP to use, with the `--php` CLI option, with its value set to the path of the PHP executable you want to use:


```
php spark serve --php=/usr/bin/php7.6.5.4
```

Hosting with Apache

A CodeIgniter4 webapp is normally hosted on a web server. Apache's httpd is the “standard” platform, and assumed in much of our documentation.

Apache is bundled with many platforms, but can also be downloaded in a bundle with a database engine and PHP from [Bitnami] (<https://bitnami.com/stacks/infrastructure>).

.htaccess

The “mod_rewrite” module enables URLs without “index.php” in them, and is assumed in our user guide.

Make sure that the rewrite module is enabled (uncommented) in the main configuration file, eg. apache2/conf/httpd.conf:

```
LoadModule rewrite_module modules/mod_rewrite.so
```

Also make sure that the default document root's <Directory> element enables this too, in the “AllowOverride” setting:

```
<Directory "/opt/lamp7.2/apache2/htdocs">  
    Options Indexes FollowSymLinks  
    AllowOverride All  
    Require all granted  
</Directory>
```

Virtual Hosting

We recommend using “virtual hosting” to run your apps. You can setup different aliases for each of the apps you work on,

Make sure that the virtual hosting module is enabled (uncommented) in the main configuration file, eg. apache2/conf/httpd.conf:

```
LoadModule vhost_alias_module modules/mod_vhost_alias.so
```

Add a host alias in your “hosts” file, typically `/etc/hosts` on unix-type platforms, or `c:/Windows/System32/drivers/etc/hosts` on Windows. Add a line to the file. This could be “myproject.local” or “myproject.test”, for instance:

```
127.0.0.1 myproject.local
```

Add a `<VirtualHost>` element for your webapp inside the virtual hosting configuration, eg. `apache2/conf/extra/httpd-vhost.conf`:

```
<VirtualHost *:80>
    DocumentRoot "/opt/lamp7.2/apache2/htdocs/myproject/public"
    ServerName myproject.local
    ErrorLog "logs/myproject-error_log"
    CustomLog "logs/myproject-access_log" common
</VirtualHost>
```

If your project folder is not a subfolder of the Apache document root, then your `<VirtualHost>` element may need a nested `<Directory>` element to grant the web server access to the files.

Testing

With the above configuration, your webapp would be accessed with the URL `http://myproject.local` in your browser.

Apache needs to be restarted whenever you change its configuration.

Hosting with Vagrant

Virtualization is an effective way to test your webapp in the environment you plan to deploy on, even if you develop on a different one. Even if you are using the same platform for both, virtualization provides an isolated environment for testing.

The codebase comes with a `vagrantFile.dist`, that can be copied to `vagrantFile` and tailored for your system, for instance enabling access to specific database or caching engines.

Setup

It assumes that you have installed [VirtualBox](https://www.virtualbox.org/wiki/Downloads) [https://www.virtualbox.org/wiki/Downloads] and [Vagrant](https://www.vagrantup.com/downloads.html) [https://www.vagrantup.com/downloads.html] for your platform.

The Vagrant configuration file assumes you have the [ubuntu/bionic64 Vagrant box](https://app.vagrantup.com/ubuntu/boxes/bionic64) [https://app.vagrantup.com/ubuntu/boxes/bionic64] setup on your system:

```
vagrant box add ubuntu/bionic64
```

Testing

Once setup, you can then launch your webapp inside a VM, with the command:

```
vagrant up
```

Your webapp will be accessible at `http://localhost:8080`, with the code coverage report for your build at `http://localhost:8081` and the user guide for it at `http://localhost:8082`.

Upgrading From a Previous Version

Please read the upgrade notes corresponding to the version you are upgrading from.

- [Upgrading from 3.x to 4.x](#)

Upgrading from 3.x to 4.x

CodeIgniter 4 is a rewrite of the framework, and is not backwards compatible. It is more appropriate to think of converting your app, rather than upgrading it. Once you have done that, upgrading from one version of CodeIgniter 4 to the next will be straightforward.

The “lean, mean and simple” philosophy has been retained, but the implementation has a lot of differences, compared to CodeIgniter 3.

There is no 12-step checklist for upgrading. Instead, start with a copy of CodeIgniter 4 in a new project folder, [however you wish to install and use it](#), and then convert and integrate your app components. We’ll try to point out the most important considerations here.

Not all of the CI3 libraries have been ported or rewritten for CI4! See the threads in the [CodeIgniter 4 Roadmap](#) [https://forum.codeigniter.com/forum-33.html] subforum for an uptodate list!

Do read the user guide before embarking on a project conversion!

Downloads

- CI4 is still available as a ready-to-run zip or tarball, which includes the user guide (though in the *docs* subfolder)
- It can also be installed using Composer

Namespaces

- CI4 is built for PHP7.2+, and everything in the framework is namespaced, except for the helpers.

Application Structure

- The framework still has app and system folders, with the same interpretation as before
- The framework now provides for a public folder, intended as the document root for your app
- There is also a writable folder, to hold cache data, logs, and session data
- The application folder looks very similar to that for CI3, with some name changes, and some subfolders moved to the writable folder
- There is no longer a nested application/core folder, as we have a different mechanism for extending framework components (see below)

Class loading

- There is no longer a CodeIgniter “superobject”, with framework component references magically injected as properties of your controller
- Classes are instantiated where needed, and components are managed by Services
- The class loader automatically handles PSR4 style class locating, within the App (application) and CodeIgniter (i.e. system) top level namespaces; with composer autoloading support, and even using educated guessing to find your models and libraries if they are in the right folder even though not namespaced
- You can configure the class loading to support whatever application structure you are most comfortable with, including the “HMVC” style

Controllers

- Controllers extend \CodeIgniter\Controller instead of CI_Controller
- They don’t use a constructor any more (to invoke CI “magic”) unless that is part of a base controller you make
- CI provides Request and Response objects for you to work with - more powerful than the CI3-way
- If you want a base controller (MY_Controller in CI3), make it where you like, e.g. BaseController extends Controller, and then have your controllers extend it

Models

- Models extend `\CodeIgniter\Model` instead of `CI_Model`
- The CI4 model has much more functionality, including automatic database connection, basic CRUD, in-model validation, and automatic pagination
- CI4 also has the `Entity` class you can build on, for richer data mapping to your database tables
- Instead of CI3's `$this->load->model(x);`, you would now use `$this->x = new X();`, following namespaced conventions for your component

Views

- Your views look much like before, but they are invoked differently ... instead of CI3's `$this->load->view(x);` you can use `echo view(x);`
- CI4 supports view “cells”, to build your response in pieces
- The template parser is still there, but substantially enhanced

Libraries

- Your app classes can still go inside `app/Libraries`, but they don't have to
- Instead of CI3's `$this->load->library(x);` you can now use `$this->x = new X();`, following namespaced conventions for your component

Helpers

- Helpers are pretty much the same as before, though some have been simplified

Extending the framework

- You don't need a `core` folder to hold `MY_...` framework component extensions or replacements
- You don't need `MY_x` classes inside your `libraries` folder to extend or replace CI4 pieces
- Make any such classes where you like, and add appropriate service methods in `app/Config/Services.php` to load your components instead of the default ones

Troubleshooting

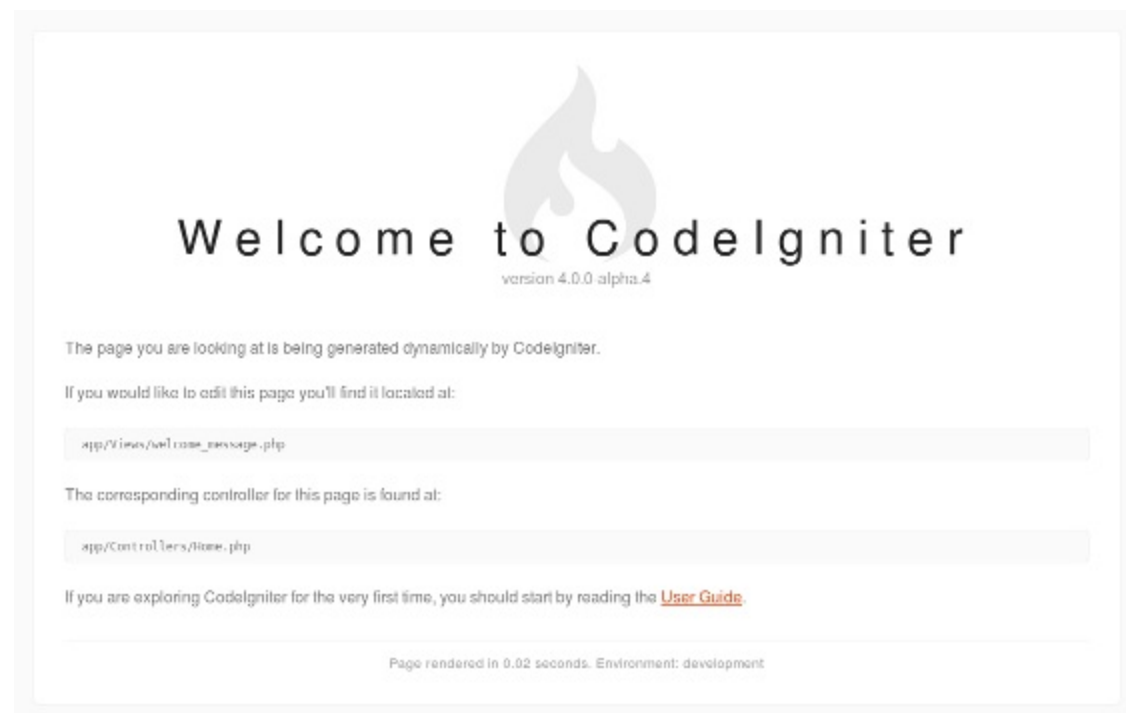
Here are some common installation problems, and suggested workarounds.

How do I know if my install is working?

From the command line, at your project root:

```
php spark serve
```

`http://localhost:8080` in your browser should then show the default welcome page:



I have to include index.php in my URL

If a URL like `/mypage/find/apple` doesn't work, but the similar URL `/index.php/mypage/find/apple` does, that sounds like your `.htaccess` rules

(for Apache) are not setup properly, or the `mod_rewrite` extension in Apache's `httpd.conf` is commented out.

Only the default page loads

If you find that no matter what you put in your URL only your default page is loading, it might be that your server does not support the `REQUEST_URI` variable needed to serve search-engine friendly URLs. As a first step, open your `app/Config/App.php` file and look for the URI Protocol information. It will recommend that you try a couple of alternate settings. If it still doesn't work after you've tried this you'll need to force CodeIgniter to add a question mark to your URLs. To do this open your `app/Config/App.php` file and change this:

```
public $indexPage = 'index.php';
```

To this:

```
public $indexPage = 'index.php?';
```

The tutorial gives 404 errors everywhere :(

You can't follow the tutorial using PHP's built-in web server. It doesn't process the `.htaccess` file needed to route requests properly.

The solution: use Apache to server your site, or else the built-in CodeIgniter equivalent, `php spark serve` from your project root.

CodeIgniter Repositories

The CodeIgniter 4 open source project has its own [Github organization](https://github.com/codeigniter4) [https://github.com/codeigniter4].

There are several development repositories, of interest to potential contributors:

Repository	Audience	Description
CodeIgniter4	contributors	Project codebase, including tests & user guide sources
devstarter	developers	Starter project (app/public/writable). Dependent on develop branch of codebase repository
translations	developers	System message translations
coding-standard	contributors	Coding style conventions & rules

There are also several deployment repositories, referenced in the installation directions. The deployment repositories are built automatically when a new version is released, and they are not directly contributed to.

Repository	Audience	Description
framework	developers	Released versions of the framework
appstarter	developers	Starter project (app/public/writable). Dependent on “framework”
userguide	anyone	Pre-built user guide

In all the above, the latest version of a repository can be downloaded by selecting the “releases” link in the secondary navbar inside the “Code” tab of its Github repository page. The current (in development) version of each can

be cloned or downloaded by selecting the “Clone or download” dropdown button on the right-hand side if the repository homepage.

Composer Packages

We also maintain composer-installable packages on packagist.org [<https://packagist.org/search/?query=codeigniter4>]. These correspond to the repositories mentioned above:

- [codeigniter4/framework](https://packagist.org/packages/codeigniter4/framework) [<https://packagist.org/packages/codeigniter4/framework>]
- [codeigniter4/appstarter](https://packagist.org/packages/codeigniter4/appstarter) [<https://packagist.org/packages/codeigniter4/appstarter>]
- [codeigniter4/devstarter](https://packagist.org/packages/codeigniter4/devstarter) [<https://packagist.org/packages/codeigniter4/devstarter>]
- [codeigniter4/userguide](https://packagist.org/packages/codeigniter4/userguide) [<https://packagist.org/packages/codeigniter4/userguide>]
- [codeigniter4/translations](https://packagist.org/packages/codeigniter4/translations) [<https://packagist.org/packages/codeigniter4/translations>]
- [codeigniter4/CodeIgniter4](https://packagist.org/packages/codeigniter4/CodeIgniter4) [<https://packagist.org/packages/codeigniter4/CodeIgniter4>]
- [codeigniter4/coding-standard](https://packagist.org/packages/codeigniter4/codeigniter4-standard) [<https://packagist.org/packages/codeigniter4/codeigniter4-standard>]

See the [Installation](#) page for more information.

Tutorial

This tutorial is intended to introduce you to the CodeIgniter4 framework and the basic principles of MVC architecture. It will show you how a basic CodeIgniter application is constructed in step-by-step fashion.

If you are not familiar with PHP, we recommend that you check out the [W3Schools PHP Tutorial](https://www.w3schools.com/php/default.asp) [https://www.w3schools.com/php/default.asp] before continuing.

In this tutorial, you will be creating a **basic news application**. You will begin by writing the code that can load static pages. Next, you will create a news section that reads news items from a database. Finally, you'll add a form to create news items in the database.

This tutorial will primarily focus on:

- Model-View-Controller basics
- Routing basics
- Form validation
- Performing basic database queries using CodeIgniter's "Query Builder"

The entire tutorial is split up over several pages, each explaining a small part of the functionality of the CodeIgniter framework. You'll go through the following pages:

- Introduction, this page, which gives you an overview of what to expect.
- [Static pages](#), which will teach you the basics of controllers, views and routing.
- [News section](#), where you'll start using models and will be doing some basic database operations.
- [Create news items](#), which will introduce more advanced database operations and form validation.
- [Conclusion](#), which will give you some pointers on further reading and other resources.

Enjoy your exploration of the CodeIgniter framework.

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Static pages

Note: This tutorial assumes you've downloaded CodeIgniter and [installed the framework](#) in your development environment.

The first thing you're going to do is set up a **controller** to handle static pages. A controller is simply a class that helps delegate work. It is the glue of your web application.

For example, when a call is made to:

```
http://example.com/news/latest/10
```

We might imagine that there is a controller named “news”. The method being called on news would be “latest”. The news method's job could be to grab 10 news items, and render them on the page. Very often in MVC, you'll see URL patterns that match:

```
http://example.com/[controller-class]/[controller-method]/[arguments]
```

As URL schemes become more complex, this may change. But for now, this is all we will need to know.

Let's make our first controller

Create a file at **app/Controllers/Pages.php** with the following code.

```
<?php namespace App\Controllers;
use CodeIgniter\Controller;

class Pages extends Controller {

    public function index()
    {
        return view('welcome_message');
    }
}
```

```

        public function showme($page = 'home')
        {
        }
    }

```

You have created a class named Pages, with a showme method that accepts one argument named \$page. It also has an index() method, the same as the default controller found in **app/Controllers/Home.php**; that method displays the CodeIgniter welcome page.

The Pages class is extending the CodeIgniter\Controller class. This means that the new Pages class can access the methods and variables defined in the CodeIgniter\Controller class (*system/Controller.php*).

The **controller is what will become the center of every request** to your web application. Like any php class, you refer to it within your controllers as \$this.

Now that you’ve created your first method, it’s time to make some basic page templates. We will be creating two “views” (page templates) that act as our page footer and header.

Create the header at **app/Views/templates/header.php** and add the following code:

```

<!doctype html>
<html>
<head>
    <title>CodeIgniter Tutorial</title>
</head>
<body>

    <h1><?= $title; ?></h1>

```

The header contains the basic HTML code that you’ll want to display before loading the main view, together with a heading. It will also output the \$title variable, which we’ll define later in the controller. Now, create a footer at **app/Views/templates/footer.php** that includes the following code:

```

<em>&copy; 2019</em>

```



```
</body>
</html>
```

Adding logic to the controller

Earlier you set up a controller with a `showme()` method. The method accepts one parameter, which is the name of the page to be loaded. The static page bodies will be located in the **app/Views/pages/** directory.

In that directory, create two files named **home.php** and **about.php**. Within those files, type some text – anything you’d like – and save them. If you like to be particularly un-original, try “Hello World!”.

In order to load those pages, you’ll have to check whether the requested page actually exists. This will be the body of the `showme()` method in the Pages controller created above:

```
public function showme($page = 'home')
{
    if ( ! is_file(APPPATH.'/Views/pages/'.$page.'.php'))
    {
        // Whoops, we don't have a page for that!
        throw new \CodeIgniter\Exceptions\PageNotFoundException

    }

    $data['title'] = ucfirst($page); // Capitalize the first

    echo view('templates/header', $data);
    echo view('pages/'.$page, $data);
    echo view('templates/footer', $data);
}
```

Now, when the requested page does exist, it is loaded, including the header and footer, and displayed to the user. If the requested page doesn’t exist, a “404 Page not found” error is shown.

The first line in this method checks whether the page actually exists. PHP’s native `is_file()` function is used to check whether the file is where it’s expected to be. The `PageNotFoundException` is a CodeIgniter exception that causes the default error page to show.

In the header template, the `$title` variable was used to customize the page title. The value of `title` is defined in this method, but instead of assigning the value to a variable, it is assigned to the `title` element in the `$data` array.

The last thing that has to be done is loading the views in the order they should be displayed. The `view()` method built-in to CodeIgniter will be used to do this. The second parameter in the `view()` method is used to pass values to the view. Each value in the `$data` array is assigned to a variable with the name of its key. So the value of `$data['title']` in the controller is equivalent to `$title` in the view.

Note

Any files and directory names passed into the **`view()`** function **MUST** match the case of the actual directory and file itself or the system will throw errors on case-sensitive platforms.

Running the App

Ready to test? You cannot run the app using PHP's built-in server, since it will not properly process the `.htaccess` rules that are provided in `public`, and which eliminate the need to specify `"index.php/"` as part of a URL. CodeIgniter has its own command that you can use though.

From the command line, at the root of your project:

```
php spark serve
```

will start a web server, accessible on port 8080. If you set the location field in your browser to `localhost:8080`, you should see the CodeIgniter welcome page.

You can now try several URLs in the browser location field, to see what the *Pages* controller you made above produces...

- `localhost:8080/pages` will show the results from the *index* method

inside our *Pages* controller, which is to display the CodeIgniter “welcome” page, because “index” is the default controller method

- `localhost:8080/pages/index` will also show the CodeIgniter “welcome” page, because we explicitly asked for the “index” method
- `localhost:8080/pages/showme` will show the “home” page that you made above, because it is the default “page” parameter to the *showme()* method.
- `localhost:8080/pages/showme/home` will also show the “home” page that you made above, because we explicitly asked for it
- `localhost:8080/pages/showme/about` will show the “about” page that you made above, because we explicitly asked for it
- `localhost:8080/pages/showme/shop` will show a “404 - File Not Found” error page, because there is no *app/Views/pages/shop.php*

Routing

The controller is now functioning!

Using custom routing rules, you have the power to map any URI to any controller and method, and break free from the normal convention:
`http://example.com/[controller-class]/[controller-method]/[arguments]`

Let’s do that. Open the routing file located at *app/Config/Routes.php* and look for the “Route Definitions” section of the configuration file.

The only uncommented line there to start with should be::

```
$routes->get('/', 'Home::index');
```

This directive says that any incoming request without any content specified should be handled by the `index` method inside the `Home` controller.

Add the following line, **after** the route directive for `/'`.

```
$routes->get('/:any)', 'Pages::showme/$1');
```

CodeIgniter reads its routing rules from top to bottom and routes the request

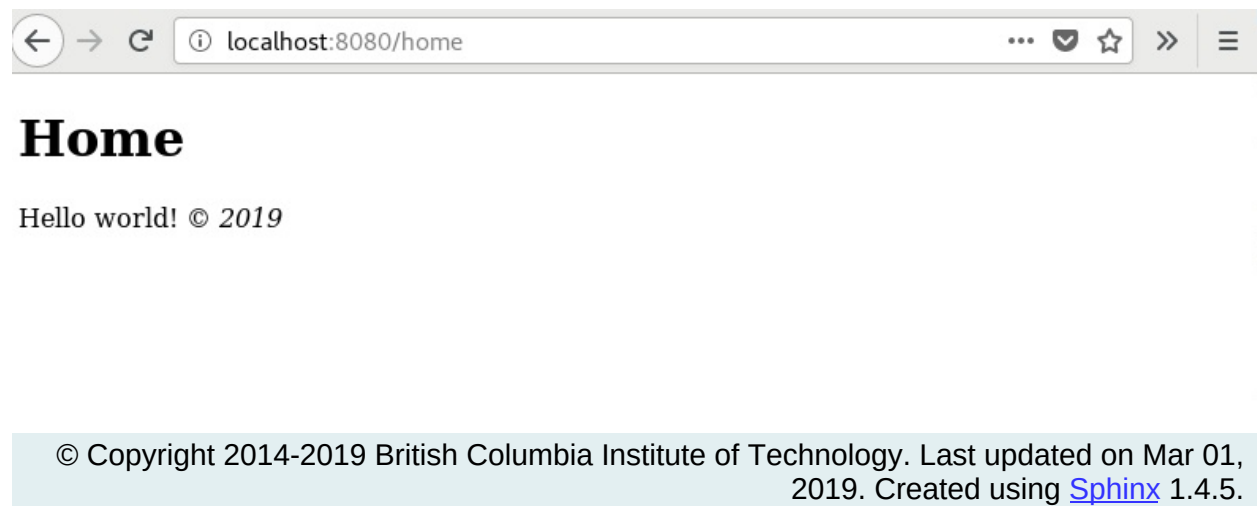
to the first matching rule. Each rule is a regular expression (left-side) mapped to a controller and method name separated by slashes (right-side). When a request comes in, CodeIgniter looks for the first match, and calls the appropriate controller and method, possibly with arguments.

More information about routing can be found in the [URI Routing documentation](#).

Here, the second rule in the `$routes` array matches **any** request using the wildcard string `(:any)`. and passes the parameter to the `view()` method of the `Pages` class.

Now visit home. Did it get routed correctly to the `showme()` method in the `pages` controller? Awesome!

You should see something like the following:



News section

In the last section, we went over some basic concepts of the framework by writing a class that references static pages. We cleaned up the URI by adding custom routing rules. Now it's time to introduce dynamic content and start using a database.

Create a database to work with

The CodeIgniter installation assumes that you have setup an appropriate database, as outlined in the [requirements](#). In this tutorial, we provide SQL code for a MySQL database, and we also assume that you have a suitable client for issuing database commands (mysql, MySQL Workbench, or phpMyAdmin).

You need to create a database that can be used for this tutorial, and then configure CodeIgniter to use it.

Using your database client, connect to your database and run the SQL command below (MySQL). Also add some seed records. For now, we'll just show you the SQL statements needed to create the table, but you should be aware that this can be done programmatically once you are more familiar with CodeIgniter; you can read about [Migrations](#) and [Seeds](#) to create more useful database setups later.

```
CREATE TABLE news (  
    id int(11) NOT NULL AUTO_INCREMENT,  
    title varchar(128) NOT NULL,  
    slug varchar(128) NOT NULL,  
    body text NOT NULL,  
    PRIMARY KEY (id),  
    KEY slug (slug)  
);
```

A note of interest: a “slug”, in the context of web publishing, is a user- and SEO-friendly short text used in a URL to identify and describe a resource.

The seed records might be something like::

```
INSERT INTO news VALUES
(1, 'Elvis sighted', 'elvis-sighted', 'Elvis was sighted at the Podu
(2, 'Say it isn\'t so!', 'say-it-isnt-so', 'Scientists conclude that
(3, 'Caffeination, Yes!', 'caffeination-yes', 'World\'s largest coff
```

Connect to your database

The local configuration file, `.env`, that you created when you installed CodeIgniter, should have the database property settings uncommented and set appropriately for the database you want to use.:

```
database.default.hostname = localhost
database.default.database = ci4tutorial
database.default.username = root
database.default.password = root
database.default.DBDriver = MySQLi
```

Setting up your model

Instead of writing database operations right in the controller, queries should be placed in a model, so they can easily be reused later. Models are the place where you retrieve, insert, and update information in your database or other data stores. They provide access to your data.

Open up the **app/Models/** directory and create a new file called **NewsModel.php** and add the following code. Make sure you've configured your database properly as described [here](#).

```
<?php namespace App\Models;

use CodeIgniter\Model;

class NewsModel extends Model
{
    protected $table = 'news';
}
```

This code looks similar to the controller code that was used earlier. It creates

a new model by extending `CodeIgniter\Model` and loads the database library. This will make the database class available through the `$this->db` object.

Now that the database and a model have been set up, you'll need a method to get all of our posts from our database. To do this, the database abstraction layer that is included with CodeIgniter — [Query Builder](#) — is used. This makes it possible to write your 'queries' once and make them work on [all supported database systems](#). The Model class also allows you to easily work with the Query Builder and provides some additional tools to make working with data simpler. Add the following code to your model.

```
public function getNews($slug = false)
{
    if ($slug === false)
    {
        return $this->findAll();
    }

    return $this->asArray()
        ->where(['slug' => $slug])
        ->first();
}
```

With this code you can perform two different queries. You can get all news records, or get a news item by its [slug](#). You might have noticed that the `$slug` variable wasn't sanitized before running the query; [Query Builder](#) does this for you.

The two methods used here, `findAll()` and `first()`, are provided by the Model class. They already know the table to use based on the `$table` property we set in **NewsModel** class, earlier. They are helper methods that use the Query Builder to run their commands on the current table, and returning an array of results in the format of your choice. In this example, `findAll()` returns an array of objects.

Display the news

Now that the queries are written, the model should be tied to the views that

are going to display the news items to the user. This could be done in our Pages controller created earlier, but for the sake of clarity, a new News controller is defined. Create the new controller at *app/Controllers/News.php*.

```
<?php namespace App\Controllers;
use App\Models\NewsModel;
use CodeIgniter\Controller;

class News extends Controller
{
    public function index()
    {
        $model = new NewsModel();

        $data['news'] = $model->getNews();
    }

    public function view($slug = null)
    {
        $model = new NewsModel();

        $data['news'] = $model->getNews($slug);
    }
}
```

Looking at the code, you may see some similarity with the files we created earlier. First, it extends a core CodeIgniter class, Controller, which provides a couple of helper methods, and makes sure that you have access to the current Request and Response objects, as well as the Logger class, for saving information to disk.

Next, there are two methods, one to view all news items, and one for a specific news item. You can see that the `$slug` variable is passed to the model's method in the second method. The model is using this slug to identify the news item to be returned.

Now the data is retrieved by the controller through our model, but nothing is displayed yet. The next thing to do is passing this data to the views. Modify the `index()` method to look like this:

```
public function index()
{
    $model = new NewsModel();
```



```

$data = [
    'news' => $model->getNews(),
    'title' => 'News archive',
];

echo view('templates/header', $data);
echo view('news/overview', $data);
echo view('templates/footer');
}

```

The code above gets all news records from the model and assigns it to a variable. The value for the title is also assigned to the `$data['title']` element and all data is passed to the views. You now need to create a view to render the news items. Create **app/Views/news/overview.php** and add the next piece of code.

```

<h2><?= $title ?></h2>

<?php if (! empty($news) && is_array($news)) : ?>

    <?php foreach ($news as $news_item): ?>

        <h3><?= $news_item['title'] ?></h3>

        <div class="main">
            <?= $news_item['body'] ?>
        </div>
        <p><a href="<?= '/news/' . $news_item['slug'] ?>">V

    <?php endforeach; ?>

<?php else : ?>

    <h3>No News</h3>

    <p>Unable to find any news for you.</p>

<?php endif ?>

```

Here, each news item is looped and displayed to the user. You can see we wrote our template in PHP mixed with HTML. If you prefer to use a template language, you can use CodeIgniter's [View Parser](#) or a third party parser.

The news overview page is now done, but a page to display individual news items is still absent. The model created earlier is made in such way that it can easily be used for this functionality. You only need to add some code to the controller and create a new view. Go back to the News controller and update the `view()` method with the following:

```
public function view($slug = NULL)
{
    $model = new NewsModel();


    $data['news'] = $model->getNews($slug);

    if (empty($data['news']))
    {
        throw new \CodeIgniter\PageNotFoundException('Can

    }

    $data['title'] = $data['news']['title'];

    echo view('templates/header', $data);
    echo view('news/view', $data);
    echo view('templates/footer');
}
```



Instead of calling the `getNews()` method without a parameter, the `$slug` variable is passed, so it will return the specific news item. The only thing left to do is create the corresponding view at **app/Views/news/view.php**. Put the following code in this file.

```
<?php
echo '<h2>'.$news['title'].'</h2>';
echo $news['body'];
```

Routing

Because of the wildcard routing rule created earlier, you need an extra route to view the controller that you just made. Modify your routing file (**app/config/routes.php**) so it looks as follows. This makes sure the requests reach the News controller instead of going directly to the Pages controller. The first line routes URI's with a slug to the `view()` method in the News controller.

```
$routes->get('news/(:segment)', 'News::view/$1');  
$routes->get('news', 'News::index');  
$routes->get('(:any)', 'Pages::view/$1');
```

Point your browser to your “news” page, i.e. `localhost:8080/news`, you should see a list of the news items, each of which has a link to display just the one article.



Create news items

You now know how you can read data from a database using CodeIgniter, but you haven't written any information to the database yet. In this section you'll expand your news controller and model created earlier to include this functionality.

Create a form

To input data into the database you need to create a form where you can input the information to be stored. This means you'll be needing a form with two fields, one for the title and one for the text. You'll derive the slug from our title in the model. Create the new view at **app/Views/news/create.php**.

```
<h2><?= esc($title); ?></h2>

<?= \Config\Services::validation()->listErrors(); ?>

<form action="/news/create">

    <label for="title">Title</label>
    <input type="input" name="title" /><br />

    <label for="body">Text</label>
    <textarea name="body"></textarea><br />

    <input type="submit" name="submit" value="Create news item" /

</form>
```

There is only one thing here that probably look unfamiliar to you: the `\Config\Services::validation()->listErrors()` function. It is used to report errors related to form validation.

Go back to your News controller. You're going to do two things here, check whether the form was submitted and whether the submitted data passed the

validation rules. You'll use the [form validation](#) library to do this.

```
public function create()
{
    helper('form');
    $model = new NewsModel();

    if (! $this->validate([
        'title' => 'required|min_length[3]|max_length[255]',
        'body'  => 'required'
    ]))
    {
        echo view('templates/header', ['title' => 'Create a news
        echo view('news/create');
        echo view('templates/footer');

    }
    else
    {
        $model->save([
            'title' => $this->request->getVar('title'),
            'slug'  => url_title($this->request->getVar('title'))
            'body'  => $this->request->getVar('body'),
        ]);
        echo view('news/success');
    }
}
```

The code above adds a lot of functionality. The first few lines load the form helper and the NewsModel. After that, the Controller-provided helper function is used to validate the \$_POST fields. In this case the title and text fields are required.

CodeIgniter has a powerful validation library as demonstrated above. You can read [more about this library here](#).

Continuing down, you can see a condition that checks whether the form validation ran successfully. If it did not, the form is displayed; if it was submitted **and** passed all the rules, the model is called. This takes care of passing the news item into the model. This contains a new function, url_title(). This function - provided by the [URL helper](#) - strips down the string you pass it, replacing all spaces by dashes (-) and makes sure

everything is in lowercase characters. This leaves you with a nice slug, perfect for creating URIs.

After this, a view is loaded to display a success message. Create a view at **app/Views/news/success.php** and write a success message.

This could be as simple as::

```
News item created successfully.
```

Model Updating

The only thing that remains is ensuring that your model is setup to allow data to be saved properly. The `save()` method that was used will determine whether the information should be inserted or if the row already exists and should be updated, based on the presence of a primary key. In this case, there is no `id` field passed to it, so it will insert a new row into its table, **news**.

However, by default the insert and update methods in the model will not actually save any data because it doesn't know what fields are safe to be updated. Edit the model to provide it a list of updatable fields in the `$allowedFields` property.

```
<?php namespace App\Models;
use CodeIgniter\Model;

class NewsModel extends Model
{
    protected $table = 'news';

    protected $allowedFields = ['title', 'slug', 'body'];
}
```

This new property now contains the fields that we allow to be saved to the database. Notice that we leave out the `id`? That's because you will almost never need to do that, since it is an auto-incrementing field in the database. This helps protect against Mass Assignment Vulnerabilities. If your model is handling your timestamps, you would also leave those out.

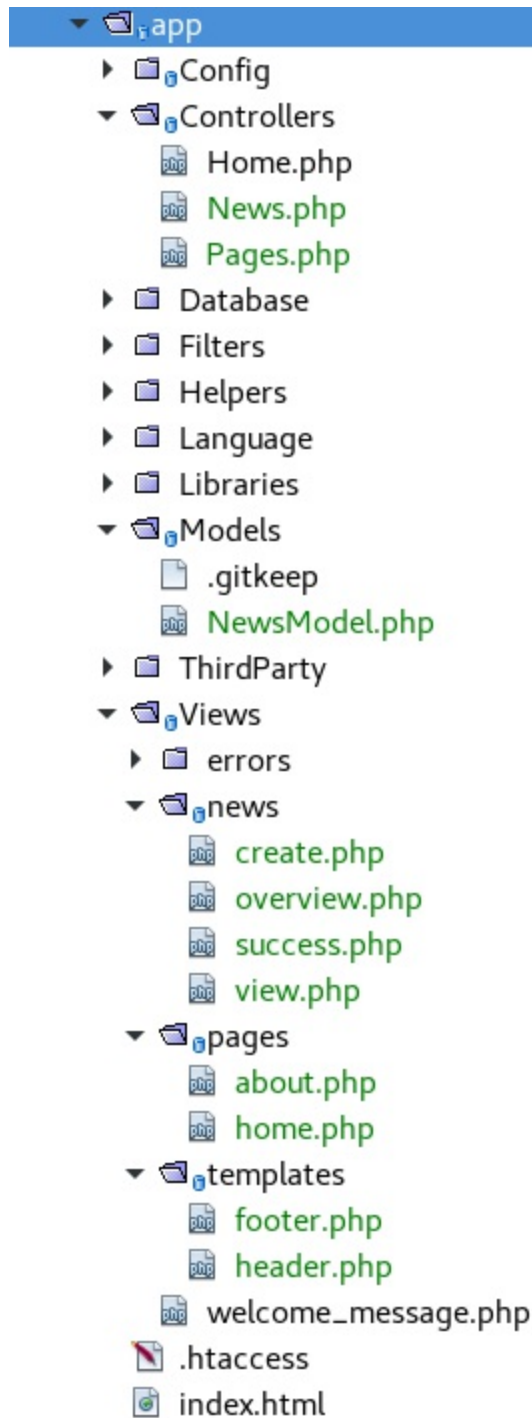
Routing

Before you can start adding news items into your CodeIgniter application you have to add an extra rule to **app/Config/Routes.php** file. Make sure your file contains the following. This makes sure CodeIgniter sees 'create' as a method instead of a news item's slug.

```
$routes->match(['get', 'post'], 'news/create', 'News::create');  
$routes->get('news/(:segment)', 'News::view/$1');  
$routes->get('news', 'News::index');  
$routes->get('/:any', 'Pages::view/$1');
```

Now point your browser to your local development environment where you installed CodeIgniter and add /news/create to the URL. Add some news and check out the different pages you made.





Congratulations

You just completed your first CodeIgniter4 application!

The image to the left shows your project's **app** folder, with all of the files

that you created in green. The two modified configuration files (Database & Routes) are not shown.

Conclusion

This tutorial did not cover all of the things you might expect of a full-fledged content management system, but it introduced you to the more important topics of routing, writing controllers, and models. We hope this tutorial gave you an insight into some of CodeIgniter’s basic design patterns, which you can expand upon.

Now that you’ve completed this tutorial, we recommend you check out the rest of the documentation. CodeIgniter is often praised because of its comprehensive documentation. Use this to your advantage and read the “Introduction” and “General Topics” sections thoroughly. You should read the class and helper references when needed.

Every intermediate PHP programmer should be able to get the hang of CodeIgniter within a few days.

If you still have questions about the framework or your own CodeIgniter code, you can:

- Check out our [forums](http://forum.codeigniter.com/) [http://forum.codeigniter.com/]

CodeIgniter4 Overview

The following pages describe the architectural concepts behind CodeIgniter4:

- [Application Structure](#)
- [Models, Views, and Controllers](#)
- [Autoloading Files](#)
- [Services](#)
- [Working With HTTP Requests](#)
- [Security Guidelines](#)

Application Structure

To get the most out of CodeIgniter, you need to understand how the application is structured, by default, and what you can change to meet the needs of your application.

Default Directories

A fresh install has six directories: /app, /system, /public, /writable, /tests and possibly /docs. Each of these directories has a very specific part to play.

app

The app directory is where all of your application code lives. This comes with a default directory structure that works well for many applications. The following folders make up the basic contents:

/app	
/Config	Stores the configuration files
/Controllers	Controllers determine the program flow
/Database	Stores the database migrations and seeds
/Filters	Stores filter classes that can run before
/Helpers	Helpers store collections of standalone f
/Language	Multiple language support reads the langu
/Libraries	Useful classes that don't fit in another
/Models	Models work with the database to represen
/ThirdParty	ThirdParty libraries that can be used in
/Views	Views make up the HTML that is displayed

Because the app directory is already namespaced, you should feel free to modify the structure of this directory to suit your application's needs. For example, you might decide to start using the Repository pattern and Entity Models to work with your data. In this case, you could rename the Models directory to Repositories, and add a new Entities directory.

Note

If you rename the `Controllers` directory, though, you will not be able to use the automatic method of routing to controllers, and will need to define all of your routes in the routes file.

All files in this directory live under the `App` namespace, though you are free to change that in **`app/Config/Constants.php`**.

system

This directory stores the files that make up the framework, itself. While you have a lot of flexibility in how you use the application directory, the files in the system directory should never be modified. Instead, you should extend the classes, or create new classes, to provide the desired functionality.

All files in this directory live under the `CodeIgniter` namespace.

public

The **public** folder holds the browser-accessible portion of your web application, preventing direct access to your source code. It contains the main **`.htaccess`** file, **`index.php`**, and any application assets that you add, like CSS, javascript, or images.

This folder is meant to be the “web root” of your site, and your web server would be configured to point to it.

writable

This directory holds any directories that might need to be written to in the course of an application’s life. This includes directories for storing cache files, logs, and any uploads a user might send. You should add any other directories that your application will need to write to here. This allows you to keep your other primary directories non-writable as an added security measure.

tests

This directory is setup to hold your test files. The `_support` directory holds various mock classes and other utilities that you can use while writing your tests. This directory does not need to be transferred to your production servers.

docs

If this directory is part of your project, it holds a local copy of the CodeIgniter4 User Guide.

Modifying Directory Locations

If you've relocated any of the main directories, you can change the configuration settings inside `app/Config/Paths`.

Please read [Managing your Applications](#)

Models, Views, and Controllers

Whenever you create an application, you have to find a way to organize the code to make it simple to locate the proper files and make it simple to maintain. Like most of the web frameworks, CodeIgniter uses the Model, View, Controller (MVC) pattern to organize the files. This keeps the data, the presentation, and flow through the application as separate parts. It should be noted that there are many views on the exact roles of each element, but this document describes our take on it. If you think of it differently, you're free to modify how you use each piece as you need.

Models manage the data of the application, and help to enforce any special business rules the application might need.

Views are simple files, with little to no logic, that display the information to the user.

Controllers act as glue code, marshaling data back and forth between the view (or the user that's seeing it) and the data storage.

At their most basic, controllers and models are simply classes that have a specific job. They are not the only class types that you can use, obviously, but they make up the core of how this framework is designed to be used. They even have designated directories in the **/app** directory for their storage, though you're free to store them wherever you desire, as long as they are properly namespaced. We will discuss that in more detail below.

Let's take a closer look at each of these three main components.

The Components

Views

Views are the simplest files and are typically HTML with very small amounts

of PHP. The PHP should be very simple, usually just displaying a variable's contents, or looping over some items and displaying their information in a table.

Views get the data to display from the controllers, who pass it to the views as variables that can be displayed with simple echo calls. You can also display other views within a view, making it pretty simple to display a common header or footer on every page.

Views are generally stored in **/app/Views**, but can quickly become unwieldy if not organized in some fashion. CodeIgniter does not enforce any type of organization, but a good rule of thumb would be to create a new directory in the **Views** directory for each controller. Then, name views by the method name. This makes them very easy find later on. For example, a user's profile might be displayed in a controller named `user`, and a method named `profile`. You might store the view file for this method in **/app/Views/User/Profile.php**.

That type of organization works great as a base habit to get into. At times you might need to organize it differently. That's not a problem. As long as CodeIgniter can find the file, it can display it.

[Find out more about views](#)

Models

A model's job is to maintain a single type of data for the application. This might be users, blog posts, transactions, etc. In this case, the model's job has two parts: enforce business rules on the data as it is pulled from, or put into, the database; and handle the actual saving and retrieval of the data from the database.

For many developers, the confusion comes in when determining what business rules are enforced. It simply means that any restrictions or requirements on the data is handled by the model. This might include normalizing raw data before it's saved to meet company standards, or formatting a column in a certain way before handing it to the controller. By keeping these business requirements in the model, you won't repeat code

throughout several controllers and accidentally miss updating an area.

Models are typically stored in **/app/Models**, though they can use a namespace to be grouped however you need.

[Find out more about models](#)

Controllers

Controllers have a couple of different roles to play. The most obvious one is that they receive input from the user and then determine what to do with it. This often involves passing the data to a model to save it, or requesting data from the model that is then passed on to the view to be displayed. This also includes loading up other utility classes, if needed, to handle specialized tasks that is outside of the purview of the model.

The other responsibility of the controller is to handle everything that pertains to HTTP requests - redirects, authentication, web safety, encoding, etc. In short, the controller is where you make sure that people are allowed to be there, and they get the data they need in a format they can use.

Controllers are typically stored in **/app/Controllers**, though they can use a namespace to be grouped however you need.

[Find out more about controllers](#)

Autoloading Files

Every application consists of a large number of classes in many different locations. The framework provides classes for core functionality. Your application will have a number of libraries, models, and other entities to make it work. You might have third-party classes that your project is using. Keeping track of where every single file is, and hard-coding that location into your files in a series of `requires()` is a massive headache and very error-prone. That's where autoloaders come in.

CodeIgniter provides a very flexible autoloader that can be used with very little configuration. It can locate individual non-namespaced classes, namespaced classes that adhere to [PSR4](http://www.php-fig.org/psr/psr-4/) [http://www.php-fig.org/psr/psr-4/] autoloading directory structures, and will even attempt to locate classes in common directories (like Controllers, Models, etc).

For performance improvement, the core CodeIgniter components have been added to the classmap.

The autoloader works great by itself, but can also work with other autoloaders, like [Composer](https://getcomposer.org) [https://getcomposer.org], or even your own custom autoloaders, if needed. Because they're all registered through [spl_autoload_register](http://php.net/manual/en/function.spl-autoload-register.php) [http://php.net/manual/en/function.spl-autoload-register.php], they work in sequence and don't get in each other's way.

The autoloader is always active, being registered with `spl_autoload_register()` at the beginning of the framework's execution.

Configuration

Initial configuration is done in **/app/Config/Autoload.php**. This file contains two primary arrays: one for the classmap, and one for PSR4-compatible namespaces.

Namespaces

The recommended method for organizing your classes is to create one or more namespaces for your application's files. This is most important for any business-logic related classes, entity classes, etc. The `psr4` array in the configuration file allows you to map the namespace to the directory those classes can be found in:

```
$psr4 = [  
    'App'           => APPPATH,  
    'CodeIgniter' => SYSTEMPATH,  
];
```

The key of each row is the namespace itself. This does not need a trailing slash. If you use double-quotes to define the array, be sure to escape the backwards slash. That means that it would be `My\\App`, not `My\App`. The value is the location to the directory the classes can be found in. They should have a trailing slash.

By default, the application folder is namespace to the `App` namespace. While you are not forced to namespace the controllers, libraries, or models in the application directory, if you do, they will be found under the `App` namespace. You may change this namespace by editing the `/app/Config/Constants.php` file and setting the new namespace value under the `APP_NAMESPACE` setting:

```
define('APP_NAMESPACE', 'App');
```

You will need to modify any existing files that are referencing the current namespace.

Important

Config files are namespaced in the `Config` namespace, not in `App\Config` as you might expect. This allows the core system files to always be able to locate them, even when the application namespace has changed.

Classmap

The classmap is used extensively by CodeIgniter to eke the last ounces of performance out of the system by not hitting the file-system with extra `is_file()` calls. You can use the classmap to link to third-party libraries that are not namespaced:

```
$classmap = [  
    'Markdown' => APPPATH . 'third_party/markdown.php'  
];
```

The key of each row is the name of the class that you want to locate. The value is the path to locate it at.

Legacy Support

If neither of the above methods find the class, and the class is not namespaced, the autoloader will look in the **/app/Libraries** and **/app/Models** directories to attempt to locate the files. This provides a measure to help ease the transition from previous versions.

There are no configuration options for legacy support.

Composer Support

Composer support is automatically initialized by default. By default it looks for Composer's autoload file at `ROOTPATH.'vendor/autoload.php'`. If you need to change the location of that file for any reason, you can modify the value defined in `Config\Constants.php`.

Note

If the same namespace is defined in both CodeIgniter and Composer, CodeIgniter's autoloader will be the first one to get a chance to locate the file.

Services

- [Introduction](#)
 - [Convenience Functions](#)
- [Defining Services](#)
 - [Allowing Parameters](#)
 - [Shared Classes](#)
 - [Service Discovery](#)

[Introduction](#)

All of the classes within CodeIgniter are provided as “services”. This simply means that, instead of hard-coding a class name to load, the classes to call are defined within a very simple configuration file. This file acts as a type of factory to create new instances of the required class.

A quick example will probably make things clearer, so imagine that you need to pull in an instance of the Timer class. The simplest method would simply be to create a new instance of that class:

```
$timer = new \CodeIgniter\Debug\Timer();
```

And this works great. Until you decide that you want to use a different timer class in its place. Maybe this one has some advanced reporting the default timer does not provide. In order to do this, you now have to locate all of the locations in your application that you have used the timer class. Since you might have left them in place to keep a performance log of your application constantly running, this might be a time-consuming and error-prone way to handle this. That’s where services come in handy.

Instead of creating the instance ourselves, we let a central class create an instance of the class for us. This class is kept very simple. It only contains a

method for each class that we want to use as a service. The method typically returns a shared instance of that class, passing any dependencies it might have into it. Then, we would replace our timer creation code with code that calls this new class:

```
$timer = \Config\Services::timer();
```

When you need to change the implementation used, you can modify the services configuration file, and the change happens automatically throughout your application without you having to do anything. Now you just need to take advantage of any new functionality and you're good to go. Very simple and error-resistant.

Note

It is recommended to only create services within controllers. Other files, like models and libraries should have the dependencies either passed into the constructor or through a setter method.

Convenience Functions

Two functions have been provided for getting a service. These functions are always available.

The first is `service()` which returns a new instance of the requested service. The only required parameter is the service name. This is the same as the method name within the Services file always returns a SHARED instance of the class, so calling the function multiple times should always return the same instance:

```
$logger = service('logger');
```

If the creation method requires additional parameters, they can be passed after the service name:

```
$renderer = service('renderer', APPPATH.'views/');
```

The second function, `single_service()` works just like `service()` but returns a new instance of the class:


```
$logger = single_service('logger');
```

Defining Services

To make services work well, you have to be able to rely on each class having a constant API, or [interface](http://php.net/manual/en/language.oop5.interfaces.php) [http://php.net/manual/en/language.oop5.interfaces.php], to use. Almost all of CodeIgniter's classes provide an interface that they adhere to. When you want to extend or replace core classes, you only need to ensure you meet the requirements of the interface and you know that the classes are compatible.

For example, the `RouterCollection` class implements the `RouterCollectionInterface`. When you want to create a replacement that provides a different way to create routes, you just need to create a new class that implements the `RouterCollectionInterface`:

```
class MyRouter implements \CodeIgniter\Router\RouteCollectionInterface
{
    // Implement required methods here.
}
```



Finally, modify `/app/Config/Services.php` to create a new instance of `MyRouter` instead of `CodeIgniter\Router\RouterCollection`:

```
public static function routes()
{
    return new \App\Router\MyRouter();
}
```

Allowing Parameters

In some instances, you will want the option to pass a setting to the class during instantiation. Since the services file is a very simple class, it is easy to make this work.

A good example is the renderer service. By default, we want this class to be able to find the views at `APPPATH.views/`. We want the developer to have the option of changing that path, though, if their needs require it. So the class accepts the `$viewPath` as a constructor parameter. The service method looks like this:

```
public static function renderer($viewPath=APPPATH.'views/')
{
    return new \CodeIgniter\View\View($viewPath);
}
```

This sets the default path in the constructor method, but allows for easily changing the path it uses:

```
$renderer = \Config\Services::renderer('/shared/views');
```

Shared Classes

There are occasions where you need to require that only a single instance of a service is created. This is easily handled with the `getSharedInstance()` method that is called from within the factory method. This handles checking if an instance has been created and saved within the class, and, if not, creates a new one. All of the factory methods provide a `$getShared = true` value as the last parameter. You should stick to the method also:

```
class Services
{
    public static function routes($getShared = false)
    {
        if (! $getShared)
        {
            return new \CodeIgniter\Router\RouteCollection();
        }

        return static::getSharedInstance('routes');
    }
}
```

Service Discovery

CodeIgniter can automatically discover any `ConfigServices.php` files you

may have created within any defined namespaces. This allows simple use of any module Services files. In order for custom Services files to be discovered, they must meet these requirements:

- It's namespace must be defined `Config\Autoload.php`
- Inside the namespace, the file must be found at `Config\Services.php`
- It must extend `CodeIgniter\Config\BaseService`

A small example should clarify this.

Imagine that you've created a new directory, `Blog` in your root directory. This will hold a blog module with controllers, models, etc, and you'd like to make some of the classes available as a service. The first step is to create a new file: `Blog\Config\Services.php`. The skeleton of the file should be:

```
<?php namespace Blog\Config;

use CodeIgniter\Config\BaseService;

class Services extends BaseService
{
    public static function postManager()
    {
        ...
    }
}
```

Now you can use this file as described above. When you want to grab the posts service from any controller, you would simply use the framework's `Config\Services` class to grab your service:

```
$postManager = Config\Services::postManager();
```

Note

If multiple Services file have the same method name, the first one found will be the instance returned.

Working With HTTP Requests

In order to get the most out of CodeIgniter, you need to have a basic understanding of how HTTP requests and responses work. Since this is what you work with while developing web applications, understanding the concepts behind HTTP is a **must** for all developers that want to be successful.

The first part of this chapter gives an overview. After the concepts are out of the way, we will discuss how to work with the requests and responses within CodeIgniter.

What is HTTP?

HTTP is simply a text-based convention that allows two machines to talk to each other. When a browser requests a page, it asks the server if it can get the page. The server then prepares the page and sends a response back to the browser that asked for it. That's pretty much it. Obviously, there are some complexities that you can use, but the basics are really pretty simple.

HTTP is the term used to describe that exchange convention. It stands for HyperText Transfer Protocol. Your goal when you develop web applications is to always understand what the browser is requesting, and be able to respond appropriately.

The Request

Whenever a client (a web browser, smartphone app, etc) makes a request, it sends a small text message to the server and waits for a response.

The request would look something like this:

```
GET / HTTP/1.1  
Host codeigniter.com
```

```
Accept: text/html
User-Agent: Chrome/46.0.2490.80
```

This message displays all of the information necessary to know what the client is requesting. It tells the method for the request (GET, POST, DELETE, etc), and the version of HTTP it supports.

The request also includes a number of optional request headers that can contain a wide variety of information such as what languages the client wants the content displayed as, the types of formats the client accepts, and much more. Wikipedia has an article that lists [all header fields](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields) [https://en.wikipedia.org/wiki/List_of_HTTP_header_fields] if you want to look it over.

The Response

Once the server receives the request, your application will take that information and generate some output. The server will bundle your output as part of its response to the client. This is also represented as a simple text message that looks something like this:

```
HTTP/1.1 200 OK
Server: nginx/1.8.0
Date: Thu, 05 Nov 2015 05:33:22 GMT
Content-Type: text/html; charset=UTF-8
```

```
<html>
. . .
</html>
```

The response tells the client what version of the HTTP specification that it's using and, probably most importantly, the status code (200). The status code is one of a number of codes that have been standardized to have a very specific meaning to the client. This can tell them that it was successful (200), or that the page wasn't found (404). Head over to IANA for a [full list of HTTP status codes](https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml) [https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml].

Working with Requests and Responses

While PHP provides ways to interact with the request and response headers, CodeIgniter, like most frameworks, abstracts them so that you have a

consistent, simple interface to them. The [IncomingRequest class](#) is an object-oriented representation of the HTTP request. It provides everything you need:

```
use CodeIgniter\HTTP\IncomingRequest;

$request = new IncomingRequest(new \Config\App(), new \CodeIgnite

// the URI being requested (i.e. /about)
$request->uri->getPath();


// Retrieve $_GET and $_POST variables
$request->getVar('foo');
$request->getGet('foo');
$request->getPost('foo');

// Retrieve JSON from AJAX calls
$request->getJSON();

// Retrieve server variables
$request->getServer('Host');

// Retrieve an HTTP Request header, with case-insensitive names
$request->getHeader('host');
$request->getHeader('Content-Type');

$request->getMethod(); // GET, POST, PUT, etc
```



The request class does a lot of work in the background for you, that you never need to worry about. The `isAJAX()` and `isSecure()` methods check several different methods to determine the correct answer.

CodeIgniter also provides a [Response class](#) that is an object-oriented representation of the HTTP response. This gives you an easy and powerful way to construct your response to the client:

```
use CodeIgniter\HTTP\Response;

$response = new Response();

$response->setStatusCode(Response::HTTP_OK);
$response->setBody($output);
$response->setHeader('Content-type', 'text/html');
$response->noCache();
```

```
// Sends the output to the browser  
$response->send();
```

In addition, the Response class allows you to work the HTTP cache layer for the best performance.

Security Guidelines

We take security seriously. CodeIgniter incorporates a number of features and techniques to either enforce good security practices, or to enable you to do so easily.

We respect the [Open Web Application Security Project \(OWASP\)](#) [<https://www.owasp.org>] and follow their recommendations as much as possible.

The following comes from [OWASP Top Ten Cheat Sheet](#) [https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet], identifying the top vulnerabilities for web applications. For each, we provide a brief description, the OWASP recommendations, and then the CodeIgniter provisions to address the problem.

A1 Injection

An injection is the inappropriate insertion of partial or complete data via the input data from the client to the application. Attack vectors include SQL, XML, ORM, code & buffer overflows.

OWASP recommendations

- Presentation: set correct content type, character set & locale
- Submission: validate fields and provide feedback
- Controller: sanitize input; positive input validation using correct character set
- Model: parameterized queries

CodeIgniter provisions

- [HTTP library](#) provides for input field filtering & content metadata
- Form validation library

A2 Weak authentication and session management

Inadequate authentication or improper session management can lead to a user getting more privileges than they are entitled to.

OWASP recommendations

- Presentation: validate authentication & role; send CSRF token with forms
- Design: only use built-in session management
- Controller: validate user, role, CSRF token
- Model: validate role
- Tip: consider the use of a request governor

CodeIgniter provisions

- [Session](#) library
- [HTTP library](#) provides for CSRF validation
- Easy to add third party authentication

A3 Cross Site Scripting (XSS)

Insufficient input validation where one user can add content to a web site that can be malicious when viewed by other users to the web site.

OWASP recommendations

- Presentation: output encode all user data as per output context; set input constraints
- Controller: positive input validation
- Tips: only process trustworthy data; do not store data HTML encoded in DB

CodeIgniter provisions

- `esc` function

- Form validation library

A4 Insecure Direct Object Reference

Insecure Direct Object References occur when an application provides direct access to objects based on user-supplied input. As a result of this vulnerability attackers can bypass authorization and access resources in the system directly, for example database records or files.

OWASP recommendations

- Presentation: don't expose internal data; use random reference maps
- Controller: obtain data from trusted sources or random reference maps
- Model: validate user roles before updating data

CodeIgniter provisions

- Form validation library
- Easy to add third party authentication

A5 Security Misconfiguration

Improper configuration of an application architecture can lead to mistakes that might compromise the security of the whole architecture.

OWASP recommendations

- Presentation: harden web and application servers; use HTTP strict transport security
- Controller: harden web and application servers; protect your XML stack
- Model: harden database servers

CodeIgniter provisions

- Sanity checks during bootstrap

A6 Sensitive Data Exposure

Sensitive data must be protected when it is transmitted through the network. Such data can include user credentials and credit cards. As a rule of thumb, if data must be protected when it is stored, it must be protected also during transmission.

OWASP recommendations

- Presentation: use TLS1.2; use strong ciphers and hashes; do not send keys or hashes to browser
- Controller: use strong ciphers and hashes
- Model: mandate strong encrypted communications with servers

CodeIgniter provisions

- Session keys stored encrypted

A7 Missing Function Level Access Control

Sensitive data must be protected when it is transmitted through the network. Such data can include user credentials and credit cards. As a rule of thumb, if data must be protected when it is stored, it must be protected also during transmission.

OWASP recommendations

- Presentation: ensure that non-web data is outside the web root; validate users and roles; send CSRF tokens
- Controller: validate users and roles; validate CSRF tokens
- Model: validate roles

CodeIgniter provisions

- Public folder, with application and system outside
- [HTTP library](#) provides for CSRF validation

A8 Cross Site Request Forgery (CSRF)

CSRF is an attack that forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated.

OWASP recommendations

- Presentation: validate users and roles; send CSRF tokens
- Controller: validate users and roles; validate CSRF tokens
- Model: validate roles

CodeIgniter provisions

- [HTTP library](#) provides for CSRF validation

A9 Using Components with Known Vulnerabilities

Many applications have known vulnerabilities and known attack strategies that can be exploited in order to gain remote control or to exploit data.

OWASP recommendations

- Don't use any of these

CodeIgniter provisions

- Third party libraries incorporated must be vetted

A10 Unvalidated Redirects and Forwards

Faulty business logic or injected actionable code could redirect the user inappropriately.

OWASP recommendations

- Presentation: don't use URL redirection; use random indirect references
- Controller: don't use URL redirection; use random indirect references
- Model: validate roles

CodeIgniter provisions

- [HTTP library](#) provides for ...
- [Session](#) library providesflashdata

General Topics

- [Working With Configuration Files](#)
- [CodeIgniter URLs](#)
- [Helper Functions](#)
- [Global Functions and Constants](#)
- [Logging Information](#)
- [Error Handling](#)
- [Web Page Caching](#)
- [Code Modules](#)
- [Managing your Applications](#)
- [Handling Multiple Environments](#)

Working With Configuration Files

Every application needs a way to define various settings that affect the application. These are handled through configuration files. Configuration files simply hold a class that contains its settings as public properties. Unlike in many other frameworks, there is no single class that you need to use to access your settings. Instead, you simply create an instance of the class and all your settings are there for you.

- [Accessing Config Files](#)
- [Creating Configuration Files](#)
- [Handling Different Environments](#)
- [Nesting Variables](#)
- [Namespaced Variables](#)
- [Incorporating Environment Variables Into a Configuration](#)
- [Treating Environment Variables as Arrays](#)
- [Registrars](#)

[Accessing Config Files](#)

You can access config files within your classes by creating a new instance or using the config function. All of the properties are public, so you access the settings like any other property:

```
// Creating new class by hand
$config = new \Config\Pager();

// Creating new class with config function
$config = config( 'Pager', false );

// Get shared instance with config function
$config = config( 'Pager' );

// Access config class with namespace
```

```
$config = config( 'Config\\Pager' );  
  
// Access settings as class properties  
$pageSize = $pager->perPage;
```

If no namespace is provided, it will look for the files in all available namespaces that have been defined, as well as **/app/Config/**. All of the configuration files that ship with CodeIgniter are namespaced with Config. Using this namespace in your application will provide the best performance since it knows exactly what directory to find the files in and doesn't have to scan several locations in the filesystem to get there.

You can locate the configuration files any place on your server by using a different namespace. This allows you to pull configuration files on the production server to a folder that is not in the web-accessible space at all, while keeping it under **/app** for ease of access during development.

Creating Configuration Files

If you need to create a new configuration file you would create a new file at your desired location, **/app/Config** by default. Then create the class and fill it with public properties that represent your settings:

```
<?php namespace Config;  
  
use CodeIgniter\Config\BaseConfig;  
  
class App extends BaseConfig  
{  
    public $siteName    = 'My Great Site';  
    public $siteEmail   = 'webmaster@example.com';  
}
```

The class should extend `\CodeIgniter\Config\BaseConfig` to ensure that it can receive environment-specific settings.

Handling Different Environments

Because your site can operate within multiple environments, such as the

developer's local machine or the server used for the production site, you can modify your values based on the environment. Within these you will have settings that might change depending on the server it's running on. This can include database settings, API credentials, and other settings that will vary between deploys.

You can store values in a **.env** file in the root directory, alongside the system and application directories. It is simply a collection of name/value pairs separated by an equal sign, much like a ".ini" file:

```
S3_BUCKET="dotenv"  
SECRET_KEY="super_secret_key"
```

If the variable exists in the environment already, it will NOT be overwritten.

Important

Make sure the **.env** file is added to **.gitignore** (or your version control system's equivalent) so it is not checked in the code. Failure to do so could result in sensitive credentials being stored in the repository for anyone to find.

You are encouraged to create a template file, like **env.example**, that has all of the variables your project needs with empty or dummy data. In each environment, you can then copy the file to **.env** and fill in the appropriate data.

When your application runs, this file will be automatically loaded and the variables will be put into the environment. This will work in any environment. These variables are then available through `getenv()`, `$_SERVER`, and `$_ENV`. Of the three, `getenv()` function is recommended since it is not case-sensitive:

```
$s3_bucket = getenv('S3_BUCKET');  
$s3_bucket = $_ENV['S3_BUCKET'];  
$s3_bucket = $_SERVER['S3_BUCKET'];
```

Note

If you are using Apache, then the `CI_ENVIRONMENT` can be set at the top of `public/.htaccess`, which comes with a commented line to do that. Change the environment setting to the one you want to use, and uncomment that line.

Nesting Variables

To save on typing, you can reuse variables that you've already specified in the file by wrapping the variable name within `${...}`:

```
BASE_DIR="/var/webroot/project-root"
CACHE_DIR="${BASE_DIR}/cache"
TMP_DIR="${BASE_DIR}/tmp"
```

Namespaced Variables

There will be times when you will have several variables with the same name. When this happens, the system has no way of knowing what the correct value should be. You can protect against this by “namespacing” the variables.

Namespaced variables use a dot notation to qualify variable names when those variables get incorporated into configuration files. This is done by including a distinguishing prefix, followed by a dot (`.`), and then the variable name itself:

```
// not namespaced variables
name = "George"
db=my_db

// namespaced variables
address.city = "Berlin"
address.country = "Germany"
frontend.db = sales
backend.db = admin
BackEnd.db = admin
```

Incorporating Environment Variables Into a

Configuration

When you instantiate a configuration file, any namespaced environment variables are considered for merging into the a configuration objects' properties.

If the prefix of a namespaced variable matches the configuration class name exactly, case-sensitive, then the trailing part of the variable name (after the dot) is treated as a configuration property name. If it matches an existing configuration property, the environment variable's value will override the corresponding one in the configuration file. If there is no match, the configuration properties are left unchanged.

The same holds for a “short prefix”, which is the name given to the case when the environment variable prefix matches the configuration class name converted to lower case.

Treating Environment Variables as Arrays

A namespaced environment variable can be further treated as an array. If the prefix matches the configuration class, then the remainder of the environment variable name is treated as an array reference if it also contains a dot:

```
// regular namespaced variable  
SimpleConfig.name = George  
  
// array namespaced variables  
SimpleConfig.address.city = "Berlin"  
SimpleConfig.address.country = "Germany"
```

If this was referring to a SimpleConfig configuration object, the above example would be treated as:

```
$address['city']    = "Berlin";  
$address['country'] = "Germany";
```

Any other elements of the \$address property would be unchanged.

You can also use the array property name as a prefix. If the environment file

held instead:

```
// array namespaced variables
SimpleConfig.address.city = "Berlin"
address.country = "Germany"
```

then the result would be the same as above.

Registrars

A configuration file can also specify any number of “registrars”, which are any other classes which might provide additional configuration properties. This is done by adding a registrars property to your configuration file, holding an array of the names of candidate registrars.:

```
protected $registrars = [
    SupportingPackageRegistrar::class
];
```

In order to act as a “registrar” the classes so identified must have a static function named the same as the configuration class, and it should return an associative array of property settings.

When your configuration object is instantiated, it will loop over the designated classes in \$registrars. For each of these classes, which contains a method name matching the configuration class, it will invoke that method, and incorporate any returned properties the same way as described for namespaced variables.

A sample configuration class setup for this:

```
<?php namespace App\Config;

use CodeIgniter\Config\BaseConfig;

class MySalesConfig extends BaseConfig
{
    public $target          = 100;
    public $campaign        = "Winter Wonderland";
    protected $registrars = [
        '\App\Models\RegionalSales';
    ];
}
```

```
    ];  
}
```

... and the associated regional sales model might look like:

```
<?php namespace App\Models;  
  
class RegionalSales  
{  
    public static function MySalesConfig()  
    {  
        return ['target' => 45, 'actual' => 72];  
    }  
}
```

With the above example, when *MySalesConfig* is instantiated, it will end up with the two properties declared, but the value of the *\$target* property will be over-ridden by treating *RegionalSalesModel* as a “registrar”. The resulting configuration properties:

```
$target    = 45;  
$campaign  = "Winter Wonderland";
```

CodeIgniter URLs

By default, URLs in CodeIgniter are designed to be search-engine and human-friendly. Rather than using the standard “query-string” approach to URLs that is synonymous with dynamic systems, CodeIgniter uses a **segment-based** approach:

`example.com/news/article/my_article`

URI Segments

The segments in the URL, in following with the Model-View-Controller approach, usually represent:

`example.com/class/method/ID`

1. The first segment represents the controller **class** that should be invoked.
2. The second segment represents the class **method** that should be called.
3. The third, and any additional segments, represent the ID and any variables that will be passed to the controller.

The [URI Library](#) and the [URL Helper](#) contain functions that make it easy to work with your URI data. In addition, your URLs can be remapped using the [URI Routing](#) feature for more flexibility.

Removing the index.php file

By default, the **index.php** file will be included in your URLs:

`example.com/index.php/news/article/my_article`

If your server supports rewriting URLs you can easily remove this file with URL rewriting. This is handled differently by different servers, but we will show examples for the two most common web servers here.

Apache Web Server

Apache must have the *mod_rewrite* extension enabled. If it does, you can use a *.htaccess* file with some simple rules. Here is an example of such a file, using the “negative” method in which everything is redirected except the specified items:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php/$1 [L]
```

In this example, any HTTP request other than those for existing directories and existing files is treated as a request for your *index.php* file.

Note

These specific rules might not work for all server configurations.

Note

Make sure to also exclude from the above rules any assets that you might need to be accessible from the outside world.

NGINX

Under NGINX, you can define a location block and use the *try_files* directive to get the same effect as we did with the above Apache configuration:

```
location / {
    try_files $uri $uri/ /index.php/$args;
}
```

This will first look for a file or directory matching the URI (constructing the full path to each file from the settings of the root and alias directives), and then sends the request to the *index.php* file along with any arguments.

Helper Functions

Helpers, as the name suggests, help you with tasks. Each helper file is simply a collection of functions in a particular category. There are **URL Helpers**, that assist in creating links, there are **Form Helpers** that help you create form elements, **Text Helpers** perform various text formatting routines, **Cookie Helpers** set and read cookies, **File Helpers** help you deal with files, etc.

- [Loading a Helper](#)
 - [Loading from Non-standard Locations](#)
- [Using a Helper](#)
- [“Extending” Helpers](#)
- [Now What?](#)

Unlike most other systems in CodeIgniter, Helpers are not written in an Object Oriented format. They are simple, procedural functions. Each helper function performs one specific task, with no dependence on other functions.

CodeIgniter does not load Helper Files by default, so the first step in using a Helper is to load it. Once loaded, it becomes globally available in your [controller](#) and [views](#).

Helpers are typically stored in your **system/Helpers**, or **app/Helpers directory**. CodeIgniter will look first in your **app/Helpers directory**. If the directory does not exist or the specified helper is not located there CI will instead look in your global *system/Helpers/* directory.

[Loading a Helper](#)

Loading a helper file is quite simple using the following method:

```
helper( 'name' );
```

Where **name** is the file name of the helper, without the .php file extension or the “helper” part.

For example, to load the **Cookie Helper** file, which is named **cookie_helper.php**, you would do this:

```
helper('cookie');
```

If you need to load more than one helper at a time, you can pass an array of file names in and all of them will be loaded:

```
helper(['cookie', 'date']);
```

A helper can be loaded anywhere within your controller methods (or even within your View files, although that’s not a good practice), as long as you load it before you use it. You can load your helpers in your controller constructor so that they become available automatically in any function, or you can load a helper in a specific function that needs it.

Note

The Helper loading method above does not return a value, so don’t try to assign it to a variable. Just use it as shown.

Note

The URL helper is always loaded so you do not need to load it yourself.

Loading from Non-standard Locations

Helpers can be loaded from directories outside of **app/Helpers** and **system/Helpers**, as long as that path can be found through a namespace that has been setup within the PSR-4 section of the [Autoloader config file](#). You would prefix the name of the Helper with the namespace that it can be located in. Within that namespaced directory, the loader expects it to live within a sub-directory named `Helpers`. An example will help understand this.

For this example, assume that we have grouped together all of our Blog-related code into its own namespace, `Example\Blog`. The files exist on our server at `/Modules/Blog/`. So, we would put our Helper files for the blog module in `/Modules/Blog/Helpers/`. A `blog_helper` file would be at `/Modules/Blog/Helpers/blog_helper.php`. Within our controller we could use the following command to load the helper for us:

```
helper('Modules\Blog\blog');
```

Note

The functions within files loaded this way are not truly namespaced. The namespace is simply used as a convenient way to locate the files.

Using a Helper

Once you've loaded the Helper File containing the function you intend to use, you'll call it the way you would a standard PHP function.

For example, to create a link using the `anchor()` function in one of your view files you would do this:

```
<?php echo anchor('blog/comments', 'Click Here');?>
```

Where “Click Here” is the name of the link, and “blog/comments” is the URI to the controller/method you wish to link to.

“Extending” Helpers

To “extend” Helpers, create a file in your `app/Helpers/` folder with an identical name to the existing Helper.


If all you need to do is add some functionality to an existing helper - perhaps add a function or two, or change how a particular helper function operates - then it's overkill to replace the entire helper with your version. In this case it's better to simply “extend” the Helper.

Note

The term “extend” is used loosely since Helper functions are procedural and discrete and cannot be extended in the traditional programmatic sense. Under the hood, this gives you the ability to add to or or to replace the functions a Helper provides.

For example, to extend the native **Array Helper** you’ll create a file named **app/Helpers/array_helper.php**, and add or override functions:

```
// any_in_array() is not in the Array Helper, so it defines a new  
function any_in_array($needle, $haystack)  
{  
    $needle = is_array($needle) ? $needle : [$needle];  
  
    foreach ($needle as $item)  
    {  
        if (in_array($item, $haystack))  
        {  
            return TRUE;  
        }  
    }  
  
    return FALSE;  
}  
  
// random_element() is included in Array Helper, so it overrides  
function random_element($array)  
{  
    shuffle($array);  
    return array_pop($array);  
}
```



The **helper()** method will scan through all PSR-4 namespaces defined in **app/Config/Autoload.php** and load in ALL matching helpers of the same name. This allows any module’s helpers to be loaded, as well as any helpers you’ve created specifically for this application. The load order is as follows:

1. app/Helpers - Files loaded here are always loaded first.
2. {namespace}/Helpers - All namespaces are looped through in the order

they are defined.

3. system/Helpers - The base file is loaded last

Now What?

In the Table of Contents you'll find a list of all the available Helper Files. Browse each one to see what they do.

Global Functions and Constants

CodeIgniter uses provides a few functions and variables that are globally defined, and are available to you at any point. These do not require loading any additional libraries or helpers.

- [Global Functions](#)
 - [Service Accessors](#)
 - [Miscellaneous Functions](#)
- [Global Constants](#)
 - [Core Constants](#)
 - [Time Constants](#)

[Global Functions](#)

[Service Accessors](#)

cache(*[\$key]*)

Parameters: • **\$key** (*string*) – The cache name of the item to retrieve from cache (Optional)

Returns: Either the cache object, or the item retrieved from the cache

Return type: mixed

If no \$key is provided, will return the Cache engine instance. If a \$key is provided, will return the value of \$key as stored in the cache currently, or false if no value is found.

Examples:

```
$foo = cache('foo');  
$cache = cache();
```

env(\$key[, \$default=null])

- Parameters:**
- **\$key** (*string*) – The name of the environment variable to retrieve
 - **\$default** (*mixed*) – The default value to return if no value is found.
- Returns:** The environment variable, the default value, or null.
- Return type:** mixed

Used to retrieve values that have previously been set to the environment, or return a default value if it is not found. Will format boolean values to actual booleans instead of string representations.

Especially useful when used in conjunction with .env files for setting values that are specific to the environment itself, like database settings, API keys, etc.

esc(\$data, \$context='html'[, \$encoding])

- Parameters:**
- **\$data** (*string|array*) – The information to be escaped.
 - **\$context** (*string*) – The escaping context. Default is 'html'.
 - **\$encoding** (*string*) – The character encoding of the string.
- Returns:** The escaped data.
- Return type:** string

Escapes data for inclusion in web pages, to help prevent XSS attacks. This uses the Zend Escaper library to handle the actual filtering of the data.

If \$data is a string, then it simply escapes and returns it. If \$data is an array, then it loops over it, escaping each 'value' of the key/value pairs.

Valid context values: html, js, css, url, attr, raw, null

helper(\$filename)

Parameters: • **\$filename** (*string|array*) – The name of the helper file to load, or an array of names.

Loads a helper file.

For full details, see the [Helper Functions](#) page.

lang(string \$line[, array \$args]): string

Parameters: • **\$line** (*string*) – The line of text to retrieve
• **\$args** (*array*) – An array of data to substitute for placeholders.

Retrieves a locale-specific file based on an alias string.

For more information, see the [Localization](#) page.

old(\$key[, \$default = null[, \$escape = 'html']])

Parameters: • **\$key** (*string*) – The name of the old form data to check for.
• **\$default** (*mixed*) – The default value to return if \$key doesn't exist.
• **\$escape** (*mixed*) – An [escape](#) context or false to disable it.

Returns: The value of the defined key, or the default value.

Return type: mixed

Provides a simple way to access “old input data” from submitting a form.

Example:

```
// in controller, checking form submittal
if (! $model->save($user))
{
    // 'withInput' is what specifies "old data"
    // should be saved.
    return redirect()->back()->withInput();
}
```

```
// In the view
<input type="email" name="email" value="<?= old('email') ?>">
// Or with arrays
<input type="email" name="user[email]" value="<?= old('user.em
< >
```

Note

If you are using the [form helper](#), this feature is built-in. You only need to use this function when not using the form helper.

`session([$key])`

Parameters: • **\$key** (*string*) – The name of the session item to check for.

Returns: An instance of the Session object if no \$key, the value found in the session for \$key, or null.

Return type: mixed

Provides a convenient way to access the session class and to retrieve a stored value. For more information, see the [Sessions](#) page.

`timer([$name])`

Parameters: • **\$name** (*string*) – The name of the benchmark point.

Returns: The Timer instance

Return type: CodeIgniterDebugTimer

A convenience method that provides quick access to the Timer class. You can pass in the name of a benchmark point as the only parameter. This will start timing from this point, or stop timing if a timer with this name is already running.

Example:

```
// Get an instance
$timer = timer();

// Set timer start and stop points
```

```
timer('controller_loading');    // Will start the timer
timer('controller_loading');    // Will stop the running timer
```

view(\$name[, \$data[, \$options]])

- Parameters:**
- **\$name** (*string*) – The name of the file to load
 - **\$data** (*array*) – An array of key/value pairs to make available within the view.
 - **\$options** (*array*) – An array of options that will be passed to the rendering class.
- Returns:** The output from the view.
- Return type:** string

Grabs the current `RendererInterface`-compatible class and tells it to render the specified view. Simply provides a convenience method that can be used in Controllers, libraries, and routed closures.

Currently, only one option is available for use within the *\$options* array, *saveData* which specifies that data will persistent between multiple calls to *view()* within the same request. By default, the data for that view is forgotten after displaying that single view file.

The *\$option* array is provided primarily to facilitate third-party integrations with libraries like Twig.

Example:

```
$data = ['user' => $user];

echo view('user_profile', $data);
```

For more details, see the [Views](#) page.

Miscellaneous Functions

csrf_token()

- Returns:** The name of the current CSRF token.

Return type: string

Returns the name of the current CSRF token.

csrf_hash()

Returns: The current value of the CSRF hash.

Return type: string

Returns the current CSRF hash value.

csrf_field()

Returns: A string with the HTML for hidden input with all required CSRF information.

Return type: string

Returns a hidden input with the CSRF information already inserted:

```
<input type="hidden" name="{csrf_token}" value="{csrf_hash}">
```

force_https(\$duration = 31536000[, \$request = null[, \$response = null]])

- Parameters:**
- **\$duration** (*int*) – The number of seconds browsers should convert links to this resource to HTTPS.
 - **\$request** (*RequestInterface*) – An instance of the current Request object.
 - **\$response** (*ResponseInterface*) – An instance of the current Response object.

Checks to see if the page is currently being accessed via HTTPS. If it is, then nothing happens. If it is not, then the user is redirected back to the current URI but through HTTPS. Will set the HTTP Strict Transport Security header, which instructs modern browsers to automatically modify any HTTP requests to HTTPS requests for the \$duration.

is_cli()

Returns: TRUE if the script is being executed from the command line or FALSE otherwise.

Return
type: bool

log_message(\$level, \$message[, array \$context])

Parameters:

- **\$level** (*string*) – The level of severity
- **\$message** (*string*) – The message that is to be logged.
- **\$context** (*array*) – An associative array of tags and their values that should be replaced in \$message

Returns: TRUE if was logged succesfully or FALSE if there was a problem logging it

Return
type: bool

Logs a message using the Log Handlers defined in **app/Config/Logger.php**.

Level can be one of the following values: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info**, or **debug**.

Context can be used to substitute values in the message string. For full details, see the [Logging Information](#) page.

redirect(*string \$uri*)

Parameters:

- **\$uri** (*string*) – The URI to redirect the user to.

Returns a RedirectResponse instance allowing you to easily create redirects:

```
// Go back to the previous page  
return redirect()->back();
```

```
// Go to specific UI  
return redirect()->to('/admin');
```

```
// Go to a named/reverse-routed URI  
return redirect()->route('named_route');
```

```
// Keep the old input values upon redirect so they can be used  
return redirect()->back()->withInput();
```

```
// Set a flash message
return redirect()->back()->with('foo', 'message');
```

When passing a URI into the function, it is treated as a reverse-route request, not a relative/full URI, treating it the same as using `redirect()->route()`:

```
// Go to a named/reverse-routed URI
return redirect('named_route');
```

remove_invisible_characters(\$str[, \$url_encoded = TRUE])

Parameters:

- **\$str** (*string*) – Input string
- **\$url_encoded** (*bool*) – Whether to remove URL-encoded characters as well

Returns: Sanitized string

Return type: string

This function prevents inserting NULL characters between ASCII characters, like `Java\0script`.

Example:

```
remove_invisible_characters('Java\\0script');
// Returns: 'Javascript'
```

route_to(\$method[, ...\$params])

Parameters:

- **\$method** (*string*) – The named route alias, or name of the controller/method to match.
- **\$params** (*mixed*) – One or more parameters to be passed to be matched in the route.

Generates a relative URI for you based on either a named route alias, or a `controller::method` combination. Will take parameters into effect, if provided.

For full details, see the [URI Routing](#) page.

service(\$name[, ...\$params])

- **\$name** (*string*) – The name of the service to load
- Parameters:** • **\$params** (*mixed*) – One or more parameters to pass to the service method.
- Returns:** An instance of the service class specified.
- Return type:** mixed

Provides easy access to any of the [Services](#) defined in the system. This will always return a shared instance of the class, so no matter how many times this is called during a single request, only one class instance will be created.

Example:

```
$logger = service('logger');  
$renderer = service('renderer', APPPATH.'views/');
```

single_service(\$name[, ...\$params])

- **\$name** (*string*) – The name of the service to load
- Parameters:** • **\$params** (*mixed*) – One or more parameters to pass to the service method.
- Returns:** An instance of the service class specified.
- Return type:** mixed

Identical to the **service()** function described above, except that all calls to this function will return a new instance of the class, where **service** returns the same instance every time.

stringify_attributes(\$attributes[, \$js])

- **\$attributes** (*mixed*) – string, array of key value pairs, or object
- Parameters:** • **\$js** (*boolean*) – TRUE if values do not need quotes (Javascript-style)
- Returns:** String containing the attribute key/value pairs, comma-separated

**Return
type:** string

Helper function used to convert a string, array, or object of attributes to a string.

Global Constants

The following constants are always available anywhere within your application.

Core Constants

constant **APPPATH**

The path to the **app** directory.

constant **ROOTPATH**

The path to the project root directory. Just above APPPATH.

constant **SYSTEMPATH**

The path to the **system** directory.

constant **FCPATH**

The path to the directory that holds the front controller.

constant **WRITEPATH**

The path to the **writable** directory.

Time Constants

constant **SECOND**

Equals 1.

constant **MINUTE**

Equals 60.

constant **HOURL**

Equals 3600.

constant **DAY**

Equals 86400.

constant **WEEK**

Equals 604800.

constant **MONTH**

Equals 2592000.

constant **YEAR**

Equals 31536000.

constant **DECADE**

Equals 315360000.

Logging Information

- [Configuration](#)
 - [Using Multiple Log Handlers](#)
- [Modifying the Message With Context](#)
- [Using Third-Party Loggers](#)
- [LoggerAware Trait](#)

You can log information to the local log files by using the `log_message()` method. You must supply the “level” of the error in the first parameter, indicating what type of message it is (debug, error, etc). The second parameter is the message itself:

```
if ($some_var == '')
{
    log_message('error', 'Some variable did not contain a val
}
```

There are eight different log levels, matching to the [RFC 5424](#)

[<http://tools.ietf.org/html/rfc5424>] levels, and they are as follows:

- **debug** - Detailed debug information.
- **info** - Interesting events in your application, like a user logging in, logging SQL queries, etc.
- **notice** - Normal, but significant events in your application.
- **warning** - Exceptional occurrences that are not errors, like the user of deprecated APIs, poor use of an API, or other undesirable things that are not necessarily wrong.
- **error** - Runtime errors that do not require immediate action but should typically be logged and monitored.
- **critical** - Critical conditions, like an application component not available, or an unexpected exception.

- **alert** - Action must be taken immediately, like when an entire website is down, the database unavailable, etc.
- **emergency** - The system is unusable.

The logging system does not provide ways to alert sysadmins or webmasters about these events, they solely log the information. For many of the more critical event levels, the logging happens automatically by the Error Handler, described above.

Configuration

You can modify which levels are actually logged, as well as assign different Loggers to handle different levels, within the `/app/Config/Logger.php` configuration file.

The `threshold` value of the config file determines which levels are logged across your application. If any levels are requested to be logged by the application, but the threshold doesn't allow them to log currently, they will be ignored. The simplest method to use is to set this value to the minimum level that you want to have logged. For example, if you want to log debug messages, and not information messages, you would set the threshold to 5. Any log requests with a level of 5 or less (which includes runtime errors, system errors, etc) would be logged and info, notices, and warnings would be ignored:

```
public $threshold = 5;
```

A complete list of levels and their corresponding threshold value is in the configuration file for your reference.

You can pick and choose the specific levels that you would like logged by assigning an array of log level numbers to the threshold value:

```
// Log only debug and info type messages  
public $threshold = [5, 8];
```

Using Multiple Log Handlers

The logging system can support multiple methods of handling logging running at the same time. Each handler can be set to handle specific levels and ignore the rest. Currently, two handlers come with a default install:

- **File Handler** is the default handler and will create a single file for every day locally. This is the recommended method of logging.
- **ChromeLogger Handler** If you have the [ChromeLogger extension](https://craig.is/writing/chrome-logger) [https://craig.is/writing/chrome-logger] installed in the Chrome web browser, you can use this handler to display the log information in Chrome's console window.

The handlers are configured in the main configuration file, in the `$handlers` property, which is simply an array of handlers and their configuration. Each handler is specified with the key being the fully name-spaced class name. The value will be an array of varying properties, specific to each handler. Each handler's section will have one property in common: `handles`, which is an array of log level *names* that the handler will log information for.

```
public $handlers = [  
  
    //-----  
    // File Handler  
    //-----  
  
    'CodeIgniter\Log\Handlers\FileHandler' => [  
        'handles' => ['critical', 'alert', 'emergency', '  
    ]  
];
```



Modifying the Message With Context

You will often want to modify the details of your message based on the context of the event being logged. You might need to log a user id, an IP address, the current POST variables, etc. You can do this by use placeholders in your message. Each placeholder must be wrapped in curly braces. In the third parameter, you must provide an array of placeholder names (without the braces) and their values. These will be inserted into the message string:

```
// Generates a message like: User 123 logged into the system from
$info = [
    'id' => $user->id,
    'ip_address' => $this->request->ip_address()
];

log_message('info', 'User {id} logged into the system from {ip_ad
```

If you want to log an Exception or an Error, you can use the key of 'exception', and the value being the Exception or Error itself. A string will be generated from that object containing the error message, the file name and line number. You must still provide the exception placeholder in the message:

```
try
{
    ... Something throws error here
}
catch (\Exception $e)
{
    log_message('error', '[ERROR] {exception}', ['exception'
}]
```

Several core placeholders exist that will be automatically expanded for you based on the current page request:

Placeholder	Inserted value
{post_vars}	\$_POST variables
{get_vars}	\$_GET variables
{session_vars}	\$_SESSION variables
{env}	Current environment name, i.e. development
{file}	The name of file calling the logger
{line}	The line in {file} where the logger was called
{env:foo}	The value of 'foo' in \$_ENV

Using Third-Party Loggers

You can use any other logger that you might like as long as it extends from

either `Psr\Log\LoggerInterface` and is [PSR3](http://www.php-fig.org/psr/psr-3/) compatible. This means that you can easily drop in use for any PSR3-compatible logger, or create your own.

You must ensure that the third-party logger can be found by the system, by adding it to either the `/app/Config/Autoload.php` configuration file, or through another autoloader, like Composer. Next, you should modify `/app/Config/Services.php` to point the logger alias to your new class name.

Now, any call that is done through the `log_message()` function will use your library instead.

[LoggerAware Trait](#)

If you would like to implement your libraries in a framework-agnostic method, you can use the `CodeIgniter\Log\LoggerAwareTrait` which implements the `setLogger()` method for you. Then, when you use your library under different environments for frameworks, your library should still be able to log as it would expect, as long as it can find a PSR3 compatible logger.

Error Handling

CodeIgniter builds error reporting into your system through Exceptions, both the [SPL collection](#) [<http://php.net/manual/en/spl.exceptions.php>], as well as a few custom exceptions that are provided by the framework. Depending on your environment's setup, the default action when an error or exception is thrown is to display a detailed error report, unless the application is running under the production environment. In this case, a more generic message is displayed to keep the best user experience for your users.

- [Using Exceptions](#)
- [Configuration](#)
 - [Logging Exceptions](#)
- [Custom Exceptions](#)
 - [PageNotFoundException](#)
 - [ConfigException](#)
 - [DatabaseException](#)

[Using Exceptions](#)

This section is a quick overview for newer programmers, or developers who are not experienced with using exceptions.

Exceptions are simply events that happen when the exception is “thrown”. This halts the current flow of the script, and execution is then sent to the error handler which displays the appropriate error page:

```
throw new \Exception("Some message goes here");
```

If you are calling a method that might throw an exception, you can catch that exception using a try/catch block:

```
try {
    $user = $userModel->find($id);
}
catch (\Exception $e)
{
    die($e->getMessage());
}
```

If the `$userModel` throws an exception, it is caught and the code within the catch block is executed. In this example, the script dies, echoing the error message that the `UserModel` defined.

In this example, we catch any type of `Exception`. If we only want to watch for specific types of exceptions, like a `UnknownFileException`, we can specify that in the catch parameter. Any other exceptions that are thrown and are not child classes of the caught exception will be passed on to the error handler:

```
catch (\CodeIgniter\UnknownFileException $e)
{
    // do something here...
}
```

This can be handy for handling the error yourself, or for performing cleanup before the script ends. If you want the error handler to function as normal, you can throw a new exception within the catch block:

```
catch (\CodeIgniter\UnknownFileException $e)
{
    // do something here...

    throw new \RuntimeException($e->getMessage(), $e->getCode
}
```

Configuration

By default, CodeIgniter will display all errors in the development and testing environments, and will not display any errors in the production environment. You can change this by locating the environment configuration portion at the top of the main `index.php` file.

Important

Disabling error reporting DOES NOT stop logs from being written if there are errors.

[Logging Exceptions](#)

By default, all Exceptions other than 404 - Page Not Found exceptions are logged. This can be turned on and off by setting the **\$log** value of Config\Exceptions:

```
class Exceptions
{
    public $log = true;
}
```

To ignore logging on other status codes, you can set the status code to ignore in the same file:

```
class Exceptions
{
    public $ignoredCodes = [ 404 ];
}
```

Note

It is possible that logging still will not happen for exceptions if your current Log settings are not setup to log **critical** errors, which all exceptions are logged as.


[Custom Exceptions](#)

The following custom exceptions are available:

[PageNotFoundException](#)

This is used to signal a 404, Page Not Found error. When thrown, the system will show the view found at `/app/views/errors/html/error_404.php`. You should customize all of the error views for your site. If, in `Config/Routes.php`, you have specified a 404 Override, that will be called instead of the standard 404 page:

```
if (! $page = $pageModel->find($id))
{
    throw \CodeIgniter\Exceptions\PageNotFoundException::forP
}
```



You can pass a message into the exception that will be displayed in place of the default message on the 404 page.

[ConfigException](#)

This exception should be used when the values from the configuration class are invalid, or when the config class is not the right type, etc:

```
throw new \CodeIgniter\Exceptions\ConfigException();
```

This provides an HTTP status code of 500, and an exit code of 3.

[DatabaseException](#)

This exception is thrown for database errors, such as when the database connection cannot be created, or when it is temporarily lost:

```
throw new \CodeIgniter\Database\Exceptions\DatabaseException();
```

This provides an HTTP status code of 500, and an exit code of 8.

Web Page Caching

CodeIgniter lets you cache your pages in order to achieve maximum performance.

Although CodeIgniter is quite fast, the amount of dynamic information you display in your pages will correlate directly to the server resources, memory, and processing cycles utilized, which affect your page load speeds. By caching your pages, since they are saved in their fully rendered state, you can achieve performance much closer to that of static web pages.

How Does Caching Work?

Caching can be enabled on a per-page basis, and you can set the length of time that a page should remain cached before being refreshed. When a page is loaded for the first time, the file will be cached using the currently configured cache engine. On subsequent page loads the cache file will be retrieved and sent to the requesting user's browser. If it has expired, it will be deleted and refreshed before being sent to the browser.

Note

The Benchmark tag is not cached so you can still view your page load speed when caching is enabled.

Enabling Caching

To enable caching, put the following tag in any of your controller methods:

```
$this->cachePage($n);
```

Where \$n is the number of **seconds** you wish the page to remain cached

between refreshes.

The above tag can go anywhere within a method. It is not affected by the order that it appears, so place it wherever it seems most logical to you. Once the tag is in place, your pages will begin being cached.

Important

If you change configuration options that might affect your output, you have to manually delete your cache files.

Note

Before the cache files can be written you must set the cache engine up by editing **app/Config/Cache.php**.

Deleting Caches

If you no longer wish to cache a file you can remove the caching tag and it will no longer be refreshed when it expires.

Note

Removing the tag will not delete the cache immediately. It will have to expire normally.

Code Modules

CodeIgniter supports a form of code modularization to help you create reusable code. Modules are typically centered around a specific subject, and can be thought of as mini-applications within your larger application. Any of the standard file types within the framework are supported, like controllers, models, views, config files, helpers, language files, etc. Modules may contain as few, or as many, of these as you like.

- [Namespaces](#)
- [Auto-Discovery](#)
 - [Enable/Disable Discover](#)
 - [Specify Discovery Items](#)
 - [Discovery and Composer](#)
- [Working With Files](#)
 - [Routes](#)
 - [Controllers](#)
 - [Config Files](#)
 - [Migrations](#)
 - [Seeds](#)
 - [Helpers](#)
 - [Language Files](#)
 - [Libraries](#)
 - [Models](#)
 - [Views](#)

[Namespaces](#)

The core element of the modules functionality comes from the [PSR4-compatible autoloading](#) that CodeIgniter uses. While any code can use the PSR4 autoloader and namespaces, the primary way to take full advantage of

modules is to namespace your code and add it to **app/Config/Autoload.php**, in the psr4 section.

For example, let's say we want to keep a simple blog module that we can re-use between applications. We might create folder with our company name, Acme, to store all of our modules within. We will put it right alongside our **application** directory in the main project root:

```
/acme           // New modules directory
/application
/public
/system
/tests
/writable
```

Open **app/Config/Autoload.php** and add the **Acme** namespace to the psr4 array property:

```
public $psr4 = [
    'Acme' => ROOTPATH.'acme'
];
```

Now that this is setup we can access any file within the **acme** folder through the Acme namespace. This alone takes care of 80% of what is needed for modules to work, so you should be sure to familiarize yourself within namespaces and become comfortable with their use. A number of the file types will be scanned for automatically through all defined namespaces here, making this crucial to working with modules at all.

A common directory structure within a module will mimic the main application folder:

```
/acme
  /Blog
    /Config
    /Controllers
    /Database
      /Migrations
      /Seeds
    /Helpers
    /Language
      /en
```

```
/Libraries  
/Models  
/Views
```

Of course, there is nothing forcing you to use this exact structure, and you should organize it in the manner that best suits your module, leaving out directories you don't need, creating new directories for Entities, Interfaces, or Repositories, etc.

[Auto-Discovery](#)

Many times, you will need to specify the full namespace to files you want to include, but CodeIgniter can be configured to make integrating modules into your applications simpler by automatically discovering many different file types, including:

- [Events](#)
- [Registrars](#)
- [Route files](#)
- [Services](#)

This is configured in the file **app/Config/Modules.php**.

The auto-discovery system works by scanning any psr4 namespaces that have been defined within **Config/Autoload.php** for familiar directories/files.

When at the **acme** namespace above, we would need to make one small adjustment to make it so the files could be found: each “module” within the namespace would have to have it's own namespace defined there. **Acme** would be changed to **AcmeBlog**. Once your module folder has been defined, the discover process would look for a Routes file, for example, at **/acme/Blog/Config/Routes.php**, just as if it was another application.

[Enable/Disable Discover](#)

You can turn on or off all auto-discovery in the system with the **\$enabled** class variable. False will disable all discovery, optimizing performance, but negating the special capabilities of your modules.

[Specify Discovery Items](#)

With the **\$activeExplorers** option, you can specify which items are automatically discovered. If the item is not present, then no auto-discovery will happen for that item, but the others in the array will still be discovered.

[Discovery and Composer](#)

Packages that were installed via Composer will also be discovered by default. This only requires that the namespace that Composer knows about is a PSR4 namespace. PSR0 namespaces will not be detected.

If you do not want all of Composer's known directories to be scanned when locating files, you can turn this off by editing the `$discoverInComposer` variable in `Config\Modules.php`:

```
public $discoverInComposer = false;
```

[Working With Files](#)

This section will take a look at each of the file types (controllers, views, language files, etc) and how they can be used within the module. Some of this information is described in more detail in the relevant location of the user guide, but is being reproduced here so that it's easier to grasp how all of the pieces fit together.

[Routes](#)

By default, [routes](#) are automatically scanned for within modules. It can be turned off in the **Modules** config file, described above.

Note

Since the files are being included into the current scope, the `$routes` instance is already defined for you. It will cause errors if you attempt to redefine that class.

Controllers

Controllers outside of the main **app/Controllers** directory cannot be automatically routed by URI detection, but must be specified within the Routes file itself:

```
// Routes.php
$routes->get('blog', 'Acme\Blog\Controllers\Blog::index');
```

To reduce the amount of typing needed here, the **group** routing feature is helpful:

```
$routes->group('blog', ['namespace' => 'Acme\Blog\Controllers'],
{
    $routes->get('/', 'Blog::index');
});
```



Config Files

No special change is needed when working with configuration files. These are still namespaced classes and loaded with the new command:

```
$config = new \Acme\Blog\Config\Blog();
```

Config files are automatically discovered whenever using the **config()** function that is always available.

Migrations

Migration files will be automatically discovered within defined namespaces. All migrations found across all namespaces will be run every time.

Seeds

Seed files can be used from both the CLI and called from within other seed files as long as the full namespace is provided. If calling on the CLI, you will need to provide double backslashes:

```
> php public/index.php migrations seed Acme\\Blog\\Database\\Seed
```

Helpers

Helpers will be located automatically from defined namespaces when using the `helper()` method, as long as it is within the namespaces **Helpers** directory:

```
helper('blog');
```

Language Files

Language files are located automatically from defined namespaces when using the `lang()` method, as long as the file follows the same directory structures as the main application directory.

Libraries

Libraries are always instantiated by their fully-qualified class name, so no special access is provided:

```
$lib = new \Acme\Blog\Libraries\BlogLib();
```

Models

Models are always instantiated by their fully-qualified class name, so no special access is provided:

```
$model = new \Acme\Blog\Models\PostModel();
```

Views

Views can be loaded using the class namespace as described in the [views](#) documentation:

```
echo view('Acme\Blog\Views\index');
```


Managing your Applications

By default it is assumed that you only intend to use CodeIgniter to manage one application, which you will build in your **application** directory. It is possible, however, to have multiple sets of applications that share a single CodeIgniter installation, or even to rename or relocate your application directory.

Renaming or Relocating the Application Directory

If you would like to rename your application directory or even move it to a different location on your server, other than your project root, open your main **app/Config/Paths.php** and set a *full server path* in the \$appDirectory variable (at about line 38):

```
public $appDirectory = '/path/to/your/application';
```

You will need to modify two additional files in your project root, so that they can find the Paths configuration file:

- /spark runs command line apps; the path is specified on or about line 36:

```
require 'app/Config/Paths.php';  
// ^^^ Change this if you move your application folder
```

- /public/index.php is the front controller for your webapp; the config path is specified on or about line 16:

```
$pathsPath = FCPATH . '../app/Config/Paths.php';  
// ^^^ Change this if you move your application folder
```

Running Multiple Applications with one CodeIgniter Installation

If you would like to share a common CodeIgniter framework installation, to manage several different applications, simply put all of the directories located inside your application directory into their own (sub)-directory.

For example, let's say you want to create two applications, named "foo" and "bar". You could structure your application project directories like this:

```
foo/app, public, tests and writable  
bar/app/, public, tests and writable  
codeigniter/system and docs
```

This would have two apps, "foo" and "bar", both having standard application directories and a public folder, and sharing a common codeigniter framework.

The `index.php` inside each application would refer to its own configuration, `.../app/Config/Paths.php`, and the `$systemDirectory` variable inside each of those would be set to refer to the shared common "system" folder.

If either of the applications had a command-line component, then you would also modify `spark` inside each application's project folder, as directed above.

Handling Multiple Environments

Developers often desire different system behavior depending on whether an application is running in a development or production environment. For example, verbose error output is something that would be useful while developing an application, but it may also pose a security issue when “live”. In development environments, you might want additional tools loaded that you don’t in production environments, etc.

The ENVIRONMENT Constant

By default, CodeIgniter comes with the environment constant set to use the value provided in `$_SERVER['CI_ENVIRONMENT']`, otherwise defaulting to ‘production’. This can be set in several ways depending on your server setup.

.env

The simplest method to set the variable is in your [.env file](#).

```
CI_ENVIRONMENT = development
```

Apache

This server variable can be set in your `.htaccess` file, or Apache config using [SetEnv](#) [https://httpd.apache.org/docs/2.2/mod/mod_env.html#setenv].

```
SetEnv CI_ENVIRONMENT development
```

nginx

Under nginx, you must pass the environment variable through the `fastcgi_params` in order for it to show up under the `$_SERVER` variable. This allows it to work on the virtual-host level, instead of using `env` to set it for the entire server, though that would work fine on a dedicated server. You

would then modify your server config to something like:

```
server {
    server_name localhost;
    include     conf/defaults.conf;
    root        /var/www;

    location    ~* \.php$ {
        fastcgi_param CI_ENVIRONMENT "production";
        include conf/fastcgi-php.conf;
    }
}
```

Alternative methods are available for nginx and other servers, or you can remove this logic entirely and set the constant based on the server's IP address (for instance).

In addition to affecting some basic framework behavior (see the next section), you may use this constant in your own development to differentiate between which environment you are running in.

Boot Files

CodeIgniter requires that a PHP script matching the environment's name is located under **APPPATH/Config/Boot**. These files can contain any customizations that you would like to make for your environment, whether it's updating the error display settings, loading additional developer tools, or anything else. These are automatically loaded by the system. The following files are already created in a fresh install:

- development.php
- production.php
- testing.php

Effects On Default Framework Behavior

There are some places in the CodeIgniter system where the `ENVIRONMENT` constant is used. This section describes how default framework behavior is affected.

Error Reporting

Setting the `ENVIRONMENT` constant to a value of ‘development’ will cause all PHP errors to be rendered to the browser when they occur. Conversely, setting the constant to ‘production’ will disable all error output. Disabling error reporting in production is a [good security practice](#).

Configuration Files

Optionally, you can have CodeIgniter load environment-specific configuration files. This may be useful for managing things like differing API keys across multiple environments. This is described in more detail in the Handling Different Environments section of the [Working with Configuration Files](#) documentation.

Controllers and Routing

Controllers handle incoming requests.

- [Controllers](#)
- [URI Routing](#)
- [Controller Filters](#)
- [HTTP Messages](#)
- [Request Class](#)
- [IncomingRequest Class](#)
- [Content Negotiation](#)
- [HTTP Method Spoofing](#)

Controllers

Controllers are the heart of your application, as they determine how HTTP requests should be handled.

- [What is a Controller?](#)
- [Let's try it: Hello World!](#)
- [Methods](#)
- [Passing URI Segments to your methods](#)
- [Defining a Default Controller](#)
- [Remapping Method Calls](#)
- [Private methods](#)
- [Organizing Your Controllers into Sub-directories](#)
- [Included Properties](#)
 - [helpers](#)
- [Validating \\$_POST data](#)
- [That's it!](#)

What is a Controller?

A Controller is simply a class file that is named in a way that it can be associated with a URI.

Consider this URI:

`example.com/index.php/blog/`

In the above example, CodeIgniter would attempt to find a controller named Blog.php and load it.

When a controller's name matches the first segment of a URI, it will be

loaded.

Let's try it: Hello World!

Let's create a simple controller so you can see it in action. Using your text editor, create a file called Blog.php, and put the following code in it:

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Blog extends Controller
{
    public function index()
    {
        echo 'Hello World!';
    }
}
```

Then save the file to your **/app/Controllers/** directory.

Important

The file must be called 'Blog.php', with a capital 'B'.

Now visit your site using a URL similar to this:

example.com/index.php/blog

If you did it right, you should see:

Hello World!

Important

Class names must start with an uppercase letter.

This is valid:


```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Blog extends Controller {

}
```

This is **not** valid:

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class blog extends Controller {

}
```

Also, always make sure your controller extends the parent controller class so that it can inherit all its methods.

Methods

In the above example the method name is `index()`. The “index” method is always loaded by default if the **second segment** of the URI is empty. Another way to show your “Hello World” message would be this:

`example.com/index.php/blog/index/`

The second segment of the URI determines which method in the controller gets called.

Let’s try it. Add a new method to your controller:

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Blog extends Controller
{
    public function index()
    {
```

```

        echo 'Hello World!';
    }

    public function comments()
    {
        echo 'Look at this!';
    }
}

```

Now load the following URL to see the comment method:

example.com/index.php/blog/comments/

You should see your new message.

Passing URI Segments to your methods

If your URI contains more than two segments they will be passed to your method as parameters.

For example, let's say you have a URI like this:

example.com/index.php/products/shoes/sandals/123

Your method will be passed URI segments 3 and 4 ("sandals" and "123"):

```

<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Products extends Controller
{
    public function shoes($sandals, $id)
    {
        echo $sandals;
        echo $id;
    }
}

```

Important

If you are using the [URI Routing](#) feature, the segments passed to your

method will be the re-routed ones.

Defining a Default Controller

CodeIgniter can be told to load a default controller when a URI is not present, as will be the case when only your site root URL is requested. To specify a default controller, open your **app/Config/Routes.php** file and set this variable:

```
$routes->setDefaultController('Blog');
```

Where ‘Blog’ is the name of the controller class you want used. If you now load your main index.php file without specifying any URI segments you’ll see your “Hello World” message by default.

For more information, please refer to the “Routes Configuration Options” section of the [URI Routing](#) documentation.

Remapping Method Calls

As noted above, the second segment of the URI typically determines which method in the controller gets called. CodeIgniter permits you to override this behavior through the use of the `_remap()` method:

```
public function _remap()  
{  
    // Some code here...  
}
```

Important

If your controller contains a method named `_remap()`, it will **always** get called regardless of what your URI contains. It overrides the normal behavior in which the URI determines which method is called, allowing you to define your own method routing rules.

The overridden method call (typically the second segment of the URI) will be passed as a parameter to the `_remap()` method:

```
public function _remap($method)
{
    if ($method === 'some_method')
    {
        $this->$method();
    }
    else
    {
        $this->default_method();
    }
}
```

Any extra segments after the method name are passed into `_remap()`. These parameters can be passed to the method to emulate CodeIgniter's default behavior.

Example:

```
public function _remap($method, ...$params)
{
    $method = 'process_'. $method;
    if (method_exists($this, $method))
    {
        return $this->$method(...$params);
    }
    throw \CodeIgniter\Exceptions\PageNotFoundException::forP
}
< >
```

Private methods

In some cases you may want certain methods hidden from public access. In order to achieve this, simply declare the method as being private or protected and it will not be served via a URL request. For example, if you were to have a method like this:

```
protected function utility()
{
    // some code
}
```

Trying to access it via the URL, like this, will not work:

```
example.com/index.php/blog/utility/
```

Organizing Your Controllers into Sub-directories

If you are building a large application you might want to hierarchically organize or structure your controllers into sub-directories. CodeIgniter permits you to do this.

Simply create sub-directories under the main *app/Controllers/* one and place your controller classes within them.

Note

When using this feature the first segment of your URI must specify the folder. For example, let's say you have a controller located here:

```
app/Controllers/products/Shoes.php
```

To call the above controller your URI will look something like this:

```
example.com/index.php/products/shoes/show/123
```

Each of your sub-directories may contain a default controller which will be called if the URL contains *only* the sub-directory. Simply put a controller in there that matches the name of your 'default_controller' as specified in your *app/Config/Routes.php* file.

CodeIgniter also permits you to remap your URIs using its [URI Routing](#) feature.

Included Properties

Every controller you create should extend `CodeIgniter\Controller` class. This class provides several features that are available to all of your controllers.

Request Object

The application's main [Request Instance](#) is always available as a class property, `$this->request`.

Response Object

The application's main [Response Instance](#) is always available as a class property, `$this->response`.

Logger Object

An instance of the [Logger](#) class is available as a class property, `$this->logger`.

forceHTTPS

A convenience method for forcing a method to be accessed via HTTPS is available within all controllers:

```
if (! $this->request->isSecure())
{
    $this->forceHTTPS();
}
```

By default, and in modern browsers that support the HTTP Strict Transport Security header, this call should force the browser to convert non-HTTPS calls to HTTPS calls for one year. You can modify this by passing the duration (in seconds) as the first parameter:

```
if (! $this->request->isSecure())
{
    $this->forceHTTPS(31536000);    // one year
}
```

Note

A number of [time-based constants](#) are always available for you to use, including YEAR, MONTH, and more.

helpers

You can define an array of helper files as a class property. Whenever the controller is loaded, these helper files will be automatically loaded into memory so that you can use their methods anywhere inside the controller:

```
namespace App\Controllers;
use CodeIgniter\Controller;


class MyController extends Controller
{
    protected $helpers = ['url', 'form'];
}
```

Validating \$_POST data

The controller also provides a convenience method to make validating `$_POST` data a little simpler, `validate()` that takes an array of rules to test against as the first parameter, and, optionally, an array of custom error messages to display if the items don't pass. Internally, this uses the controller's `$this->request` instance to get the POST data through. The [Validation Library docs](#) has details on the format of the rules and messages arrays, as well as available rules.:

```
public function updateUser(int $userID)
{
    if (! $this->validate([
        'email' => "required|is_unique[users.email,id,{ $userID}]",
        'name'  => 'required|alpha_numeric_spaces'
    ]))
    {
        return view('users/update', [
            'errors' => $this->validator->getErrors()
        ]);
    }

    // do something here if successful...
}
```



If you find it simpler to keep the rules in the configuration file, you can replace the `$rules` array with the name of the group, as defined in

Config\Validation.php:

```
public function updateUser(int $userID)
{
    if (! $this->validate('userRules'))
    {
        return view('users/update', [
            'errors' => $this->validator->getErrors()
        ]);
    }

    // do something here if successful...
}
```

Note

Validation can also be handled automatically in the model. Where you handle validation is up to you, and you will find that some situations are simpler in the controller than then model, and vice versa.

That's it!

That, in a nutshell, is all there is to know about controllers.

URI Routing

- [Setting your own routing rules](#)
- [Placeholders](#)
- [Examples](#)
- [Custom Placeholders](#)
- [Regular Expressions](#)
- [Closures](#)
- [Mapping multiple routes](#)
- [Redirecting Routes](#)
- [Grouping Routes](#)
- [Environment Restrictions](#)
- [Reverse Routing](#)
- [Using Named Routes](#)
- [Using HTTP verbs in routes](#)
- [Command-Line only Routes](#)
- [Resource Routes](#)
 - [Change the Controller Used](#)
 - [Change the Placeholder Used](#)
 - [Limit the Routes Made](#)
- [Global Options](#)
 - [Assigning Namespace](#)
 - [Limit to Hostname](#)
 - [Limit to Subdomains](#)
 - [Offsetting the Matched Parameters](#)
- [Routes Configuration Options](#)
 - [Default Namespace](#)
 - [Default Controller](#)
 - [Default Method](#)
 - [Translate URI Dashes](#)
 - [Use Defined Routes Only](#)
 - [404 Override](#)

Typically there is a one-to-one relationship between a URL string and its corresponding controller class/method. The segments in a URI normally follow this pattern:

```
example.com/class/function/id/
```

In some instances, however, you may want to remap this relationship so that a different class/method can be called instead of the one corresponding to the URL.

For example, let's say you want your URLs to have this prototype:

```
example.com/product/1/  
example.com/product/2/  
example.com/product/3/  
example.com/product/4/
```

Normally the second segment of the URL is reserved for the method name, but in the example above it instead has a product ID. To overcome this, CodeIgniter allows you to remap the URI handler.

[Setting your own routing rules](#)

Routing rules are defined in the **app/config/Routes.php** file. In it you'll see that it creates an instance of the RouteCollection class that permits you to specify your own routing criteria. Routes can be specified using placeholders or Regular Expressions.

A route simply takes the URI on the left, and maps it to the controller and method on the right, along with any parameters that should be passed to the controller. The controller and method should be listed in the same way that you would use a static method, by separating the fully-namespaced class and its method with a double-colon, like `Users::list`. If that method requires parameters to be passed to it, then they would be listed after the method name, separated by forward-slashes:

```
// Calls the $Users->list()  
Users::list
```

```
// Calls $Users->list(1, 23)
Users::list/1/23
```

Placeholders

A typical route might look something like this:

```
$routes->add('product/(:num)', 'App\Catalog::productLookup');
```

In a route, the first parameter contains the URI to be matched, while the second parameter contains the destination it should be re-routed to. In the above example, if the literal word “product” is found in the first segment of the URL, and a number is found in the second segment, the “AppCatalog” class and the “productLookup” method are used instead.

Placeholders are simply strings that represent a Regular Expression pattern. During the routing process, these placeholders are replaced with the value of the Regular Expression. They are primarily used for readability.

The following placeholders are available for you to use in your routes:

- **(:any)** will match all characters from that point to the end of the URI. This may include multiple URI segments.
- **(:segment)** will match any character except for a forward slash (/) restricting the result to a single segment.
- **(:num)** will match any integer.
- **(:alpha)** will match any string of alphabetic characters
- **(:alphanum)** will match any string of alphabetic characters or integers, or any combination of the two.
- **(:hash)** is the same as **:segment**, but can be used to easily see which routes use hashed ids (see the [Model](#) docs).

Note

{locale} cannot be used as a placeholder or other part of the route, as it is reserved for use in [localization](#).

Examples

Here are a few basic routing examples:

```
$routes->add('journals', 'App\Blogs');
```

A URL containing the word “journals” in the first segment will be remapped to the “AppBlogs” class, and the default method, which is usually `index()`:

```
$routes->add('blog/joe', 'Blogs::users/34');
```

A URL containing the segments “blog/joe” will be remapped to the “Blogs” class and the “users” method. The ID will be set to “34”:

```
$routes->add('product/(:any)', 'Catalog::productLookup');
```

A URL with “product” as the first segment, and anything in the second will be remapped to the “Catalog” class and the “productLookup” method:

```
$routes->add('product/(:num)', 'Catalog::productLookupByID/$1');
```

A URL with “product” as the first segment, and a number in the second will be remapped to the “Catalog” class and the “productLookupByID” method passing in the match as a variable to the method.

Important

While the `add()` method is convenient, it is recommended to always use the HTTP-verb-based routes, described below, as it is more secure. It will also provide a slight performance increase, since only routes that match the current request method are stored, resulting in less routes to scan through when trying to find a match.

Custom Placeholders

You can create your own placeholders that can be used in your routes file to fully customize the experience and readability.

You add new placeholders with the `addPlaceholder` method. The first parameter is the string to be used as the placeholder. The second parameter is the Regular Expression pattern it should be replaced with. This must be called before you add the route:

```
$routes->addPlaceholder('uuid', '[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{8}');  
$routes->add('users/(:uuid)', 'Users::show/$1');
```

Regular Expressions

If you prefer you can use regular expressions to define your routing rules. Any valid regular expression is allowed, as are back-references.

Important

Note: If you use back-references you must use the dollar syntax rather than the double backslash syntax. A typical RegEx route might look something like this:

```
$routes->add('products/([a-z]+)/(\d+)', '$1::id_$2');
```

In the above example, a URI similar to `products/shirts/123` would instead call the “Shirts” controller class and the “`id_123`” method.

With regular expressions, you can also catch a segment containing a forward slash (`/`), which would usually represent the delimiter between multiple segments.

For example, if a user accesses a password protected area of your web application and you wish to be able to redirect them back to the same page after they log in, you may find this example useful:

```
$routes->add('login/(.+)', 'Auth::login/$1');
```

For those of you who don’t know regular expressions and want to learn more about them, [regular-expressions.info](http://www.regular-expressions.info/) [http://www.regular-expressions.info/] might be a good starting point.

Important

Note: You can also mix and match wildcards with regular expressions.

Closures

You can use an anonymous function, or Closure, as the destination that a route maps to. This function will be executed when the user visits that URI. This is handy for quickly executing small tasks, or even just showing a simple view:

```
$routes->add('feed', function()  
{  
    $rss = new RSSFeeder();  
    return $rss->feed('general');  
});
```

Mapping multiple routes

While the add() method is simple to use, it is often handier to work with multiple routes at once, using the map() method. Instead of calling the add() method for each route that you need to add, you can define an array of routes and then pass it as the first parameter to the map() method:

```
$routes = [];  
$routes['product/(:num)'] = 'Catalog::productLookupById';  
$routes['product/(:alphanum)'] = 'Catalog::productLookupByName';  
  
$collection->map($routes);
```

Redirecting Routes

Any site that lives long enough is bound to have pages that move. You can specify routes that should redirect to other routes with the addRedirect() method. The first parameter is the URI pattern for the old route. The second parameter is either the new URI to redirect to, or the name of a named route. The third parameter is the HTTP status code that should be sent along with

the redirect. The default value is 302 which is a temporary redirect and is recommended in most cases:

```
$routes->add('users/profile', 'Users::profile', ['as' => 'profile'  
  
// Redirect to a named route  
$routes->addRedirect('users/about', 'profile');  
// Redirect to a URI  
$routes->addRedirect('users/about', 'users/profile');
```

If a redirect route is matched during a page load, the user will be immediately redirected to the new page before a controller can be loaded.

Grouping Routes

You can group your routes under a common name with the `group()` method. The group name becomes a segment that appears prior to the routes defined inside of the group. This allows you to reduce the typing needed to build out an extensive set of routes that all share the opening string, like when building an admin area:

```
$routes->group('admin', function($routes)  
{  
    $routes->add('users', 'Admin\Users::index');  
    $routes->add('blog', 'Admin\Blog::index');  
});
```

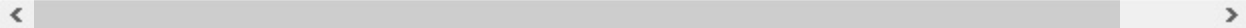
This would prefix the ‘users’ and ‘blog’ URIs with “admin”, handling URLs like `/admin/users` and `/admin/blog`. It is possible to nest groups within groups for finer organization if you need it:

```
$routes->group('admin', function($routes)  
{  
    $routes->group('users', function($routes)  
    {  
        $routes->add('list', 'Admin\Users::list');  
    });  
});
```

This would handle the URL at `admin/users/list`.

If you need to assign options to a group, like a [namespace](#), do it before the callback:

```
$routes->group('api', ['namespace' => 'App\API\v1'], function($ro
{
    $routes->resource('users');
});
```



This would handle a resource route to the App\API\v1\Users controller with the /api/users URI.

You can also use ensure that a specific [filter](#) gets ran for a group of routes. This will always run the filter before or after the controller. This is especially handy during authentication or api logging:

```
$routes->group('api', ['filter' => 'api-auth'], function($routes)
{
    $routes->resource('users');
});
```

The value for the filter must match one of the aliases defined within app/Config/Filters.php.

[Environment Restrictions](#)

You can create a set of routes that will only be viewable under a certain environment. This allows you to create tools that only the developer can use on their local machines that are not reachable on testing or production servers. This can be done with the environment() method. The first parameter is the name of the environment. Any routes defined within this closure are only accessible from the given environment:

```
$routes->environment('development', function($routes)
{
    $routes->add('builder', 'Tools\Builder::index');
});
```

[Reverse Routing](#)

Reverse routing allows you to define the controller and method, as well as any parameters, that a link should go to, and have the router lookup the current route to it. This allows route definitions to change without you having to update your application code. This is typically used within views to create links.

For example, if you have a route to a photo gallery that you want to link to, you can use the `route_to()` helper function to get the current route that should be used. The first parameter is the fully qualified Controller and method, separated by a double colon (`::`), much like you would use when writing the initial route itself. Any parameters that should be passed to the route are passed in next:

```
// The route is defined as:
$routes->add('users/(:id)/gallery(:any)', 'App\Controllers\Galler

// Generate the relative URL to link to user ID 15, gallery 12
// Generates: /users/15/gallery/12
<a href="<?= route_to('App\Controllers\Galleries::showUserGallery
< >
```

Using Named Routes

You can name routes to make your application less fragile. This applies a name to a route that can be called later, and even if the route definition changes, all of the links in your application built with `route_to` will still work without you having to make any changes. A route is named by passing in the `as` option with the name of the route:

```
// The route is defined as:
$routes->add('users/(:id)/gallery(:any)', 'Galleries::showUserGal

// Generate the relative URL to link to user ID 15, gallery 12
// Generates: /users/15/gallery/12
<a href="<?= route_to('user_gallery', 15, 12) ?>">View Gallery</a
< >
```

This has the added benefit of making the views more readable, too.

Using HTTP verbs in routes

It is possible to use HTTP verbs (request method) to define your routing rules. This is particularly useful when building RESTFUL applications. You can use any standard HTTP verb (GET, POST, PUT, DELETE, etc). Each verb has its own method you can use:

```
$routes->get('products', 'Product::feature');
$routes->post('products', 'Product::feature');
$routes->put('products/(:num)', 'Product::feature');
$routes->delete('products/(:num)', 'Product::feature');
```

You can supply multiple verbs that a route should match by passing them in as an array to the match method:

```
$routes->match(['get', 'put'], 'products', 'Product::feature');
```

Command-Line only Routes

You can create routes that work only from the command-line, and are inaccessible from the web browser, with the `cli()` method. This is great for building cronjobs or CLI-only tools. Any route created by any of the HTTP-verb-based route methods will also be inaccessible from the CLI, but routes created by the `any()` method will still be available from the command line:

```
$routes->cli('migrate', 'App\Database::migrate');
```

Resource Routes

You can quickly create a handful of RESTful routes for a single resource with the `resource()` method. This creates the five most common routes needed for full CRUD of a resource: create a new resource, update an existing one, list all of that resource, show a single resource, and delete a single resource. The first parameter is the resource name:

```
$routes->resource('photos');
```

// Equivalent to the following:

```
$routes->get('photos', 'Photos::index');
$routes->get('photos/new', 'Photos::new');
$routes->get('photos/(:segment)/edit', 'Photos::edit/$1');
$routes->get('photos/(:segment)', 'Photos::show/$1');
```

```
$routes->post('photos', 'Photos::create');
$routes->patch('photos/(:segment)', 'Photos::update/$1');
$routes->put('photos/(:segment)', 'Photos::update/$1');
$routes->delete('photos/(:segment)', 'Photos::delete/$1');
```

Important

The routes are matched in the order they are specified, so if you have a resource photos above a get 'photos/poll' the show action's route for the resource line will be matched before the get line. To fix this, move the get line above the resource line so that it is matched first.

The second parameter accepts an array of options that can be used to modify the routes that are generated. While these routes are geared toward API-usage, where more methods are allowed, you can pass in the 'websafe' option to have it generate update and delete methods that work with HTML forms:

```
$routes->resource('photos', ['websafe' => 1]);

// The following equivalent routes are created:
$routes->post('photos/(:segment)', 'Photos::update/$1');
$routes->post('photos/(:segment)/delete', 'Photos::delete/$1');
```

Change the Controller Used

You can specify the controller that should be used by passing in the controller option with the name of the controller that should be used:

```
$routes->resource('photos', ['controller' => 'App\Gallery']);

// Would create routes like:
$routes->get('photos', 'App\Gallery::index');
```

Change the Placeholder Used

By default, the segment placeholder is used when a resource ID is needed. You can change this by passing in the placeholder option with the new string to use:

```
$routes->resource('photos', ['placeholder' => '(:id)']);  
  
// Generates routes like:  
$routes->get('photos/(:id)', 'Photos::show/$1');
```

Limit the Routes Made

You can restrict the routes generated with the `only` option. This should be an array or comma separated list of method names that should be created. Only routes that match one of these methods will be created. The rest will be ignored:

```
$routes->resource('photos', ['only' => ['index', 'show']]);
```

Otherwise you can remove unused routes with the `except` option. This option run after `only`:

```
$routes->resource('photos', ['except' => 'new,edit']);
```

Valid methods are: `index`, `show`, `create`, `update`, `new`, `edit` and `delete`.

Global Options

All of the methods for creating a route (`add`, `get`, `post`, `resource`, etc) can take an array of options that can modify the generated routes, or further restrict them. The `$options` array is always the last parameter:

```
$routes->add('from', 'to', $options);  
$routes->get('from', 'to', $options);  
$routes->post('from', 'to', $options);  
$routes->put('from', 'to', $options);  
$routes->head('from', 'to', $options);  
$routes->options('from', 'to', $options);  
$routes->delete('from', 'to', $options);  
$routes->patch('from', 'to', $options);  
$routes->match(['get', 'put'], 'from', 'to', $options);  
$routes->resource('photos', $options);  
$routes->map($array, $options);  
$routes->group('name', $options, function());
```

Assigning Namespace

While a default namespace will be prepended to the generated controllers (see below), you can also specify a different namespace to be used in any options array, with the namespace option. The value should be the namespace you want modified:

```
// Routes to \Admin\Users::index()  
$routes->add('admin/users', 'Users::index', ['namespace' => 'Admi  
< >
```

The new namespace is only applied during that call for any methods that create a single route, like get, post, etc. For any methods that create multiple routes, the new namespace is attached to all routes generated by that function or, in the case of group(), all routes generated while in the closure.

Limit to Hostname

You can restrict groups of routes to function only in certain domain or sub-domains of your application by passing the “hostname” option along with the desired domain to allow it on as part of the options array:

```
$collection->get('from', 'to', ['hostname' => 'accounts.example.c  
< >
```

This example would only allow the specified hosts to work if the domain exactly matched “accounts.example.com”. It would not work under the main site at “example.com”.

Limit to Subdomains

When the subdomain option is present, the system will restrict the routes to only be available on that sub-domain. The route will only be matched if the subdomain is the one the application is being viewed through:

```
// Limit to media.example.com  
$routes->add('from', 'to', ['subdomain' => 'media']);
```

You can restrict it to any subdomain by setting the value to an asterisk, (*). If you are viewing from a URL that does not have any subdomain present, this will not be matched:

```
// Limit to any sub-domain
$routes->add('from', 'to', ['subdomain' => '*']);
```

Important

The system is not perfect and should be tested for your specific domain before being used in production. Most domains should work fine but some edge case ones, especially with a period in the domain itself (not used to separate suffixes or www) can potentially lead to false positives.

Offsetting the Matched Parameters

You can offset the matched parameters in your route by any numeric value with the offset option, with the value being the number of segments to offset.

This can be beneficial when developing API's with the first URI segment being the version number. It can also be used when the first parameter is a language string:

```
$routes->get('users/(:num)', 'users/show/$1', ['offset' => 1]);

// Creates:
$routes['users/(:num)'] = 'users/show/$2';
```

Routes Configuration Options

The RoutesCollection class provides several options that affect all routes, and can be modified to meet your application's needs. These options are available at the top of `/app/Config/Routes.php`.

Default Namespace

When matching a controller to a route, the router will add the default namespace value to the front of the controller specified by the route. By default, this value is empty, which leaves each route to specify the fully namespaced controller:

```
$routes->setDefaultNamespace('');  
  
// Controller is \Users  
$routes->add('users', 'Users::index');  
  
// Controller is \Admin\Users  
$routes->add('users', 'Admin\Users::index');
```

If your controllers are not explicitly namespaced, there is no need to change this. If you namespace your controllers, then you can change this value to save typing:

```
$routes->setDefaultNamespace('App');  
  
// Controller is \App\Users  
$routes->add('users', 'Users::index');  
  
// Controller is \App\Admin\Users  
$routes->add('users', 'Admin\Users::index');
```

Default Controller

When a user visits the root of your site (i.e. example.com) the controller to use is determined by the value set by the `setDefaultController()` method, unless a route exists for it explicitly. The default value for this is `Home` which matches the controller at `/app/Controllers/Home.php`:

```
// example.com routes to app/Controllers/Welcome.php  
$routes->setDefaultController('Welcome');
```

The default controller is also used when no matching route has been found, and the URI would point to a directory in the controllers directory. For example, if the user visits `example.com/admin`, if a controller was found at `/app/Controllers/admin/Home.php` it would be used.

Default Method

This works similar to the default controller setting, but is used to determine the default method that is used when a controller is found that matches the URI, but no segment exists for the method. The default value is `index`:

```
$routes->setDefaultMethod('listAll');
```

In this example, if the user were to visit `example.com/products`, and a `Products::listAll()` method would be executed.

Translate URI Dashes

This option enables you to automatically replace dashes ('-') with underscores in the controller and method URI segments, thus saving you additional route entries if you need to do that. This is required, because the dash isn't a valid class or method name character and would cause a fatal error if you try to use it:

```
$routes->setTranslateURIDashes(true);
```

Use Defined Routes Only

When no defined route is found that matches the URI, the system will attempt to match that URI against the controllers and methods as described above. You can disable this automatic matching, and restrict routes to only those defined by you, by setting the `setAutoRoute()` option to `false`:

```
$routes->setAutoRoute(false);
```

404 Override

When a page is not found that matches the current URI, the system will show a generic 404 view. You can change what happens by specifying an action to happen with the `set404override()` option. The value can be either a valid class/method pair, just like you would show in any route, or a Closure:

```
// Would execute the show404 method of the App\Errors class
$routes->set404override('App\Errors::show404');

// Will display a custom view
$routes->set404override(function()
{
    echo view('my_errors/not_found.html');
});
```


Controller Filters

- [Creating a Filter](#)
 - [Before Filters](#)
 - [After Filters](#)
- [Configuring Filters](#)
 - [\\$aliases](#)
 - [\\$globals](#)
 - [\\$methods](#)
 - [\\$filters](#)
- [Provided Filters](#)

Controller Filters allow you to perform actions either before or after the controllers execute. Unlike [events](#), you can very simply choose which URI's in your application have the filters applied to them. Incoming filters may modify the Request, while after filters can act on and even modify the Response, allowing for a lot of flexibility and power. Some common examples of tasks that might be performed with filters are:

- Performing CSRF protection on the incoming requests
- Restricting areas of your site based upon their Role
- Perform rate limiting on certain endpoints
- Display a “Down for Maintenance” page
- Perform automatic content negotiation
- and more..

[Creating a Filter](#)

Filters are simple classes that implement `CodeIgniter\Filters\FilterInterface`. They contain two methods: `before()` and `after()` which hold the code that will run before and after the

controller respectively. Your class must contain both methods but may leave the methods empty if they are not needed. A skeleton filter class looks like:


```
<?php namespace App\Filters;

use CodeIgniter\HTTP\RequestInterface;
use CodeIgniter\HTTP\ResponseInterface;
use CodeIgniter\Filters\FilterInterface;

class MyFilter implements FilterInterface
{
    public function before(RequestInterface $request)
    {
        // Do something here
    }

    //-----

    public function after(RequestInterface $request, ResponseInterface $response)
    {
        // Do something here
    }
}
```



Before Filters

From any filter, you can return the \$request object and it will replace the current Request, allowing you to make changes that will still be present when the controller executes.

Since before filters are executed prior to your controller being executed, you may at times want to stop the actions in the controller from happening. You can do this by passing back anything that is not the request object. This is typically used to perform redirects, like in this example:

```
public function before(RequestInterface $request)
{
    $auth = service('auth');

    if (! $auth->isLoggedIn())
    {
        return redirect('login');
    }
}
```

```
}
```

If a Response instance is returned, the Response will be sent back to the client and script execution will stop. This can be useful for implementing rate limiting for API's. See **app/Filters/Throttle.php** for an example.

After Filters

After filters are nearly identical to before filters, except that you can only return the \$response object, and you cannot stop script execution. This does allow you to modify the final output, or simply do something with the final output. This could be used to ensure certain security headers were set the correct way, or to cache the final output, or even to filter the final output with a bad words filter.

Configuring Filters

Once you've created your filters, you need to configure when they get run. This is done in app/Config/Filters.php. This file contains four properties that allow you to configure exactly when the filters run.

\$aliases

The \$aliases array is used to associate a simple name with one or more fully-qualified class names that are the filters to run:

```
public $aliases = [  
    'csrf' => \CodeIgniter\Filters\CSRF::class  
];
```

Aliases are mandatory and if you try to use a full class name later, the system will throw an error. Defining them in this way makes it simple to switch out the class used. Great for when you decided you need to change to a different authentication system since you only change the filter's class and you're done.

You can combine multiple filters into one alias, making complex sets of filters simple to apply:

```
public $aliases = [
    'apiPrep' => [
        \App\Filters\Negotiate::class,
        \App\Filters\ApiAuth::class
    ]
];
```

You should define as many aliases as you need.

[\\$globals](#)

The second section allows you to define any filters that should be applied to every request made by the framework. You should take care with how many you use here, since it could have performance implications to have too many run on every request. Filters can be specified by adding their alias to either the before or after array:

```
public $globals = [
    'before' => [
        'csrf'
    ],
    'after' => []
];
```

There are times where you want to apply a filter to almost every request, but have a few that should be left alone. One common example is if you need to exclude a few URI's from the CSRF protection filter to allow requests from third-party websites to hit one or two specific URI's, while keeping the rest of them protected. To do this, add an array with the 'except' key and a uri to match as the value alongside the alias:

```
public $globals = [
    'before' => [
        'csrf' => ['except' => 'api/*']
    ],
    'after' => []
];
```

Any place you can use a URI in the filter settings, you can use a regular expression or, like in this example, use an asterisk for a wildcard that will match all characters after that. In this example, any URL's starting with api/ would be exempted from CSRF protection, but the site's forms would all be

protected. If you need to specify multiple URI's you can use an array of URI patterns:

```
public $globals = [  
    'before' => [  
        'csrf' => ['except' => ['foo/*', 'bar/*']]  
    ],  
    'after' => []  
];
```

\$methods

You can apply filters to all requests of a certain HTTP method, like POST, GET, PUT, etc. In this array, you would specify the method name in lowercase. It's value would be an array of filters to run. Unlike the `$globals` or the `$filters` properties, these will only run as before filters:

```
public $methods = [  
    'post' => ['foo', 'bar'],  
    'get' => ['baz']  
]
```

In addition to the standard HTTP methods, this also supports two special cases: 'cli', and 'ajax'. The names are self-explanatory here, but 'cli' would apply to all requests that were run from the command line, while 'ajax' would apply to every AJAX request.

\$filters

This property is an array of filter aliases. For each alias you can specify before and after arrays that contain a list of URI patterns that filter should apply to:

```
public filters = [  
    'foo' => ['before' => ['admin/*'], 'after' => ['users/*']],  
    'bar' => ['before' => ['api/*', 'admin/*']]  
];
```

Provided Filters

Three filters are bundled with CodeIgniter4: Honeypot, Security, and DebugToolbar.

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

HTTP Messages

The Message class provides an interface to the portions of an HTTP message that are common to both requests and responses, including the message body, protocol version, utilities for working with the headers, and methods for handling content negotiation.

This class is the parent class that both the [Request Class](#) and the [Response Class](#) extend from. As such, some methods, such as the content negotiation methods, may apply only to a request or response, and not the other one, but they have been included here to keep the header methods together.

What is Content Negotiation?

At its heart Content Negotiation is simply a part of the HTTP specification that allows a single resource to serve more than one type of content, allowing the clients to request the type of data that works best for them.

A classic example of this is a browser that cannot display PNG files can request only GIF or JPEG images. When the `getServer` receives the request, it looks at the available file types the client is requesting and selects the best match from the image formats that it supports, in this case likely choosing a JPEG image to return.

This same negotiation can happen with four types of data:

- **Media/Document Type** - this could be image format, or HTML vs. XML or JSON.
- **Character Set** - The character set the returned document should be set in. Typically is UTF-8.
- **Document Encoding** - Typically the type of compression used on the results.
- **Document Language** - For sites that support multiple languages, this

helps determine which to return.

Class Reference

CodeIgniter\HTTP\Message

body()

Returns: The current message body

Return type: string

Returns the current message body, if any has been set. If not body exists, returns null:

```
echo $message->body();
```

setBody([\$str])

Parameters: • **\$str** (*string*) – The body of the message.

Returns: the Message instance to allow methods to be chained together.

Return type: CodeIgniter\HTTP\Message instance.
Sets the body of the current request.

populateHeaders()

Returns: void

Scans and parses the headers found in the SERVER data and stores it for later access. This is used by the [IncomingRequest Class](#) to make the current request's headers available.

The headers are any SERVER data that starts with HTTP_, like HTTP_HOST. Each message is converted from it's standard uppercase and underscore format to a ucwords and dash format. The preceding HTTP_ is removed from the string. So HTTP_ACCEPT_LANGUAGE becomes Accept-Language.

headers()

Returns: An array of all of the headers found.

Return type: array

Returns an array of all headers found or previously set.

header([*\$name*[, *\$filter* = null]])

Parameters:

- **\$name** (*string*) – The name of the header you want to retrieve the value of.
- **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).

Returns: The current value of the header. If the header has multiple values, they will be returned as an array.

Return type: string|array|null

Allows you to retrieve the current value of a single message header. *\$name* is the case-insensitive header name. While the header is converted internally as described above, you can access the header with any type of case:

```
// These are all the same:
$message->header('HOST');
$message->header('Host');
$message->header('host');
```

If the header has multiple values, the values will return as an array of values. You can use the `headerLine()` method to retrieve the values as a string:

```
echo $message->header('Accept-Language');
```

```
// Outputs something like:
[
    'en',
    'en-US'
]
```

You can filter the header by passing a filter value in as the second parameter:

```
$message->header('Document-URI', FILTER_SANITIZE_URL);
```

headerLine(\$name)

Parameters: • **\$name** (*string*) – The name of the header to retrieve.

Returns: A string representing the header value.

Return type: string

Returns the value(s) of the header as a string. This method allows you to easily get a string representation of the header values when the header has multiple values. The values are appropriately joined:

```
echo $message->headerLine('Accept-Language');
```

```
// Outputs:  
en, en-US
```

setHeader([\$name[, \$value]])

Parameters: • **\$name** (*string*) – The name of the header to set the value for.

• **\$value** (*mixed*) – The value to set the header to.

Returns: The current message instance

Return type: CodeIgniter\HTTP\Message

Sets the value of a single header. \$name is the case-insensitive name of the header. If the header doesn't already exist in the collection, it will be created. The \$value can be either a string or an array of strings:

```
$message->setHeader('Host', 'codeigniter.com');
```

removeHeader([\$name])

Parameters: • **\$name** (*string*) – The name of the header to remove.

Returns: The current message instance

Return type: CodeIgniter\HTTP\Message

Removes the header from the Message. \$name is the case-insensitive

name of the header:

```
$message->remove('Host');
```

appendHeader([\$name[, \$value]])

Parameters:

- **\$name** (*string*) – The name of the header to modify
- **\$value** (*mixed*) – The value to add to the header.

Returns: The current message instance

Return type: CodeIgniter\HTTP\Message

Adds a value to an existing header. The header must already be an array of values instead of a single string. If it is a string then a LogicException will be thrown.

```
$message->appendHeader('Accept-Language', 'en-US; q=0.8');
```

protocolVersion()

Returns: The current HTTP protocol version

Return type: string

Returns the message's current HTTP protocol. If none has been set, will return null. Acceptable values are 1.0 and 1.1.

setProtocolVersion(\$version)

Parameters:

- **\$version** (*string*) – The HTTP protocol version

Returns: The current message instance

Return type: CodeIgniter\HTTP\Message

Sets the HTTP protocol version this Message uses. Valid values are 1.0 or 1.1:

```
$message->setProtocolVersion('1.1');
```

negotiateMedia(\$supported[, \$strictMatch=false])

Parameters:

- **\$supported** (*array*) – An array of media types the application supports
- **\$strictMatch** (*bool*) – Whether it should force an

exact match to happen.

Returns: The supported media type that best matches what is requested.

Return type: string

Parses the Accept header and compares with the application's supported media types to determine the best match. Returns the appropriate media type. The first parameter is an array of application supported media types that should be compared against header values:

```
$supported = [  
    'image/png',  
    'image/jpg',  
    'image/gif'  
];  
$imageType = $message->negotiateMedia($supported);
```

The \$supported array should be structured so that the application's preferred format is the first in the array, with the rest following in descending order of priority. If no match can be made between the header values and the supported values, the first element of the array will be returned.

Per the [RFC](http://tools.ietf.org/html/rfc7231#section-5.3) [http://tools.ietf.org/html/rfc7231#section-5.3] the match has the option of returning a default value, like this method does, or to return an empty string. If you need to have an exact match and would like an empty string returned instead, pass true as the second parameter:

```
// Returns empty string if no match.  
$imageType = $message->negotiateMedia($supported, true);
```

The matching process takes into account the priorities and specificity of the RFC. This means that the more specific header values will have a higher order of precedence, unless modified by a different q value.

For more details, please read the [appropriate section of the RFC](http://tools.ietf.org/html/rfc7231#section-5.3.2)

[http://tools.ietf.org/html/rfc7231#section-5.3.2].

negotiateCharset(\$supported)

Parameters: • **\$supported** (array) – An array of character sets

the application supports.

Returns: The supported character set that best matches what is required..

Return type: string

This is used identically to the `negotiateMedia()` method, except that it matches against the `Accept-Charset` header string:

```
$supported = [  
    'utf-8',  
    'iso-8895-9'  
];  
$charset = $message->negotiateCharset($supported);
```

If no match is found, the system will default to `utf-8`.

negotiateEncoding(*\$supported*)

Parameters: • **\$supported** (*array*) – An array of character encodings the application supports.

Returns: The supported character set that best matches what is required..

Return type: string

Determines the best match between the application-supported values and the `Accept-Encoding` header value. If no match is found, will return the first element of the `$supported` array:

```
$supported = [  
    'gzip',  
    'compress'  
];  
$encoding = $message->negotiateEncoding($supported);
```

negotiateLanguage(*\$supported*)

Parameters: • **\$supported** (*array*) – An array of languages the application supports.

Returns: The supported language that best matches what is required..

Return string
type:

Determines the best match between the application-supported languages and the Accept-Language header value. If no match is found, will return the first element of the \$supported array:

```
$supported = [  
    'en',  
    'fr',  
    'x-pig-latin'  
];  
$language = $message->negotiateLanguage($supported);
```

More information about the language tags are available in [RFC 1766](https://www.ietf.org/rfc/rfc1766.txt)
[<https://www.ietf.org/rfc/rfc1766.txt>].

Request Class

The request class is an object-oriented representation of an HTTP request. This is meant to work for both incoming, such as a request to the application from a browser, and outgoing requests, like would be used to send a request from the application to a third-party application. This class provides the common functionality they both need, but both cases have custom classes that extend from the Request class to add specific functionality.

See the documentation for the [IncomingRequest Class](#) and [CURLRequest Class](#) for more usage details.

Class Reference

CodeIgniter\HTTP\Request

getIPAddress()

The user's IP Address, if it can be detected, or null. If the
Returns: IP address is not a valid IP address, then will return
0.0.0.0

**Return
type:** string

Returns the IP address for the current user. If the IP address is not valid, the method will return '0.0.0.0':

```
echo $request->getIPAddress();
```

Important

This method takes into account the App->proxyIPs setting and will return the reported HTTP_X_FORWARDED_FOR, HTTP_CLIENT_IP, HTTP_X_CLIENT_IP, or HTTP_X_CLUSTER_CLIENT_IP address for the allowed IP address.

isValidIP(\$ip[, \$which = "])

Parameters:

- **\$ip** (*string*) – IP address
- **\$which** (*string*) – IP protocol ('ipv4' or 'ipv6')

Returns: true if the address is valid, false if not

Return type: bool

Takes an IP address as input and returns true or false (boolean) depending on whether it is valid or not.

Note

The `$request->getIPAddress()` method above automatically validates the IP address.

```
if ( ! $request->isValidIP($ip) )
{
    echo 'Not Valid';
}
else
{
    echo 'Valid';
}
```

Accepts an optional second string parameter of 'ipv4' or 'ipv6' to specify an IP format. The default checks for both formats.

getMethod([\$upper = FALSE])

Parameters:

- **\$upper** (*bool*) – Whether to return the request method name in upper or lower case

Returns: HTTP request method

Return type: string

Returns the `$_SERVER['REQUEST_METHOD']`, with the option to set it in uppercase or lowercase.

```
echo $request->getMethod(TRUE); // Outputs: POST
echo $request->getMethod(FALSE); // Outputs: post
```

```
echo $request->getMethod(); // Outputs: post
```

```
getServer([$index = null[, $filter = null[, $flags = null]])
```

- **\$index** (*mixed*) – Value name
- **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
- **\$flags** (*int*) – Flags to apply. A list of flags can be found [here](#).

Parameters: \$_SERVER item value if found, NULL if not

Return
type: mixed

This method is identical to the `post()`, `get()` and `cookie()` methods from the [IncomingRequest Class](#), only it fetches `getServer` data (`$_SERVER`):

```
$request->getServer('some_data');
```

To return an array of multiple `$_SERVER` values, pass all the required keys as an array.

```
$request->getServer(['SERVER_PROTOCOL', 'REQUEST_URI']);
```

IncomingRequest Class

The IncomingRequest class provides an object-oriented representation of an HTTP request from a client, like a browser. It extends from, and has access to all the methods of the [Request](#) and [Message](#) classes, in addition to the methods listed below.

- [Accessing the Request](#)
- [Determining Request Type](#)
- [Retrieving Input](#)
- [Retrieving Headers](#)
- [The Request URL](#)
- [Uploaded Files](#)
- [Content Negotiation](#)
 - [Class Reference](#)

[Accessing the Request](#)

An instance of the request class already populated for you if the current class is a descendant of CodeIgniter\Controller and can be accessed as a class property:

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class UserController extends Controller
{
    public function index()
    {
        if ($this->request->isAJAX())
        {
            . . .
        }
    }
}
```

```

    }
}

```

If you are not within a controller, but still need access to the application's Request object, you can get a copy of it through the [Services class](#):

```
$request = \Config\Services::request();
```

It's preferable, though, to pass the request in as a dependency if the class is anything other than the controller, where you can save it as a class property:

```

<?php
use CodeIgniter\HTTP\RequestInterface;

class SomeClass
{
    protected $request;

    public function __construct(RequestInterface $request)
    {
        $this->request = $request;
    }
}

$someClass = new SomeClass(\Config\Services::request());

```

Determining Request Type

A request could be of several types, including an AJAX request or a request from the command line. This can be checked with the `isAJAX()` and `isCLI()` methods:

```

// Check for AJAX request.
if ($request->isAJAX())
{
    . . .
}

// Check for CLI Request
if ($request->isCLI())
{
    . . .
}

```

You can check the HTTP method that this request represents with the `method()` method:

```
// Returns 'post'
$method = $request->getMethod();
```

By default, the method is returned as a lower-case string (i.e. 'get', 'post', etc). You can get an uppercase version by passing in `true` as the only parameter:

```
// Returns 'GET'
$method = $request->getMethod(true);
```

You can also check if the request was made through an HTTPS connection with the `isSecure()` method:

```
if (! $request->isSecure())
{
    force_https();
}
```

Retrieving Input

You can retrieve input from `$_SERVER`, `$_GET`, `$_POST`, `$_ENV`, and `$_SESSION` through the Request object. The data is not automatically filtered and returns the raw input data as passed in the request. The main advantages to using these methods instead of accessing them directly (`$_POST['something']`), is that they will return null if the item doesn't exist, and you can have the data filtered. This lets you conveniently use data without having to test whether an item exists first. In other words, normally you might do something like this:

```
$something = isset($_POST['foo']) ? $_POST['foo'] : NULL;
```

With CodeIgniter's built in methods you can simply do this:

```
$something = $request->getVar('foo');
```

The `getVar()` method will pull from `$_REQUEST`, so will return any data from `$_GET`, `$_POST`, or `$_COOKIE`. While this is convenient, you will

often need to use a more specific method, like:

- `$request->getGet()`
- `$request->getPost()`
- `$request->getServer()`
- `$request->getCookie()`

In addition, there are a few utility methods for retrieving information from either `$_GET` or `$_POST`, while maintaining the ability to control the order you look for it:

- `$request->getPostGet()` - checks `$_POST` first, then `$_GET`
- `$request->getGetPost()` - checks `$_GET` first, then `$_POST`

Getting JSON data

You can grab the contents of `php://input` as a JSON stream with `getJSON()`.

Note

This has no way of checking if the incoming data is valid JSON or not, you should only use this method if you know that you're expecting JSON.

```
$json = $request->getJSON();
```

By default, this will return any objects in the JSON data as objects. If you want that converted to associative arrays, pass in `true` as the first parameter.

The second and third parameters match up to the `depth` and `options` arguments of the [json_decode](http://php.net/manual/en/function.json-decode.php) [http://php.net/manual/en/function.json-decode.php] PHP function.

Retrieving Raw data (PUT, PATCH, DELETE)

Finally, you can grab the contents of `php://input` as a raw stream with `getRawInput()`:

```
$data = $request->getRawInput();
```

This will retrieve data and convert it to an array. Like this:

```
var_dump($request->getRawInput());  
  
[  
    'Param1' => 'Value1',  
    'Param2' => 'Value2'  
]
```

Filtering Input Data

To maintain security of your application, you will want to filter all input as you access it. You can pass the type of filter to use in as the last parameter of any of these methods. The native `filter_var()` function is used for the filtering. Head over to the PHP manual for a list of [valid filter types](http://php.net/manual/en/filter.filters.php)

[<http://php.net/manual/en/filter.filters.php>].

Filtering a POST variable would look like this:

```
$email = $request->getVar('email', FILTER_SANITIZE_EMAIL);
```

All of the methods mentioned above support the filter type passed in as the last parameter, with the exception of `getJSON()`.

Retrieving Headers

You can get access to any header that was sent with the request with the `getHeaders()` method, which returns an array of all headers, with the key as the name of the header, and the value being an instance of `CodeIgniter\HTTP\Header`:

```
var_dump($request->getHeaders());  
  
[  
    'Host' => CodeIgniter\HTTP\Header,  
    'Cache-Control' => CodeIgniter\HTTP\Header,  
    'Accept' => CodeIgniter\HTTP\Header,  
]
```

If you only need a single header, you can pass the name into the `getHeader()` method. This will grab the specified header object in a case-insensitive

manner if it exists. If not, then it will return null:

```
// these are all equivalent
$host = $request->getHeader('host');
$host = $request->getHeader('Host');
$host = $request->getHeader('HOST');
```

You can always use `hasHeader()` to see if the header existed in this request:

```
if ($request->hasHeader('DNT'))
{
    // Don't track something...
}
```

If you need the value of header as a string with all values on one line, you can use the `getHeaderLine()` method:

```
// Accept-Encoding: gzip, deflate, sdch
echo 'Accept-Encoding: '.$request->getHeaderLine('accept-encoding')
< >
```

If you need the entire header, with the name and values in a single string, simply cast the header as a string:

```
echo (string)$header;
```

The Request URL

You can retrieve a [URI](#) object that represents the current URI for this request through the `$request->uri` property. You can cast this object as a string to get a full URL for the current request:

```
$uri = (string)$request->uri;
```

The object gives you full abilities to grab any part of the request on its own:

```
$uri = $request->uri;

echo $uri->getScheme();           // http
echo $uri->getAuthority();        // snoopy:password@example.com:88
echo $uri->getUserInfo();         // snoopy:password
echo $uri->getHost();             // example.com
```



```
echo $uri->getPort();           // 88
echo $uri->getPath();           // /path/to/page
echo $uri->getQuery();          // foo=bar&bar=baz
echo $uri->getSegments();       // ['path', 'to', 'page']
echo $uri->getSegment(1);       // 'path'
echo $uri->getTotalSegments();  // 3
```

Uploaded Files

Information about all uploaded files can be retrieved through `$request->getFiles()`, which returns a [FileCollection](#) instance. This helps to ease the pain of working with uploaded files, and uses best practices to minimize any security risks.

```
$files = $request->getFiles();

// Grab the file by name given in HTML form
if ($files->hasFile('uploadedFile'))
{
    $file = $files->getFile('uploadedfile');

    // Generate a new secure name
    $name = $file->getRandomName();

    // Move the file to it's new home
    $file->move('/path/to/dir', $name);

    echo $file->getSize('mb');           // 1.23
    echo $file->getExtension();          // jpg
    echo $file->getType();                // image/jpg
}
```

You can also retrieve a single file based on the filename given in the HTML file input:

```
$file = $request->getFile('uploadedfile');
```

Content Negotiation

You can easily negotiate content types with the request through the `negotiate()` method:

```
$language      = $request->negotiate('language', ['en-US', 'en-GB'],  
$imageType     = $request->negotiate('media', ['image/png', 'image/  
$charset       = $request->negotiate('charset', ['UTF-8', 'UTF-16']  
$contentType   = $request->negotiate('media', ['text/html', 'text/x  
$encoding      = $request->negotiate('encoding', ['gzip', 'compress
```

See the [Content Negotiation](#) page for more details.

[Class Reference](#)

Note

In addition to the methods listed here, this class inherits the methods from the [Request Class](#) and the [Message Class](#).

The methods provided by the parent classes that are available are:

- **CodeIgniter\HTTP\Request::getIPAddress()**
- **CodeIgniter\HTTP\Request::validIP()**
- **CodeIgniter\HTTP\Request::getMethod()**
- **CodeIgniter\HTTP\Request::getServer()**
- **CodeIgniter\HTTP\Message::body()**
- **CodeIgniter\HTTP\Message::setBody()**
- **CodeIgniter\HTTP\Message::populateHeaders()**
- **CodeIgniter\HTTP\Message::headers()**
- **CodeIgniter\HTTP\Message::header()**
- **CodeIgniter\HTTP\Message::headerLine()**
- **CodeIgniter\HTTP\Message::setHeader()**
- **CodeIgniter\HTTP\Message::removeHeader()**
- **CodeIgniter\HTTP\Message::appendHeader()**
- **CodeIgniter\HTTP\Message::protocolVersion()**
- **CodeIgniter\HTTP\Message::setProtocolVersion()**
- **CodeIgniter\HTTP\Message::negotiateMedia()**
- **CodeIgniter\HTTP\Message::negotiateCharset()**
- **CodeIgniter\HTTP\Message::negotiateEncoding()**
- **CodeIgniter\HTTP\Message::negotiateLanguage()**

- `CodeIgniter\HTTP\Message::negotiateLanguage()`

`CodeIgniter\HTTP\IncomingRequest`

`isCLI()`

Returns: True if the request was initiated from the command line, otherwise false.

Return type: bool

`isAJAX()`

Returns: True if the request is an AJAX request, otherwise false.

Return type: bool

`isSecure()`

Returns: True if the request is an HTTPS request, otherwise false.

Return type: bool

`getVar([$index = null[, $filter = null[, $flags = null]])`

- Parameters:**
- ***\$index*** (*string*) – The name of the variable/key to look for.
 - ***\$filter*** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
 - ***\$flags*** (*int*) – Flags to apply. A list of flags can be found [here](#).

Returns: `$_REQUEST` if no parameters supplied, otherwise the `REQUEST` value if found, or null if not

Return type: mixed|null

The first parameter will contain the name of the `REQUEST` item you are looking for:

```
$request->getVar('some_data');
```

The method returns null if the item you are attempting to retrieve does not exist.

The second optional parameter lets you run the data through the PHP's filters. Pass in the desired filter type as the second parameter:

```
$request->getVar('some_data', FILTER_SANITIZE_STRING);
```

To return an array of all POST items call without any parameters.

To return all POST items and pass them through the filter, set the first parameter to null while setting the second parameter to the filter you want to use:

```
$request->getVar(null, FILTER_SANITIZE_STRING); // returns
```



To return an array of multiple POST parameters, pass all the required keys as an array:

```
$request->getVar(['field1', 'field2']);
```

Same rule applied here, to retrieve the parameters with filtering, set the second parameter to the filter type to apply:

```
$request->getVar(['field1', 'field2'], FILTER_SANITIZE_STRI
```



getGet(*[\$index = null*, *\$filter = null*, *\$flags = null]*)

- Parameters:**
- **\$index** (*string*) – The name of the variable/key to look for.
 - **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
 - **\$flags** (*int*) – Flags to apply. A list of flags can be found [here](#).

Returns: `$_GET` if no parameters supplied, otherwise the GET value if found, or null if not

Return type: mixed|null

This method is identical to `getVar()`, only it fetches GET data.

getPost(*[\$index = null[, \$filter = null[, \$flags = null]]]*)

- Parameters:**
- **\$index** (*string*) – The name of the variable/key to look for.
 - **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
 - **\$flags** (*int*) – Flags to apply. A list of flags can be found [here](#).
- Returns:** `$_POST` if no parameters supplied, otherwise the POST value if found, or null if not
mixed|null
- Return type:** This method is identical to `getVar()`, only it fetches POST data.

getPostGet(*[\$index = null[, \$filter = null[, \$flags = null]]]*)

- Parameters:**
- **\$index** (*string*) – The name of the variable/key to look for.
 - **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
 - **\$flags** (*int*) – Flags to apply. A list of flags can be found [here](#).
- Returns:** `$_POST` if no parameters supplied, otherwise the POST value if found, or null if not
mixed|null
- Return type:** mixed|null

This method works pretty much the same way as `getPost()` and `getGet()`, only combined. It will search through both POST and GET streams for data, looking first in POST, and then in GET:

```
$request->getPostGet('field1');
```

getGetPost(*[\$index = null[, \$filter = null[, \$flags = null]]]*)

- **\$index** (*string*) – The name of the variable/key to look for.

Parameters:

- **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
- **\$flags** (*int*) – Flags to apply. A list of flags can be found [here](#).

Returns: \$_POST if no parameters supplied, otherwise the POST value if found, or null if not

Return type: mixed|null

This method works pretty much the same way as `getPost()` and `getGet()`, only combined. It will search through both POST and GET streams for data, looking first in GET, and then in POST:

```
$request->getGetPost('field1');
```

getCookie([*\$index* = null[, *\$filter* = null[, *\$flags* = null]]])

Noindex:

Parameters:

- **\$index** (*mixed*) – COOKIE name
- **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
- **\$flags** (*int*) – Flags to apply. A list of flags can be found [here](#).

Returns: \$_COOKIE if no parameters supplied, otherwise the COOKIE value if found or null if not

Return type: mixed

This method is identical to `getPost()` and `getGet()`, only it fetches cookie data:

```
$request->getCookie('some_cookie');  

$request->getCookie('some_cookie', FILTER_SANITIZE_STRING);
```

To return an array of multiple cookie values, pass all the required keys as an array:

```
$request->getCookie(['some_cookie', 'some_cookie2']);
```

Note

Unlike the [Cookie Helper](#) function `get_cookie()`, this method does NOT prepend your configured `$config['cookie_prefix']` value.

getServer(`[$index = null[, $filter = null[, $flags = null]]]`)

- Parameters:**
- **\$index** (*mixed*) – Value name
 - **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
 - **\$flags** (*int*) – Flags to apply. A list of flags can be found [here](#).
- Returns:** `$_SERVER` item value if found, NULL if not
- Return type:** mixed

This method is identical to the `getPost()`, `getGet()` and `getCookie()` methods, only it fetches `getServer` data (`$_SERVER`):

```
$request->getServer('some_data');
```

To return an array of multiple `$_SERVER` values, pass all the required keys as an array.

```
$request->getServer(['SERVER_PROTOCOL', 'REQUEST_URI']);
```

getUserAgent(`[$filter = null]`)

- Parameters:**
- **\$filter** (*int*) – The type of filter to apply. A list of filters can be found [here](#).
- Returns:** The User Agent string, as found in the `SERVER` data, or null if not found.
- Return type:** mixed

This method returns the User Agent string from the `SERVER` data:

```
$request->getUserAgent();
```


Content Negotiation

Content negotiation is a way to determine what type of content to return to the client based on what the client can handle, and what the server can handle. This can be used to determine whether the client is wanting HTML or JSON returned, whether the image should be returned as a jpg or png, what type of compression is supported and more. This is done by analyzing four different headers which can each support multiple value options, each with their own priority. Trying to match this up manually can be pretty challenging. CodeIgniter provides the `Negotiator` class that can handle this for you.

Loading the Class

You can load an instance of the class manually through the Service class:

```
$negotiator = \Config\Services::negotiator();
```

This will grab the current request instance and automatically inject it into the `Negotiator` class.

This class does not need to be loaded on it's own. Instead, it can be accessed through this request's `IncomingRequest` instance. While you cannot access it directly this way, you can easily access all of methods through the `negotiate()` method:

```
$request->negotiate('media', ['foo', 'bar']);
```

When accessed this way, the first parameter is the type of content you're trying to find a match for, while the second is an array of supported values.

Negotiating

In this section we will discuss the 4 types of content that can be negotiated and show how that would look using both of the methods described above to access the negotiator.

Media

The first aspect to look at is handling ‘media’ negotiations. These are provided by the Accept header and is one of the most complex headers available. A common example is the client telling the server what format it wants the data in. This is especially common in API’s. For example, a client might request JSON formatted data from an API endpoint:

```
GET /foo HTTP/1.1
Accept: application/json
```

The server now needs to provide a list of what type of content it can provide. In this example, the API might be able to return data as raw HTML, JSON, or XML. This list should be provided in order of preference:

```
$supported = [
    'application/json',
    'text/html',
    'application/xml'
];

$format = $request->negotiate('media', $supported);
// or
$format = $negotiate->media($supported);
```

In this case, both the client and the server can agree on formatting the data as JSON so ‘json’ is returned from the negotiate method. By default, if no match is found, the first element in the \$supported array would be returned. In some cases, though, you might need to enforce the format to be a strict match. If you pass true as the final value, it will return an empty string if no match is found:

```
$format = $request->negotiate('media', $supported, true);
// or
$format = $negotiate->media($supported, true);
```

Language

Another common usage is to determine the language the content should be served in. If you are running only a single language site, this obviously isn't going to make much difference, but any site that can offer up multiple translations of content will find this useful, since the browser will typically send the preferred language along in the Accept-Language header:

```
GET /foo HTTP/1.1
Accept-Language: fr; q=1.0, en; q=0.5
```

In this example, the browser would prefer French, with a second choice of English. If your website supports English and German you would do something like:

```
$supported = [
    'en',
    'de'
];

$lang = $request->negotiate('language', $supported);
// or
$lang = $negotiate->language($supported);
```

In this example, 'en' would be returned as the current language. If no match is found, it will return the first element in the \$supported array, so that should always be the preferred language.

Encoding

The Accept-Encoding header contains the character sets the client prefers to receive, and is used to specify the type of compression the client supports:

```
GET /foo HTTP/1.1
Accept-Encoding: compress, gzip
```

Your web server will define what types of compression you can use. Some, like Apache, only support **gzip**:

```
$type = $request->negotiate('encoding', ['gzip']);
// or
```

```
$type = $negotiate->encoding(['gzip']);
```

See more at [Wikipedia](https://en.wikipedia.org/wiki/HTTP_compression) [https://en.wikipedia.org/wiki/HTTP_compression].

Character Set

The desired character set is passed through the Accept-Charset header:

```
GET /foo HTTP/1.1  
Accept-Charset: utf-16, utf-8
```

By default, if no matches are found, **utf-8** will be returned:

```
$charset = $request->negotiate('charset', ['utf-8']);  
// or  
$charset = $negotiate->charset(['utf-8']);
```

HTTP Method Spoofing

When working with HTML forms you can only use GET or POST HTTP verbs. In most cases this is just fine. However, to support REST-ful routing you need to support other, more correct, verbs, like DELETE or PUT. Since the browsers don't support this, CodeIgniter provides you with a way to spoof the method that is being used. This allows you to make a POST request, but tell the application that it should be treated as a different request type.

To spoof the method, a hidden input is added to the form with the name of `_method`. It's value is the HTTP verb that you want the request to be:

```
<form action="" method="post">  
  <input type="hidden" name="_method" value="PUT" />  
  
</form>
```

This form is converted into a PUT request and is a true PUT request as far as the routing and the `IncomingRequest` class are concerned.

The form that you are using must be a POST request. GET requests cannot be spoofed.

Note

Be sure to check your web server's configuration as some servers do not support all HTTP verbs with the default configuration, and must have additional packages enabled to work.

Building Responses

View components are used to build what is returned to the user.

- [Views](#)
- [View Cells](#)
- [View Renderer](#)
- [View Layouts](#)
- [View Parser](#)
- [HTTP Responses](#)
- [API Response Trait](#)
- [Localization](#)
- [Alternate PHP Syntax for View Files](#)

Views

- [Creating a View](#)
- [Displaying a View](#)
- [Loading Multiple Views](#)
- [Storing Views within Sub-directories](#)
- [Namespaced Views](#)
- [Caching Views](#)
- [Adding Dynamic Data to the View](#)
- [Creating Loops](#)

A view is simply a web page, or a page fragment, like a header, footer, sidebar, etc. In fact, views can flexibly be embedded within other views (within other views, etc.) if you need this type of hierarchy.

Views are never called directly, they must be loaded by a controller. Remember that in an MVC framework, the Controller acts as the traffic cop, so it is responsible for fetching a particular view. If you have not read the [Controllers](#) page, you should do so before continuing.

Using the example controller you created in the controller page, let's add a view to it.

[Creating a View](#)

Using your text editor, create a file called `BlogView.php` and put this in it:

```
<html>
<head>
    <title>My Blog</title>
</head>
<body>
    <h1>Welcome to my Blog!</h1>
```

```
</body>
</html>
```

Then save the file in your **app/Views** directory.

Displaying a View

To load and display a particular view file you will use the following function:

```
echo view('name');
```

Where *name* is the name of your view file.

Important

If the file extension is omitted, then the views are expected to end with the .php extension.

Now, open the controller file you made earlier called `Blog.php`, and replace the echo statement with the view function:

```
<?php namespace App\Controllers;

class Blog extends \CodeIgniter\Controller
{
    public function index()
    {
        echo view('BlogView');
    }
}
```

If you visit your site using the URL you did earlier you should see your new view. The URL was similar to this:

`example.com/index.php/blog/`

Note

While all of the examples show echo the view directly, you can also return

the output from the view, instead, and it will be appended to any captured output.

Loading Multiple Views

CodeIgniter will intelligently handle multiple calls to `view()` from within a controller. If more than one call happens they will be appended together. For example, you may wish to have a header view, a menu view, a content view, and a footer view. That might look something like this:

```
<?php namespace App\Controllers;

class Page extends \CodeIgniter\Controller
{
    public function index()
    {
        $data = [
            'page_title' => 'Your title'
        ];

        echo view('header');
        echo view('menu');
        echo view('content', $data);
        echo view('footer');
    }
}
```

In the example above, we are using “dynamically added data”, which you will see below.

Storing Views within Sub-directories

Your view files can also be stored within sub-directories if you prefer that type of organization. When doing so you will need to include the directory name loading the view. Example:

```
echo view('directory_name/file_name');
```

Namespaced Views

You can store views under a **View** directory that is namespaced, and load that view as if it was namespaced. While PHP does not support loading non-class files from a namespace, CodeIgniter provides this feature to make it possible to package your views together in a module-like fashion for easy re-use or distribution.

If you have Blog directory that has a PSR-4 mapping setup in the [Autoloader](#) living under the namespace Example\Blog, you could retrieve view files as if they were namespaced also. Following this example, you could load the **BlogView** file from **/blog/views** by prepending the namespace to the view name:

```
echo view('Example\Blog\Views\BlogView');
```

Caching Views

You can cache a view with the view command by passing a cache option with the number of seconds to cache the view for, in the third parameter:

```
// Cache the view for 60 seconds  
echo view('file_name', $data, ['cache' => 60]);
```

By default, the view will be cached using the same name as the view file itself. You can customize this by passing along cache_name and the cache ID you wish to use:

```
// Cache the view for 60 seconds  
echo view('file_name', $data, ['cache' => 60, 'cache_name' => 'my  
< >
```

Adding Dynamic Data to the View

Data is passed from the controller to the view by way of an array in the second parameter of the view function. Here's an example:

```
$data = [  
    'title'    => 'My title',  
    'heading' => 'My Heading',  
    'message' => 'My Message'
```

```
];  
  
echo view('blogview', $data);
```

Let's try it with your controller file. Open it and add this code:

```
<?php namespace App\Controllers;  
  
class Blog extends \CodeIgniter\Controller  
{  
    public function index()  
    {  
        $data['title']    = "My Real Title";  
        $data['heading'] = "My Real Heading";  
  
        echo view('blogview', $data);  
    }  
}
```

Now open your view file and change the text to variables that correspond to the array keys in your data:

```
<html>  
<head>  
    <title><?= $title ?></title>  
</head>  
<body>  
    <h1><?= $heading ?></h1>  
</body>  
</html>
```

Then load the page at the URL you've been using and you should see the variables replaced.

The data passed in is only available during one call to *view*. If you call the function multiple times in a single request, you will have to pass the desired data to each view. This keeps any data from “bleeding” into other views, potentially causing issues. If you would prefer the data to persist, you can pass the *saveData* option into the *\$option* array in the third parameter.

```
$data = [  
    'title'    => 'My title',  
    'heading' => 'My Heading',  
    'message' => 'My Message'
```

```
];
```

```
echo view('blogview', $data, ['saveData' => true]);
```

Additionally, if you would like the default functionality of the view method to be that it does save the data between calls, you can set `$saveData` to **true** in **app/Config/Views.php**.

Creating Loops

The data array you pass to your view files is not limited to simple variables. You can pass multi dimensional arrays, which can be looped to generate multiple rows. For example, if you pull data from your database it will typically be in the form of a multi-dimensional array.

Here's a simple example. Add this to your controller:

```
<?php namespace App\Controllers;

class Blog extends \CodeIgniter\Controller
{
    public function index()
    {
        $data = [
            'todo_list' => ['Clean House', 'Call Mom']
            'title'      => "My Real Title",
            'heading'     => "My Real Heading"
        ];

        echo view('blogview', $data);
    }
}
```

Now open your view file and create a loop:

```
<html>
<head>
    <title><?= $title ?></title>
</head>
<body>
    <h1><?= $heading ?></h1>
```

```
<h3>My Todo List</h3>

<ul>
<?php foreach ($todo_list as $item):?>

    <li><?= $item ?></li>

<?php endforeach;?>
</ul>

</body>
</html>
```

View Cells

View Cells allow you to insert HTML that is generated outside of your controller. It simply calls the specified class and method, which must return a string of valid HTML. This method could be in any callable method, found in any class that the autoloader can locate. The only restriction is that the class can not have any constructor parameters. This is intended to be used within views, and is a great aid to modularizing your code.

```
<?= view_cell('\App\Libraries\Blog::recentPosts') ?>
```

In this example, the class `App\Libraries\Blog` is loaded, and the method `recentPosts()` is run. The method must return the generated HTML as a string. The method can be either a static method or not. Either way works.

Cell Parameters


You can further refine the call by passing a list of parameters in the second parameter to the method. The values passed can be an array of key/value pairs, or a comma-separated string of key/value pairs:

```
// Passing Parameter Array
<?= view_cell('\App\Libraries\Blog::recentPosts', ['category' =>

// Passing Parameter String
<?= view_cell('\App\Libraries\Blog::recentPosts', 'category=codei

public function recentPosts(array $params=[])
{
    $posts = $this->blogModel->where('category', $params['category']
                                ->orderBy('published_on', 'desc')
                                ->limit($params['limit'])
                                ->get();

    return view('recentPosts', ['posts' => $posts]);
}
```



Additionally, you can use parameter names that match the parameter variables in the method for better readability. When you use it this way, all of the parameters must always be specified in the view cell call:

```
<?= view_cell('\App\Libraries\Blog::recentPosts', 'category=codei  
public function recentPosts(int $limit, string $category)  
{  
    $posts = $this->blogModel->where('category', $category)  
        ->orderBy('published_on', 'desc')  
        ->limit($limit)  
        ->get();  
  
    return view('recentPosts', ['posts' => $posts]);  
}
```

Cell Caching

You can cache the results of the view cell call by passing the number of seconds to cache the data for as the third parameter. This will use the currently configured cache engine.

```
// Cache the view for 5 minutes  
<?= view_cell('\App\Libraries\Blog::recentPosts', 'limit=5', 300)
```

You can provide a custom name to use instead of the auto-generated one if you like, by passing the new name as the fourth parameter:

```
// Cache the view for 5 minutes  
<?= view_cell('\App\Libraries\Blog::recentPosts', 'limit=5', 300,
```

View Renderer

- [Using the View Renderer](#)
 - [What It Does](#)
 - [Method Chaining](#)
 - [Escaping Data](#)
 - [View Renderer Options](#)
- [Class Reference](#)

[Using the View Renderer](#)

The `view()` function is a convenience function that grabs an instance of the renderer service, sets the data, and renders the view. While this is often exactly what you want, you may find times where you want to work with it more directly. In that case you can access the View service directly:

```
$view = \Config\Services::renderer();
```

Alternately, if you are not using the view class as your default renderer, you can instantiate it directly:

```
$view = new \CodeIgniter\View\View();
```

Important

You should create services only within controllers. If you need access to the View class from a library, you should set that as a dependency in your library's constructor.

Then you can use any of the three standard methods that it provides: **`render(viewpath, options, save)`**, **`setVar(name, value, context)`** and

setData(data, context).

What It Does

The `view` class processes conventional HTML/PHP scripts stored in the application's view path, after extracting view parameters into PHP variables, accessible inside the scripts. This means that your view parameter names need to be legal PHP variable names.

The `View` class uses an associative array internally, to accumulate view parameters until you call its `render()`. This means that your parameter (or variable) names need to be unique, or a later variable setting will over-ride an earlier one.

This also impacts escaping parameter values for different contexts inside your script. You will have to give each escaped value a unique parameter name.

No special meaning is attached to parameters whose value is an array. It is up to you to process the array appropriately in your PHP code.

Method Chaining

The `setVar()` and `setData()` methods are chainable, allowing you to combine a number of different calls together in a chain:

```
$view->setVar('one', $one)  
->setVar('two', $two)  
->render('myView');
```

Escaping Data

When you pass data to the `setVar()` and `setData()` functions you have the option to escape the data to protect against cross-site scripting attacks. As the last parameter in either method, you can pass the desired context to escape the data for. See below for context descriptions.

If you don't want the data to be escaped, you can pass *null* or *raw* as the final parameter to each function:

```
$view->setVar('one', $one, 'raw');
```

If you choose not to escape data, or you are passing in an object instance, you can manually escape the data within the view with the `esc()` function. The first parameter is the string to escape. The second parameter is the context to escape the data for (see below):


```
<?= \esc($object->getStat()) ?>
```

Escaping Contexts

By default, the `esc()` and, in turn, the `setVar()` and `setData()` functions assume that the data you want to escape is intended to be used within standard HTML. However, if the data is intended for use in Javascript, CSS, or in an href attribute, you would need different escaping rules to be effective. You can pass in the name of the context as the second parameter. Valid contexts are 'html', 'js', 'css', 'url', and 'attr':

```
<a href="<?= esc($url, 'url') ?>" data-foo="<?= esc($bar, 'attr')
<script>
    var siteName = '<?= esc($siteName, 'js') ?>';
</script>

<style>
    body {
        background-color: <?= esc('bgColor', 'css') ?>
    }
</style>
```



[View Renderer Options](#)

Several options can be passed to the `render()` or `renderString()` methods:

- `cache` - the time in seconds, to save a view's results; ignored for `renderString()`
- `cache_name` - the ID used to save/retrieve a cached view result; defaults to the viewpath;
ignored for `renderString()`

- `saveData` - true if the view data parameters should be retained for subsequent calls

Class Reference

CodeIgniter\View\View

render(\$view[, \$options[, \$saveData=false]])

- **\$view** (*string*) – File name of the view source
 - **\$options** (*array*) – Array of options, as key/value pairs
- Parameters:**
- **\$saveData** (*boolean*) – If true, will save data for use with any other calls, if false, will clean the data after rendering the view.
- Returns:** The rendered text for the chosen view
- Return type:** string

Builds the output based upon a file name and any data that has already been set:

```
echo $view->render('myview');
```

renderString(\$view[, \$options[, \$saveData=false]])

- **\$view** (*string*) – Contents of the view to render, for instance content retrieved from a database
 - **\$options** (*array*) – Array of options, as key/value pairs
- Parameters:**
- **\$saveData** (*boolean*) – If true, will save data for use with any other calls, if false, will clean the data after rendering the view.
- Returns:** The rendered text for the chosen view
- Return type:** string

Builds the output based upon a view fragment and any data that has already been set:

```
echo $view->renderString('<div>My Sharona</div>');
```

This could be used for displaying content that might have been stored in a database, but you need to be aware that this is a potential security vulnerability, and that you **must** validate any such data, and probably escape it appropriately!

setData([*\$data*[, *\$context*=null]])

Parameters:

- **\$data** (*array*) – Array of view data strings, as key/value pairs
- **\$context** (*string*) – The context to use for data escaping.

Returns: The Renderer, for method chaining

Return type: CodeIgniter\View\RendererInterface.

Sets several pieces of view data at once:

```
$view->setData(['name'=>'George', 'position'=>'Boss']);
```

Supported escape contexts: html, css, js, url, or attr or raw. If ‘raw’, no escaping will happen.

Each call adds to the array of data that the object is accumulating, until the view is rendered.

setVar(*\$name*[, *\$value*=null[, *\$context*=null]])

Parameters:

- **\$name** (*string*) – Name of the view data variable
- **\$value** (*mixed*) – The value of this view data
- **\$context** (*string*) – The context to use for data escaping.

Returns: The Renderer, for method chaining

Return type: CodeIgniter\View\RendererInterface.

Sets a single piece of view data:

```
$view->setVar('name', 'Joe', 'html');
```

Supported escape contexts: html, css, js, url, attr or raw. If ‘raw’, no

escaping will happen.

If you use the a view data variable that you have previously used for this object, the new value will replace the existing one.

View Layouts

- [Creating A Layout](#)
- [Using Layouts in Views](#)
- [Rendering the View](#)

CodeIgniter supports a simple, yet very flexible, layout system that makes it simple to use one or more base page layouts across your application. Layouts support sections of content that can be inserted from any view being rendered. You could create different layouts to support one-column, two-column, blog archive pages, and more. Layouts are never directly rendered. Instead, you render a view, which specifies the layout that it wants to extend.

[Creating A Layout](#)

Layouts are views like any other. The only difference is their intended usage. Layouts are the only view files that would make use of the `renderSection()` method. This method acts as a placeholder for content.

```
<!doctype html>
<html>
<head>
    <title>My Layout</title>
</head>
<body>
    <?= $this->renderSection('content') ?>
</body>
</html>
```

The `renderSection()` method only has one argument - the name of the section. That way any child views know what to name the content section.

[Using Layouts in Views](#)

Whenever a view wants to be inserted into a layout, it must use the `extend()` method at the top of the file:

```
<?= $this->extend('default') ?>
```

The `extend` method takes the name of any view file that you wish to use. Since they are standard views, they will be located just like a view. By default, it will look in the application's View directory, but will also scan other PSR-4 defined namespaces. You can include a namespace to locate the view in particular namespace View directory:

```
<?= $this->extend('Blog\Views\default') ?>
```

All content within a view that extends a layout must be included within `section($name)` and `endSection()` method calls. Any content between these calls will be inserted into the layout wherever the `renderSection($name)` call that matches the section name exists.:

```
<?= $this->extend('default') ?>

<?= $this->section('content') ?>
    <h1>Hello World!</h1>
<?= $this->endSection() ?>
```

The `endSection()` does not need the section name. It automatically knows which one to close.

Rendering the View

Rendering the view and its layout is done exactly as any other view would be displayed within a controller:

```
public function index()
{
    echo view('some_view');
}
```

The renderer is smart enough to detect whether the view should be rendered on its own, or if it needs a layout.

View Parser

- [Using the View Parser Class](#)
 - [What It Does](#)
 - [Parser templates](#)
 - [Parser Configuration Options](#)
- [Substitution Variations](#)
 - [Loop Substitutions](#)
 - [Nested Substitutions](#)
 - [Comments](#)
 - [Cascading Data](#)
 - [Preventing Parsing](#)
 - [Conditional Logic](#)
 - [Escaping Data](#)
 - [Filters](#)
 - [Parser Plugins](#)
- [Usage Notes](#)
 - [View Fragments](#)
- [Class Reference](#)

The View Parser can perform simple text substitution for pseudo-variables contained within your view files. It can parse simple variables or variable tag pairs.

Pseudo-variable names or control constructs are enclosed in braces, like this:

```
<html>
<head>
    <title>{blog_title}</title>
</head>
<body>
    <h3>{blog_heading}</h3>

    {blog_entries}
```

```
        <h5>{title}</h5>
        <p>{body}</p>
    {/blog_entries}

</body>
</html>
```

These variables are not actual PHP variables, but rather plain text representations that allow you to eliminate PHP from your templates (view files).

Note

CodeIgniter does **not** require you to use this class since using pure PHP in your view pages (for instance using the [View renderer](#)) lets them run a little faster. However, some developers prefer to use some form of template engine if they work with designers who they feel would find some confusion working with PHP.

Using the View Parser Class

The simplest method to load the parser class is through its service:

```
$parser = \Config\Services::parser();
```

Alternately, if you are not using the Parser class as your default renderer, you can instantiate it directly:

```
$parser = new \CodeIgniter\View\Parser();
```

Then you can use any of the three standard rendering methods that it provides: **render(viewpath, options, save)**, **setVar(name, value, context)** and **setData(data, context)**. You will also be able to specify delimiters directly, through the **setDelimiters(left,right)** method.

Using the Parser, your view templates are processed only by the Parser itself, and not like a conventional view PHP script. PHP code in such a script is ignored by the parser, and only substitutions are performed.

This is purposeful: view files with no PHP.

What It Does

The Parser class processes “PHP/HTML scripts” stored in the application’s view path. These scripts have a .php extension, but can not contain any PHP.

Each view parameter (which we refer to as a pseudo-variable) triggers a substitution, based on the type of value you provided for it. Pseudo-variables are not extracted into PHP variables; instead their value is accessed through the pseudo-variable syntax, where its name is referenced inside braces.

The Parser class uses an associative array internally, to accumulate pseudo-variable settings until you call its render(). This means that your pseudo-variable names need to be unique, or a later parameter setting will over-ride an earlier one.

This also impacts escaping parameter values for different contexts inside your script. You will have to give each escaped value a unique parameter name.

Parser templates

You can use the render() method to parse (or render) simple templates, like this:

```
$data = [  
    'blog_title'    => 'My Blog Title',  
    'blog_heading' => 'My Blog Heading'  
];  
  
echo $parser->setData($data)  
    ->render('blog_template');
```

View parameters are passed to setData() as an associative array of data to be replaced in the template. In the above example, the template would contain two variables: {blog_title} and {blog_heading} The first parameter to render() contains the name of the [view file](#) (in this example the file would be called blog_template.php),

Parser Configuration Options

Several options can be passed to the `render()` or `renderString()` methods.

- `cache` - the time in seconds, to save a view's results; ignored for `renderString()`
- `cache_name` - the ID used to save/retrieve a cached view result; defaults to the viewpath;
ignored for `renderString()`
- `saveData` - true if the view data parameters should be retained for subsequent calls;
default is **false**
- `cascadeData` - true if pseudo-variable settings should be passed on to nested
substitutions; default is **true**

```
echo $parser->render('blog_template', [  
    'cache' => HOUR,  
    'cache_name' => 'something_unique',  
]);
```

Substitution Variations

There are three types of substitution supported: simple, looping, and nested. Substitutions are performed in the same sequence that pseudo-variables were added.

The **simple substitution** performed by the parser is a one-to-one replacement of pseudo-variables where the corresponding data parameter has either a scalar or string value, as in this example:

```
$template = '<head><title>{blog_title}</title></head>';  
$data     = ['blog_title' => 'My ramblings'];  
  
echo $parser->setData($data)->renderString($template);  
  
// Result: <head><title>My ramblings</title></head>
```

The Parser takes substitution a lot further with “variable pairs”, used for nested substitutions or looping, and with some advanced constructs for conditional substitution.

When the parser executes, it will generally

- handle any conditional substitutions
- handle any nested/looping substitutions
- handle the remaining single substitutions

Loop Substitutions

A loop substitution happens when the value for a pseudo-variable is a sequential array of arrays, like an array of row settings.

The above example code allows simple variables to be replaced. What if you would like an entire block of variables to be repeated, with each iteration containing new values? Consider the template example we showed at the top of the page:

```
<html>
<head>
  <title>{blog_title}</title>
</head>
<body>
  <h3>{blog_heading}</h3>

  {blog_entries}
    <h5>{title}</h5>
    <p>{body}</p>
  {/blog_entries}

</body>
</html>
```

In the above code you’ll notice a pair of variables: {blog_entries} data... {/blog_entries}. In a case like this, the entire chunk of data between these pairs would be repeated multiple times, corresponding to the number of rows in the “blog_entries” element of the parameters array.

Parsing variable pairs is done using the identical code shown above to parse

single variables, except, you will add a multi-dimensional array corresponding to your variable pair data. Consider this example:

```
$data = [  
    'blog_title'    => 'My Blog Title',  
    'blog_heading' => 'My Blog Heading',  
    'blog_entries' => [  
        ['title' => 'Title 1', 'body' => 'Body 1'],  
        ['title' => 'Title 2', 'body' => 'Body 2'],  
        ['title' => 'Title 3', 'body' => 'Body 3'],  
        ['title' => 'Title 4', 'body' => 'Body 4'],  
        ['title' => 'Title 5', 'body' => 'Body 5']  
    ]  
];  
  
echo $parser->setData($data)  
    ->render('blog_template');
```

The value for the pseudo-variable `blog_entries` is a sequential array of associative arrays. The outer level does not have keys associated with each of the nested “rows”.

If your “pair” data is coming from a database result, which is already a multi-dimensional array, you can simply use the database `getResultArray()` method:

```
$query = $db->query("SELECT * FROM blog");  
  
$data = [  
    'blog_title'    => 'My Blog Title',  
    'blog_heading' => 'My Blog Heading',  
    'blog_entries' => $query->getResultArray()  
];  
  
echo $parser->setData($data)  
    ->render('blog_template');
```

If the array you are trying to loop over contains objects instead of arrays, the parser will first look for an `asArray` method on the object. If it exists, that method will be called and the resulting array is then looped over just as described above. If no `asArray` method exists, the object will be cast as an array and its public properties will be made available to the Parser.

This is especially useful with the Entity classes, which has an `asArray` method that returns all public and protected properties (minus the `_options` property) and makes them available to the Parser.

Nested Substitutions

A nested substitution happens when the value for a pseudo-variable is an associative array of values, like a record from a database:

```
$data = [  
    'blog_title'    => 'My Blog Title',  
    'blog_heading' => 'My Blog Heading',  
    'blog_entry'    => [  
        'title' => 'Title 1', 'body' => 'Body 1'  
    ]  
];  
  
echo $parser->setData($data)  
    ->render('blog_template');
```

The value for the pseudo-variable `blog_entry` is an associative array. The key/value pairs defined inside it will be exposed inside the variable pair loop for that variable.

A `blog_template` that might work for the above:

```
<h1>{blog_title} - {blog_heading}</h1>  
{blog_entry}  
    <div>  
        <h2>{title}</h2>  
        <p>{body}</p>  
    </div>  
{/blog_entry}
```

If you would like the other pseudo-variables accessible inside the “`blog_entry`” scope, then make sure that the “`cascadeData`” option is set to `true`.

Comments

You can place comments in your templates that will be ignored and removed

during parsing by wrapping the comments in a {# #} symbols.

```
{# This comment is removed during parsing. #}  
{blog_entry}  
    <div>  
        <h2>{title}</h2>  
        <p>{body}</p>  
    </div>  
{/blog_entry}
```

Cascading Data

With both a nested and a loop substitution, you have the option of cascading data pairs into the inner substitution.

The following example is not impacted by cascading:

```
$template = '{name} lives in {location}{city} on {planet}'  
$data = [  
    'name' => 'George',  
    'location' => [ 'city' => 'Red City', 'planet' => 'Mars'  
];  
  
echo $parser->setData($data)->renderString($template);  
// Result: George lives in Red City on Mars.
```

This example gives different results, depending on cascading:

```
$template = '{location}{name} lives in {city} on {planet}'  
$data = [  
    'name' => 'George',  
    'location' => [ 'city' => 'Red City', 'planet' => 'Mars'  
];  
  
echo $parser->setData($data)->renderString($template, ['cascadeDa  
// Result: {name} lives in Red City on Mars.  
  
echo $parser->setData($data)->renderString($template, ['cascadeDa  
// Result: George lives in Red City on Mars.
```


Preventing Parsing

You can specify portions of the page to not be parsed with the {noparse} {/noparse} tag pair. Anything in this section will stay exactly as it is, with no variable substitution, looping, etc, happening to the markup between the brackets.

```
{noparse}
    <h1>Untouched Code</h1>
{/noparse}
```

Conditional Logic

The Parser class supports some basic conditionals to handle if, else, and elseif syntax. All if blocks must be closed with an endif tag:

```
{if $role=='admin'}
    <h1>Welcome, Admin!</h1>
{endif}
```

This simple block is converted to the following during parsing:

```
<?php if ($role=='admin'): ?>
    <h1>Welcome, Admin!</h1>
<?php endif ?>
```

All variables used within if statements must have been previously set with the same name. Other than that, it is treated exactly like a standard PHP conditional, and all standard PHP rules would apply here. You can use any of the comparison operators you would normally, like ==, ===, !==, <, >, etc.

```
{if $role=='admin'}
    <h1>Welcome, Admin</h1>
{elseif $role=='moderator'}
    <h1>Welcome, Moderator</h1>
{else}
    <h1>Welcome, User</h1>
{endif}
```

Note

In the background, conditionals are parsed using an **eval()**, so you must ensure that you take care with the user data that is used within conditionals, or you could open your application up to security risks.

Escaping Data

By default, all variable substitution is escaped to help prevent XSS attacks on your pages. CodeIgniter's **esc** method supports several different contexts, like general **html**, when it's in an HTML **attr***, in ****css**, etc. If nothing else is specified, the data will be assumed to be in an HTML context. You can specify the context used by using the **esc** filter:

```
{ user_styles | esc(css) }  
<a href="{ user_link | esc(attr) }">{ title }</a>
```

There will be times when you absolutely need something to be used and NOT escaped. You can do this by adding exclamation marks to the opening and closing braces:

```
{! unescaped_var !}
```

Filters

Any single variable substitution can have one or more filters applied to it to modify the way it is presented. These are not intended to drastically change the output, but provide ways to reuse the same variable data but with different presentations. The **esc** filter discussed above is one example. Dates are another common use case, where you might need to format the same data differently in several sections on the same page.

Filters are commands that come after the pseudo-variable name, and are separated by the pipe symbol, |:

```
// -55 is displayed as 55  
{ value|abs }
```

If the parameter takes any arguments, they must be separated by commas and enclosed in parentheses:

```
{ created_at|date(Y-m-d) }
```

Multiple filters can be applied to the value by piping multiple ones together. They are processed in order, from left to right:

```
{ created_at|date_modify(+5 days)|date(Y-m-d) }
```

Provided Filters

The following filters are available when using the parser:

Filter	Arguments	Description	Example
abs		Displays the absolute value of a number.	{ v abs }
capitalize		Displays the string in sentence case: all lowercase with firstletter capitalized.	{ v capitalize }
date	format (Y-m-d)	A PHP date -compatible formatting string.	{ v date(Y-m-d) }
date_modify	value to add / subtract	A strtotime compatible string to modify the date, like +5 day or -1 week.	{ v date_modify(+1 day) }
default	default value	Displays the default value if the variable is empty or undefined.	{ v default(just in case) }
esc	html, attr, css, js	Specifies the context to escape the data.	{ v esc(attr) }
Returns the text			

excerpt	phrase, radius	within a radius of words from a given phrase. Same as excerpt helper function.	{ v excerpt(green giant, 2
highlight	phrase	Highlights a given phrase within the text using ‘<mark> </mark>’ tags.	{ v highlight(view parser
highlight_code		Highlights code samples with HTML/CSS.	{ v highlight_code }
limit_chars	limit	Limits the number of chracters to \$limit.	{ v limit_chars(100) }
limit_words	limit	Limits the number of words to \$limit.	{ v limit_words(20) }
local_currency	currency, locale	Displays a localized version of a currency. “currency” value is any 3-letter ISO 4217 currency code.	{ v local_currency(EUR,
local_number	type, precision, locale	Displays a localized version of a number. “type” can be one of: decimal, currency, percent, scientific, spellout, ordinal, duration.	{ v local_number(decimal, }
lower		Converts a string to lowercase.	{ v lower }
nl2br		Replaces all newline characters	{ v nl2br }

		(n) to an HTML tag.	
number_format	places	Wraps PHP number_format function for use within the parser.	{ v number_format(3) }
prose		Takes a body of text and uses the auto_typography() method to turn it into prettier, easier- to-read, prose.	{ v prose }
round	places, type	Rounds a number to the specified places. Types of ceil and floor can be passed to use those functions instead.	{ v round(3) } { v round(
strip_tags	allowed chars	Wraps PHP strip_tags . Can accept a string of allowed tags.	{ v strip_tags() }
title		Displays a “title case” version of the string, with all lowercase, and each word capitalized.	{ v title }
upper		Displays the string in all uppercase.	{ v upper }

See [PHP’s NumberFormatter](http://php.net/manual/en/numberformatter.create.php) [http://php.net/manual/en/numberformatter.create.php] for details relevant to the “local_number” filter.

Custom Filters

You can easily create your own filters by editing **app/Config/View.php** and adding new entries to the `$filters` array. Each key is the name of the filter is called by in the view, and its value is any valid PHP callable:

```
public $filters = [  
    'abs'          => '\CodeIgniter\View\Filters::abs',  
    'capitalize' => '\CodeIgniter\View\Filters::capitalize',  
];
```

PHP Native functions as Filters

You can easily use native php function as filters by editing **app/Config/View.php** and adding new entries to the `$filters` array. Each key is the name of the native PHP function is called by in the view, and its value is any valid native PHP function prefixed with:

```
public $filters = [  
    'str_repeat' => '\str_repeat',  
];
```

Parser Plugins

Plugins allow you to extend the parser, adding custom features for each project. They can be any PHP callable, making them very simple to implement. Within templates, plugins are specified by `{+ +}` tags:

```
{+ foo +} inner content {+ /foo +}
```

This example shows a plugin named **foo**. It can manipulate any of the content between its opening and closing tags. In this example, it could work with the text " inner content ". Plugins are processed before any pseudo-variable replacements happen.

While plugins will often consist of tag pairs, like shown above, they can also be a single tag, with no closing tag:

```
{+ foo +}
```

Opening tags can also contain parameters that can customize how the plugin works. The parameters are represented as key/value pairs:

```
{+ foo bar=2 baz="x y" }
```

Parameters can also be single values:

```
{+ include somefile.php +}
```

Provided Plugins

The following plugins are available when using the parser:

Plugin	Arguments	Description	Example
current_url		Alias for the current_url helper function.	{+ current_url +}
previous_url		Alias for the previous_url helper function.	{+ previous_url +}
site_url		Alias for the site_url helper function.	{+ site_url "login" +}
mailto	email, title, attributes	Alias for the mailto helper function.	{+ mailto email=foo@example.c title="Stranger Things" +}
safe_mailto	email, title, attributes	Alias for the safe_mailto helper function.	{+ safe_mailto email=foo@example.c title="Stranger Things" +}
lang	language string	Alias for the lang helper function.	{+ lang number.terabyteAbbr +}

validation_errors	fieldname(optional)	Returns either error string for the field (if specified) or all validation errors.	{+ validation_errors +} {+ validation_errors field="email" +}
route	route name	Alias for the route_to helper function.	{+ route "login" +}

Registering a Plugin

At its simplest, all you need to do to register a new plugin and make it ready for use is to add it to the **app/Config/View.php**, under the **\$plugins** array. The key is the name of the plugin that is used within the template file. The value is any valid PHP callable, including static class methods, and closures:

```
public $plugins = [
    'foo' => '\Some\Class::methodName',
    'bar' => function($str, array $params=[]) {
        return $str;
    },
];
```

If the callable is on its own, it is treated as a single tag, not a open/close one. It will be replaced by the return value from the plugin:

```
public $plugins = [
    'foo' => '\Some\Class::methodName'
];
```

```
// Tag is replaced by the return value of Some\Class::methodName
{+ foo +}
```

If the callable is wrapped in an array, it is treated as an open/close tag pair that can operate on any of the content between its tags:


```
public $plugins = [
    'foo' => ['\Some\Class::methodName']
];

{+ foo +} inner content {+ /foo +}
```

Usage Notes

If you include substitution parameters that are not referenced in your template, they are ignored:

```
$template = 'Hello, {firstname} {lastname}';
$data = [
    'title' => 'Mr',
    'firstname' => 'John',
    'lastname' => 'Doe'
];
echo $parser->setData($data)
    ->renderString($template);
```

// Result: Hello, John Doe

If you do not include a substitution parameter that is referenced in your template, the original pseudo-variable is shown in the result:

```
$template = 'Hello, {firstname} {initials} {lastname}';
$data = [
    'title' => 'Mr',
    'firstname' => 'John',
    'lastname' => 'Doe'
];
echo $parser->setData($data)
    ->renderString($template);
```

// Result: Hello, John {initials} Doe

If you provide a string substitution parameter when an array is expected, i.e. for a variable pair, the substitution is done for the opening variable pair tag, but the closing variable pair tag is not rendered properly:

```
$template = 'Hello, {firstname} {lastname} ({degrees}{degree} {/d
$data = [
    'degrees' => 'Mr',
    'firstname' => 'John',
```

```

        'lastname' => 'Doe',
        'titles' => [
            ['degree' => 'BSc'],
            ['degree' => 'PhD']
        ]
    ];
    echo $parser->setData($data)
        ->renderString($template);

// Result: Hello, John Doe (Mr{degree} {/degrees})

```

View Fragments

You do not have to use variable pairs to get the effect of iteration in your views. It is possible to use a view fragment for what would be inside a variable pair, and to control the iteration in your controller instead of in the view.

An example with the iteration controlled in the view:

```

$template = '<ul>{menuitems}
    <li><a href="{link}">{title}</a></li>
{/menuitems}</ul>';

$data = [
    'menuitems' => [
        ['title' => 'First Link', 'link' => '/first'],
        ['title' => 'Second Link', 'link' => '/second'],
    ]
];
echo $parser->setData($data)
    ->renderString($template);

```

Result:

```

<ul>
    <li><a href="/first">First Link</a></li>
    <li><a href="/second">Second Link</a></li>
</ul>

```

An example with the iteration controlled in the controller, using a view fragment:

```

$temp = '';
$template1 = '<li><a href="{link}">{title}</a></li>';
$data1 = [
    ['title' => 'First Link', 'link' => '/first'],
    ['title' => 'Second Link', 'link' => '/second'],
];

foreach ($data1 as $menuitem)
{
    $temp .= $parser->setData($menuItem)->renderString();
}

$template = '<ul>{menuitems}</ul>';
$data = [
    'menuitems' => $temp
];
echo $parser->setData($data)
    ->renderString($template);

```

Result:

```

<ul>
    <li><a href="/first">First Link</a></li>
    <li><a href="/second">Second Link</a></li>
</ul>

```

[Class Reference](#)

CodeIgniter\View\Parser

render(\$view[, \$options[, \$saveData=false]])

- **\$view** (*string*) – File name of the view source
- **\$options** (*array*) – Array of options, as key/value pairs
- **\$saveData** (*boolean*) – If true, will save data for use with any other calls, if false, will clean the data after rendering the view.

Parameters:

Returns: The rendered text for the chosen view

Return type: string

Builds the output based upon a file name and any data that has already been set:

```
echo $parser->render('myview');
```

Options supported:

- `cache` - the time in seconds, to save a view's results
- `cache_name` - the ID used to save/retrieve a cached view result; defaults to the viewpath
- `cascadeData` - true if the data pairs in effect when a nested or loop substitution occurs should be propagated
- `saveData` - true if the view data parameter should be retained for subsequent calls
- `leftDelimiter` - the left delimiter to use in pseudo-variable syntax
- `rightDelimiter` - the right delimiter to use in pseudo-variable syntax

Any conditional substitutions are performed first, then remaining substitutions are performed for each data pair.

renderString(\$template[, \$options[, \$saveData=false]])

- Parameters:**
- **\$template** (*string*) – View source provided as a string
 - **\$options** (*array*) – Array of options, as key/value pairs
 - **\$saveData** (*boolean*) – If true, will save data for use with any other calls, if false, will clean the data after rendering the view.

Returns: The rendered text for the chosen view

Return type: string

Builds the output based upon a provided template source and any data that has already been set:

```
echo $parser->render('myview');
```

Options supported, and behavior, as above.

setData([\$data[, \$context=null]])

Parameters:

- **\$data** (*array*) – Array of view data strings, as key/value pairs
- **\$context** (*string*) – The context to use for data escaping.

Returns: The Renderer, for method chaining

Return type: CodeIgniter\View\RendererInterface.

Sets several pieces of view data at once:

```
$renderer->setData(['name'=>'George', 'position'=>'Boss']);
```

Supported escape contexts: html, css, js, url, or attr or raw.

If 'raw', no escaping will happen.

setVar(\$name[, \$value=null[, \$context=null]])

Parameters:

- **\$name** (*string*) – Name of the view data variable
- **\$value** (*mixed*) – The value of this view data
- **\$context** (*string*) – The context to use for data escaping.

Returns: The Renderer, for method chaining

Return type: CodeIgniter\View\RendererInterface.

Sets a single piece of view data:

```
$renderer->setVar('name', 'Joe', 'html');
```

Supported escape contexts: html, css, js, url, attr or raw.

If 'raw', no escaping will happen.

setDelimiters(\$leftDelimiter = '{', \$rightDelimiter = '}')

Parameters:

- **\$leftDelimiter** (*string*) – Left delimiter for substitution fields
- **\$rightDelimiter** (*string*) – right delimiter for substitution fields

Returns: The Renderer, for method chaining

Return

type: CodeIgniter\View\RendererInterface.

Over-ride the substitution field delimiters:

```
$renderer->setDelimiters(['', '']);
```

HTTP Responses

The Response class extends the [HTTP Message Class](#) with methods only appropriate for a server responding to the client that called it.

- [Working with the Response](#)
 - [Setting the Output](#)
 - [Setting Headers](#)
- [Force File Download](#)
- [HTTP Caching](#)
- [Content Security Policy](#)
 - [Turning CSP On](#)
 - [Runtime Configuration](#)
 - [Inline Content](#)

[Working with the Response](#)

A Response class is instantiated for you and passed into your controllers. It can be accessed through `$this->response`. Many times you will not need to touch the class directly, since CodeIgniter takes care of sending the headers and the body for you. This is great if the page successfully created the content it was asked to. When things go wrong, or you need to send very specific status codes back, or even take advantage of the powerful HTTP caching, it's there for you.

[Setting the Output](#)

When you need to set the output of the script directly, and not rely on CodeIgniter to automatically get it, you do it manually with the `setBody` method. This is usually used in conjunction with setting the status code of the response:

```
$this->response->setStatusCode(404)
    ->setBody($body);
```

The reason phrase ('OK', 'Created', 'Moved Permanently') will be automatically added, but you can add custom reasons as the second parameter of the `setStatusCode()` method:

```
$this->response->setStatusCode(404, 'Nope. Not here.');
```

You can set format an array into either JSON or XML and set the content type header to the appropriate mime with the `setJSON` and `setXML` methods. Typically, you will send an array of data to be converted:

```
$data = [
    'success' => true,
    'id' => 123
];

return $this->response->setJSON($data);
    or
return $this->response->setXML($data);
```

Setting Headers

Often, you will need to set headers to be set for the response. The Response class makes this very simple to do, with the `setHeader()` method. The first parameter is the name of the header. The second parameter is the value, which can be either a string or an array of values that will be combined correctly when sent to the client. Using these functions instead of using the native PHP functions allows you to ensure that no headers are sent prematurely, causing errors, and makes testing possible.

```
$response->setHeader('Location', 'http://example.com')
    ->setHeader('WWW-Authenticate', 'Negotiate');
```

If the header exists and can have more than one value, you may use the `appendHeader()` and `prependHeader()` methods to add the value to the end or beginning of the values list, respectively. The first parameter is the name of the header, while the second is the value to append or prepend.

```
$response->setHeader('Cache-Control', 'no-cache')
```



```
->appendHeader('Cache-Control', 'must-revalidate');
```

Headers can be removed from the response with the `removeHeader()` method, which takes the header name as the only parameter. This is not case-sensitive.

```
$response->removeHeader('Location');
```

Force File Download

The Response class provides a simple way to send a file to the client, prompting the browser to download the data to your computer. This sets the appropriate headers to make it happen.

The first parameter is the **name you want the downloaded file to be named**, the second parameter is the file data.

If you set the second parameter to NULL and `$filename` is an existing, readable file path, then its content will be read instead.

If you set the third parameter to boolean TRUE, then the actual file MIME type (based on the filename extension) will be sent, so that if your browser has a handler for that type - it can use it.

Example:

```
$data = 'Here is some text!';  
$name = 'mytext.txt';  
return $response->download($name, $data);
```

If you want to download an existing file from your server you'll need to do the following:

```
// Contents of photo.jpg will be automatically read  
return $response->download('/path/to/photo.jpg', NULL);
```

Note

The response object MUST be returned for the download to be sent to the

client. This allows the response to be passed through all **after** filters before being sent to the client.

HTTP Caching

Built into the HTTP specification are tools help the client (often the web browser) cache the results. Used correctly, this can lend a huge performance boost to your application because it will tell the client that they don't need to contact the `getServer` at all since nothing has changed. And you can't get faster than that.

This are handled through the `Cache-Control` and `ETag` headers. This guide is not the proper place for a thorough introduction to all of the cache headers power, but you can get a good understanding over at [Google Developers](https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching) [https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching] and the [Mobify Blog](https://www.mobify.com/blog/beginners-guide-to-http-cache-headers/) [https://www.mobify.com/blog/beginners-guide-to-http-cache-headers/].

By default, all response objects sent through CodeIgniter have HTTP caching turned off. The options and exact circumstances are too varied for us to be able to create a good default other than turning it off. It's simple to set the Cache values to what you need, though, through the `setCache()` method:

```
$options = [
    'max-age'   => 300,
    's-maxage'  => 900
    'etag'      => 'abcde',
];
$this->response->setCache($options);
```

The `$options` array simply takes an array of key/value pairs that are, with a couple of exceptions, assigned to the `Cache-Control` header. You are free to set all of the options exactly as you need for you specific situation. While most of the options are applied to the `Cache-Control` header, it intelligently handles the `etag` and `last-modified` options to their appropriate header.

Content Security Policy

One of the best protections you have against XSS attacks is to implement a Content Security Policy on the site. This forces you to whitelist every single source of content that is pulled in from your site's HTML, including images, stylesheets, javascript files, etc. The browser will refuse content from sources that don't meet the whitelist. This whitelist is created within the response's Content-Security-Policy header and has many different ways it can be configured.

This sounds complex, and on some sites, can definitely be challenging. For many simple sites, though, where all content is served by the same domain (<http://example.com>), it is very simple to integrate.

As this is a complex subject, this user guide will not go over all of the details. For more information, you should visit the following sites:

- [Content Security Policy main site](http://content-security-policy.com/) [http://content-security-policy.com/]
- [W3C Specification](https://www.w3.org/TR/CSP) [https://www.w3.org/TR/CSP]
- [Introduction at HTML5Rocks](http://www.html5rocks.com/en/tutorials/security/content-security-policy/) [http://www.html5rocks.com/en/tutorials/security/content-security-policy/]
- [Article at SitePoint](https://www.sitepoint.com/improving-web-security-with-the-content-security-policy/) [https://www.sitepoint.com/improving-web-security-with-the-content-security-policy/]

Turning CSP On

By default, support for this is off. To enable support in your application, edit the CSPEnabled value in **app/Config/App.php**:

```
public $CSPEnabled = true;
```

When enabled, the response object will contain an instance of `CodeIgniter\HTTP\ContentSecurityPolicy`. The values set in **app/Config/ContentSecurityPolicy.php** are applied to that instance and, if no changes are needed during runtime, then the correctly formatted header is sent and you're all done.

With CSP enabled, two header lines are added to the HTTP response: a Content-Security-Policy header, with policies identifying content types or origins that are explicitly allowed for different contexts, and a Content-

Security-Policy-Report-Only header, which identifies content types or origins that will be allowed but which will also be reported to the destination of your choice.

Our implementation provides for a default treatment, changeable through the `reportOnly()` method. When an additional entry is added to a CSP directive, as shown below, it will be added to the CSP header appropriate for blocking or preventing. That can be over-ridden on a per call basis, by providing an optional second parameter to the adding method call.

Runtime Configuration

If your application needs to make changes at run-time, you can access the instance at `$response->CSP`. The class holds a number of methods that map pretty clearly to the appropriate header value that you need to set. Examples are shown below, with different combinations of parameters, though all accept either a directive name or an array of them.:

```
// specify the default directive treatment
$response->CSP->reportOnly(false);

// specify the origin to use if none provided for a directive
$response->CSP->setDefaultSrc('cdn.example.com');
// specify the URL that "report-only" reports get sent to
$response->CSP->setReportURI('http://example.com/csp/reports');
// specify that HTTP requests be upgraded to HTTPS
$response->CSP->upgradeInsecureRequests(true);

// add types or origins to CSP directives
// assuming that the default treatment is to block rather than ju
$response->CSP->addBaseURI('example.com', true); // report only
$response->CSP->addChildSrc('https://youtube.com'); // blocked
$response->CSP->addConnectSrc('https://*.facebook.com', false); /
$response->CSP->addFontSrc('fonts.example.com');
$response->CSP->addFormAction('self');
$response->CSP->addFrameAncestor('none', true); // report this on
$response->CSP->addImageSrc('cdn.example.com');
$response->CSP->addMediaSrc('cdn.example.com');
$response->CSP->addManifestSrc('cdn.example.com');
$response->CSP->addObjectSrc('cdn.example.com', false); // reject
$response->CSP->addPluginType('application/pdf', false); // rejec
$response->CSP->addScriptSrc('scripts.example.com', true); // all
$response->CSP->addStyleSrc('css.example.com');
```

```
$response->CSP->addSandbox(['allow-forms', 'allow-scripts']);
```

The first parameter to each of the “add” methods is an appropriate string value, or an array of them.

The `reportOnly` method allows you to specify the default reporting treatment for subsequent sources, unless over-ridden. For instance, you could specify that `youtube.com` was allowed, and then provide several allowed but reported sources:

```
$response->addChildSrc('https://youtube.com'); // allowed
$response->reportOnly(true);
$response->addChildSrc('https://metube.com'); // allowed but repo
$response->addChildSrc('https://ourtube.com', false); // allowed
```

Inline Content

It is possible to set a website to not protect even inline scripts and styles on its own pages, since this might have been the result of user-generated content. To protect against this, CSP allows you to specify a nonce within the `<style>` and `<script>` tags, and to add those values to the response’s header. This is a pain to handle in real life, and is most secure when generated on the fly. To make this simple, you can include a `{csp-style-nonce}` or `{csp-script-nonce}` placeholder in the tag and it will be handled for you automatically:

```
// Original
<script {csp-script-nonce}>
    console.log("Script won't run as it doesn't contain a nonce a
</script>
```

```
// Becomes
<script nonce="Eskdikejidojdk978Ad8jf">
    console.log("Script won't run as it doesn't contain a nonce a
</script>
```

```
// OR
<style {csp-style-nonce}>
    . . .
</style>
```

Class Reference

Note

In addition to the methods listed here, this class inherits the methods from the [Message Class](#).

The methods provided by the parent class that are available are:

- `CodeIgniter\HTTP\Message::body()`
- `CodeIgniter\HTTP\Message::setBody()`
- `CodeIgniter\HTTP\Message::populateHeaders()`
- `CodeIgniter\HTTP\Message::headers()`
- `CodeIgniter\HTTP\Message::header()`
- `CodeIgniter\HTTP\Message::headerLine()`
- `CodeIgniter\HTTP\Message::setHeader()`
- `CodeIgniter\HTTP\Message::removeHeader()`
- `CodeIgniter\HTTP\Message::appendHeader()`
- `CodeIgniter\HTTP\Message::protocolVersion()`
- `CodeIgniter\HTTP\Message::setProtocolVersion()`
- `CodeIgniter\HTTP\Message::negotiateMedia()`
- `CodeIgniter\HTTP\Message::negotiateCharset()`
- `CodeIgniter\HTTP\Message::negotiateEncoding()`
- `CodeIgniter\HTTP\Message::negotiateLanguage()`
- `CodeIgniter\HTTP\Message::negotiateLanguage()`

CodeIgniter\HTTP\Response

`getStatusCode()`

Returns: The current HTTP status code for this response

Return type: int

Returns the currently status code for this response. If no status code has been set, a `BadMethodCallException` will be thrown:

```
echo $response->getStatusCode();
```

getStatusCode(\$code[, \$reason=""])

Parameters:

- **\$code** (*int*) – The HTTP status code
- **\$reason** (*string*) – An optional reason phrase.

Returns: The current Response instance

Return type: CodeIgniter\HTTP\Response

Sets the HTTP status code that should be sent with this response:

```
$response->getStatusCode(404);
```

The reason phrase will be automatically generated based upon the official lists. If you need to set your own for a custom status code, you can pass the reason phrase as the second parameter:

```
$response->getStatusCode(230, "Tardis initiated");
```

getReason()

Returns: The current reason phrase.

Return type: string

Returns the current status code for this response. If not status has been set, will return an empty string:

```
echo $response->getReason();
```

setDate(\$date)

Parameters:

- **\$date** (*DateTime*) – A DateTime instance with the time to set for this response.

Returns: The current response instance.

Return type: CodeIgniter\HTTP\Response

Sets the date used for this response. The \$date argument must be an instance of DateTime:

```
$date = DateTime::createFromFormat('j-M-Y', '15-Feb-2016');  
$response->setDate($date);
```

setContentType(\$mime[, \$charset='UTF-8'])

- **\$mime** (*string*) – The content type this response represents.

Parameters:

- **\$charset** (*string*) – The character set this response uses.

Returns: The current response instance.

Return type: CodeIgniterHTTPResponse

Sets the content type this response represents:

```
$response->setContentType('text/plain');
$response->setContentType('text/html');
$response->setContentType('application/json');
```

By default, the method sets the character set to UTF-8. If you need to change this, you can pass the character set as the second parameter:

```
$response->setContentType('text/plain', 'x-pig-latin');
```

noCache()

Returns: The current response instance.

Return type: CodeIgniterHTTPResponse

Sets the Cache-Control header to turn off all HTTP caching. This is the default setting of all response messages:

```
$response->noCache();
```

```
// Sets the following header:
Cache-Control: no-store, max-age=0, no-cache
```

setCache(\$options)

- **\$options** (*array*) – An array of key/value cache control settings

Parameters:

Returns: The current response instance.

Return type: CodeIgniterHTTPResponse

Sets the Cache-Control headers, including ETags and Last-Modified. Typical keys are:

- etag
- last-modified
- max-age
- s-maxage
- private
- public
- must-revalidate
- proxy-revalidate
- no-transform

When passing the last-modified option, it can be either a date string, or a DateTime object.

setLastModified(\$date)

Parameters: • **\$date** (*string|DateTime*) – The date to set the Last-Modified header to

Returns: The current response instance.

Return type: CodeIgniterHTTPResponse

Sets the Last-Modified header. The \$date object can be either a string or a DateTime instance:

```
$response->setLastModified(date('D, d M Y H:i:s'));
$response->setLastModified(DateTime::createFromFormat('u',
< >
```

send()

Returns: The current response instance.

Return type: CodeIgniterHTTPResponse

Tells the response to send everything back to the client. This will first send the headers, followed by the response body. For the main application response, you do not need to call this as it is handled automatically by CodeIgniter.

```
setCookie($name = "[, $value = "[, $expire = "[, $domain = "[, $path
= '/[, $prefix = "[, $secure = FALSE[, $httponly = FALSE]]]]]]])
```

- **\$name** (*mixed*) – Cookie name or an array of parameters
 - **\$value** (*string*) – Cookie value
 - **\$expire** (*int*) – Cookie expiration time in seconds
 - **\$domain** (*string*) – Cookie domain
 - **\$path** (*string*) – Cookie path
 - **\$prefix** (*string*) – Cookie name prefix
 - **\$secure** (*bool*) – Whether to only transfer the cookie through HTTPS
 - **\$httponly** (*bool*) – Whether to only make the cookie accessible for HTTP requests (no JavaScript)
- Parameters:**

Return type: void

Sets a cookie containing the values you specify. There are two ways to pass information to this method so that a cookie can be set: Array Method, and Discrete Parameters:

Array Method

Using this method, an associative array is passed as the first parameter:

```
$cookie = [
    'name'    => 'The Cookie Name',
    'value'   => 'The Value',
    'expire'  => '86500',
    'domain'  => '.some-domain.com',
    'path'    => '/',
    'prefix'  => 'myprefix_',
    'secure'  => TRUE,
    'httponly' => FALSE
];
```

```
$response->setCookie($cookie);
```

Notes

Only the name and value are required. To delete a cookie set it with the expiration blank.

The expiration is set in **seconds**, which will be added to the current time. Do not include the time, but rather only the number of seconds from *now* that you wish the cookie to be valid. If the expiration is set to zero the cookie will only last as long as the browser is open.

For site-wide cookies regardless of how your site is requested, add your URL to the **domain** starting with a period, like this: .your-domain.com

The path is usually not needed since the method sets a root path.

The prefix is only needed if you need to avoid name collisions with other identically named cookies for your server.

The secure boolean is only needed if you want to make it a secure cookie by setting it to TRUE.

Discrete Parameters

If you prefer, you can set the cookie by passing data using individual parameters:

```
$response->setCookie($name, $value, $expire, $domain, $path
```

```
deleteCookie($name = "[, $domain = "[, $path = '/'[, $prefix = "]]])
```

- **\$name** (*mixed*) – Cookie name or an array of parameters

- Parameters:**
- **\$domain** (*string*) – Cookie domain
 - **\$path** (*string*) – Cookie path
 - **\$prefix** (*string*) – Cookie name prefix

Return type: void

Delete an existing cookie by setting its expiry to blank.

Notes

Only the name is required.

The prefix is only needed if you need to avoid name collisions with other identically named cookies for your server.

Provide a prefix if cookies should only be deleted for that subset. Provide a domain name if cookies should only be deleted for that domain. Provide a path name if cookies should only be deleted for that path.

If any of the optional parameters are empty, then the same-named cookie will be deleted across all that apply.

Example:

```
$response->deleteCookie($name);
```

hasCookie(\$name = "[, \$value = null[, \$prefix = "])

- Parameters:**
- **\$name** (*mixed*) – Cookie name or an array of parameters
 - **\$value** (*string*) – cookie value
 - **\$prefix** (*string*) – Cookie name prefix

Return type: boolean

Checks to see if the Response has a specified cookie or not.

Notes

Only the name is required. If a prefix is specified, it will be prepended to the cookie name.

If no value is given, the method just checks for the existence of the named cookie. If a value is given, then the method checks that the cookie exists, and that it has the prescribed value.

Example:

```
if ($response->hasCookie($name)) ...
```

getCookie(\$name = "[, \$prefix = "])

Parameters:

- **\$name** (*mixed*) – Cookie name
- **\$prefix** (*string*) – Cookie name prefix

Return type: boolean

Returns the named cookie, if found, or null.

If no name is given, returns the array of cookies.

Each cookie is returned as an associative array.

Example:

```
$cookie = $response->getCookie($name);
```

API Response Trait

Much of modern PHP development requires building API's, whether simply to provide data for a javascript-heavy single page application, or as a standalone product. CodeIgniter provides an API Response trait that can be used with any controller to make common response types simple, with no need to remember which HTTP status code should be returned for which response types.

- [Example Usage](#)
- [Handling Response Types](#)
 - [Class Reference](#)

[Example Usage](#)

The following example shows a common usage pattern within your controllers.

```
<?php namespace App\Controllers;

use CodeIgniter\API\ResponseTrait;

class Users extends \CodeIgniter\Controller
{
    use ResponseTrait;

    public function createUser()
    {
        $model = new UserModel();
        $user   = $model->save($this->request->getPost());

        // Respond with 201 status code
        return $this->respondCreated();
    }
}
```

In this example, an HTTP status code of 201 is returned, with the generic status message, 'Created'. Methods exist for the most common use cases:

```
// Generic response method
respond($data, 200);
// Generic failure response
fail($errors, 400);
// Item created response
respondCreated($data);
// Item successfully deleted
respondDeleted($data);
// Client isn't authorized
failUnauthorized($description);
// Forbidden action
failForbidden($description);
// Resource Not Found
failNotFound($description);
// Data did not validate
failValidationError($description);
// Resource already exists
failResourceExists($description);
// Resource previously deleted
failResourceGone($description);
// Client made too many requests
failTooManyRequests($description);
```

Handling Response Types

When you pass your data in any of these methods, they will determine the data type to format the results as based on the following criteria:

- If \$data is a string, it will be treated as HTML to send back to the client.
- If \$data is an array, it will try to negotiate the content type with what the client asked for, defaulting to JSON
 - if nothing else has been specified within ConfigAPI.php, the \$supportedResponseFormats property.

To define the formatter that is used, edit **Config/Format.php**. The \$supportedResponseFormats contains a list of mime types that your application can automatically format the response for. By default, the system knows how to format both XML and JSON responses:

```
public $supportedResponseFormats = [
    'application/json',
    'application/xml'
];
```

This is the array that is used during [Content Negotiation](#) to determine which type of response to return. If no matches are found between what the client requested and what you support, the first format in this array is what will be returned.

Next, you need to define the class that is used to format the array of data. This must be a fully qualified class name, and the class must implement **CodeIgniter\Format\FormatterInterface**. Formatters come out of the box that support both JSON and XML:

```
public $formatters = [
    'application/json' => \CodeIgniter\Format\JSONFormatter::class
    'application/xml'  => \CodeIgniter\Format\XMLFormatter::class
];
```

So, if your request asks for JSON formatted data in an **Accept** header, the data array you pass any of the `respond*` or `fail*` methods will be formatted by the **CodeIgniter\API\JSONFormatter** class. The resulting JSON data will be sent back to the client.

[Class Reference](#)

respond(\$data[, \$statusCode=200[, \$message=""]])

- Parameters:**
- **\$data** (*mixed*) – The data to return to the client. Either string or array.
 - **\$statusCode** (*int*) – The HTTP status code to return. Defaults to 200
 - **\$message** (*string*) – A custom “reason” message to return.

This is the method used by all other methods in this trait to return a response to the client.

The \$data element can be either a string or an array. By default, a string

will be returned as HTML, while an array will be run through `json_encode` and returned as JSON, unless [Content Negotiation](#) determines it should be returned in a different format.

If a `$message` string is passed, it will be used in place of the standard IANA reason codes for the response status. Not every client will respect the custom codes, though, and will use the IANA standards that match the status code.

Note

Since it sets the status code and body on the active Response instance, this should always be the final method in the script execution.

```
fail($messages[, int $status=400[, string $code=null[, string  
$message=""]]])
```

- **\$messages** (*mixed*) – A string or array of strings that contain error messages encountered.
- **\$status** (*int*) – The HTTP status code to return.

Parameters: Defaults to 400.

- **\$code** (*string*) – A custom, API-specific, error code.
- **\$message** (*string*) – A custom “reason” message to return.

Returns: A multi-part response in the client’s preferred format.

This is the generic method used to represent a failed response, and is used by all of the other “fail” methods.

The `$messages` element can be either a string or an array of strings.

The `$status` parameter is the HTTP status code that should be returned.

Since many APIs are better served using custom error codes, a custom error code can be passed in the third parameter. If no value is present, it will be the same as `$status`.

If a `$message` string is passed, it will be used in place of the standard

IANA reason codes for the response status. Not every client will respect the custom codes, though, and will use the IANA standards that match the status code.

The response is an array with two elements: error and messages. The error element contains the status code of the error. The messages element contains an array of error messages. It would look something like:

```
$response = [
    'status' => 400,
    'code' => '321a',
    'messages' => [
        'Error message 1',
        'Error message 2'
    ]
];
```

respondCreated(\$data = null[, string \$message = "])

- **\$data** (*mixed*) – The data to return to the client. Either string or array.
- Parameters:**
- **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when a new resource was created, typically 201.:

```
$user = $userModel->insert($data);
return $this->respondCreated($user);
```

respondDeleted(\$data = null[, string \$message = "])

- **\$data** (*mixed*) – The data to return to the client. Either string or array.
- Parameters:**
- **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when a new resource was deleted as the result of this API call, typically 200.

```
$user = $userModel->delete($id);  
return $this->respondDeleted(['id' => $id]);
```

failUnauthorized(*string \$description* = 'Unauthorized'[, *string \$code*=null[, *string \$message* = "]])

- **\$description** (*mixed*) – The error message to show the user.

Parameters:

- **\$code** (*string*) – A custom, API-specific, error code.
- **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when the user either has not been authorized, or has incorrect authorization. Status code is 401.

```
return $this->failUnauthorized('Invalid Auth token');
```

failForbidden(*string \$description* = 'Forbidden'[, *string \$code*=null[, *string \$message* = "]])

- **\$description** (*mixed*) – The error message to show the user.

Parameters:

- **\$code** (*string*) – A custom, API-specific, error code.
- **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Unlike failUnauthorized, this method should be used when the requested API endpoint is never allowed. Unauthorized implies the client is encouraged to try again with different credentials. Forbidden means the client should not try again because it won’t help. Status code is 403.

```
return $this->failForbidden('Invalid API endpoint.');
```

failNotFound(*string \$description* = 'Not Found'[, *string \$code*=null[, *string \$message* = "]])

- **\$description** (*mixed*) – The error message to show the

Parameters: user.
• **\$code** (*string*) – A custom, API-specific, error code.
• **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when the requested resource cannot be found. Status code is 404.

```
return $this->failNotFound('User 13 cannot be found.');
```

failValidationError(*string* \$description = 'Bad Request'[, *string* \$code=null[, *string* \$message = "]])

• **\$description** (*mixed*) – The error message to show the user.

Parameters: • **\$code** (*string*) – A custom, API-specific, error code.
• **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when data the client sent did not pass validation rules. Status code is typically 400.

```
return $this->failValidationError($validation->getErrors());
```

failResourceExists(*string* \$description = 'Conflict'[, *string* \$code=null[, *string* \$message = "]])

• **\$description** (*mixed*) – The error message to show the user.

Parameters: • **\$code** (*string*) – A custom, API-specific, error code.
• **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when the resource the client is trying to create already exists. Status code is typically 409.

```
return $this->failResourceExists('A user already exists with t
```

```
< >
```

failResourceGone(*string \$description* = 'Gone'[, *string \$code*=null[,
string \$message = "]])

- **\$description** (*mixed*) – The error message to show the user.

Parameters:

- **\$code** (*string*) – A custom, API-specific, error code.
- **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when the requested resource was previously deleted and is no longer available. Status code is typically 410.

```
return $this->failResourceGone('That user has been previously
```

```
< >
```

failTooManyRequests(*string \$description* = 'Too Many Requests'[, *string \$code*=null[, *string \$message* = "]])

- **\$description** (*mixed*) – The error message to show the user.

Parameters:

- **\$code** (*string*) – A custom, API-specific, error code.
- **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when the client has called an API endpoint too many times. This might be due to some form of throttling or rate limiting. Status code is typically 400.

```
return $this->failTooManyRequests('You must wait 15 seconds be
```

```
< >
```

failServerError(*string \$description* = 'Internal Server Error'[, *string \$code* = null[, *string \$message* = "]])

- **\$description** (*string*) – The error message to show the user.

Parameters:

- **\$code** (*string*) – A custom, API-specific, error code.
- **\$message** (*string*) – A custom “reason” message to return.

Returns: The value of the Response object’s send() method.

Sets the appropriate status code to use when there is a server error.

```
return $this->failServerError('Server error.');
```

Localization

- [Working With Locales](#)
 - [Configuring the Locale](#)
 - [Locale Detection](#)
 - [Retrieving the Current Locale](#)
- [Language Localization](#)
 - [Creating Language Files](#)
 - [Basic Usage](#)
 - [Language Fallback](#)
 - [Message Translations](#)

[Working With Locales](#)

CodeIgniter provides several tools to help you localize your application for different languages. While full localization of an application is a complex subject, it's simple to swap out strings in your application with different supported languages.

Language strings are stored in the **app/Language** directory, with a sub-directory for each supported language:

```
/app
  /Language
    /en
      app.php
    /fr
      app.php
```

Important

Locale detection only works for web-based requests that use the `IncomingRequest` class. Command-line requests will not have these

features.

Configuring the Locale

Every site will have a default language/locale they operate in. This can be set in **Config/App.php**:

```
public $defaultLocale = 'en';
```

The value can be any string that your application uses to manage text strings and other formats. It is recommended that a [BCP 47](http://www.rfc-editor.org/rfc/bcp/bcp47.txt) [http://www.rfc-editor.org/rfc/bcp/bcp47.txt] language code is used. This results in language codes like en-US for American English, or fr-FR, for French/France. A more readable introduction to this can be found on the [W3C's site](https://www.w3.org/International/articles/language-tags/) [https://www.w3.org/International/articles/language-tags/].

The system is smart enough to fallback to more generic language codes if an exact match cannot be found. If the locale code was set to **en-US** and we only have language files setup for **en** then those will be used since nothing exists for the more specific **en-US**. If, however, a language directory existed at **app/Language/en-US** then that would be used first.

Locale Detection

There are two methods supported to detect the correct locale during the request. The first is a “set and forget” method that will automatically perform [content negotiation](#) for you to determine the correct locale to use. The second method allows you to specify a segment in your routes that will be used to set the locale.

Content Negotiation

You can setup content negotiation to happen automatically by setting two additional settings in Config/App. The first value tells the Request class that we do want to negotiate a locale, so simply set it to true:


```
public $negotiateLocale = true;
```

Once this is enabled, the system will automatically negotiate the correct language based upon an array of locales that you have defined in `$supportedLocales`. If no match is found between the languages that you support, and the requested language, the first item in `$supportedLocales` will be used. In the following example, the **en** locale would be used if no match is found:

```
public $supportedLocales = ['en', 'es', 'fr-FR'];
```

In Routes

The second method uses a custom placeholder to detect the desired locale and set it on the Request. The placeholder `{locale}` can be placed as a segment in your route. If present, the contents of the matching segment will be your locale:

```
$routes->get('{locale}/books', 'App\Books::index');
```

In this example, if the user tried to visit `http://example.com/fr/books`, then the locale would be set to `fr`, assuming it was configured as a valid locale.

Note

If the value doesn't match a valid locale as defined in the App configuration file, the default locale will be used in its place.

Retrieving the Current Locale

The current locale can always be retrieved from the `IncomingRequest` object, through the `getLocale()` method. If your controller is extending `CodeIgniter\Controller`, this will be available through `$this->request`:

```
<?php namespace App\Controllers;

class UserController extends \CodeIgniter\Controller
{
```

```

    public function index()
    {
        $locale = $this->request->getLocale();
    }
}

```

Alternatively, you can use the [Services class](#) to retrieve the current request:

```
$locale = service('request')->getLocale();
```

Language Localization

Creating Language Files

Languages do not have any specific naming convention that are required. The file should be named logically to describe the type of content it holds. For example, let's say you want to create a file containing error messages. You might name it simply: **Errors.php**.

Within the file you would return an array, where each element in the array has a language key and the string to return:

```
'language_key' => 'The actual message to be shown.'
```

Note

It's good practice to use a common prefix for all messages in a given file to avoid collisions with similarly named items in other files. For example, if you are creating error messages you might prefix them with error_

```

return [
    'errorEmailMissing'    => 'You must submit an email address',
    'errorURLMissing'      => 'You must submit a URL',
    'errorUsernameMissing' => 'You must submit a username',
];

```

Basic Usage

You can use the lang() helper function to retrieve text from any of the

language files, by passing the filename and the language key as the first parameter, separated by a period (.). For example, to load the `errorMessageMissing` string from the `Errors` language file, you would do the following:

```
echo lang('Errors.errorMessageMissing');
```

If the requested language key doesn't exist in the file for the current locale, the string will be passed back, unchanged. In this example, it would return `'Errors.errorMessageMissing'` if it didn't exist.

Replacing Parameters

Note

The following functions all require the [intl](http://php.net/manual/en/book.intl.php) [http://php.net/manual/en/book.intl.php] extension to be loaded on your system in order to work. If the extension is not loaded, no replacement will be attempted. A great overview can be found over at [Sitepoint](https://www.sitepoint.com/localization-demystified-understanding-php-intl/) [https://www.sitepoint.com/localization-demystified-understanding-php-intl/].

You can pass an array of values to replace placeholders in the language string as the second parameter to the `lang()` function. This allows for very simple number translations and formatting:

```
// The language file, Tests.php:
return [
    "apples"      => "I have {0, number} apples.",
    "men"         => "I have {1, number} men out-performed the re
    "namedApples" => "I have {number_apples, number, integer} app
];

// Displays "I have 3 apples."
echo lang('Tests.apples', [ 3 ]);
```

The first item in the placeholder corresponds to the index of the item in the array, if it's numerical:

```
// Displays "The top 23 men out-performed the remaining 20"
echo lang('Tests.men', [20, 23]);
```

You can also use named keys to make it easier to keep things straight, if you'd like:

```
// Displays "I have 3 apples."
echo lang("Tests.namedApples", ['number_apples' => 3]);
```

Obviously, you can do more than just number replacement. According to the [official ICU docs](http://icu-project.org/apiref/icu4c/classMessageFormat.html#details) [http://icu-project.org/apiref/icu4c/classMessageFormat.html#details] for the underlying library, the following types of data can be replaced:

- numbers - integer, currency, percent
- dates - short, medium, long, full
- time - short, medium, long, full
- spellout - spells out numbers (i.e. 34 becomes thirty-four)
- ordinal
- duration

Here are a few examples:

```
// The language file, Tests.php
return [
    'shortTime'    => 'The time is now {0, time, short}.',
    'mediumTime'   => 'The time is now {0, time, medium}.',
    'longTime'     => 'The time is now {0, time, long}.',
    'fullTime'     => 'The time is now {0, time, full}.',
    'shortDate'    => 'The date is now {0, date, short}.',
    'mediumDate'   => 'The date is now {0, date, medium}.',
    'longDate'     => 'The date is now {0, date, long}.',
    'fullDate'     => 'The date is now {0, date, full}.',
    'spelledOut'   => '34 is {0, spellout}',
    'ordinal'      => 'The ordinal is {0, ordinal}',
    'duration'     => 'It has been {0, duration}',
];

// Displays "The time is now 11:18 PM"
echo lang('Tests.shortTime', [time()]);
// Displays "The time is now 11:18:50 PM"
echo lang('Tests.mediumTime', [time()]);
// Displays "The time is now 11:19:09 PM CDT"
echo lang('Tests.longTime', [time()]);
// Displays "The time is now 11:19:26 PM Central Daylight Time"
```

```

echo lang('Tests.fullTime', [time()]);

// Displays "The date is now 8/14/16"
echo lang('Tests.shortDate', [time()]);
// Displays "The date is now Aug 14, 2016"
echo lang('Tests.mediumDate', [time()]);
// Displays "The date is now August 14, 2016"
echo lang('Tests.longDate', [time()]);
// Displays "The date is now Sunday, August 14, 2016"
echo lang('Tests.fullDate', [time()]);

// Displays "34 is thirty-four"
echo lang('Tests.spelledOut', [34]);

// Displays "It has been 408,676:24:35"
echo lang('Tests.ordinal', [time()]);

```

You should be sure to read up on the `MessageFormatter` class and the underlying ICU formatting to get a better idea on what capabilities it has, like performing conditional replacement, pluralization, and more. Both of the links provided earlier will give you an excellent idea as to the options available.

Specifying Locale

To specify a different locale to be used when replacing parameters, you can pass the locale in as the third parameter to the `lang()` method.

```

// Displays "The time is now 23:21:28 GMT-5"
echo lang('Test.longTime', [time()], 'ru-RU');

// Displays "£7.41"
echo lang('{price, number, currency}', ['price' => 7.41], 'en-GB')
// Displays "$7.41"
echo lang('{price, number, currency}', ['price' => 7.41], 'en-US')

```

Nested Arrays

Language files also allow nested arrays to make working with lists, etc... easier.

```
// Language/en/Fruit.php

return [
    'list' => [
        'Apples',
        'Bananas',
        'Grapes',
        'Lemons',
        'Oranges',
        'Strawberries'
    ]
];

// Displays "Apples, Bananas, Grapes, Lemons, Oranges, Strawberry
echo implode(', ', lang('Fruit.list'));
```

Language Fallback

If you have a set of messages for a given locale, for instance `Language/en/app.php`, you can add language variants for that locale, each in its own folder, for instance `Language/en-US/app.php`.

You only need to provide values for those messages that would be localized differently for that locale variant. Any missing message definitions will be automatically pulled from the main locale settings.

It gets better - the localization can fall all the way back to English, in case new messages are added to the framework and you haven't had a chance to translate them yet for your locale.

So, if you are using the locale `fr-CA`, then a localized message will first be sought in `Language/fr/CA`, then in `Language/fr`, and finally in `Language/en`.

Message Translations

We have an “official” set of translations in their [own repository](https://github.com/codeigniter4/translations) [<https://github.com/codeigniter4/translations>].

You can download that repository, and copy its `Language` folder into your app. The incorporated translations will be automatically picked up because

the App namespace is mapped to your app folder.

Alternately, you could use `composer install codeigniter4/translations` inside your project, and the translated messages will be automatically picked up because the Translations namespace gets mapped appropriately.

Alternate PHP Syntax for View Files

If you do not utilize a templating engine to simplify output, you'll be using pure PHP in your View files. To minimize the PHP code in these files, and to make it easier to identify the code blocks it is recommended that you use PHP's alternative syntax for control structures and short tag echo statements. If you are not familiar with this syntax, it allows you to eliminate the braces from your code, and eliminate "echo" statements.

Alternative Echos

Normally to echo, or print out a variable you would do this:

```
<?php echo $variable; ?>
```

With the alternative syntax you can instead do it this way:

```
<?= $variable ?>
```

Alternative Control Structures

Control structures, like if, for, foreach, and while can be written in a simplified format as well. Here is an example using foreach:

```
<ul>

<?php foreach ($todo as $item) : ?>

    <li><?= $item ?></li>

<?php endforeach ?>

</ul>
```


Notice that there are no braces. Instead, the end brace is replaced with endforeach. Each of the control structures listed above has a similar closing syntax: endif, endfor, endforeach, and endwhile

Also notice that instead of using a semicolon after each structure (except the last one), there is a colon. This is important!

Here is another example, using if/elseif/else. Notice the colons:

```
<?php if ($username === 'sally') : ?>
    <h3>Hi Sally</h3>
<?php elseif ($username === 'joe') : ?>
    <h3>Hi Joe</h3>
<?php else : ?>
    <h3>Hi unknown user</h3>
<?php endif ?>
```

Working With Databases

CodeIgniter comes with a full-featured and very fast abstracted database class that supports both traditional structures and Query Builder patterns. The database functions offer clear, simple syntax.

- [Quick Start: Usage Examples](#)
- [Database Configuration](#)
- [Connecting to a Database](#)
- [Running Queries](#)
- [Generating Query Results](#)
- [Query Helper Functions](#)
- [Query Builder Class](#)
- [Transactions](#)
- [Getting MetaData](#)
- [Custom Function Calls](#)
- [Database Events](#)
- [Database Utilities](#)

Database Quick Start: Example Code

The following page contains example code showing how the database class is used. For complete details please read the individual pages describing each function.

Initializing the Database Class

The following code loads and initializes the database class based on your [configuration](#) settings:

```
$db = \Config\Database::connect();
```

Once loaded the class is ready to be used as described below.

Note: If all your pages require database access you can connect automatically. See the [connecting](#) page for details.

Standard Query With Multiple Results (Object Version)

```
$query = $db->query('SELECT name, title, email FROM my_table');
$results = $query->getResult();

foreach ($results as $row)
{
    echo $row->title;
    echo $row->name;
    echo $row->email;
}

echo 'Total Results: ' . count($results);
```

The above getResult() function returns an array of **objects**. Example: \$row->title

Standard Query With Multiple Results (Array Version)

```
$query = $db->query('SELECT name, title, email FROM my_table');
$results = $query->getResultArray();

foreach ($results as $row)
{
    echo $row['title'];
    echo $row['name'];
    echo $row['email'];
}
```

The above getResultArray() function returns an array of standard array indexes. Example: \$row['title']

Standard Query With Single Result

```
$query = $db->query('SELECT name FROM my_table LIMIT 1');
$row = $query->getRow();
echo $row->name;
```

The above getRow() function returns an **object**. Example: \$row->name

Standard Query With Single Result (Array version)

```
$query = $db->query('SELECT name FROM my_table LIMIT 1');
$row = $query->getRowArray();
echo $row['name'];
```

The above getRowArray() function returns an **array**. Example: \$row['name']

Standard Insert

```
$sql = "INSERT INTO mytable (title, name) VALUES (". $db->escape($
$db->query($sql);
```

```
echo $db->getAffectedRows();
```

Query Builder Query

The [Query Builder Pattern](#) gives you a simplified means of retrieving data:

```
$query = $db->table('table_name')->get();
```

```
foreach ($query->getResult() as $row)
{
    echo $row->title;
}
```

The above `get()` function retrieves all the results from the supplied table. The [Query Builder](#) class contains a full compliment of functions for working with data.

Query Builder Insert

```
$data = [
    'title' => $title,
    'name'  => $name,
    'date'  => $date
];
```

```
$db->table('mytable')->insert($data); // Produces: INSERT INTO m
```

Database Configuration

- [Configuring With .env File](#)
- [Explanation of Values:](#)

CodeIgniter has a config file that lets you store your database connection values (username, password, database name, etc.). The config file is located at `app/Config/Database.php`. You can also set database connection values in the `.env` file. See below for more details.

The config settings are stored in a class property that is an array with this prototype:

```
public $default = [  
    'DSN'          => '',  
    'hostname'     => 'localhost',  
    'username'     => 'root',  
    'password'     => '',  
    'database'     => 'database_name',  
    'DBDriver'     => 'MySQLi',  
    'DBPrefix'     => '',  
    'pConnect'     => TRUE,  
    'DBDebug'      => TRUE,  
    'cacheOn'      => FALSE,  
    'cacheDir'     => '',  
    'charset'      => 'utf8',  
    'DBCollat'     => 'utf8_general_ci',  
    'swapPre'      => '',  
    'encrypt'      => FALSE,  
    'compress'     => FALSE,  
    'strictOn'     => FALSE,  
    'failover'     => [],  
];
```

The name of the class property is the connection name, and can be used while connecting to specify a group name.

Some database drivers (such as PDO, PostgreSQL, Oracle, ODBC) might require a full DSN string to be provided. If that is the case, you should use the ‘DSN’ configuration setting, as if you’re using the driver’s underlying native PHP extension, like this:

```
// PDO
$default['DSN'] = 'pgsql:host=localhost;port=5432;dbname=database';

// Oracle
$default['DSN'] = '//localhost/XE';
```

Note

If you do not specify a DSN string for a driver that requires it, CodeIgniter will try to build it with the rest of the provided settings.

Note

If you provide a DSN string and it is missing some valid settings (e.g. the database character set), which are present in the rest of the configuration fields, CodeIgniter will append them.

You can also specify failovers for the situation when the main connection cannot connect for some reason. These failovers can be specified by setting the failover for a connection like this:

```
$default['failover'] = [
    [
        'hostname' => 'localhost1',
        'username' => '',
        'password' => '',
        'database' => '',
        'DBDriver' => 'MySQLi',
        'DBPrefix' => '',
        'pConnect' => TRUE,
        'DBDebug'  => TRUE,
        'cacheOn'  => FALSE,
        'cacheDir' => '',
        'charset'  => 'utf8',
    ]
];
```

```

        'DBCollat' => 'utf8_general_ci',
        'swapPre'  => '',
        'encrypt'  => FALSE,
        'compress' => FALSE,
        'strictOn' => FALSE
    ],
    [
        'hostname' => 'localhost2',
        'username'  => '',
        'password'  => '',
        'database'  => '',
        'DBDriver'  => 'MySQLi',
        'DBPrefix'  => '',
        'pConnect'  => TRUE,
        'DBDebug'   => TRUE,
        'cacheOn'   => FALSE,
        'cacheDir'  => '',
        'charset'   => 'utf8',
        'DBCollat'  => 'utf8_general_ci',
        'swapPre'   => '',
        'encrypt'   => FALSE,
        'compress'  => FALSE,
        'strictOn'  => FALSE
    ]
];

```

You can specify as many failovers as you like.

You may optionally store multiple sets of connection values. If, for example, you run multiple environments (development, production, test, etc.) under a single installation, you can set up a connection group for each, then switch between groups as needed. For example, to set up a “test” environment you would do this:

```

public $test = [
    'DSN'          => '',
    'hostname'     => 'localhost',
    'username'     => 'root',
    'password'     => '',
    'database'     => 'database_name',
    'DBDriver'     => 'MySQLi',
    'DBPrefix'     => '',
    'pConnect'     => TRUE,
    'DBDebug'      => TRUE,
    'cacheOn'      => FALSE,
    'cacheDir'     => '',

```



```

        'charset'    => 'utf8',
        'DBCollat'   => 'utf8_general_ci',
        'swapPre'     => '',
        'compress'    => FALSE,
        'encrypt'     => FALSE,
        'strictOn'    => FALSE,
        'failover'    => []
    );

```

Then, to globally tell the system to use that group you would set this variable located in the config file:

```
$defaultGroup = 'test';
```

Note

The name ‘test’ is arbitrary. It can be anything you want. By default we’ve used the word “default” for the primary connection, but it too can be renamed to something more relevant to your project.

You could modify the config file to detect the environment and automatically update the *defaultGroup* value to the correct one by adding the required logic within the class’ constructor:

```

class Database
{
    public $development = [...];
    public $test        = [...];
    public $production  = [...];

    public function __construct()
    {
        $this->defaultGroup = ENVIRONMENT;
    }
}

```

Configuring With .env File

You can also save your configuration values within a .env file with the current server’s database settings. You only need to enter the values that

change from what is in the default group's configuration settings. The values should be name following this format, where default is the group name:

```
database.default.username = 'root';  
database.default.password = '';  
database.default.database = 'ci4';
```

As with all other

Explanation of Values:

Name Config	Description
dsn	The DSN connect string (an all-in-one configuration sequence).
hostname	The hostname of your database server. Often this is 'localhost'.
username	The username used to connect to the database.
password	The password used to connect to the database.
database	The name of the database you want to connect to.
DBDriver	The database type. eg: MySQLi, Postgre, etc. The case must match the driver name
DBPrefix	An optional table prefix which will added to the table name when running Query Builder queries. This permits multiple CodeIgniter installations to share one database.
pConnect	TRUE/FALSE (boolean) - Whether to use a persistent connection.
DBDebug	TRUE/FALSE (boolean) - Whether database errors should be displayed.
cacheOn	TRUE/FALSE (boolean) - Whether database query caching is enabled.
cacheDir	The absolute server path to your database query cache directory.
charset	The character set used in communicating with the database.
	The character collation used in communicating with the database

DBCollat

Note

Only used in the 'MySQLi' driver.

swapPre

A default table prefix that should be swapped with dbprefix. This is useful for distributed applications where you might run manually written queries, and need the prefix to still be customizable by the end user.

schema

The database schema, defaults to 'public'. Used by PostgreSQL and ODBC drivers.

encrypt

Whether or not to use an encrypted connection.

- 'sqlsrv' and 'pdo/sqlsrv' drivers accept TRUE/FALSE
- 'MySQLi' and 'pdo/mysql' drivers accept an array with the following options:
 - 'ssl_key' - Path to the private key file
 - 'ssl_cert' - Path to the public key certificate file
 - 'ssl_ca' - Path to the certificate authority file
 - 'ssl_capath' - Path to a directory containing trusted CA certificates in PEM format
 - 'ssl_cipher' - List of *allowed* ciphers to be used for the encryption, separated by colons (':')
 - 'ssl_verify' - TRUE/FALSE; Whether to verify the server certificate or not ('MySQLi' only)

compress

Whether or not to use client compression (MySQL only).

strictOn

TRUE/FALSE (boolean) - Whether to force "Strict Mode" connections, good for ensuring strict SQL while developing an application.

port

The database port number. To use this value you have to add a line to the database config array.

```
$default['port'] = 5432;
```

Note

Depending on what database platform you are using (MySQL, PostgreSQL, etc.) not all values will be needed. For example, when using SQLite you will not need to supply a username or password, and the database name will be the path to your database file. The information above assumes you are using MySQL.

Connecting to your Database

You can connect to your database by adding this line of code in any function where it is needed, or in your class constructor to make the database available globally in that class.

```
$db = \Config\Database::connect();
```

If the above function does **not** contain any information in the first parameter it will connect to the default group specified in your database config file. For most people, this is the preferred method of use.

A convenience method exists that is purely a wrapper around the above line and is provided for your convenience:

```
$db = db_connect();
```

Available Parameters

1. The database group name, a string that must match the config class' property name. Default value is `$config->defaultGroup`.
2. TRUE/FALSE (boolean). Whether to return the a shared connection (see Connecting to Multiple Databases below).

Manually Connecting to a Database

The first parameter of this function can **optionally** be used to specify a particular database group from your config file. Examples:

To choose a specific group from your config file you can do this:

```
$db = \Config\Database::connect('group_name');
```

Where `group_name` is the name of the connection group from your config

file.

Multiple Connections to Same Database

By default, the `connect()` method will return the same instance of the database connection every time. If you need to have a separate connection to the same database, send `false` as the second parameter:

```
$db = \Config\Database::connect('group_name', false);
```

Connecting to Multiple Databases

If you need to connect to more than one database simultaneously you can do so as follows:

```
$db1 = \Config\Database::connect('group_one');  
$db  = \Config\Database::connect('group_two');
```

Note: Change the words “group_one” and “group_two” to the specific group names you are connecting to.

Note

You don’t need to create separate database configurations if you only need to use a different database on the same connection. You can switch to a different database when you need to, like this:

```
$db->setDatabase($database2_name);
```

Connecting with Custom Settings

You can pass in an array of database settings instead of a group name to get a connection that uses your custom settings. The array passed in must be the same format as the groups are defined in the configuration file:

```
$custom = [  
    'DSN'      => '',
```

```

        'hostname' => 'localhost',
        'username' => '',
        'password' => '',
        'database' => '',
        'DBDriver' => 'MySQLi',
        'DBPrefix' => '',
        'pConnect' => false,
        'DBDebug' => (ENVIRONMENT !== 'production'),
        'cacheOn' => false,
        'cacheDir' => '',
        'charset' => 'utf8',
        'DBCollat' => 'utf8_general_ci',
        'swapPre' => '',
        'encrypt' => false,
        'compress' => false,
        'strictOn' => false,
        'failover' => [],
        'port' => 3306,
    ];
$db = \Config\Database::connect($custom);

```

Reconnecting / Keeping the Connection Alive

If the database server's idle timeout is exceeded while you're doing some heavy PHP lifting (processing an image, for instance), you should consider pinging the server by using the `reconnect()` method before sending further queries, which can gracefully keep the connection alive or re-establish it.

Important

If you are using MySQLi database driver, the `reconnect()` method does not ping the server but it closes the connection then connects again.

```
$db->reconnect();
```

Manually closing the Connection

While CodeIgniter intelligently takes care of closing your database connections, you can explicitly close the connection.

```
$db->close();
```


Queries

- [Query Basics](#)
 - [Regular Queries](#)
 - [Simplified Queries](#)
- [Working with Database prefixes manually](#)
- [Protecting identifiers](#)
- [Escaping Queries](#)
- [Query Bindings](#)
 - [Named Bindings](#)
- [Handling Errors](#)
- [Prepared Queries](#)
 - [Preparing the Query](#)
 - [Executing the Query](#)
 - [Other Methods](#)
- [Working with Query Objects](#)
 - [The Query Class](#)

[Query Basics](#)

[Regular Queries](#)

To submit a query, use the **query** function:

```
$db->query('YOUR QUERY HERE');
```

The query() function returns a database result **object** when “read” type queries are run which you can use to [show your results](#). When “write” type queries are run it simply returns TRUE or FALSE depending on success or failure. When retrieving data you will typically assign the query to your own variable, like this:

```
$query = $db->query('YOUR QUERY HERE');
```

Simplified Queries

The **simpleQuery** method is a simplified version of the `$db->query()` method. It DOES NOT return a database result set, nor does it set the query timer, or compile bind data, or store your query for debugging. It simply lets you submit a query. Most users will rarely use this function.

It returns whatever the database drivers' "execute" function returns. That typically is TRUE/FALSE on success or failure for write type queries such as INSERT, DELETE or UPDATE statements (which is what it really should be used for) and a resource/object on success for queries with fetchable results.

```
if ($db->simpleQuery('YOUR QUERY'))
{
    echo "Success!";
}
else
{
    echo "Query failed!";
}
```

Note

PostgreSQL's `pg_exec()` function (for example) always returns a resource on success even for write type queries. So keep that in mind if you're looking for a boolean value.

Working with Database prefixes manually

If you have configured a database prefix and would like to prepend it to a table name for use in a native SQL query for example, then you can use the following:

```
$db->prefixTable('tablename'); // outputs prefix_tablename
```

If for any reason you would like to change the prefix programmatically

without needing to create a new connection you can use this method:

```
$db->setPrefix('newprefix');  
$db->prefixTable('tablename'); // outputs newprefix_tablename
```

Protecting identifiers

In many databases it is advisable to protect table and field names - for example with backticks in MySQL. **Query Builder queries are automatically protected**, but if you need to manually protect an identifier you can use:

```
$db->protectIdentifiers('table_name');
```

Important

Although the Query Builder will try its best to properly quote any field and table names that you feed it. Note that it is NOT designed to work with arbitrary user input. DO NOT feed it with unsanitized user data.

This function will also add a table prefix to your table, assuming you have a prefix specified in your database config file. To enable the prefixing set TRUE (boolean) via the second parameter:

```
$db->protectIdentifiers('table_name', TRUE);
```

Escaping Queries

It's a very good security practice to escape your data before submitting it into your database. CodeIgniter has three methods that help you do this:

1. **\$db->escape()** This function determines the data type so that it can escape only string data. It also automatically adds single quotes around the data so you don't have to:

```
$sql = "INSERT INTO table (title) VALUES('$db->escape($title  
< >
```

2. **\$db->escapeString()** This function escapes the data passed to it, regardless of type. Most of the time you'll use the above function rather than this one. Use the function like this:

```
$sql = "INSERT INTO table (title) VALUES('".$db->escapeString
```

3. **\$db->escapeLikeString()** This method should be used when strings are to be used in LIKE conditions so that LIKE wildcards ('%', '_') in the string are also properly escaped.

```
$search = '20% raise';  
$sql = "SELECT id FROM table WHERE column LIKE '%" .  
$db->escapeLikeString($search)."%' ESCAPE '!'";
```

Important

The `escapeLikeString()` method uses '!' (exclamation mark) to escape special characters for *LIKE* conditions. Because this method escapes partial strings that you would wrap in quotes yourself, it cannot automatically add the `ESCAPE '!'` condition for you, and so you'll have to manually do that.

Query Bindings

Bindings enable you to simplify your query syntax by letting the system put the queries together for you. Consider the following example:

```
$sql = "SELECT * FROM some_table WHERE id = ? AND status = ? AND  
$db->query($sql, [3, 'live', 'Rick']);
```

The question marks in the query are automatically replaced with the values in the array in the second parameter of the query function.

Binding also work with arrays, which will be transformed to IN sets:

```
$sql = "SELECT * FROM some_table WHERE id IN ? AND status = ? AND  
$db->query($sql, [[3, 6], 'live', 'Rick']);
```

The resulting query will be:

```
SELECT * FROM some_table WHERE id IN (3,6) AND status = 'live' AN
```

The secondary benefit of using binds is that the values are automatically escaped producing safer queries. You don't have to remember to manually escape data — the engine does it automatically for you.

[Named Bindings](#)

Instead of using the question mark to mark the location of the bound values, you can name the bindings, allowing the keys of the values passed in to match placeholders in the query:

```
$sql = "SELECT * FROM some_table WHERE id = :id: AND status = :st
$db->query($sql, [
    'id'      => 3,
    'status'  => 'live',
    'name'    => 'Rick'
]);
```

Note

Each name in the query **MUST** be surrounded by colons.

[Handling Errors](#)

```
$db->error();
```

If you need to get the last error that has occurred, the `error()` method will return an array containing its code and message. Here's a quick example:

```
if ( ! $db->simpleQuery('SELECT `example_field` FROM `example_tab
{
    $error = $db->error(); // Has keys 'code' and 'message'
}
```

Prepared Queries

Most database engines support some form of prepared statements, that allow you to prepare a query once, and then run that query multiple times with new sets of data. This eliminates the possibility of SQL injection since the data is passed to the database in a different format than the query itself. When you need to run the same query multiple times it can be quite a bit faster, too. However, to use it for every query can have major performance hits, since you're calling out to the database twice as often. Since the Query Builder and Database connections already handle escaping the data for you, the safety aspect is already taken care of for you. There will be times, though, when you need to ability to optimize the query by running a prepared statement, or prepared query.

Preparing the Query

This can be easily done with the `prepare()` method. This takes a single parameter, which is a Closure that returns a query object. Query objects are automatically generated by any of the “final” type queries, including **insert**, **update**, **delete**, **replace**, and **get**. This is handled the easiest by using the Query Builder to run a query. The query is not actually run, and the values don't matter since they're never applied, acting instead as placeholders. This returns a `PreparedQuery` object:

```
$pQuery = $db->prepare(function($db)
{
    return $db->table('user')
        ->insert([
            'name'      => 'x',
            'email'     => 'y',
            'country'   => 'US'
        ]);
});
```

If you don't want to use the Query Builder you can create the Query object manually using question marks for value placeholders:

```
use CodeIgniter\Database\Query;
```

```
$pQuery = $db->prepare(function($db)
{
    $sql = "INSERT INTO user (name, email, country) VALUES (?, ?,
    return (new Query($db))->setQuery($sql);
});
```

If the database requires an array of options passed to it during the prepare statement phase you can pass that array through in the second parameter:

```
use CodeIgniter\Database\Query;

$pQuery = $db->prepare(function($db)
{
    $sql = "INSERT INTO user (name, email, country) VALUES (?, ?,
    return (new Query($db))->setQuery($sql);
}, $options);
```

Executing the Query

Once you have a prepared query you can use the `execute()` method to actually run the query. You can pass in as many variables as you need in the query parameters. The number of parameters you pass must match the number of placeholders in the query. They must also be passed in the same order as the placeholders appear in the original query:

```
// Prepare the Query
$pQuery = $db->prepare(function($db)
{
    return $db->table('user')
        ->insert([
            'name' => 'x',
            'email' => 'y',
            'country' => 'US'
        ]);
});

// Collect the Data
$name = 'John Doe';
$email = 'j.doe@example.com';
$country = 'US';
```

```
// Run the Query  
$results = $pQuery->execute($name, $email, $country);
```

This returns a standard [result set](#).

Other Methods

In addition to these two primary methods, the prepared query object also has the following methods:

close()

While PHP does a pretty good job of closing all open statements with the database it's always a good idea to close out the prepared statement when you're done with it:

```
$pQuery->close();
```

getQueryString()

This returns the prepared query as a string.

hasError()

Returns boolean true/false if the last execute() call created any errors.

getErrorCode() getErrorMessage()

If any errors were encountered these methods can be used to retrieve the error code and string.

Working with Query Objects

Internally, all queries are processed and stored as instances of CodeIgniterDatabaseQuery. This class is responsible for binding the parameters, otherwise preparing the query, and storing performance data about its query.

getLastQuery()

When you just need to retrieve the last Query object, use the `getLastQuery()` method:

```
$query = $db->getLastQuery();  
echo (string)$query;
```

The Query Class

Each query object stores several pieces of information about the query itself. This is used, in part, by the Timeline feature, but is available for your use as well.

getQuery()

Returns the final query after all processing has happened. This is the exact query that was sent to the database:

```
$sql = $query->getQuery();
```

This same value can be retrieved by casting the Query object to a string:

```
$sql = (string)$query;
```

getOriginalQuery()

Returns the raw SQL that was passed into the object. This will not have any binds in it, or prefixes swapped out, etc:

```
$sql = $query->getOriginalQuery();
```

hasError()

If an error was encountered during the execution of this query this method will return true:

```
if ($query->hasError())  
{  
    echo 'Code: ' . $query->getErrorCode();  
    echo 'Error: ' . $query->getErrorMessage();  
}
```

```
}
```

isWriteType()

Returns true if the query was determined to be a write-type query (i.e. INSERT, UPDATE, DELETE, etc):

```
if ($query->isWriteType())  
{  
    ... do something  
}
```

swapPrefix()

Replaces one table prefix with another value in the final SQL. The first parameter is the original prefix that you want replaced, and the second parameter is the value you want it replaced with:

```
$sql = $query->swapPrefix('ci3_', 'ci4_');
```

getStartTime()

Gets the time the query was executed in seconds with microseconds:

```
$microtime = $query->getStartTime();
```

getDuration()

Returns a float with the duration of the query in seconds with microseconds:

```
$microtime = $query->getDuration();
```

Generating Query Results

There are several ways to generate query results:

- [Result Arrays](#)
- [Result Rows](#)
- [Custom Result Objects](#)
- [Result Helper Methods](#)
- [Class Reference](#)

[Result Arrays](#)

getResult()

This method returns the query result as an array of **objects**, or **an empty array** on failure. Typically you'll use this in a foreach loop, like this:

```
$query = $db->query("YOUR QUERY");

foreach ($query->getResult() as $row)
{
    echo $row->title;
    echo $row->name;
    echo $row->body;
}
```

The above method is an alias of `getResultObject()`.

You can pass in the string 'array' if you wish to get your results as an array of arrays:

```
$query = $db->query("YOUR QUERY");

foreach ($query->getResult('array') as $row)
```

```
{
    echo $row['title'];
    echo $row['name'];
    echo $row['body'];
}
```

The above usage is an alias of `getResultArray()`.

You can also pass a string to `getResult()` which represents a class to instantiate for each result object

```
$query = $db->query("SELECT * FROM users;");

foreach ($query->getResult('User') as $user)
{
    echo $user->name; // access attributes
    echo $user->reverseName(); // or methods defined on the 'User'
}
```

The above method is an alias of `getCustomResultObject()`.

getResultArray()

This method returns the query result as a pure array, or an empty array when no result is produced. Typically you'll use this in a foreach loop, like this:

```
$query = $db->query("YOUR QUERY");

foreach ($query->getResultArray() as $row)
{
    echo $row['title'];
    echo $row['name'];
    echo $row['body'];
}
```

Result Rows

getRow()

This method returns a single result row. If your query has more than one row, it returns only the first row. The result is returned as an **object**. Here's a

usage example:

```
$query = $db->query("YOUR QUERY");  
  
$row = $query->getRow();  
  
if (isset($row))  
{  
    echo $row->title;  
    echo $row->name;  
    echo $row->body;  
}
```

If you want a specific row returned you can submit the row number as a digit in the first parameter:

```
$row = $query->getRow(5);
```

You can also add a second String parameter, which is the name of a class to instantiate the row with:

```
$query = $db->query("SELECT * FROM users LIMIT 1;");  
$row = $query->getRow(0, 'User');  
  
echo $row->name; // access attributes  
echo $row->reverse_name(); // or methods defined on the 'User' class
```

getRowArray()

Identical to the above row() method, except it returns an array. Example:

```
$query = $db->query("YOUR QUERY");  
  
$row = $query->getRowArray();  
  
if (isset($row))  
{  
    echo $row['title'];  
    echo $row['name'];  
    echo $row['body'];  
}
```

If you want a specific row returned you can submit the row number as a digit

in the first parameter:

```
$row = $query->getRowArray(5);
```

In addition, you can walk forward/backwards/first/last through your results using these variations:

```
$row = $query->getFirstRow()
$row = $query->getLastRow()
$row = $query->getNextRow()
$row = $query->getPreviousRow()
```

By default they return an object unless you put the word “array” in the parameter:

```
$row = $query->getFirstRow('array')
$row = $query->getLastRow('array')
$row = $query->getNextRow('array')
$row = $query->getPreviousRow('array')
```

Note

All the methods above will load the whole result into memory (prefetching). Use `getUnbufferedRow()` for processing large result sets.

`getUnbufferedRow()`

This method returns a single result row without prefetching the whole result in memory as `row()` does. If your query has more than one row, it returns the current row and moves the internal data pointer ahead.

```
$query = $db->query("YOUR QUERY");

while ($row = $query->getUnbufferedRow())
{
    echo $row->title;
    echo $row->name;
    echo $row->body;
}
```

You can optionally pass 'object' (default) or 'array' in order to specify the returned value's type:

```
$query->getUnbufferedRow();           // object
$query->getUnbufferedRow('object');    // object
$query->getUnbufferedRow('array');      // associative array
```

Custom Result Objects

You can have the results returned as an instance of a custom class instead of a stdClass or array, as the getResult() and getResultArray() methods allow. If the class is not already loaded into memory, the Autoloader will attempt to load it. The object will have all values returned from the database set as properties. If these have been declared and are non-public then you should provide a __set() method to allow them to be set.

Example:

```
class User
{
    public $id;
    public $email;
    public $username;

    protected $last_login;

    public function lastLogin($format)
    {
        return $this->lastLogin->format($format);
    }

    public function __set($name, $value)
    {
        if ($name === 'lastLogin')
        {
            $this->lastLogin = DateTime::createFromFo
        }
    }

    public function __get($name)
    {
        if (isset($this->$name))
        {
```

```

        return $this->$name;
    }
}

```

In addition to the two methods listed below, the following methods also can take a class name to return the results as: `getFirstRow()`, `getLastRow()`, `getNextRow()`, and `getPreviousRow()`.

getCustomResultObject()

Returns the entire result set as an array of instances of the class requested. The only parameter is the name of the class to instantiate.

Example:

```

$query = $db->query("YOUR QUERY");

$rows = $query->getCustomResultObject('User');

foreach ($rows as $row)
{
    echo $row->id;
    echo $row->email;
    echo $row->last_login('Y-m-d');
}

```

getCustomRowObject()

Returns a single row from your query results. The first parameter is the row number of the results. The second parameter is the class name to instantiate.

Example:

```

$query = $db->query("YOUR QUERY");

$row = $query->getCustomRowObject(0, 'User');

if (isset($row))
{
    echo $row->email;    // access attributes
    echo $row->last_login('Y-m-d');    // access class methods
}

```


You can also use the `getRow()` method in exactly the same way.

Example:

```
$row = $query->getCustomRowObject(0, 'User');
```

Result Helper Methods

getFieldCount()

The number of FIELDS (columns) returned by the query. Make sure to call the method using your query result object:

```
$query = $db->query('SELECT * FROM my_table');  
  
echo $query->getFieldCount();
```

getFieldNames()

Returns an array with the names of the FIELDS (columns) returned by the query. Make sure to call the method using your query result object:

```
$query = $db->query('SELECT * FROM my_table');  
  
    echo $query->getFieldNames();
```

freeResult()

It frees the memory associated with the result and deletes the result resource ID. Normally PHP frees its memory automatically at the end of script execution. However, if you are running a lot of queries in a particular script you might want to free the result after each query result has been generated in order to cut down on memory consumption.

Example:

```
$query = $thisdb->query('SELECT title FROM my_table');  
  
foreach ($query->getResult() as $row)  
{
```

```

        echo $row->title;
    }

$query->freeResult(); // The $query result object will no longer

$query2 = $db->query('SELECT name FROM some_table');

$row = $query2->getRow();
echo $row->name;
$query2->freeResult(); // The $query2 result object will no longer

```

dataSeek()

This method sets the internal pointer for the next result row to be fetched. It is only useful in combination with `getUnbufferedRow()`.

It accepts a positive integer value, which defaults to 0 and returns TRUE on success or FALSE on failure.

```

$query = $db->query('SELECT `field_name` FROM `table_name`');
$query->dataSeek(5); // Skip the first 5 rows
$row = $query->getUnbufferedRow();

```

Note

Not all database drivers support this feature and will return FALSE. Most notably - you won't be able to use it with PDO.

Class Reference

class **CodeIgniterDatabaseBaseResult**

getResult(*[\$type = 'object']*)

Parameters:

- **\$type** (*string*) – Type of requested results - array, object, or class name

Returns: Array containing the fetched rows

Return

type: array

A wrapper for the `getResultArray()`, `getResultObject()` and `getCustomResultObject()` methods.

Usage: see [Result Arrays](#).

getResultArray()

Returns: Array containing the fetched rows

Return type: array

Returns the query results as an array of rows, where each row is itself an associative array.

Usage: see [Result Arrays](#).

getResultObject()

Returns: Array containing the fetched rows

Return type: array

Returns the query results as an array of rows, where each row is an object of type `stdClass`.

Usage: see [Result Arrays](#).

getCustomResultObject(*\$class_name*)

Parameters:

- ***\$class_name*** (*string*) – Class name for the resulting rows

Returns: Array containing the fetched rows

Return type: array

Returns the query results as an array of rows, where each row is an instance of the specified class.

getRow(*\$n* = 0[, *\$type* = 'object'])

• ***\$n*** (*int*) – Index of the query results row to be

Parameters: returned

- **\$type** (*string*) – Type of the requested result - array, object, or class name

Returns: The requested row or NULL if it doesn't exist

Return type: mixed

A wrapper for the `getRowArray()`, `getRowObject()` and `getCustomRowObject()` methods.

Usage: see [Result Rows](#).

getUnbufferedRow([*\$type* = 'object'])

Parameters: • **\$type** (*string*) – Type of the requested result - array, object, or class name

Returns: Next row from the result set or NULL if it doesn't exist

Return type: mixed

Fetches the next result row and returns it in the requested form.

Usage: see [Result Rows](#).

getRowArray([*\$n* = 0])

Parameters: • **\$n** (*int*) – Index of the query results row to be returned

Returns: The requested row or NULL if it doesn't exist

Return type: array

Returns the requested result row as an associative array.

Usage: see [Result Rows](#).

getRowObject([*\$n* = 0])

- **\$n** (*int*) – Index of the query results row to be

Parameters: returned

Returns: The requested row or NULL if it doesn't exist

**Return
type:** stdClass

Returns the requested result row as an object of type stdClass.

Usage: see [Result Rows](#).

getCustomRowObject(\$n, \$type)

Parameters:

- **\$n** (*int*) – Index of the results row to return
- **\$class_name** (*string*) – Class name for the resulting row

Returns: The requested row or NULL if it doesn't exist

**Return
type:** \$type

Returns the requested result row as an instance of the requested class.

dataSeek([\$n = 0])

Parameters:

- **\$n** (*int*) – Index of the results row to be returned next

Returns: TRUE on success, FALSE on failure

**Return
type:** bool

Moves the internal results row pointer to the desired offset.

Usage: see [Result Helper Methods](#).

setRow(\$key[, \$value = NULL])

Parameters:

- **\$key** (*mixed*) – Column name or array of key/value pairs
- **\$value** (*mixed*) – Value to assign to the column, \$key is a single field name

Return void
type:

Assigns a value to a particular column.

getNextRow([*\$type* = 'object'])

Parameters: • **\$type** (*string*) – Type of the requested result -
array, object, or class name

Returns: Next row of result set, or NULL if it doesn't
exist

Return
type: mixed

Returns the next row from the result set.

getPreviousRow([*\$type* = 'object'])

Parameters: • **\$type** (*string*) – Type of the requested result -
array, object, or class name

Returns: Previous row of result set, or NULL if it doesn't
exist

Return
type: mixed

Returns the previous row from the result set.

getFirstRow([*\$type* = 'object'])

Parameters: • **\$type** (*string*) – Type of the requested result -
array, object, or class name

Returns: First row of result set, or NULL if it doesn't
exist

Return
type: mixed

Returns the first row from the result set.

getLastRow([*\$type* = 'object'])

• **\$type** (*string*) – Type of the requested result -

Parameters: array, object, or class name

Returns: Last row of result set, or NULL if it doesn't exist

Return type: mixed

Returns the last row from the result set.

getFieldCount()

Returns: Number of fields in the result set

Return type: int

Returns the number of fields in the result set.

Usage: see [Result Helper Methods](#).

getFieldNames()

returns: Array of column names

rtype: array

Returns an array containing the field names in the result set.

getFieldData()

Returns: Array containing field meta-data

Return type: array

Generates an array of stdClass objects containing field meta-data.

freeResult()

Return type: void

Frees a result set.

Usage: see [Result Helper Methods](#).

Query Helper Methods

Information From Executing a Query

`$db->insertID()`

The insert ID number when performing database inserts.

Note

If using the PDO driver with PostgreSQL, or using the Interbase driver, this function requires a \$name parameter, which specifies the appropriate sequence to check for the insert id.

`$db->affectedRows()`

Displays the number of affected rows, when doing “write” type queries (insert, update, etc.).

Note

In MySQL “DELETE FROM TABLE” returns 0 affected rows. The database class has a small hack that allows it to return the correct number of affected rows. By default this hack is enabled but it can be turned off in the database driver file.

`$db->getLastQuery()`

Returns a Query object that represents the last query that was run (the query string, not the result).

Information About Your Database

\$db->countAll()

Permits you to determine the number of rows in a particular table. Submit the table name in the first parameter. This is part of Query Builder. Example:

```
echo $db->table('my_table')->countAll();
```

```
// Produces an integer, like 25
```

\$db->getPlatform()

Outputs the database platform you are running (MySQL, MS SQL, Postgres, etc...):

```
echo $db->getPlatform();
```

\$db->getVersion()

Outputs the database version you are running:

```
echo $db->getVersion();
```

Query Builder Class

CodeIgniter gives you access to a Query Builder class. This pattern allows information to be retrieved, inserted, and updated in your database with minimal scripting. In some cases only one or two lines of code are necessary to perform a database action. CodeIgniter does not require that each database table be its own class file. It instead provides a more simplified interface.

Beyond simplicity, a major benefit to using the Query Builder features is that it allows you to create database independent applications, since the query syntax is generated by each database adapter. It also allows for safer queries, since the values are escaped automatically by the system.

- [Loading the Query Builder](#)
- [Selecting Data](#)
- [Looking for Specific Data](#)
- [Looking for Similar Data](#)
- [Ordering results](#)
- [Limiting or Counting Results](#)
- [Query grouping](#)
- [Inserting Data](#)
- [Updating Data](#)
- [Deleting Data](#)
- [Method Chaining](#)
- [Resetting Query Builder](#)
- [Class Reference](#)

[Loading the Query Builder](#)

The Query Builder is loaded through the `table()` method on the database connection. This sets the `FROM` portion of the query for you and returns a new

instance of the Query Builder class:

```
$db      = \Config\Database::connect();  
$builder = $db->table('users');
```

The Query Builder is only loaded into memory when you specifically request the class, so no resources are used by default.

Selecting Data

The following functions allow you to build SQL **SELECT** statements.

\$builder->get()

Runs the selection query and returns the result. Can be used by itself to retrieve all records from a table:

```
$builder = $db->table('mytable');  
$query   = $builder->get(); // Produces: SELECT * FROM mytable
```

The first and second parameters enable you to set a limit and offset clause:

```
$query = $builder->get(10, 20);  
  
// Executes: SELECT * FROM mytable LIMIT 20, 10  
// (in MySQL. Other databases have slightly different syntax)
```

You'll notice that the above function is assigned to a variable named `$query`, which can be used to show the results:

```
$query = $builder->get();  
  
foreach ($query->getResult() as $row)  
{  
    echo $row->title;  
}
```

Please visit the [result functions](#) page for a full discussion regarding result generation.

\$builder->getCompiledSelect()

Compiles the selection query just like **\$builder->get()** but does not *run* the query. This method simply returns the SQL query as a string.

Example:

```
$sql = $builder->getCompiledSelect();  
echo $sql;  
  
// Prints string: SELECT * FROM mytable
```

The first parameter enables you to set whether or not the query builder query will be reset (by default it will be reset, just like when using **\$builder->get()**):

```
echo $builder->limit(10,20)->getCompiledSelect(false);  
  
// Prints string: SELECT * FROM mytable LIMIT 20, 10  
// (in MySQL. Other databases have slightly different syntax)  
  
echo $builder->select('title, content, date')->getCompiledSelect(  
// Prints string: SELECT title, content, date FROM mytable LIMIT  
< >
```

The key thing to notice in the above example is that the second query did not utilize **\$builder->from()** and did not pass a table name into the first parameter. The reason for this outcome is because the query has not been executed using **\$builder->get()** which resets values or reset directly using **\$builder->resetQuery()**.

\$builder->getWhere()

Identical to the **get()** function except that it permits you to add a “where” clause in the first parameter, instead of using the **db->where()** function:

```
$query = $builder->getWhere(['id' => $id], $limit, $offset);
```

Please read the about the where function below for more information.

\$builder->select()

Permits you to write the SELECT portion of your query:

```
$builder->select('title, content, date');  
$query = $builder->get();  
  
// Executes: SELECT title, content, date FROM mytable
```

Note

If you are selecting all (*) from a table you do not need to use this function. When omitted, CodeIgniter assumes that you wish to select all fields and automatically adds 'SELECT *'.


`$builder->select()` accepts an optional second parameter. If you set it to `FALSE`, CodeIgniter will not try to protect your field or table names. This is useful if you need a compound select statement where automatic escaping of fields may break them.

```
$builder->select('(SELECT SUM(payments.amount) FROM payments WHERE  
$query = $builder->get();
```



`$builder->selectMax()`

Writes a `SELECT MAX(field)` portion for your query. You can optionally include a second parameter to rename the resulting field.

```
$builder->selectMax('age');  
$query = $builder->get(); // Produces: SELECT MAX(age) as age FR  
  
$builder->selectMax('age', 'member_age');  
$query = $builder->get(); // Produces: SELECT MAX(age) as member_  

```

`$builder->selectMin()`

Writes a “`SELECT MIN(field)`” portion for your query. As with `selectMax()`, You can optionally include a second parameter to rename the resulting field.

```
$builder->selectMin('age');  
$query = $builder->get(); // Produces: SELECT MIN(age) as age FRO  

```

`$builder->selectAvg()`

Writes a “SELECT AVG(field)” portion for your query. As with selectMax(), You can optionally include a second parameter to rename the resulting field.

```
$builder->selectAvg('age');  
$query = $builder->get(); // Produces: SELECT AVG(age) as age FROM  
< >
```

`$builder->selectSum()`

Writes a “SELECT SUM(field)” portion for your query. As with selectMax(), You can optionally include a second parameter to rename the resulting field.

```
$builder->selectSum('age');  
$query = $builder->get(); // Produces: SELECT SUM(age) as age FROM  
< >
```

`$builder->from()`

Permits you to write the FROM portion of your query:

```
$builder->select('title, content, date');  
$builder->from('mytable');  
$query = $builder->get(); // Produces: SELECT title, content, da  
< >
```

Note

As shown earlier, the FROM portion of your query can be specified in the `$db->table()` function. Additional calls to `from()` will add more tables to the FROM portion of your query.

`$builder->join()`

Permits you to write the JOIN portion of your query:

```
$builder->db->table('blog');  
$builder->select('*');  
$builder->join('comments', 'comments.id = blogs.id');  
$query = $builder->get();
```

```
// Produces:  
// SELECT * FROM blogs JOIN comments ON comments.id = blogs.id
```

Multiple function calls can be made if you need several joins in one query.

If you need a specific type of JOIN you can specify it via the third parameter of the function. Options are: left, right, outer, inner, left outer, and right outer.

```
$builder->join('comments', 'comments.id = blogs.id', 'left');  
// Produces: LEFT JOIN comments ON comments.id = blogs.id
```

Looking for Specific Data

\$builder->where()

This function enables you to set **WHERE** clauses using one of four methods:

Note

All values passed to this function are escaped automatically, producing safer queries.

1. Simple key/value method:

```
$builder->where('name', $name); // Produces: WHERE name =  
< >
```

Notice that the equal sign is added for you.

If you use multiple function calls they will be chained together with AND between them:

```
$builder->where('name', $name);  
$builder->where('title', $title);  
$builder->where('status', $status);  
// WHERE name = 'Joe' AND title = 'boss' AND status = 'ac  
< >
```

2. Custom key/value method:

You can include an operator in the first parameter in order to control the comparison:

```
$builder->where('name !=', $name);  
$builder->where('id <', $id); // Produces: WHERE name !=  
< >
```

3. Associative array method:

```
$array = ['name' => $name, 'title' => $title, 'status' =>  
$builder->where($array);  
// Produces: WHERE name = 'Joe' AND title = 'boss' AND st  
< >
```

You can include your own operators using this method as well:

```
$array = ['name !=' => $name, 'id <' => $id, 'date >' =>  
$builder->where($array);  
< >
```

4. Custom string:

You can write your own clauses manually:

```
$where = "name='Joe' AND status='boss' OR status='active'"  
$builder->where($where);  
< >
```

`$builder->where()` accepts an optional third parameter. If you set it to `FALSE`, CodeIgniter will not try to protect your field or table names.

```
$builder->where('MATCH (field) AGAINST ("value")', NULL, FALSE);
```

`$builder->orWhere()`

This function is identical to the one above, except that multiple instances are joined by OR:

```
$builder->where('name !=', $name);  
$builder->orWhere('id >', $id); // Produces: WHERE name != 'Joe'  
< >
```


\$builder->whereIn()

Generates a WHERE field IN ('item', 'item') SQL query joined with AND if appropriate

```
$names = ['Frank', 'Todd', 'James'];  
$builder->whereIn('username', $names);  
// Produces: WHERE username IN ('Frank', 'Todd', 'James')
```

\$builder->orWhereIn()

Generates a WHERE field IN ('item', 'item') SQL query joined with OR if appropriate

```
$names = ['Frank', 'Todd', 'James'];  
$builder->orWhereIn('username', $names);  
// Produces: OR username IN ('Frank', 'Todd', 'James')
```

\$builder->whereNotIn()

Generates a WHERE field NOT IN ('item', 'item') SQL query joined with AND if appropriate

```
$names = ['Frank', 'Todd', 'James'];  
$builder->whereNotIn('username', $names);  
// Produces: WHERE username NOT IN ('Frank', 'Todd', 'James')
```

\$builder->orWhereNotIn()

Generates a WHERE field NOT IN ('item', 'item') SQL query joined with OR if appropriate

```
$names = ['Frank', 'Todd', 'James'];  
$builder->orWhereNotIn('username', $names);  
// Produces: OR username NOT IN ('Frank', 'Todd', 'James')
```

Looking for Similar Data

\$builder->like()

This method enables you to generate **LIKE** clauses, useful for doing

searches.

Note

All values passed to this method are escaped automatically.

Note

All `like*` method variations can be forced to be perform case-insensitive searches by passing a fifth parameter of `true` to the method. This will use platform-specific features where available otherwise, will force the values to be lowercase, i.e. `WHERE LOWER(column) LIKE '%search%'`. This may require indexes to be made for `LOWER(column)` instead of `column` to be effective.

1. Simple key/value method:

```
$builder->like('title', 'match');  
// Produces: WHERE `title` LIKE '%match%' ESCAPE '!'
```

If you use multiple method calls they will be chained together with `AND` between them:

```
$builder->like('title', 'match');  
$builder->like('body', 'match');  
// WHERE `title` LIKE '%match%' ESCAPE '!' AND `body` LI  
< >
```

If you want to control where the wildcard (%) is placed, you can use an optional third argument. Your options are ‘before’, ‘after’ and ‘both’ (which is the default).

```
$builder->like('title', 'match', 'before'); // Produc  
$builder->like('title', 'match', 'after'); // Produc  
$builder->like('title', 'match', 'both'); // Produc  
< >
```

2. Associative array method:

```
$array = ['title' => $match, 'page1' => $match, 'page2' => $match];  
$builder->like($array);  
// WHERE `title` LIKE '%match%' ESCAPE '!' AND `page1` LIKE '%match%' ESCAPE '!'
```

\$builder->orLike()

This method is identical to the one above, except that multiple instances are joined by OR:

```
$builder->like('title', 'match'); $builder->orLike('body', $match);  
// WHERE `title` LIKE '%match%' ESCAPE '!' OR `body` LIKE '%match%' ESCAPE '!'
```

\$builder->notLike()

This method is identical to like(), except that it generates NOT LIKE statements:

```
$builder->notLike('title', 'match'); // WHERE `title` NOT LIKE '%match%' ESCAPE '!'
```

\$builder->orNotLike()

This method is identical to notLike(), except that multiple instances are joined by OR:

```
$builder->like('title', 'match');  
$builder->orNotLike('body', 'match');  
// WHERE `title` LIKE '%match%' ESCAPE '!' OR `body` NOT LIKE '%match%' ESCAPE '!'
```

\$builder->groupBy()

Permits you to write the GROUP BY portion of your query:

```
$builder->groupBy("title"); // Produces: GROUP BY title
```

You can also pass an array of multiple values as well:

```
$builder->groupBy(["title", "date"]); // Produces: GROUP BY title, date
```

\$builder->distinct()


Adds the “DISTINCT” keyword to a query

```
$builder->distinct();  
$builder->get(); // Produces: SELECT DISTINCT * FROM mytable
```

\$builder->having()

Permits you to write the HAVING portion of your query. There are 2 possible syntaxes, 1 argument or 2:

```
$builder->having('user_id = 45'); // Produces: HAVING user_id =  
$builder->having('user_id', 45); // Produces: HAVING user_id =
```




You can also pass an array of multiple values as well:

```
$builder->having(['title =' => 'My Title', 'id <' => $id]);  
// Produces: HAVING title = 'My Title', id < 45
```

If you are using a database that CodeIgniter escapes queries for, you can prevent escaping content by passing an optional third argument, and setting it to FALSE.

```
$builder->having('user_id', 45); // Produces: HAVING `user_id`  
$builder->having('user_id', 45, FALSE); // Produces: HAVING use
```



\$builder->orHaving()

Identical to having(), only separates multiple clauses with “OR”.

Ordering results

\$builder->orderBy()

Lets you set an ORDER BY clause.

The first parameter contains the name of the column you would like to order by.

The second parameter lets you set the direction of the result. Options are **ASC**, **DESC** AND **RANDOM**.

```
$builder->orderBy('title', 'DESC');  
// Produces: ORDER BY `title` DESC
```

You can also pass your own string in the first parameter:

```
$builder->orderBy('title DESC, name ASC');  
// Produces: ORDER BY `title` DESC, `name` ASC
```

Or multiple function calls can be made if you need multiple fields.

```
$builder->orderBy('title', 'DESC');  
$builder->orderBy('name', 'ASC');  
// Produces: ORDER BY `title` DESC, `name` ASC
```

If you choose the **RANDOM** direction option, then the first parameters will be ignored, unless you specify a numeric seed value.

```
$builder->orderBy('title', 'RANDOM');  
// Produces: ORDER BY RAND()  
  
$builder->orderBy(42, 'RANDOM');  
// Produces: ORDER BY RAND(42)
```

Note

Random ordering is not currently supported in Oracle and will default to ASC instead.

Limiting or Counting Results

\$builder->limit()

Lets you limit the number of rows you would like returned by the query:

```
$builder->limit(10); // Produces: LIMIT 10
```

The second parameter lets you set a result offset.

```
$builder->limit(10, 20); // Produces: LIMIT 20, 10 (in MySQL. 0
```

`$builder->countAllResults()`

Permits you to determine the number of rows in a particular Query Builder query. Queries will accept Query Builder restrictors such as `where()`, `orWhere()`, `like()`, `orLike()`, etc. Example:

```
echo $builder->countAllResults(); // Produces an integer, like 2
$builder->like('title', 'match');
$builder->from('my_table');
echo $builder->countAllResults(); // Produces an integer, like 17
```

However, this method also resets any field values that you may have passed to `select()`. If you need to keep them, you can pass `FALSE` as the first parameter.

```
echo $builder->countAllResults(false); // Produces an integer, like 17
```

`$builder->countAll()`

Permits you to determine the number of rows in a particular table. Example:

```
echo $builder->countAll(); // Produces an integer, like 25
```

As is in `countAllResult` method, this method resets any field values that you may have passed to `select()` as well. If you need to keep them, you can pass `FALSE` as the first parameter.

[Query grouping](#)

Query grouping allows you to create groups of `WHERE` clauses by enclosing them in parentheses. This will allow you to create queries with complex `WHERE` clauses. Nested groups are supported. Example:

```
$builder->select('*')->from('my_table')
    ->groupStart()
        ->where('a', 'a')
    ->orGroupStart()
```

```

        ->where('b', 'b')
        ->where('c', 'c')
    ->groupEnd()
    ->groupEnd()
    ->where('d', 'd')
->get();

// Generates:
// SELECT * FROM (`my_table`) WHERE ( `a` = 'a' OR ( `b` = 'b' AN
< >

```

Note

groups need to be balanced, make sure every groupStart() is matched by a groupEnd().

\$builder->groupStart()

Starts a new group by adding an opening parenthesis to the WHERE clause of the query.

\$builder->orGroupStart()

Starts a new group by adding an opening parenthesis to the WHERE clause of the query, prefixing it with 'OR'.

\$builder->notGroupStart()

Starts a new group by adding an opening parenthesis to the WHERE clause of the query, prefixing it with 'NOT'.

\$builder->orNotGroupStart()

Starts a new group by adding an opening parenthesis to the WHERE clause of the query, prefixing it with 'OR NOT'.

\$builder->groupEnd()

Ends the current group by adding a closing parenthesis to the WHERE clause of the query.

Inserting Data

\$builder->insert()

Generates an insert string based on the data you supply, and runs the query. You can either pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = [  
    'title' => 'My title',  
    'name'  => 'My Name',  
    'date'  => 'My date'  
];  
  
$builder->insert($data);  
// Produces: INSERT INTO mytable (title, name, date) VALUES ('My  
< >
```

The first parameter is an associative array of values.

Here is an example using an object:

```
/*  
class MyClass {  
    public $title    = 'My Title';  
    public $content  = 'My Content';  
    public $date     = 'My Date';  
}  
*/  
  
$object = new MyClass;  
$builder->insert($object);  
// Produces: INSERT INTO mytable (title, content, date) VALUES ('  
< >
```

The first parameter is an object.

Note

All values are escaped automatically producing safer queries.

`$builder->getCompiledInsert()`

Compiles the insertion query just like `$builder->insert()` but does not *run* the query. This method simply returns the SQL query as a string.

Example:

```
$data = [
    'title' => 'My title',
    'name'  => 'My Name',
    'date'  => 'My date'
];

$sql = $builder->set($data)->getCompiledInsert('mytable');
echo $sql;
```

// Produces string: INSERT INTO mytable (`title`, `name`, `date`)




The second parameter enables you to set whether or not the query builder query will be reset (by default it will be—just like `$builder->insert()`):

```
echo $builder->set('title', 'My Title')->getCompiledInsert('mytab
```

// Produces string: INSERT INTO mytable (`title`) VALUES ('My Tit

```
echo $builder->set('content', 'My Content')->getCompiledInsert();
```

// Produces string: INSERT INTO mytable (`title`, `content`) VALU



The key thing to notice in the above example is that the second query did not utilize `$builder->from()` nor did it pass a table name into the first parameter. The reason this worked is because the query has not been executed using `$builder->insert()` which resets values or reset directly using `$builder->resetQuery()`.

Note

This method doesn't work for batched inserts.

\$builder->insertBatch()

Generates an insert string based on the data you supply, and runs the query. You can either pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = [
    [
        'title' => 'My title',
        'name'  => 'My Name',
        'date'  => 'My date'
    ],
    [
        'title' => 'Another title',
        'name'  => 'Another Name',
        'date'  => 'Another date'
    ]
];

$builder->insertBatch($data);
// Produces: INSERT INTO mytable (title, name, date) VALUES ('My
< >
```

The first parameter is an associative array of values.

Note

All values are escaped automatically producing safer queries.

Updating Data

\$builder->replace()

This method executes a REPLACE statement, which is basically the SQL standard for (optional) DELETE + INSERT, using *PRIMARY* and *UNIQUE* keys as the determining factor. In our case, it will save you from the need to implement complex logics with different combinations of `select()`, `update()`, `delete()` and `insert()` calls.

Example:

```

$data = [
    'title' => 'My title',
    'name'   => 'My Name',
    'date'   => 'My date'
];

$builder->replace($data);

// Executes: REPLACE INTO mytable (title, name, date) VALUES ('My

```

In the above example, if we assume that the *title* field is our primary key, then if a row containing 'My title' as the *title* value, that row will be deleted with our new row data replacing it.

Usage of the `set()` method is also allowed and all fields are automatically escaped, just like with `insert()`.

`$builder->set()`

This function enables you to set values for inserts or updates.

It can be used instead of passing a data array directly to the insert or update functions:

```

$builder->set('name', $name);
$builder->insert(); // Produces: INSERT INTO mytable (`name`) VA

```

If you use multiple function called they will be assembled properly based on whether you are doing an insert or an update:

```

$builder->set('name', $name);
$builder->set('title', $title);
$builder->set('status', $status);
$builder->insert();

```

set() will also accept an optional third parameter (`$escape`), that will prevent data from being escaped if set to `FALSE`. To illustrate the difference, here is `set()` used both with and without the escape parameter.

```

$builder->set('field', 'field+1', FALSE);
$builder->where('id', 2);

```

```

$builder->update(); // gives UPDATE mytable SET field = field+1 W
$builder->set('field', 'field+1');
$builder->where('id', 2);
$builder->update(); // gives UPDATE `mytable` SET `field` = 'fiel

```

You can also pass an associative array to this function:

```

$array = [
    'name'    => $name,
    'title'   => $title,
    'status' => $status
];

$builder->set($array);
$builder->insert();

```

Or an object:

```

/*
class MyClass {
    public $title    = 'My Title';
    public $content  = 'My Content';
    public $date     = 'My Date';
}
*/

$object = new MyClass;
$builder->set($object);
$builder->insert();

```

\$builder->update()

Generates an update string and runs the query based on the data you supply. You can pass an **array** or an **object** to the function. Here is an example using an array:

```

$data = [
    'title' => $title,
    'name'  => $name,
    'date'  => $date
];

$builder->where('id', $id);

```

```
$builder->update($data);
// Produces:
//
//      UPDATE mytable
//      SET title = '{$title}', name = '{$name}', date = '{$date}'
//      WHERE id = $id
```

Or you can supply an object:

```
/*
class MyClass {
    public $title    = 'My Title';
    public $content  = 'My Content';
    public $date     = 'My Date';
}
*/

$object = new MyClass;
$builder->where('id', $id);
$builder->update($object);
// Produces:
//
// UPDATE `mytable`
// SET `title` = '{$title}', `name` = '{$name}', `date` = '{$date}'
// WHERE id = `id`
```

Note

All values are escaped automatically producing safer queries.

You'll notice the use of the `$builder->where()` function, enabling you to set the WHERE clause. You can optionally pass this information directly into the update function as a string:

```
$builder->update($data, "id = 4");
```

Or as an array:

```
$builder->update($data, ['id' => $id]);
```

You may also use the `$builder->set()` function described above when

performing updates.

\$builder->updateBatch()

Generates an update string based on the data you supply, and runs the query. You can either pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = [
    [
        'title' => 'My title' ,
        'name'  => 'My Name 2' ,
        'date'  => 'My date 2'
    ],
    [
        'title' => 'Another title' ,
        'name'  => 'Another Name 2' ,
        'date'  => 'Another date 2'
    ]
];

$builder->updateBatch($data, 'title');
```

```
// Produces:
// UPDATE `mytable` SET `name` = CASE
// WHEN `title` = 'My title' THEN 'My Name 2'
// WHEN `title` = 'Another title' THEN 'Another Name 2'
// ELSE `name` END,
// `date` = CASE
// WHEN `title` = 'My title' THEN 'My date 2'
// WHEN `title` = 'Another title' THEN 'Another date 2'
// ELSE `date` END
// WHERE `title` IN ('My title','Another title')
```

The first parameter is an associative array of values, the second parameter is the where key.

Note

All values are escaped automatically producing safer queries.

Note

`affectedRows()` won't give you proper results with this method, due to the very nature of how it works. Instead, `updateBatch()` returns the number of rows affected.

`$builder->getCompiledUpdate()`

This works exactly the same way as `$builder->getCompiledInsert()` except that it produces an UPDATE SQL string instead of an INSERT SQL string.

For more information view documentation for *`$builder->getCompiledInsert()`*.

Note


This method doesn't work for batched updates.

Deleting Data

`$builder->delete()`

Generates a delete SQL string and runs the query.

```
$builder->delete(['id' => $id]); // Produces: // DELETE FROM myt
```



The first parameter is the where clause. You can also use the `where()` or `or_where()` functions instead of passing the data to the first parameter of the function:

```
$builder->where('id', $id);  
$builder->delete();
```

```
// Produces:  
// DELETE FROM mytable  
// WHERE id = $id
```

If you want to delete all data from a table, you can use the `truncate()` function, or `empty_table()`.

`$builder->emptyTable()`

Generates a delete SQL string and runs the query:

```
$builder->emptyTable('mytable'); // Produces: DELETE FROM mytable
```

`$builder->truncate()`

Generates a truncate SQL string and runs the query.

```
$builder->truncate();
```

```
// Produce:  
// TRUNCATE mytable
```

Note

If the TRUNCATE command isn't available, `truncate()` will execute as "DELETE FROM table".

`$builder->getCompiledDelete()`

This works exactly the same way as `$builder->getCompiledInsert()` except that it produces a DELETE SQL string instead of an INSERT SQL string.

For more information view documentation for `$builder->getCompiledInsert()`.

Method Chaining

Method chaining allows you to simplify your syntax by connecting multiple functions. Consider this example:

```
$query = $builder->select('title')
```



```
->where('id', $id)
->limit(10, 20)
->get();
```

Resetting Query Builder

\$builder->resetQuery()

Resetting Query Builder allows you to start fresh with your query without executing it first using a method like `$builder->get()` or `$builder->insert()`.


This is useful in situations where you are using Query Builder to generate SQL (ex. `$builder->getCompiledSelect()`) but then choose to, for instance, run the query:

```
// Note that the second parameter of the get_compiled_select meth
$sql = $builder->select(['field1','field2'])
        ->where('field3',5)
        ->getCompiledSelect(false);

// ...
// Do something crazy with the SQL code... like add it to a cron
// later execution or something...
// ...

$data = $builder->get()->getResultArray();

// Would execute and return an array of results of the following
// SELECT field1, field1 from mytable where field3 = 5;
```



Class Reference

class **CodeIgniterDatabaseBaseBuilder**

resetQuery()

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Resets the current Query Builder state. Useful when you want to build a query that can be canceled under certain conditions.

countAllResults(*[\$reset = TRUE]*)

Parameters: • **\$reset** (*bool*) – Whether to reset values for SELECTs

Returns: Number of rows in the query result

Return type: int

Generates a platform-specific query string that counts all records returned by an Query Builder query.

countAll(*[\$reset = TRUE]*)

Parameters: • **\$reset** (*bool*) – Whether to reset values for SELECTs

Returns: Number of rows in the query result

Return type: int

Generates a platform-specific query string that counts all records returned by an Query Builder query.

get(*[\$limit = NULL[, \$offset = NULL]]*)

Parameters: • **\$limit** (*int*) – The LIMIT clause
• **\$offset** (*int*) – The OFFSET clause

Returns: CodeIgniterDatabaseResultInterface instance (method chaining)

Return type: CodeIgniterDatabaseResultInterface

Compiles and runs SELECT statement based on the already called Query Builder methods.

getWhere(*[\$where = NULL[, \$limit = NULL[, \$offset = NULL]]]*)

Parameters: • **\$where** (*string*) – The WHERE clause
• **\$limit** (*int*) – The LIMIT clause
• **\$offset** (*int*) – The OFFSET clause

Returns: CodeIgniterDatabaseResultInterface instance (method chaining)

Return type: CodeIgniterDatabaseResultInterface

Same as `get()`, but also allows the WHERE to be added directly.

select([*\$select* = '*', *\$escape* = *NULL*])

Parameters:

- **\$select** (*string*) – The SELECT portion of a query
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a SELECT clause to a query.

selectAvg([*\$select* = "", *\$alias* = "])

Parameters:

- **\$select** (*string*) – Field to compute the average of
- **\$alias** (*string*) – Alias for the resulting value name

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a SELECT AVG(field) clause to a query.

selectMax([*\$select* = "", *\$alias* = "])

Parameters:

- **\$select** (*string*) – Field to compute the maximum of
- **\$alias** (*string*) – Alias for the resulting value name

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a SELECT MAX(field) clause to a query.

selectMin([\$select = "[, \$alias = "]])

Parameters:

- **\$select** (*string*) – Field to compute the minimum of
- **\$alias** (*string*) – Alias for the resulting value name

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a SELECT MIN(field) clause to a query.

selectSum([\$select = "[, \$alias = "]])

Parameters:

- **\$select** (*string*) – Field to compute the sum of
- **\$alias** (*string*) – Alias for the resulting value name

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a SELECT SUM(field) clause to a query.

distinct([\$val = TRUE])

Parameters:

- **\$val** (*bool*) – Desired value of the “distinct” flag

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Sets a flag which tells the query builder to add a DISTINCT clause to the SELECT portion of the query.

from(\$from)

Parameters:

- **\$from** (*mixed*) – Table name(s); string or array

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Specifies the FROM clause of a query.

join(\$table, \$cond[, \$type = "[, \$escape = NULL])

- **\$table** (*string*) – Table name to join
- **\$cond** (*string*) – The JOIN ON condition
- **\$type** (*string*) – The JOIN type
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a JOIN clause to a query.

where(\$key[, \$value = NULL[, \$escape = NULL])

- **\$key** (*mixed*) – Name of field to compare, or associative array
- **\$value** (*mixed*) – If a single key, compared to this value
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance

Return type: object

Generates the WHERE portion of the query. Separates multiple calls with 'AND'.

orWhere(\$key[, \$value = NULL[, \$escape = NULL])

- **\$key** (*mixed*) – Name of field to compare, or associative array
- **\$value** (*mixed*) – If a single key, compared to this value

- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance

Return type: object

Generates the WHERE portion of the query. Separates multiple calls with ‘OR’.

orWhereIn(*[\$key = NULL[, \$values = NULL[, \$escape = NULL]]]*)

- **\$key** (*string*) – The field to search
- **\$values** (*array*) – The values searched on
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance

Return type: object

Generates a WHERE field IN(‘item’, ‘item’) SQL query, joined with ‘OR’ if appropriate.

orWhereNotIn(*[\$key = NULL[, \$values = NULL[, \$escape = NULL]]]*)

- **\$key** (*string*) – The field to search
- **\$values** (*array*) – The values searched on
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance

Return type: object

Generates a WHERE field NOT IN(‘item’, ‘item’) SQL query, joined with ‘OR’ if appropriate.

whereIn(*[\$key = NULL[, \$values = NULL[, \$escape = NULL]]]*)

- **\$key** (*string*) – Name of field to examine
- **\$values** (*array*) – Array of target values
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance

Return type: object

Generates a WHERE field IN('item', 'item') SQL query, joined with 'AND' if appropriate.

whereNotIn(\$key = NULL[, \$values = NULL[, \$escape = NULL]]])

- **\$key** (*string*) – Name of field to examine
- **\$values** (*array*) – Array of target values
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance

Return type: object

Generates a WHERE field NOT IN('item', 'item') SQL query, joined with 'AND' if appropriate.

groupStart()

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Starts a group expression, using ANDs for the conditions inside it.

orGroupStart()

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Starts a group expression, using ORs for the conditions inside it.

notGroupStart()

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Starts a group expression, using AND NOTs for the conditions inside it.

orNotGroupStart()

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Starts a group expression, using OR NOTs for the conditions inside it.

groupEnd()

Returns: BaseBuilder instance

Return type: object

Ends a group expression.

like(\$field[, \$match = "[, \$side = 'both'[, \$escape = NULL]]])

- Parameters:**
- **\$field** (*string*) – Field name
 - **\$match** (*string*) – Text portion to match
 - **\$side** (*string*) – Which side of the expression to put the ‘%’ wildcard on
 - **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a LIKE clause to a query, separating multiple calls with AND.

orLike(\$field[, \$match = "[, \$side = 'both'[, \$escape = NULL]]])

- **\$field** (*string*) – Field name
- **\$match** (*string*) – Text portion to match

- Parameters:**
- **\$side** (*string*) – Which side of the expression to put the ‘%’ wildcard on
 - **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a LIKE clause to a query, separating multiple class with OR.

```
notLike($field[, $match = "[, $side = 'both'[, $escape =  
NULL]]])
```

- Parameters:**
- **\$field** (*string*) – Field name
 - **\$match** (*string*) – Text portion to match
 - **\$side** (*string*) – Which side of the expression to put the ‘%’ wildcard on
 - **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a NOT LIKE clause to a query, separating multiple calls with AND.

```
orNotLike($field[, $match = "[, $side = 'both'[, $escape =  
NULL]]])
```

- Parameters:**
- **\$field** (*string*) – Field name
 - **\$match** (*string*) – Text portion to match
 - **\$side** (*string*) – Which side of the expression to put the ‘%’ wildcard on
 - **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

type:

Adds a NOT LIKE clause to a query, separating multiple calls with OR.

having(\$key[, \$value = NULL[, \$escape = NULL]])

- Parameters:**
- **\$key** (*mixed*) – Identifier (string) or associative array of field/value pairs
 - **\$value** (*string*) – Value sought if \$key is an identifier
 - **\$escape** (*string*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a HAVING clause to a query, separating multiple calls with AND.

orHaving(\$key[, \$value = NULL[, \$escape = NULL]])

- Parameters:**
- **\$key** (*mixed*) – Identifier (string) or associative array of field/value pairs
 - **\$value** (*string*) – Value sought if \$key is an identifier
 - **\$escape** (*string*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a HAVING clause to a query, separating multiple calls with OR.

groupBy(\$by[, \$escape = NULL])

- Parameters:**
- **\$by** (*mixed*) – Field(s) to group by; string or array

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds a GROUP BY clause to a query.

orderBy(\$orderby[, \$direction = "[, \$escape = NULL]])

- **\$orderby** (*string*) – Field to order by
- **\$direction** (*string*) – The order requested -

Parameters: ASC, DESC or random

- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds an ORDER BY clause to a query.

limit(\$value[, \$offset = 0])

- **\$value** (*int*) – Number of rows to limit the results to

Parameters:

- **\$offset** (*int*) – Number of rows to skip

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds LIMIT and OFFSET clauses to a query.

offset(\$offset)

Parameters: • **\$offset** (*int*) – Number of rows to skip

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds an OFFSET clause to a query.

set(\$key[, \$value = "[, \$escape = NULL]])

- **\$key** (*mixed*) – Field name, or an array of field/value pairs

Parameters:

- **\$value** (*string*) – Field value, if \$key is a single field
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds field/value pairs to be passed later to `insert()`, `update()` or `replace()`.

insert([*\$set* = *NULL*[, *\$escape* = *NULL*]])

Parameters:

- **\$set** (*array*) – An associative array of field/value pairs
- **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: TRUE on success, FALSE on failure

Return type: bool

Compiles and executes an INSERT statement.

insertBatch([*\$set* = *NULL*[, *\$escape* = *NULL*[, *\$batch_size* = 100]]])

Parameters:

- **\$set** (*array*) – Data to insert
- **\$escape** (*bool*) – Whether to escape values and identifiers
- **\$batch_size** (*int*) – Count of rows to insert at once

Returns: Number of rows inserted or FALSE on failure

Return type: mixed

Compiles and executes batch INSERT statements.

Note

When more than `$batch_size` rows are provided, multiple INSERT queries will be executed, each trying to insert up to `$batch_size` rows.

setInsertBatch(*\$key*[, *\$value* = "[, \$escape = NULL]]

- Parameters:**
- **\$key** (*mixed*) – Field name or an array of field/value pairs
 - **\$value** (*string*) – Field value, if \$key is a single field
 - **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds field/value pairs to be inserted in a table later via `insertBatch()`.

update([*\$set* = NULL[, *\$where* = NULL[, *\$limit* = NULL]]])

- Parameters:**
- **\$set** (*array*) – An associative array of field/value pairs
 - **\$where** (*string*) – The WHERE clause
 - **\$limit** (*int*) – The LIMIT clause

Returns: TRUE on success, FALSE on failure

Return type: bool

Compiles and executes an UPDATE statement.

updateBatch([*\$set* = NULL[, *\$value* = NULL[, *\$batch_size* = 100]]])

- Parameters:**
- **\$set** (*array*) – Field name, or an associative array of field/value pairs
 - **\$value** (*string*) – Field value, if \$set is a single field

- **\$batch_size** (*int*) – Count of conditions to group in a single query

Returns: Number of rows updated or FALSE on failure

Return type: mixed

Compiles and executes batch UPDATE statements.

Note

When more than \$batch_size field/value pairs are provided, multiple queries will be executed, each handling up to \$batch_size field/value pairs.

setUpdateBatch(\$key[, \$value = "[, \$escape = NULL])

- Parameters:**
- **\$key** (*mixed*) – Field name or an array of field/value pairs
 - **\$value** (*string*) – Field value, if \$key is a single field
 - **\$escape** (*bool*) – Whether to escape values and identifiers

Returns: BaseBuilder instance (method chaining)

Return type: BaseBuilder

Adds field/value pairs to be updated in a table later via updateBatch().

replace([\$set = NULL])

- Parameters:**
- **\$set** (*array*) – An associative array of field/value pairs

Returns: TRUE on success, FALSE on failure

Return type: bool

Compiles and executes a REPLACE statement.

delete(*[\$where = "[, \$limit = NULL[, \$reset_data = TRUE]]]*)

Parameters:

- **\$where** (*string*) – The WHERE clause
- **\$limit** (*int*) – The LIMIT clause
- **\$reset_data** (*bool*) – TRUE to reset the query “write” clause

Returns: BaseBuilder instance (method chaining) or FALSE on failure

Return type: mixed

Compiles and executes a DELETE query.

increment(*\$column[, \$value = 1]*)

Parameters:

- **\$column** (*string*) – The name of the column to increment
- **\$value** (*int*) – The amount to increment the column by

Increments the value of a field by the specified amount. If the field is not a numeric field, like a VARCHAR, it will likely be replaced with \$value.

decrement(*\$column[, \$value = 1]*)

Parameters:

- **\$column** (*string*) – The name of the column to decrement
- **\$value** (*int*) – The amount to decrement the column by

Decrements the value of a field by the specified amount. If the field is not a numeric field, like a VARCHAR, it will likely be replaced with \$value.

truncate()

Returns: TRUE on success, FALSE on failure

Return type: bool

Executes a TRUNCATE statement on a table.

Note

If the database platform in use doesn't support TRUNCATE, a DELETE statement will be used instead.

emptyTable()

Returns: TRUE on success, FALSE on failure

Return type: bool

Deletes all records from a table via a DELETE statement.

getCompiledSelect(*[\$reset = TRUE]*)

Parameters: • **\$reset** (*bool*) – Whether to reset the current QB values or not

Returns: The compiled SQL statement as a string

Return type: string

Compiles a SELECT statement and returns it as a string.

getCompiledInsert(*[\$reset = TRUE]*)

Parameters: • **\$reset** (*bool*) – Whether to reset the current QB values or not

Returns: The compiled SQL statement as a string

Return type: string

Compiles an INSERT statement and returns it as a string.

getCompiledUpdate(*[\$reset = TRUE]*)

Parameters: • **\$reset** (*bool*) – Whether to reset the current QB values or not

Returns: The compiled SQL statement as a string

Return type: string

Compiles an UPDATE statement and returns it as a string.

getCompiledDelete([\$reset = *TRUE*])

Parameters: • **\$reset** (*bool*) – Whether to reset the current QB values or not

Returns: The compiled SQL statement as a string

Return type: string

Compiles a DELETE statement and returns it as a string.

Transactions

CodeIgniter's database abstraction allows you to use transactions with databases that support transaction-safe table types. In MySQL, you'll need to be running InnoDB or BDB table types rather than the more common MyISAM. Most other database platforms support transactions natively.

If you are not familiar with transactions we recommend you find a good online resource to learn about them for your particular database. The information below assumes you have a basic understanding of transactions.

CodeIgniter's Approach to Transactions

CodeIgniter utilizes an approach to transactions that is very similar to the process used by the popular database class ADODB. We've chosen that approach because it greatly simplifies the process of running transactions. In most cases all that is required are two lines of code.

Traditionally, transactions have required a fair amount of work to implement since they demand that you keep track of your queries and determine whether to commit or rollback based on the success or failure of your queries. This is particularly cumbersome with nested queries. In contrast, we've implemented a smart transaction system that does all this for you automatically (you can also manage your transactions manually if you choose to, but there's really no benefit).

Running Transactions

To run your queries using transactions you will use the `$this->db->transStart()` and `$this->db->transComplete()` functions as follows:

```
$this->db->transStart();  
$this->db->query('AN SQL QUERY...');
```

```
$this->db->query('ANOTHER QUERY...');  
$this->db->query('AND YET ANOTHER QUERY...');  
$this->db->transComplete();
```

You can run as many queries as you want between the start/complete functions and they will all be committed or rolled back based on success or failure of any given query.

Strict Mode

By default CodeIgniter runs all transactions in Strict Mode. When strict mode is enabled, if you are running multiple groups of transactions, if one group fails all groups will be rolled back. If strict mode is disabled, each group is treated independently, meaning a failure of one group will not affect any others.


Strict Mode can be disabled as follows:

```
$this->db->transStrict(false);
```

Managing Errors

If you have error reporting enabled in your Config/Database.php file you'll see a standard error message if the commit was unsuccessful. If debugging is turned off, you can manage your own errors like this:

```
$this->db->transStart();  
$this->db->query('AN SQL QUERY...');  
$this->db->query('ANOTHER QUERY...');  
$this->db->transComplete();  
  
if ($this->db->transStatus() === FALSE)  
{  
    // generate an error... or use the log_message() function  
}
```



Disabling Transactions

Transactions are enabled by default. If you would like to disable transactions you can do so using `$this->db->transOff()`:

```
$this->db->transOff();

$this->db->transStart();
$this->db->query('AN SQL QUERY...');
$this->db->transComplete();
```

When transactions are disabled, your queries will be auto-committed, just as they are when running queries without transactions.

Test Mode

You can optionally put the transaction system into “test mode”, which will cause your queries to be rolled back – even if the queries produce a valid result. To use test mode simply set the first parameter in the `$this->db->transStart()` function to `TRUE`:

```
$this->db->transStart(true); // Query will be rolled back
$this->db->query('AN SQL QUERY...');
$this->db->transComplete();
```

Running Transactions Manually

If you would like to run transactions manually you can do so as follows:

```
$this->db->transBegin();

$this->db->query('AN SQL QUERY...');
$this->db->query('ANOTHER QUERY...');
$this->db->query('AND YET ANOTHER QUERY...');

if ($this->db->transStatus() === FALSE)
{
    $this->db->transRollback();
}
else
{
    $this->db->transCommit();
}
```

Note

Make sure to use `$this->db->transBegin()` when running manual transactions, **NOT** `$this->db->transStart()`.

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Database Metadata

- [Table MetaData](#)
 - [List the Tables in Your Database](#)
 - [Determine If a Table Exists](#)
- [Field MetaData](#)
 - [List the Fields in a Table](#)
 - [Determine If a Field is Present in a Table](#)
 - [Retrieve Field Metadata](#)
 - [List the Indexes in a Table](#)

[Table MetaData](#)

These functions let you fetch table information.

[List the Tables in Your Database](#)

```
$db->listTables();
```

Returns an array containing the names of all the tables in the database you are currently connected to. Example:

```
$tables = $db->listTables();  
  
foreach ($tables as $table)  
{  
    echo $table;  
}
```

[Determine If a Table Exists](#)

```
$db->tableExists();
```

Sometimes it's helpful to know whether a particular table exists before running an operation on it. Returns a boolean TRUE/FALSE. Usage example:

```
if ($db->tableExists('table_name'))
{
    // some code...
}
```

Note

Replace *table_name* with the name of the table you are looking for.

Field MetaData

List the Fields in a Table

\$db->getFieldNames()

Returns an array containing the field names. This query can be called two ways:

1. You can supply the table name and call it from the \$db-> object:

```
$fields = $db->getFieldNames('table_name');

foreach ($fields as $field)
{
    echo $field;
}
```

2. You can gather the field names associated with any query you run by calling the function from your query result object:

```
$query = $db->query('SELECT * FROM some_table');

foreach ($query->getFieldNames() as $field)
{
    echo $field;
}
```

Determine If a Field is Present in a Table

\$db->fieldExists()

Sometimes it's helpful to know whether a particular field exists before performing an action. Returns a boolean TRUE/FALSE. Usage example:

```
if ($db->fieldExists('field_name', 'table_name'))
{
    // some code...
}
```

Note

Replace *field_name* with the name of the column you are looking for, and replace *table_name* with the name of the table you are looking for.

Retrieve Field Metadata

\$db->getFieldData()

Returns an array of objects containing field information.

Sometimes it's helpful to gather the field names or other metadata, like the column type, max length, etc.

Note

Not all databases provide meta-data.

Usage example:

```
$fields = $db->getFieldData('table_name');

foreach ($fields as $field)
{
    echo $field->name;
```



```

        echo $field->type;
        echo $field->max_length;
        echo $field->primary_key;
    }

```

If you have run a query already you can use the result object instead of supplying the table name:

```

$query = $db->query("YOUR QUERY");
$fields = $query->fieldData();

```

The following data is available from this function if supported by your database:

- name - column name
- max_length - maximum length of the column
- primary_key - 1 if the column is a primary key
- type - the type of the column

List the Indexes in a Table

\$db->getIndexData()

Returns an array of objects containing index information.

Usage example:

```

$keys = $db->getIndexData('table_name');

foreach ($keys as $key)
{
    echo $key->name;
    echo $key->type;
    echo $key->fields; // array of field names
}

```

The key types may be unique to the database you are using. For instance, MySQL will return one of primary, fulltext, spatial, index or unique for each key associated with a table.

Custom Function Calls

`$db->callFunction();`

This function enables you to call PHP database functions that are not natively included in CodeIgniter, in a platform independent manner. For example, let's say you want to call the `mysql_get_client_info()` function, which is **not** natively supported by CodeIgniter. You could do so like this:

```
$db->callFunction('get_client_info');
```

You must supply the name of the function, **without** the `mysql_` prefix, in the first parameter. The prefix is added automatically based on which database driver is currently being used. This permits you to run the same function on different database platforms. Obviously not all function calls are identical between platforms, so there are limits to how useful this function can be in terms of portability.

Any parameters needed by the function you are calling will be added to the second parameter.

```
$db->callFunction('some_function', $param1, $param2, etc..);
```

Often, you will either need to supply a database connection ID or a database result ID. The connection ID can be accessed using:

```
$db->connID;
```

The result ID can be accessed from within your result object, like this:

```
$query = $db->query("SOME QUERY");
```

```
$query->resultID;
```

Database Events

The Database classes contain a few [Events](#) that you can tap into in order to learn more about what is happening during the database execution. These events can be used to collect data for analysis and reporting. The [Debug Toolbar](#) uses this to collect the queries to display in the Toolbar.

The Events

DBQuery

This event is triggered whenever a new query has been run, whether successful or not. The only parameter is a [Query](#) instance of the current query. You could use this to display all queries in STDOUT, or logging to a file, or even creating tools to do automatic query analysis to help you spot potentially missing indexes, slow queries, etc. An example usage might be:

```
// In Config\Events.php
Events::on('DBQuery', 'CodeIgniter\Debug\Toolbar\Collectors\Datab

// Collect the queries so something can be done with them later.
public static function collect(CodeIgniter\Database\Query $query)
{
    static::$queries[] = $query;
}
```

◀

Utilities

The Database Utility Class contains methods that help you manage your database.

- [Get XML FROM Result](#)

[Get XML FROM Result](#)

getXMLFromResult()

This method returns the xml result from database result. You can do like this:

```
$model = new class extends \CodeIgniter\Model {
    protected $table      = 'foo';
    protected $primaryKey = 'id';
};
$db = \Closure::bind(function ($model) {
    return $model->db;
}, null, $model)($model);

$util = (new \CodeIgniter\Database\Database())->loadUtils($db);
echo $util->getXMLFromResult($model->get());
```

and it will get the following xml result:

```
<root>
  <element>
    <id>1</id>
    <name>bar</name>
  </element>
</root>
```

Modeling Data

CodeIgniter comes with rich tools for modeling and working with your database tables and records.

- [Using CodeIgniter's Model](#)
- [Using Entity Classes](#)

Using CodeIgniter's Model

- [Manual Model Creation](#)
- [CodeIgniter's Model](#)
- [Creating Your Model](#)
 - [Connecting to the Database](#)
 - [Configuring Your Model](#)
- [Working With Data](#)
 - [Finding Data](#)
 - [Saving Data](#)
 - [Deleting Data](#)
 - [Validating Data](#)
 - [Validation Placeholders](#)
 - [Protecting Fields](#)
 - [Working With Query Builder](#)
 - [Runtime Return Type Changes](#)
 - [Processing Large Amounts of Data](#)
- [Model Events](#)
 - [Defining Callbacks](#)
 - [Specifying Callbacks To Run](#)
 - [Event Parameters](#)

[Manual Model Creation](#)

You do not need to extend any special class to create a model for your application. All you need is to get an instance of the database connection and you're good to go.

```
<?php namespace App\Models;  
  
use CodeIgniter\Database\ConnectionInterface;  
  
class UserModel
```

```

{
    protected $db;

    public function __construct(ConnectionInterface &$db)
    {
        $this->db =& $db;
    }
}

```

CodeIgniter's Model

CodeIgniter does provide a model class that provides a few nice features, including:

- automatic database connection
- basic CRUD methods
- in-model validation
- automatic pagination
- and more

This class provides a solid base from which to build your own models, allowing you to rapidly build out your application's model layer.

Creating Your Model

To take advantage of CodeIgniter's model, you would simply create a new model class that extends CodeIgniter\Model:

```

<?php namespace App\Models;

use CodeIgniter\Model;

class UserModel extends Model
{
}

```

This empty class provides convenient access to the database connection, the Query Builder, and a number of additional convenience methods.

Connecting to the Database

When the class is first instantiated, if no database connection instance is passed to constructor, it will automatically connect to the default database group, as set in the configuration. You can modify which group is used on a per-model basis by adding the DBGroup property to your class. This ensures that within the model any references to `$this->db` are made through the appropriate connection.

```
<?php namespace App\Models;

use CodeIgniter\Model;

class UserModel extends Model
{
    protected $DBGroup = 'group_name';
}
```

You would replace “group_name” with the name of a defined database group from the database configuration file.

Configuring Your Model

The model class has a few configuration options that can be set to allow the class’ methods to work seamlessly for you. The first two are used by all of the CRUD methods to determine what table to use and how we can find the required records:

```
<?php namespace App\Models;

use CodeIgniter\Model;

class UserModel extends Model
{
    protected $table      = 'users';
    protected $primaryKey = 'id';

    protected $returnType = 'array';
    protected $useSoftDeletes = true;

    protected $allowedFields = ['name', 'email'];
}
```

```

        protected $useTimestamps = false;

        protected $validationRules    = [];
        protected $validationMessages = [];
        protected $skipValidation     = false;
    }

```

\$table

Specifies the database table that this model primarily works with. This only applies to the built-in CRUD methods. You are not restricted to using only this table in your own queries.

\$primaryKey

This is the name of the column that uniquely identifies the records in this table. This does not necessarily have to match the primary key that is specified in the database, but is used with methods like `find()` to know what column to match the specified value to.

\$returnType

The Model's CRUD methods will take a step of work away from you and automatically return the resulting data, instead of the Result object. This setting allows you to define the type of data that is returned. Valid values are 'array', 'object', or the fully qualified name of a class that can be used with the Result object's `getCustomResultObject()` method.

\$useSoftDeletes

If true, then any `delete*` method calls will simply set a flag in the database, instead of actually deleting the row. This can preserve data when it might be referenced elsewhere, or can maintain a "recycle bin" of objects that can be restored, or even simply preserve it as part of a security trail. If true, the `find*` methods will only return non-deleted rows, unless the `withDeleted()` method is called prior to calling the `find*` method.

This requires an INT or TINYINT field to be present in the table for storing state. The default field name is `deleted` however this name can be configured to any name of your choice by using `$deletedField` property.

\$allowedFields

This array should be updated with the field names that can be set during save, insert, or update methods. Any field names other than these will be discarded. This helps to protect against just taking input from a form and throwing it all at the model, resulting in potential mass assignment vulnerabilities.

\$useTimestamps

This boolean value determines whether the current date is automatically added to all inserts and updates. If true, will set the current time in the format specified by \$dateFormat. This requires that the table have columns named 'created_at' and 'updated_at' in the appropriate data type.

\$dateFormat

This value works with \$useTimestamps to ensure that the correct type of date value gets inserted into the database. By default, this creates DATETIME values, but valid options are: datetime, date, or int (a PHP timestamp).

\$validationRules

Contains either an array of validation rules as described in [How to save your rules](#) or a string containing the name of a validation group, as described in the same section. Described in more detail below.

\$validationMessages

Contains an array of custom error messages that should be used during validation, as described in [Setting Custom Error Messages](#). Described in more detail below.

\$skipValidation

Whether validation should be skipped during all inserts and updates. The default value is false, meaning that data will always attempt to be validated. This is primarily used by the skipValidation() method, but may be changed to true so this model will never validate.

\$beforeInsert \$afterInsert \$beforeUpdate \$afterUpdate afterFind afterDelete

These arrays allow you to specify callback methods that will be run on the data at the time specified in the property name.

Working With Data

Finding Data

Several functions are provided for doing basic CRUD work on your tables, including find(), insert(), update(), delete() and more.

find()

Returns a single row where the primary key matches the value passed in as the first parameter:

```
$user = $userModel->find($user_id);
```

The value is returned in the format specified in \$returnType.

You can specify more than one row to return by passing an array of primaryKey values instead of just one:

```
$users = $userModel->find([1, 2, 3]);
```

If no parameters are passed in, will return all rows in that model's table, effectively acting like findAll(), though less explicit.

findAll()

Returns all results:

```
$users = $userModel->findAll();
```

This query may be modified by interjecting Query Builder commands as needed prior to calling this method:

```
$users = $userModel->where('active', 1)
->findAll();
```

You can pass in a limit and offset values as the first and second parameters, respectively:

```
$users = $userModel->findAll($limit, $offset);
```

first()

Returns the first row in the result set. This is best used in combination with the query builder.

```
$user = $userModel->where('deleted', 0)
->first();
```

withDeleted()

If \$useSoftDeletes is true, then the find* methods will not return any rows where 'deleted' = 1'. To temporarily override this, you can use the withDeleted() method prior to calling the find* method.

```
// Only gets non-deleted rows (deleted = 0)
$activeUsers = $userModel->findAll();

// Gets all rows
$allUsers = $userModel->withDeleted()
->findAll();
```

onlyDeleted()

Whereas withDeleted() will return both deleted and not-deleted rows, this method modifies the next find* methods to return only soft deleted rows:

```
$deletedUsers = $userModel->onlyDeleted()
->findAll();
```

Saving Data

insert()

An associative array of data is passed into this method as the only parameter to create a new row of data in the database. The array's keys must match the name of the columns in \$table, while the array's values are the values to save for that key:

```
$data = [  
    'username' => 'darth',  
    'email'     => 'd.vader@theempire.com'  
];  
  
$userModel->insert($data);
```

update()

Updates an existing record in the database. The first parameter is the \$primaryKey of the record to update. An associative array of data is passed into this method as the second parameter. The array's keys must match the name of the columns in \$table, while the array's values are the values to save for that key:

```
$data = [  
    'username' => 'darth',  
    'email'     => 'd.vader@theempire.com'  
];  
  
$userModel->update($id, $data);
```

Multiple records may be updated with a single call by passing an array of primary keys as the first parameter:

```
$data = [  
    'active' => 1  
];  
  
$userModel->update([1, 2, 3], $data);
```

When you need a more flexible solution, you can leave the parameters empty and it functions like the Query Builder's update command, with the added benefit of validation, events, etc:

```
$userModel  
->whereIn('id', [1, 2, 3])
```

```
->set(['active' => 1]
->update());
```

save()

This is a wrapper around the insert() and update() methods that handles inserting or updating the record automatically, based on whether it finds an array key matching the \$primaryKey value:

```
// Defined as a model property
$primaryKey = 'id';

// Does an insert()
$data = [
    'username' => 'darth',
    'email'     => 'd.vader@theempire.com'
];

$userModel->save($data);

// Performs an update, since the primary key, 'id', is found.
$data = [
    'id'         => 3,
    'username'   => 'darth',
    'email'      => 'd.vader@theempire.com'
];
$userModel->save($data);
```

The save method also can make working with custom class result objects much simpler by recognizing a non-simple object and grabbing its public and protected values into an array, which is then passed to the appropriate insert or update method. This allows you to work with Entity classes in a very clean way. Entity classes are simple classes that represent a single instance of an object type, like a user, a blog post, job, etc. This class is responsible for maintaining the business logic surrounding the object itself, like formatting elements in a certain way, etc. They shouldn't have any idea about how they are saved to the database. At their simplest, they might look like this:

```
namespace App\Entities;

class Job
{
    protected $id;
```

```

    protected $name;
    protected $description;

    public function __get($key)
    {
        if (property_exists($this, $key))
        {
            return $this->$key;
        }
    }

    public function __set($key, $value)
    {
        if (property_exists($this, $key))
        {
            $this->$key = $value;
        }
    }
}

```

A very simple model to work with this might look like:

```

use CodeIgniter\Model;

class JobModel extends Model
{
    protected $table = 'jobs';
    protected $returnType = '\App\Entities\Job';
    protected $allowedFields = [
        'name', 'description'
    ];
}

```

This model works with data from the jobs table, and returns all results as an instance of App\Entities\Job. When you need to persist that record to the database, you will need to either write custom methods, or use the model's save() method to inspect the class, grab any public and private properties, and save them to the database:

```

// Retrieve a Job instance
$job = $model->find(15);

// Make some changes
$job->name = "Foobar";

```



```
// Save the changes  
$model->save($job);
```

Note

If you find yourself working with Entities a lot, CodeIgniter provides a built-in [Entity class](#) that provides several handy features that make developing Entities simpler.

Deleting Data

delete()

Takes a primary key value as the first parameter and deletes the matching record from the model's table:

```
$userModel->delete(12);
```

If the model's `$useSoftDeletes` value is true, this will update the row to set 'deleted = 1'. You can force a permanent delete by setting the second parameter as true.

An array of primary keys can be passed in as the first parameter to delete multiple records at once:

```
$userModel->delete([1, 2, 3]);
```

If no parameters are passed in, will act like the Query Builder's delete method, requiring a where call previously:

```
$userModel->where('id', 12)->delete();
```

purgeDeleted()

Cleans out the database table by permanently removing all rows that have 'deleted = 1'.

```
$userModel->purgeDeleted();
```

Validating Data

For many people, validating data in the model is the preferred way to ensure the data is kept to a single standard, without duplicating code. The Model class provides a way to automatically have all data validated prior to saving to the database with the `insert()`, `update()`, or `save()` methods.

The first step is to fill out the `$validationRules` class property with the fields and rules that should be applied. If you have custom error message that you want to use, place them in the `$validationMessages` array:

```
class UserModel extends Model
{
    protected $validationRules = [
        'username' => 'required|alpha_numeric_space|m
        'email'     => 'required|valid_email|is_unique
        'password'  => 'required|min_length[8]',
        'pass_confirm' => 'required_with[password]|matche
    ];

    protected $validationMessages = [
        'email' => [
            'is_unique' => 'Sorry. That email has alr
        ]
    ];
}
```



Now, whenever you call the `insert()`, `update()`, or `save()` methods, the data will be validated. If it fails, the model will return boolean **false**. You can use the `errors()` method to retrieve the validation errors:

```
if ($model->save($data) === false)
{
    return view('updateUser', ['errors' => $model->errors()]);
}
```

This returns an array with the field names and their associated errors that can be used to either show all of the errors at the top of the form, or to display them individually:

```
<?php if (! empty($errors)) : ?>
    <div class="alert alert-danger">
```

```

        <?php foreach ($errors as $field => $error) : ?>
            <p><?= $error ?></p>
        <?php endforeach ?>
    </div>
<?php endif ?>

```

If you'd rather organize your rules and error messages within the Validation configuration file, you can do that and simply set `$validationRules` to the name of the validation rule group you created:

```

class UserModel extends Model
{
    protected $validationRules = 'users';
}

```

Validation Placeholders

The model provides a simple method to replace parts of your rules based on data that's being passed into it. This sounds fairly obscure but can be especially handy with the `is_unique` validation rule. Placeholders are simply the name of the field (or array key) that was passed in as `$data` surrounded by curly brackets. It will be replaced by the **value** of the matched incoming field. An example should clarify this:

```

protected $validationRules = [
    'email' => 'required|valid_email|is_unique[users.email,id,{id}];
];

```

In this set of rules, it states that the email address should be unique in the database, except for the row that has an id matching the placeholder's value. Assuming that the form POST data had the following:

```

$_POST = [
    'id' => 4,
    'email' => 'foo@example.com'
]

```

then the `{id}` placeholder would be replaced with the number **4**, giving this revised rule:

```

protected $validationRules = [

```

```
'email' => 'required|valid_email|is_unique[users.email,id,4]'
```

```
];
```

So it will ignore the row in the database that has `id=4` when it verifies the email is unique.

This can also be used to create more dynamic rules at runtime, as long as you take care that any dynamic keys passed in don't conflict with your form data.

Protecting Fields

To help protect against Mass Assignment Attacks, the Model class **requires** that you list all of the field names that can be changed during inserts and updates in the `$allowedFields` class property. Any data provided in addition to these will be removed prior to hitting the database. This is great for ensuring that timestamps, or primary keys do not get changed.

```
protected $allowedFields = ['name', 'email', 'address'];
```

Occasionally, you will find times where you need to be able to change these elements. This is often during testing, migrations, or seeds. In these cases, you can turn the protection on or off:

```
$model->protect(false)  
->insert($data)  
->protect(true);
```

Working With Query Builder

You can get access to a shared instance of the Query Builder for that model's database connection any time you need it:

```
$builder = $userModel->builder();
```

This builder is already setup with the model's `$table`.

You can also use Query Builder methods and the Model's CRUD methods in the same chained call, allowing for very elegant use:

```
$users = $userModel->where('status', 'active')
```

```
->orderBy('last_login', 'asc')  
->findAll();
```

Note

You can also access the model's database connection seamlessly:

```
$user_name = $userModel->escape($name);
```

Runtime Return Type Changes

You can specify the format that data should be returned as when using the `find*()` methods as the class property, `$returnType`. There may be times that you would like the data back in a different format, though. The Model provides methods that allow you to do just that.

Note

These methods only change the return type for the next `find*()` method call. After that, it is reset to its default value.

asArray()

Returns data from the next `find*()` method as associative arrays:


```
$users = $userModel->asArray()->where('status', 'active')->findAll
```



asObject()

Returns data from the next `find*()` method as standard objects or custom class instances:

```
// Return as standard objects  
$users = $userModel->asObject()->where('status', 'active')->findAll  
  
// Return as custom class instances  
$users = $userModel->asObject('User')->where('status', 'active')->findAll
```



Processing Large Amounts of Data

Sometimes, you need to process large amounts of data and would run the risk of running out of memory. To make this simpler, you may use the `chunk()` method to get smaller chunks of data that you can then do your work on. The first parameter is the number of rows to retrieve in a single chunk. The second parameter is a Closure that will be called for each row of data.

This is best used during cronjobs, data exports, or other large tasks.

```
$userModel->chunk(100, function ($data)
{
    // do something.
    // $data is a single row of data.
});
```

Model Events

There are several points within the model's execution that you can specify multiple callback methods to run. These methods can be used to normalize data, hash passwords, save related entities, and much more. The following points in the model's execution can be affected, each through a class property: **\$beforeInsert**, **\$afterInsert**, **\$beforeUpdate**, **afterUpdate**, **afterFind**, and **afterDelete**.

Defining Callbacks

You specify the callbacks by first creating a new class method in your model to use. This class will always receive a `$data` array as its only parameter. The exact contents of the `$data` array will vary between events, but will always contain a key named **data** that contains the primary data passed to original method. In the case of the `insert*` or `update*` methods, that will be the key/value pairs that are being inserted into the database. The main array will also contain the other values passed to the method, and be detailed later. The callback method must return the original `$data` array so other callbacks have the full information.

```
protected function hashPassword(array $data)
{
    if (! isset($data['data']['password'])) return $data;

    $data['data']['password_hash'] = password_hash($data['data']['password']);
    unset($data['data']['password']);

    return $data;
}
```

Specifying Callbacks To Run

You specify when to run the callbacks by adding the method name to the appropriate class property (beforeInsert, afterUpdate, etc). Multiple callbacks can be added to a single event and they will be processed one after the other. You can use the same callback in multiple events:

```
protected $beforeInsert = ['hashPassword'];
protected $beforeUpdate = ['hashPassword'];
```

Event Parameters

Since the exact data passed to each callback varies a bit, here are the details on what is in the \$data parameter passed to each event:

Event	\$data contents
beforeInsert	data = the key/value pairs that are being inserted. If an object or Entity class is passed to the insert method, it is first converted to an array.
afterInsert	data = the original key/value pairs being inserted. result = the results of the insert() method used through the Query Builder.
beforeUpdate	id = the primary key of the row being updated. data = the key/value pairs that are being inserted. If an object or Entity class is passed to the insert method, it is first converted to an array.
afterUpdate	id = the primary key of the row being updated. data = the original key/value pairs being updated. result = the results

	of the update() method used through the Query Builder.
afterFind	Varies by find* method. See the following:
• find()	id = the primary key of the row being searched for. data = The resulting row of data, or null if no result found.
• findAll()	data = the resulting rows of data, or null if no result found. limit = the number of rows to find. offset = the number of rows to skip during the search.
• first()	data = the resulting row found during the search, or null if none found.
beforeDelete	Varies by delete* method. See the following:
• delete()	id = primary key of row being deleted. purge = boolean whether soft-delete rows should be hard deleted.
afterDelete	Varies by delete* method. See the following:
• delete()	id = primary key of row being deleted. purge = boolean whether soft-delete rows should be hard deleted. result = the result of the delete() call on the Query Builder. data = unused.

Working With Entities

CodeIgniter supports Entity classes as a first-class citizen in it's database layer, while keeping them completely optional to use. They are commonly used as part of the Repository pattern, but can be used directly with the [Model](#) if that fits your needs better.

- [Entity Usage](#)
 - [Create the Entity Class](#)
 - [Create the Model](#)
 - [Working With the Entity Class](#)
 - [Filling Properties Quickly](#)
- [Handling Business Logic](#)
- [Data Mapping](#)
- [Mutators](#)
 - [Date Mutators](#)
 - [Property Casting](#)
 - [Array/Json Casting](#)

[Entity Usage](#)

At its core, an Entity class is simply a class that represents a single database row. It has class properties to represent the database columns, and provides any additional methods to implement the business logic for that row. The core feature, though, is that it doesn't know anything about how to persist itself. That's the responsibility of the model or the repository class. That way, if anything changes on how you need to save the object, you don't have to change how that object is used throughout the application. This makes it possible to use JSON or XML files to store the objects during a rapid prototyping stage, and then easily switch to a database when you've proven the concept works.

Lets walk through a very simple User Entity and how we'd work with it to help make things clear.

Assume you have a database table named `users` that has the following schema:

<code>id</code>	- integer
<code>username</code>	- string
<code>email</code>	- string
<code>password</code>	- string
<code>created_at</code>	- datetime

Create the Entity Class

Now create a new Entity class. Since there's no default location to store these classes, and it doesn't fit in with the existing directory structure, create a new directory at **app/Entities**. Create the Entity itself at **app/Entities/User.php**.

```
<?php namespace App\Entities;

use CodeIgniter\Entity;

class User extends Entity
{
    protected $id;
    protected $username;
    protected $email;
    protected $password;
    protected $created_at;
    protected $updated_at;
}
```

At its simplest, this is all you need to do, though we'll make it more useful in a minute. Note that all of the database columns are represented in the Entity. This is required for the Model to populate the fields.

Create the Model

Create the model first at **app/Models/UserModel.php** so that we can interact with it:

```
<?php namespace App\Models;
```

```

use CodeIgniter\Model;

class UserModel extends Model
{
    protected $table          = 'users';
    protected $allowedFields = [
        'username', 'email', 'password'
    ];
    protected $returnType     = 'App\Entities\User';
    protected $useTimestamps = true;
}

```

The model uses the users table in the database for all of its activities. We've set the \$allowedFields property to include all of the fields that we want outside classes to change. The id, created_at, and updated_at fields are handled automatically by the class or the database, so we don't want to change those. Finally, we've set our Entity class as the \$returnType. This ensures that all methods on the model that return rows from the database will return instances of our User Entity class instead of an object or array like normal.

Working With the Entity Class

Now that all of the pieces are in place, you would work with the Entity class as you would any other class:

```

$user = $userModel->find($id);

// Display
echo $user->username;
echo $user->email;

// Updating
unset($user->username);
if (! isset($user->username))
{
    $user->username = 'something new';
}
$userModel->save($user);

// Create
$user = new App\Entities\User();
$user->username = 'foo';

```

```
$user->email      = 'foo@example.com';  
$userModel->save($user);
```

You may have noticed that the User class has all of the properties as **protected** not **public**, but you can still access them as if they were public properties. The base class, **CodeIgniterEntity**, takes care of this for you, as well as providing the ability to check the properties with **isset()**, or **unset()** the property.

When the User is passed to the model's **save()** method, it automatically takes care of reading the protected properties and saving any changes to columns listed in the model's **\$allowedFields** property. It also knows whether to create a new row, or update an existing one.

Filling Properties Quickly

The Entity class also provides a method, **fill()** that allows you to shove an array of key/value pairs into the class and populate the class properties. Only properties that already exist on the class can be populated in this way.

```
$data = $this->request->getPost();  
  
$user = new App\Entities\User();  
$user->fill($data);  
$userModel->save($user);
```

Handling Business Logic

While the examples above are convenient, they don't help enforce any business logic. The base Entity class implements some smart **__get()** and **__set()** methods that will check for special methods and use those instead of using the class properties directly, allowing you to enforce any business logic or data conversion that you need.

Here's an updated User entity to provide some examples of how this could be used:

```
<?php namespace App\Entities;
```

```

use CodeIgniter\Entity;
use CodeIgniter\I18n\Time;

class User extends Entity
{
    protected $id;
    protected $username;
    protected $email;
    protected $password;
    protected $created_at;
    protected $updated_at;

    public function setPassword(string $pass)
    {
        $this->password = password_hash($pass, PASSWORD_BCRYPT);

        return $this;
    }

    public function setCreatedAt(string $dateString)
    {
        $this->created_at = new Time($dateString, 'UTC');

        return $this;
    }

    public function getCreatedAt(string $format = 'Y-m-d H:i:s')
    {
        // Convert to CodeIgniter\I18n\Time object
        $this->created_at = $this->mutateDate($this->created_at);

        $timezone = $this->timezone ?? app_timezone();

        $this->created_at->setTimezone($timezone);

        return $this->created_at->format($format);
    }
}

```

The first thing to notice is the name of the methods we've added. For each one, the class expects the snake_case column name to be converted into PascalCase, and prefixed with either set or get. These methods will then be automatically called whenever you set or retrieve the class property using the direct syntax (i.e. \$user->email). The methods do not need to be public unless you want them accessed from other classes. For example, the created_at

class property will be accessed through the `setCreatedAt()` and `getCreatedAt()` methods.

Note

This only works when trying to access the properties from outside of the track. Any methods internal to the class must call the `setX()` and `getX()` methods directly.

In the `setPassword()` method we ensure that the password is always hashed.

In `setCreatedAt()` we convert the string we receive from the model into a `DateTime` object, ensuring that our timezone is UTC so we can easily convert the the viewer's current timezone. In `getCreatedAt()`, it converts the time to a formatted string in the application's current timezone.

While fairly simple, these examples show that using Entity classes can provide a very flexible way to enforce business logic and create objects that are pleasant to use.

```
// Auto-hash the password - both do the same thing  
$user->password = 'my great password';  
$user->setPassword('my great password');
```

Data Mapping

At many points in your career, you will run into situations where the use of an application has changed and the original column names in the database no longer make sense. Or you find that your coding style prefers camelCase class properties, but your database schema required snake_case names. These situations can be easily handled with the Entity class' data mapping features.

As an example, imagine your have the simplified User Entity that is used throughout your application:

```
<?php namespace App\Entities;
```

```

use CodeIgniter\Entity;

class User extends Entity
{
    protected $id;
    protected $name;           // Represents a username
    protected $email;
    protected $password;
    protected $created_at;
    protected $updated_at;
}

```

Your boss comes to you and says that no one uses usernames anymore, so you're switching to just use emails for login. But they do want to personalize the application a bit, so they want you to change the name field to represent a user's full name now, not their username like it does currently. To keep things tidy and ensure things continue making sense in the database you whip up a migration to rename the *name* field to *full_name* for clarity.

Ignoring how contrived this example is, we now have two choices on how to fix the User class. We could modify the class property from *\$name* to *\$full_name*, but that would require changes throughout the application. Instead, we can simply map the *full_name* column in the database to the *\$name* property, and be done with the Entity changes:

```

<?php namespace App\Entities;

use CodeIgniter\Entity;

class User extends Entity
{
    protected $id;
    protected $name;           // Represents a full name now
    protected $email;
    protected $password;
    protected $created_at;
    protected $updated_at;

    protected $_options = [
        'datamap' => [
            'full_name' => 'name'
        ],
        'dates' => ['created_at', 'updated_at', 'deleted_at'],
        'casts' => []
    ]
}

```

```
    ];  
}
```

By adding our new database name to the `$datamap` array, we can tell the class what class property the database column should be accessible through. The key of the array is the name of the column in the database, where the value in the array is class property to map it to.

In this example, when the model sets the `full_name` field on the `User` class, it actually assigns that value to the class' `$name` property, so it can be set and retrieved through `$user->name`. The value will still be accessible through the original `$user->full_name`, also, as this is needed for the model to get the data back out and save it to the database. However, `unset` and `isset` only work on the mapped property, `$name`, not on the original name, `full_name`.

Mutators

Date Mutators

By default, the `Entity` class will convert fields named `created_at`, `updated_at`, or `deleted_at` into [Time](#) instances whenever they are set or retrieved. The `Time` class provides a large number of helpful methods in a immutable, localized way.

You can define which properties are automatically converted by adding the name to the **`options['dates']`** array:

```
<?php namespace App\Entities;  
  
use CodeIgniter\Entity;  
  
class User extends Entity  
{  
    protected $id;  
    protected $name;           // Represents a full name now  
    protected $email;  
    protected $password;  
    protected $created_at;  
    protected $updated_at;
```



```

        protected $_options = [
            'dates' => ['created_at', 'updated_at', 'deleted_at'],
            'casts' => [],
            'datamap' => []
        ];
    }

```

Now, when any of those properties are set, they will be converted to a Time instance, using the application's current timezone, as set in **app/Config/App.php**:

```

$user = new App\Entities\User();

// Converted to Time instance
$user->created_at = 'April 15, 2017 10:30:00';

// Can now use any Time methods:
echo $user->created_at->humanize();
echo $user->created_at->setTimezone('Europe/London')->toDateStrin
<

```

Property Casting

You can specify that properties in your Entity should be converted to common data types with the **casts** entry in the **\$_options** property. The **casts** option should be an array where the key is the name of the class property, and the value is the data type it should be cast to. Casting only affects when values are read. No conversions happen that affect the permanent value in either the entity or the database. Properties can be cast to any of the following data types: **integer**, **float**, **double**, **string**, **boolean**, **object**, **array**, **datetime**, and **timestamp**. Add question mark at the beginning of type to mark property as nullable, i.e. **?string**, **?integer**.

For example, if you had a User entity with an **is_banned** property, you can cast it as a boolean:

```

<?php namespace App\Entities;

use CodeIgniter\Entity;

class User extends Entity
{

```

```

protected $is_banned;

protected $_options = [
    'casts' => [
        'is_banned' => 'boolean',
        'is_banned_nullable' => '?boolean'
    ],
    'dates' => ['created_at', 'updated_at', 'deleted_at'],
    'datamap' => []
];
}

```

Array/Json Casting

Array/Json casting is especially useful with fields that store serialized arrays or json in them. When cast as:

- an **array**, they will automatically be unserialized,
- a **json**, they will automatically be set as an value of `json_decode($value, false)`,
- a **json-array**, they will automatically be set as an value of `json_decode($value, true)`,

when you read the property's value. Unlike the rest of the data types that you can cast properties into, the:

- **array** cast type will serialize,
- **json** and **json-array** cast will use `json_encode` function on

the value whenever the property is set:

```

<?php namespace App\Entities;

use CodeIgniter\Entity;

class User extends Entity
{
    protected $options;

    protected $_options = [
        'casts' => [
            'options' => 'array',
            'options_object' => 'json',

```

```
        'options_array' => 'json-array'
    ],
    'dates' => ['created_at', 'updated_at', 'deleted_at'],
    'datamap' => []
];

$user      = $userModel->find(15);
$options   = $user->options;

$options['foo'] = 'bar';

$user->options = $options;
$userModel->save($user);
```

Managing Databases

CodeIgniter comes with tools to restructure or seed your database.

- [Database Manipulation with Database Forge](#)
- [Database Migrations](#)
- [Database Seeding](#)

Database Forge Class

The Database Forge Class contains methods that help you manage your database.

- [Initializing the Forge Class](#)
- [Creating and Dropping Databases](#)
- [Creating and Dropping Tables](#)
 - [Adding fields](#)
 - [Adding Keys](#)
 - [Adding Foreign Keys](#)
 - [Creating a table](#)
 - [Dropping a table](#)
 - [Dropping a Foreign Key](#)
 - [Renaming a table](#)
- [Modifying Tables](#)
 - [Adding a Column to a Table](#)
 - [Dropping a Column From a Table](#)
 - [Modifying a Column in a Table](#)
- [Class Reference](#)

[Initializing the Forge Class](#)

Important

In order to initialize the Forge class, your database driver must already be running, since the forge class relies on it.

Load the Forge Class as follows:

```
$forge = \Config\Database::forge();
```

You can also pass another database group name to the DB Forge loader, in case the database you want to manage isn't the default one:

```
$this->myforge = $this->load->dbforge('other_db');
```

In the above example, we're passing a the name of a different database group to connect to as the first parameter.

Creating and Dropping Databases

`$forge->createDatabase('db_name')`

Permits you to create the database specified in the first parameter. Returns TRUE/FALSE based on success or failure:

```
if ($forge->createDatabase('my_db'))
{
    echo 'Database created!';
}
```

`$forge->dropDatabase('db_name')`

Permits you to drop the database specified in the first parameter. Returns TRUE/FALSE based on success or failure:

```
if ($forge->dropDatabase('my_db'))
{
    echo 'Database deleted!';
}
```

Creating and Dropping Tables

There are several things you may wish to do when creating tables. Add fields, add keys to the table, alter columns. CodeIgniter provides a mechanism for this.

Adding fields

Fields are normally created via an associative array. Within the array you

must include a 'type' key that relates to the datatype of the field. For example, INT, VARCHAR, TEXT, etc. Many datatypes (for example VARCHAR) also require a 'constraint' key.

```
$fields = [  
    'users' => [  
        'type'      => 'VARCHAR',  
        'constraint' => 100,  
    ],  
];  
// will translate to "users VARCHAR(100)" when the field is added
```

Additionally, the following key/values can be used:

- unsigned/true : to generate “UNSIGNED” in the field definition.
- default/value : to generate a default value in the field definition.
- null/true : to generate “NULL” in the field definition. Without this, the field will default to “NOT NULL”.
- auto_increment/true : generates an auto_increment flag on the field.
Note that the field type must be a type that supports this, such as integer.
- unique/true : to generate a unique key for the field definition.

```
$fields = [  
    'id' => [  
        'type'      => 'INT',  
        'constraint' => 5,  
        'unsigned'   => true,  
        'auto_increment' => true  
    ],  
    'title' => [  
        'type'      => 'VARCHAR',  
        'constraint' => '100',  
        'unique'     => true,  
    ],  
    'author' => [  
        'type'      => 'VARCHAR',  
        'constraint' => 100,  
        'default'    => 'King of Town',  
    ],  
    'description' => [  
        'type'      => 'TEXT',  
        'null'       => true,  
    ],  
];
```

```

        'status'      => [
                        'type'      => 'ENUM',
                        'constraint' => ['publish', 'pending', 'draft',
                        'default'    => 'pending',
        ],
    ];

```

After the fields have been defined, they can be added using `$forge->addField($fields);` followed by a call to the `createTable()` method.

`$forge->addField()`

The add fields method will accept the above array.

Passing strings as fields

If you know exactly how you want a field to be created, you can pass the string into the field definitions with `addField()`

```

$forge->addField("label varchar(100) NOT NULL DEFAULT 'default la

```

Note

Passing raw strings as fields cannot be followed by `addKey()` calls on those fields.

Note

Multiple calls to `addField()` are cumulative.

Creating an id field

There is a special exception for creating id fields. A field with type `id` will automatically be assigned as an `INT(9)` auto_incrementing Primary Key.

```

$forge->addField('id');

```



```
// gives id INT(9) NOT NULL AUTO_INCREMENT
```

Adding Keys

Generally speaking, you'll want your table to have Keys. This is accomplished with `$forge->addKey('field')`. The optional second parameter set to `TRUE` will make it a primary key and the third parameter set to `TRUE` will make it a unique key. Note that `addKey()` must be followed by a call to `createTable()`.

Multiple column non-primary keys must be sent as an array. Sample output below is for MySQL.

```
$forge->addKey('blog_id', TRUE);  
// gives PRIMARY KEY `blog_id` (`blog_id`)  
  
$forge->addKey('blog_id', TRUE);  
$forge->addKey('site_id', TRUE);  
// gives PRIMARY KEY `blog_id_site_id` (`blog_id`, `site_id`)  
  
$forge->addKey('blog_name');  
// gives KEY `blog_name` (`blog_name`)  
  
$forge->addKey(['blog_name', 'blog_label']);  
// gives KEY `blog_name_blog_label` (`blog_name`, `blog_label`)  
  
$forge->addKey(['blog_id', 'uri'], FALSE, TRUE);  
// gives UNIQUE KEY `blog_id_uri` (`blog_id`, `uri`)
```

To make code reading more objective it is also possible to add primary and unique keys with specific methods:

```
$forge->addPrimaryKey('blog_id');  
// gives PRIMARY KEY `blog_id` (`blog_id`)  
  
$forge->addUniqueKey(['blog_id', 'uri']);  
// gives UNIQUE KEY `blog_id_uri` (`blog_id`, `uri`)
```

Adding Foreign Keys

Foreign Keys help to enforce relationships and actions across your tables. For tables that support Foreign Keys, you may add them directly in forge:

```
$forge->addForeignKey('users_id','users','id');  
// gives CONSTRAINT `TABLENAME_users_foreign` FOREIGN KEY(`users_`  
< >
```

You can specify the desired action for the “on delete” and “on update” properties of the constraint:

```
$forge->addForeignKey('users_id','users','id','CASCADE','CASCADE'  
// gives CONSTRAINT `TABLENAME_users_foreign` FOREIGN KEY(`users_`  
< >
```

Creating a table

After fields and keys have been declared, you can create a new table with

```
$forge->createTable('table_name');  
// gives CREATE TABLE table_name
```

An optional second parameter set to TRUE adds an “IF NOT EXISTS” clause into the definition

```
$forge->createTable('table_name', TRUE);  
// gives CREATE TABLE IF NOT EXISTS table_name
```

You could also pass optional table attributes, such as MySQL’s ENGINE:

```
$attributes = ['ENGINE' => 'InnoDB'];  
$forge->createTable('table_name', FALSE, $attributes);  
// produces: CREATE TABLE `table_name` (...) ENGINE = InnoDB DEFA  
< >
```

Note

Unless you specify the CHARACTER SET and/or COLLATE attributes, createTable() will always add them with your configured *charset* and *DBCollat* values, as long as they are not empty (MySQL only).

Dropping a table

Execute a DROP TABLE statement and optionally add an IF EXISTS clause.

```
// Produces: DROP TABLE table_name
$forge->dropTable('table_name');

// Produces: DROP TABLE IF EXISTS table_name
$forge->dropTable('table_name', TRUE);
```

Dropping a Foreign Key

Execute a DROP FOREIGN KEY.

```
// Produces: ALTER TABLE 'tablename' DROP FOREIGN KEY 'users_fore
$forge->dropForeignKey('tablename', 'users_foreign');
```



Note

SQLite database driver does not support dropping of foreign keys.

Renaming a table

Executes a TABLE rename

```
$forge->renameTable('old_table_name', 'new_table_name');
// gives ALTER TABLE old_table_name RENAME TO new_table_name
```

Modifying Tables

Adding a Column to a Table

\$forge->addColumn()

The addColumn() method is used to modify an existing table. It accepts the same field array as above, and can be used for an unlimited number of additional fields.

```
$fields = [
    'preferences' => ['type' => 'TEXT']
];
$forge->addColumn('table_name', $fields);
```

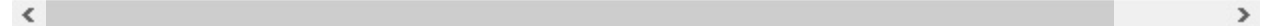
```
// Executes: ALTER TABLE table_name ADD preferences TEXT
```

If you are using MySQL or CUBIRD, then you can take advantage of their AFTER and FIRST clauses to position the new column.

Examples:

```
// Will place the new column after the `another_field` column:
$fields = [
    'preferences' => ['type' => 'TEXT', 'after' => 'another_f
];

// Will place the new column at the start of the table definition
$fields = [
    'preferences' => ['type' => 'TEXT', 'first' => TRUE]
];
```



Dropping a Column From a Table

\$forge->dropColumn()

Used to remove a column from a table.

```
$forge->dropColumn('table_name', 'column_to_drop');
```

Modifying a Column in a Table

\$forge->modifyColumn()

The usage of this method is identical to addColumn(), except it alters an existing column rather than adding a new one. In order to change the name you can add a “name” key into the field defining array.

```
$fields = [
    'old_name' => [
        'name' => 'new_name',
        'type' => 'TEXT',
    ],
];
$forge->modifyColumn('table_name', $fields);
// gives ALTER TABLE table_name CHANGE old_name new_name TEXT
```

Class Reference

class **CodeIgniterDatabaseForge**

addColumn(\$table[, \$field = []])

Parameters:

- **\$table** (*string*) – Table name to add the column to
- **\$field** (*array*) – Column definition(s)

Returns: TRUE on success, FALSE on failure

Return type: bool

Adds a column to a table. Usage: See [Adding a Column to a Table](#).

addField(\$field)

Parameters:

- **\$field** (*array*) – Field definition to add

Returns: CodeIgniterDatabaseForge instance (method chaining)

Return type: CodeIgniterDatabaseForge

Adds a field to the set that will be used to create a table. Usage: See [Adding fields](#).

addKey(\$key[, \$primary = FALSE[, \$unique = FALSE]])

Parameters:

- **\$key** (*mixed*) – Name of a key field or an array of fields
- **\$primary** (*bool*) – Set to TRUE if it should be a primary key or a regular one
- **\$unique** (*bool*) – Set to TRUE if it should be a unique key or a regular one

Returns: CodeIgniterDatabaseForge instance (method chaining)

Return type: CodeIgniterDatabaseForge

Adds a key to the set that will be used to create a table. Usage: See [Adding Keys](#).

addPrimaryKey(*\$key*)

Parameters: • ***\$key*** (*mixed*) – Name of a key field or an array of fields

Returns: CodeIgniterDatabaseForge instance (method chaining)

Return type: CodeIgniterDatabaseForge

Adds a primary key to the set that will be used to create a table.

Usage: See [Adding Keys](#).

addUniqueKey(*\$key*)

Parameters: • ***\$key*** (*mixed*) – Name of a key field or an array of fields

Returns: CodeIgniterDatabaseForge instance (method chaining)

Return type: CodeIgniterDatabaseForge

Adds an unique key to the set that will be used to create a table.

Usage: See [Adding Keys](#).

createDatabase(*\$db_name*)

Parameters: • ***\$db_name*** (*string*) – Name of the database to create

Returns: TRUE on success, FALSE on failure

Return type: bool

Creates a new database. Usage: See [Creating and Dropping Databases](#).

createTable(*\$table*[, *\$if_not_exists* = FALSE[, array *\$attributes* = []]])

- ***\$table*** (*string*) – Name of the table to create
- ***\$if_not_exists*** (*string*) – Set to TRUE to add an

Parameters: • 'IF NOT EXISTS' clause
• **\$attributes** (*string*) – An associative array of table attributes

Returns: TRUE on success, FALSE on failure

Return type: bool

Creates a new table. Usage: See [Creating a table](#).

dropColumn(\$table, \$column_name)

Parameters: • **\$table** (*string*) – Table name
• **\$column_name** (*array*) – The column name to drop

Returns: TRUE on success, FALSE on failure

Return type: bool

Drops a column from a table. Usage: See [Dropping a Column From a Table](#).

dropDatabase(\$db_name)

Parameters: • **\$db_name** (*string*) – Name of the database to drop

Returns: TRUE on success, FALSE on failure

Return type: bool

Drops a database. Usage: See [Creating and Dropping Databases](#).

dropTable(\$table_name[, \$if_exists = FALSE])

Parameters: • **\$table** (*string*) – Name of the table to drop
• **\$if_exists** (*string*) – Set to TRUE to add an 'IF EXISTS' clause

Returns: TRUE on success, FALSE on failure

Return type: bool

Drops a table. Usage: See [Dropping a table](#).

modifyColumn(\$table, \$field)

Parameters:

- **\$table** (*string*) – Table name
- **\$field** (*array*) – Column definition(s)

Returns: TRUE on success, FALSE on failure

Return type: bool

Modifies a table column. Usage: See [Modifying a Column in a Table](#).

renameTable(\$table_name, \$new_table_name)

Parameters:

- **\$table** (*string*) – Current of the table
- **\$new_table_name** (*string*) – New name of the table

Returns: TRUE on success, FALSE on failure

Return type: bool

Renames a table. Usage: See [Renaming a table](#).

Database Migrations

Migrations are a convenient way for you to alter your database in a structured and organized manner. You could edit fragments of SQL by hand but you would then be responsible for telling other developers that they need to go and run them. You would also have to keep track of which changes need to be run against the production machines next time you deploy.

The database table **migration** tracks which migrations have already been run so all you have to do is update your application files and call `$migration->current()` to work out which migrations should be run. The current version is found in **app/Config/Migrations.php**.

- [Migration file names](#)
- [Create a Migration](#)
 - [Using \\$currentVersion](#)
 - [Database Groups](#)
 - [Namespaces](#)
- [Usage Example](#)
- [Command-Line Tools](#)
- [Migration Preferences](#)
- [Class Reference](#)

[Migration file names](#)

Each Migration is run in numeric order forward or backwards depending on the method taken. Two numbering styles are available:

- **Sequential:** each migration is numbered in sequence, starting with **001**. Each number must be three digits, and there must not be any gaps in the sequence. (This was the numbering scheme prior to CodeIgniter 3.0.)
- **Timestamp:** each migration is numbered using the timestamp when the

migration was created, in **YYYYMMDDHHIISS** format (e.g. **20121031100537**). This helps prevent numbering conflicts when working in a team environment, and is the preferred scheme in CodeIgniter 3.0 and later.

The desired style may be selected using the `$type` setting in your `app/Config/Migrations.php` file. The default setting is `timestamp`.

Regardless of which numbering style you choose to use, prefix your migration files with the migration number followed by an underscore and a descriptive name for the migration. For example:

- `001_add_blog.php` (sequential numbering)
- `20121031100537_add_blog.php` (timestamp numbering)

Create a Migration

This will be the first migration for a new site which has a blog. All migrations go in the **app/Database/Migrations/** directory and have names such as `20121031100537_Add_blog.php`.

```
<?php namespace App\Database\Migrations;
```

```
class Migration_Add_blog extends \CodeIgniter\Database\Migration
```

```
    public function up()
    {
        $this->forge->addField([
            'blog_id' => [
                'type' => 'INT',
                'constraint' => 5,
                'unsigned' => TRUE,
                'auto_increment' => TRUE
            ],
            'blog_title' => [
                'type' => 'VARCHAR',
                'constraint' => '100',
            ],
            'blog_description' => [
                'type' => 'TEXT',
                'null' => TRUE,
            ],
        ],
```

```
    });  
    $this->forge->addKey('blog_id', TRUE);  
    $this->forge->createTable('blog');  
}  
  
public function down()  
{  
    $this->forge->dropTable('blog');  
}  
}
```

Then in **app/Config/Migrations.php** set `$currentVersion = 20121031100537;`.

The database connection and the database Forge class are both available to you through `$this->db` and `$this->forge`, respectively.

Alternatively, you can use a command-line call to generate a skeleton migration file. See below for more details.

Using \$currentVersion

The `$currentVersion` setting allows you to mark a location that your main application namespace should be set at. This is especially helpful for use in a production setting. In your application, you can always update the migration to the current version, and not latest to ensure your production and staging servers are running the correct schema. On your development servers, you can add additional migrations for code that is not ready for production, yet. By using the `latest()` method, you can be assured that your development machines are always running the bleeding edge schema.

Database Groups

A migration will only be run against a single database group. If you have multiple groups defined in **app/Config/Database.php**, then it will run against the `$defaultGroup` as specified in that same configuration file. There may be times when you need different schemas for different database groups. Perhaps you have one database that is used for all general site information, while another database is used for mission critical data. You can ensure that

migrations are run only against the proper group by setting the \$DBGroup property on your migration. This name must match the name of the database group exactly:

```
<?php namespace App\Database\Migrations;

class Migration_Add_blog extends \CodeIgniter\Database\Migration
{
    protected $DBGroup = 'alternate_db_group';

    public function up() { . . . }

    public function down() { . . . }
}
```

Namespaces

The migration library can automatically scan all namespaces you have defined within **app/Config/Autoload.php** and its \$psr4 property for matching directory names. It will include all migrations it finds in Database/Migrations.

Each namespace has it's own version sequence, this will help you upgrade and downgrade each module (namespace) without affecting other namespaces.

For example, assume that we have the the following namespaces defined in our Autoload configuration file:

```
$psr4 = [
    'App'           => APPPATH,
    'MyCompany'     => ROOTPATH.'MyCompany'
];
```

This will look for any migrations located at both **APPPATH/Database/Migrations** and **ROOTPATH/Database/Migrations**. This makes it simple to include migrations in your re-usable, modular code suites.

Usage Example

In this example some simple code is placed in **app/Controllers/Migrate.php** to update the schema:

```
<?php namespace App\Controllers;

class Migrate extends \CodeIgniter\Controller
{
    public function index()
    {
        $migrate = \Config\Services::migrations();

        try
        {
            $migrate->current();
        }
        catch (\Exception $e)
        {
            // Do something with the error here...
        }
    }
}
```

Command-Line Tools

CodeIgniter ships with several [commands](#) that are available from the command line to help you work with migrations. These tools are not required to use migrations but might make things easier for those of you that wish to use them. The tools primarily provide access to the same methods that are available within the MigrationRunner class.

latest

Migrates all database groups to the latest available migrations:

```
> php spark migrate:latest
```

You can use (latest) with the following options:

- (-g) to chose database group, otherwise default database group will be used.

- (-n) to choose namespace, otherwise (App) namespace will be used.
- (-all) to migrate all namespaces to the latest migration

This example will migrate Blog namespace to latest:

```
> php spark migrate:latest -g test -n Blog
```

current

Migrates the (App) namespace to match the version set in \$currentVersion. This will migrate both up and down as needed to match the specified version:

```
> php spark migrate:current
```

You can use (current) with the following options:

- (-g) to chose database group, otherwise default database group will be used.

version

Migrates to the specified version. If no version is provided, you will be prompted for the version.

```
// Asks you for the version...
```

```
> php spark migrate:version  
Version:
```

```
// Sequential
```

```
> php spark migrate:version 007
```

```
// Timestamp
```

```
> php spark migrate:version 20161426211300
```

You can use (version) with the following options:

- (-g) to chose database group, otherwise default database group will be used.
- (-n) to choose namespace, , otherwise (App) namespace will be used.

rollback

Rolls back all migrations, taking all database groups to a blank slate, effectively migration 0:

```
> php spark migrate:rollback
```

You can use (rollback) with the following options:

- (-g) to chose database group, otherwise default database group will be used.
- (-n) to choose namespace, otherwise (App) namespace will be used.
- (-all) to migrate all namespaces to the latest migration

refresh

Refreshes the database state by first rolling back all migrations, and then migrating to the latest version:

```
> php spark migrate:refresh
```

You can use (refresh) with the following options:

- (-g) to chose database group, otherwise default database group will be used.
- (-n) to choose namespace, otherwise (App) namespace will be used.
- (-all) to migrate all namespaces to the latest migration

status

Displays a list of all migrations and the date and time they ran, or ‘–’ if they have not been run:

```
> php spark migrate:status
Filename           Migrated On
First_migration.php 2016-04-25 04:44:22
```

You can use (refresh) with the following options:

- (-g) to chose database group, otherwise default database group will be used.

create

Creates a skeleton migration file in **app/Database/Migrations**.

- When migration type is timestamp, using the YYYYMMDDHHIISS format:

```
> php spark migrate:create [filename]
```

- When migration type is sequential, using the numbered in sequence, default with 001:

```
> php spark migrate:create [filename] 001
```

You can use (create) with the following options:

- (-n) to choose namespace, otherwise (App) namespace will be used.

Migration Preferences

The following is a table of all the config options for migrations, available in **app/Config/Migrations.php**.

Preference	Default	Options	Description
enabled	FALSE	TRUE / FALSE	Enable or disable migrations.
path	'Database/Migrations/'	None	The path to your migrations folder.
currentVersion	0	None	The current version your database should use.
table	migrations	None	The table name for storing the schema version number.
type	'timestamp'	'timestamp' /	The type of numeric identifier used to name

‘sequential’ migration files.

Class Reference

class **CodeIgniterDatabaseMigrationRunner**

current(\$group)

Parameters: • **\$group** (*mixed*) – database group name, if null (App) namespace will be used.

Returns: TRUE if no migrations are found, current version string on success, FALSE on failure

Return type: mixed

Migrates up to the current version (whatever is set for \$currentVersion in *app/Config/Migrations.php*).

findMigrations()

Returns: An array of migration files

Return type: array

An array of migration filenames are returned that are found in the **path** property.

latest(\$namespace, \$group)

Parameters: • **\$namespace** (*mixed*) – application namespace, if null (App) namespace will be used.

• **\$group** (*mixed*) – database group name, if null default database group will be used.

Returns: Current version string on success, FALSE on failure

Return type: mixed

This works much the same way as `current()` but instead of looking for the \$currentVersion the Migration class will use the very newest migration found in the filesystem.

latestAll(\$group)

Parameters: • **\$group** (*mixed*) – database group name, if null default database group will be used.

Returns: TRUE on success, FALSE on failure

Return type: mixed

This works much the same way as `latest()` but instead of looking for one namespace, the Migration class will use the very newest migration found for all namespaces.

version(\$target_version, \$namespace, \$group)

Parameters:

- **\$namespace** (*mixed*) – application namespace, if null (App) namespace will be used.
- **\$group** (*mixed*) – database group name, if null default database group will be used.
- **\$target_version** (*mixed*) – Migration version to process

Returns: TRUE if no migrations are found, current version string on success, FALSE on failure

Return type: mixed

Version can be used to roll back changes or step forwards programmatically to specific versions. It works just like `current()` but ignores `$currentVersion`.

```
$migration->version(5);
```

setNamespace(\$namespace)

Parameters: • **\$namespace** (*string*) – application namespace.

Returns: The current MigrationRunner instance

Return type: CodeIgniterDatabaseMigrationRunner

Sets the path the library should look for migration files:

```
$migration->setNamespace($path)  
->latest();
```

setGroup(\$group)

Parameters: • **\$group** (*string*) – database group name.

Returns: The current MigrationRunner instance

Return type: CodeIgniterDatabaseMigrationRunner

Sets the path the library should look for migration files:

```
$migration->setNamespace($path)
->latest();
```

Database Seeding

Database seeding is a simple way to add data into your database. It is especially useful during development where you need to populate the database with sample data that you can develop against, but it is not limited to that. Seeds can contain static data that you don't want to include in a migration, like countries, or geo-coding tables, event or setting information, and more.

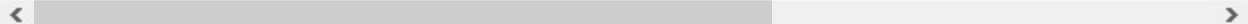
Database seeds are simple classes that must have a **run()** method, and extend **CodeIgniterDatabaseSeeder**. Within the **run()** the class can create any form of data that it needs to. It has access to the database connection and the forge through `$this->db` and `$this->forge`, respectively. Seed files must be stored within the **app/Database/Seeds** directory. The name of the file must match the name of the class.

```
<?php namespace App\Database\Seeds;

class SimpleSeeder extends \CodeIgniter\Database\Seeder
{
    public function run()
    {
        $data = [
            'username' => 'darth',
            'email'     => 'darth@theempire.com'
        ];

        // Simple Queries
        $this->db->query("INSERT INTO users (username, em
            $data
        );

        // Using Query Builder
        $this->db->table('users')->insert($data);
    }
}
```



Nesting Seeders

Seeders can call other seeders, with the **call()** method. This allows you to easily organize a central seeder, but organize the tasks into separate seeder files:

```
<?php namespace App\Database\Seeds;

class TestSeeder extends \CodeIgniter\Database\Seeder
{
    public function run()
    {
        $this->call('UserSeeder');
        $this->call('CountrySeeder');
        $this->call('JobSeeder');
    }
}
```

You can also use a fully-qualified class name in the **call()** method, allowing you to keep your seeders anywhere the autoloader can find them. This is great for more modular code bases:

```
public function run()
{
    $this->call('UserSeeder');
    $this->call('My\Database\Seeds\CountrySeeder');
}
```

Using Seeders

You can grab a copy of the main seeder through the database config class:

```
$seeder = \Config\Database::seeder();
$seeder->call('TestSeeder');
```

Command Line Seeding

You can also seed data from the command line, as part of the Migrations CLI tools, if you don't want to create a dedicated controller:

```
> php spark db:seed TestSeeder
```


Library Reference

- [Caching Driver](#)
- [CURLRequest Class](#)
- [Working with Files](#)
- [Honeypot Class](#)
- [Image Manipulation Class](#)
- [Pagination](#)
- [Security Class](#)
- [Session Library](#)
- [Throttler](#)
- [Dates and Times](#)
- [Typography](#)
- [Working with Uploaded Files](#)
- [Working with URIs](#)
- [User Agent Class](#)
- [Validation](#)

Caching Driver

CodeIgniter features wrappers around some of the most popular forms of fast and dynamic caching. All but file-based caching require specific server requirements, and a Fatal Exception will be thrown if server requirements are not met.

- [Example Usage](#)
 - [Configuring the Cache](#)
- [Class Reference](#)
- [Drivers](#)
 - [File-based Caching](#)
 - [Memcached Caching](#)
 - [WinCache Caching](#)
 - [Redis Caching](#)
 - [Dummy Cache](#)

[Example Usage](#)

The following example shows a common usage pattern within your controllers.

```
if ( ! $foo = cache('foo'))
{
    echo 'Saving to the cache!<br />';
    $foo = 'foobarbaz!';

    // Save into the cache for 5 minutes
    cache()->save('foo', $foo, 300);
}

echo $foo;
```

You can grab an instance of the cache engine directly through the Services

class:

```
$cache = \Config\Services::cache();  
$foo = $cache->get('foo');
```

Configuring the Cache

All configuration for the cache engine is done in **app/Config/Cache.php**. In that file, the following items are available.

\$handler

This is the name of the handler that should be used as the primary handler when starting up the engine. Available names are: dummy, file, memcached, redis, wincache.

\$backupHandler

In the case that the first choice \$handler is not available, this is the next cache handler to load. This is commonly the **file** handler since the file system is always available, but may not fit more complex, multi-server setups.

\$prefix

If you have more than one application using the same cache storage, you can add a custom prefix here that is prepended to all key names.

\$path

This is used by the file handler to show where it should save the cache files to.

\$memcached

This is an array of servers that will be used when using the Memcache(d) handler.

\$redis

The settings for the Redis server that you wish to use when using the Redis handler.

Class Reference

isSupported()

Returns: TRUE if supported, FALSE if not

Return type: bool

get(\$key)

Parameters: • **\$key** (*string*) – Cache item name

Returns: Item value or FALSE if not found

Return type: mixed

This method will attempt to fetch an item from the cache store. If the item does not exist, the method will return FALSE.

Example:

```
$foo = $cache->get('my_cached_item');
```

save(\$key, \$data[, \$ttl = 60[, \$raw = FALSE]])

Parameters:

- **\$key** (*string*) – Cache item name
- **\$data** (*mixed*) – the data to save
- **\$ttl** (*int*) – Time To Live, in seconds (default 60)
- **\$raw** (*bool*) – Whether to store the raw value

Returns: TRUE on success, FALSE on failure

Return type: string

This method will save an item to the cache store. If saving fails, the method will return FALSE.

Example:

```
$cache->save('cache_item_id', 'data_to_cache');
```

Note

The `$raw` parameter is only utilized by Memcache, in order to allow usage of `increment()` and `decrement()`.

delete(*\$key*)

Parameters: • **\$key** (*string*) – name of cached item

Returns: TRUE on success, FALSE on failure

Return type: bool

This method will delete a specific item from the cache store. If item deletion fails, the method will return FALSE.

Example:

```
$cache->delete('cache_item_id');
```

increment(*\$key* [, *\$offset* = 1])

Parameters: • **\$key** (*string*) – Cache ID

• **\$offset** (*int*) – Step/value to add

Returns: New value on success, FALSE on failure

Return type: mixed

Performs atomic incrementation of a raw stored value.

Example:

```
// 'iterator' has a value of 2
```

```
$cache->increment('iterator'); // 'iterator' is now 3
```

```
$cache->increment('iterator', 3); // 'iterator' is now 6
```

decrement(*\$key* [, *\$offset* = 1])

Parameters: • **\$key** (*string*) – Cache ID

• **\$offset** (*int*) – Step/value to reduce by

Returns: New value on success, FALSE on failure

Return type: mixed

Performs atomic decrementation of a raw stored value.

Example:

```
// 'iterator' has a value of 6  
$cache->decrement('iterator'); // 'iterator' is now 5  
$cache->decrement('iterator', 2); // 'iterator' is now 3
```

clean()

Returns: TRUE on success, FALSE on failure

Return type: bool

This method will ‘clean’ the entire cache. If the deletion of the cache files fails, the method will return FALSE.

Example:

```
$cache->clean();
```

cache_info()

Returns: Information on the entire cache database

Return type: mixed

This method will return information on the entire cache.

Example:

```
var_dump($cache->cache_info());
```

Note

The information returned and the structure of the data is dependent on which adapter is being used.

getMetadata(\$key)

Parameters: • **\$key** (*string*) – Cache item name

Returns: Metadata for the cached item

Return type: mixed

This method will return detailed information on a specific item in the cache.

Example:

```
var_dump($cache->getMetadata('my_cached_item'));
```

Note

The information returned and the structure of the data is dependent on which adapter is being used.

Drivers

File-based Caching

Unlike caching from the Output Class, the driver file-based caching allows for pieces of view files to be cached. Use this with care, and make sure to benchmark your application, as a point can come where disk I/O will negate positive gains by caching. This requires a writable cache directory to be really writable (0777).

Memcached Caching

Multiple Memcached servers can be specified in the cache configuration file.

For more information on Memcached, please see <http://php.net/memcached>.

WinCache Caching

Under Windows, you can also utilize the WinCache driver.

For more information on WinCache, please see <http://php.net/wincache>.

Redis Caching

Redis is an in-memory key-value store which can operate in LRU cache mode. To use it, you need [Redis server and phpredis PHP extension](https://github.com/phpredis/phpredis) [https://github.com/phpredis/phpredis].

Config options to connect to redis server must be stored in the app/Config/redis.php file. Available options are:

```
$config['host'] = '127.0.0.1';  
$config['password'] = NULL;  
$config['port'] = 6379;  
$config['timeout'] = 0;  
$config['database'] = 0;
```

For more information on Redis, please see <http://redis.io>.

Dummy Cache

This is a caching backend that will always ‘miss.’ It stores no data, but lets you keep your caching code in place in environments that don’t support your chosen cache.

CURLRequest Class

The CURLRequest class is a lightweight HTTP client based on CURL that allows you to talk to other web sites and servers. It can be used to get the contents of a Google search, retrieve a web page or image, or communicate with an API, among many other things.

- [Loading the Library](#)
- [Working with the Library](#)
 - [Making Requests](#)
 - [Using Responses](#)
- [Request Options](#)
 - [allow_redirects](#)
 - [auth](#)
 - [body](#)
 - [cert](#)
 - [connect_timeout](#)
 - [cookie](#)
 - [debug](#)
 - [delay](#)
 - [form_params](#)
 - [headers](#)
 - [http_errors](#)
 - [json](#)
 - [multipart](#)
 - [query](#)
 - [timeout](#)
 - [verify](#)
 - [version](#)

This class is modelled after the [Guzzle HTTP Client](#)

[<http://docs.guzzlephp.org/en/latest/>] library since it is one of the more widely used

libraries. Where possible, the syntax has been kept the same so that if your application needs something a little more powerful than what this library provides, you will have to change very little to move over to use Guzzle.

Note

This class requires the [cURL Library](http://php.net/manual/en/book.curl.php) [http://php.net/manual/en/book.curl.php] to be installed in your version of PHP. This is a very common library that is typically available but not all hosts will provide it, so please check with your host to verify if you run into problems.

Loading the Library

The library can be loaded either manually or through the [Services class](#).

To load with the Services class call the `curlrequest()` method:

```
$client = \Config\Services::curlrequest();
```

You can pass in an array of default options as the first parameter to modify how cURL will handle the request. The options are described later in this document:

```
$options = [  
    'base_uri' => 'http://example.com/api/v1/',  
    'timeout'  => 3  
];  
$client = \Config\Services::curlrequest($options);
```

When creating the class manually, you need to pass a few dependencies in. The first parameter is an instance of the `\Config\App` class. The second parameter is a `\CodeIgniter\HTTP\URI` instance. The third parameter is a `\CodeIgniter\HTTP\Response` object. The fourth parameter is the optional `$options` array:

```
$client = new \CodeIgniter\HTTP\CURLRequest(  
    new \Config\App(),  
    new \CodeIgniter\HTTP\URI(),  
    new \CodeIgniter\HTTP\Response(new \Config\App()),  
    $options
```



```
);
```


Working with the Library

Working with CURL requests is simply a matter of creating the Request and getting a [Response object](#) back. It is meant to handle the communications. After that you have complete control over how the information is handled.

Making Requests

Most communication is done through the `request()` method, which fires off the request, and then returns a Response instance to you. This takes the HTTP method, the url and an array of options as the parameters.

```
$client = \Config\Services::curlrequest();  
  
$response = $client->request('GET', 'https://api.github.com/user'  
    'auth' => ['user', 'pass']  
]);
```



Since the response is an instance of `CodeIgniter\HTTP\Response` you have all of the normal information available to you:

```
echo $response->getStatusCode();  
echo $response->getBody();  
echo $response->getHeader('Content-Type');  
$language = $response->negotiateLanguage(['en', 'fr']);
```

While the `request()` method is the most flexible, you can also use the following shortcut methods. They each take the URL as the first parameter and an array of options as the second:

```
* $client->get('http://example.com');  
* $client->delete('http://example.com');  
* $client->head('http://example.com');  
* $client->options('http://example.com');  
* $client->patch('http://example.com');  
* $client->put('http://example.com');  
* $client->post('http://example.com');
```

Base URI

A `base_uri` can be set as one of the options during the instantiation of the class. This allows you to set a base URI, and then make all requests with that client using relative URLs. This is especially handy when working with APIs:

```
$client = \Config\Services::curlrequest([
    'base_uri' => 'https://example.com/api/v1/'
]);

// GET http:example.com/api/v1/photos
$client->get('photos');

// GET http:example.com/api/v1/photos/13
$client->delete('photos/13');
```

When a relative URI is provided to the `request()` method or any of the shortcut methods, it will be combined with the `base_uri` according to the rules described by [RFC 2986, section 2](http://tools.ietf.org/html/rfc3986#section-5.2) [http://tools.ietf.org/html/rfc3986#section-5.2]. To save you some time, here are some examples of how the combinations are resolved.

base_uri	URI	Result
http://foo.com	/bar	http://foo.com/bar
http://foo.com/foo	/bar	http://foo.com/bar
http://foo.com/foo	bar	http://foo.com/bar
http://foo.com/foo/	bar	http://foo.com/foo/bar
http://foo.com	http://baz.com	http://baz.com
http://foo.com/?bar	bar	http://foo.com/bar

Using Responses

Each `request()` call returns a `Response` object that contains a lot of useful information and some helpful methods. The most commonly used methods let you determine the response itself.

You can get the status code and reason phrase of the response:

```
$code    = $response->getStatusCode();    // 200
$reason  = $response->getReason();        // OK
```

You can retrieve headers from the response:

```
// Get a header line
echo $response->getHeaderLine('Content-Type');

// Get all headers
foreach ($response->getHeaders() as $name => $value)
{
    echo $name . ': ' . $response->getHeaderLine($name) . "\n";
}
```

The body can be retrieved using the `getBody()` method:

```
$body = $response->getBody();
```

The body is the raw body provided by the remote `getServer`. If the content type requires formatting, you will need to ensure that your script handles that:

```
if (strpos($response->getHeader('content-type'), 'application/jso
{
    $body = json_decode($body);
}
< >
```

[Request Options](#)

This section describes all of the available options you may pass into the constructor, the `request()` method, or any of the shortcut methods.

[allow_redirects](#)

By default, cURL will follow all “Location:” headers the remote servers send back. The `allow_redirects` option allows you to modify how that works.

If you set the value to `false`, then it will not follow any redirects at all:

```
$client->request('GET', 'http://example.com', ['allow_redirects'
< >
```

Setting it to true will apply the default settings to the request:

```
$client->request('GET', 'http://example.com', ['allow_redirects'
// Sets the following defaults:
'max'          => 5, // Maximum number of redirects to follow before
'strict'       => true, // Ensure POST requests stay POST requests t
'protocols'    => ['http', 'https'] // Restrict redirects to one or
```

You can pass in array as the value of the `allow_redirects` option to specify new settings in place of the defaults:

```
$client->request('GET', 'http://example.com', ['allow_redirects'
    'max'          => 10,
    'protocols'    => ['https'] // Force HTTPS domains only.
]);
```

Note

Following redirects does not work when PHP is in `safe_mode` or `open_basedir` is enabled.

[auth](#)

Allows you to provide Authentication details for [HTTP Basic](http://www.ietf.org/rfc/rfc2069.txt) [http://www.ietf.org/rfc/rfc2069.txt] and [Digest](http://www.ietf.org/rfc/rfc2069.txt) [http://www.ietf.org/rfc/rfc2069.txt] and authentication. Your script may have to do extra to support Digest authentication - this simply passes the username and password along for you. The value must be an array where the first element is the username, and the second is the password. The third parameter should be the type of authentication to use, either basic or digest:

```
$client->request('GET', 'http://example.com', ['auth' => ['userna
```

[body](#)

There are two ways to set the body of the request for request types that support them, like PUT, OR POST. The first way is to use the `setBody()` method:

```
$client->setBody($body)
    ->request('put', 'http://example.com');
```

The second method is by passing a body option in. This is provided to maintain Guzzle API compatibility, and functions the exact same way as the previous example. The value must be a string:

```
$client->request('put', 'http://example.com', ['body' => $body]);
```

cert

To specify the location of a PEM formatted client-side certificate, pass a string with the full path to the file as the `cert` option. If a password is required, set the value to an array with the first element as the path to the certificate, and the second as the password:

```
$client->request('get', '/', ['cert' => ['/path/getServer.pem', '
< >
```

connect_timeout

By default, CodeIgniter does not impose a limit for cURL to attempt to connect to a website. If you need to modify this value, you can do so by passing the amount of time in seconds with the `connect_timeout` option. You can pass 0 to wait indefinitely:

```
$response->request('GET', 'http://example.com', ['connect_timeout
< >
```

cookie

This specifies the filename that CURL should use to read cookie values from, and to save cookie values to. This is done using the `CURL_COOKIEJAR` and `CURL_COOKIEFILE` options. An example:

```
$response->request('GET', 'http://example.com', ['cookie' => WRIT  
< >
```

debug

When debug is passed and set to true, this will enable additional debugging to echo to STDOUT during the script execution. This is done by passing CURLOPT_VERBOSE and echoing the output:

```
$response->request('GET', 'http://example.com', ['debug' => true]  
< >
```

You can pass a filename as the value for debug to have the output written to a file:

```
$response->request('GET', 'http://example.com', ['debug' => '/usr  
< >
```

delay

Allows you to pause a number of milliseconds before sending the request:

```
// Delay for 2 seconds  
$response->request('GET', 'http://example.com', ['delay' => 2000]  
< >
```

form_params

You can send form data in an application/x-www-form-urlencoded POST request by passing an associative array in the form_params option. This will set the Content-Type header to application/x-www-form-urlencoded if it's not already set:

```
$client->request('POST', '/post', [  
    'form_params' => [  
        'foo' => 'bar',  
        'baz' => ['hi', 'there']  
    ],  
]);
```

Note

form_params cannot be used with the multipart option. You will need to use one or the other. Use form_params for application/x-www-form-urlencoded request, and multipart for multipart/form-data requests.

[headers](#)

While you can set any headers this request needs by using the `setHeader()` method, you can also pass an associative array of headers in as an option. Each key is the name of a header, and each value is a string or array of strings representing the header field values:

```
$client->request('get', '/', [
    'headers' => [
        'User-Agent' => 'testing/1.0',
        'Accept'      => 'application/json',
        'X-Foo'       => ['Bar', 'Baz']
    ]
]);
```

If headers are passed into the constructor they are treated as default values that will be overridden later by any further headers arrays or calls to `setHeader()`.

[http_errors](#)

By default, `CURLRequest` will fail if the HTTP code returned is greater than or equal to 400. You can set `http_errors` to `false` to return the content instead:

```
$client->request('GET', '/status/500');
// Will fail verbosely

$res = $client->request('GET', '/status/500', ['http_errors' => false]);
echo $res->getStatusCode();
// 500
```



[json](#)

The `json` option is used to easily upload JSON encoded data as the body of a request. A Content-Type header of `application/json` is added, overwriting any Content-Type that might be already set. The data provided to this option can be any value that `json_encode()` accepts:

```
$response = $client->request('PUT', '/put', ['json' => ['foo' =>
```

Note

This option does not allow for any customization of the `json_encode()` function, or the Content-Type header. If you need that ability, you will need to encode the data manually, passing it through the `setBody()` method of `CURLRequest`, and set the Content-Type header with the `setHeader()` method.

multipart

When you need to send files and other data via a POST request, you can use the `multipart` option, along with the [CURLFile Class](http://php.net/manual/en/class.curlfile.php) [http://php.net/manual/en/class.curlfile.php]. The values should be an associative array of POST data to send. For safer usage, the legacy method of uploading files by prefixing their name with an `@` has been disabled. Any files that you want to send must be passed as instances of `CURLFile`:

```
$post_data = [  
    'foo'      => 'bar',  
    'userfile' => new \CURLFile('/path/to/file.txt')  
];
```

Note

`multipart` cannot be used with the `form_params` option. You can only use one or the other. Use `form_params` for `application/x-www-form-urlencoded` requests, and `multipart` for `multipart/form-data` requests.

[query](#)

You can pass along data to send as query string variables by passing an associative array as the query option:

```
// Send a GET request to /get?foo=bar  
$client->request('GET', '/get', ['query' => ['foo' => 'bar']]);
```

[timeout](#)

By default, cURL functions are allowed to run as long as they take, with no time limit. You can modify this with the `timeout` option. The value should be the number of seconds you want the functions to execute for. Use 0 to wait indefinitely:

```
$response->request('GET', 'http://example.com', ['timeout' => 5])  
<  >
```

[verify](#)

This option describes the SSL certificate verification behavior. If the `verify` option is `true`, it enables the SSL certificate verification and uses the default CA bundle provided by the operating system. If set to `false` it will disable the certificate verification (this is insecure, and allows man-in-the-middle attacks!). You can set it to a string that contains the path to a CA bundle to enable verification with a custom certificate. The default value is `true`:

```
// Use the system's CA bundle (this is the default setting)  
$client->request('GET', '/', ['verify' => true]);  
  
// Use a custom SSL certificate on disk.  
$client->request('GET', '/', ['verify' => '/path/to/cert.pem']);  
  
// Disable validation entirely. (Insecure!)  
$client->request('GET', '/', ['verify' => false]);
```

[version](#)

To set the HTTP protocol to use, you can pass a string or float with the version number (typically either 1.0 or 1.1, 2.0 is currently unsupported.):

```
// Force HTTP/1.0  
$client->request('GET', '/', ['version' => 1.0]);
```

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Working with Files

CodeIgniter provides a File class that wraps the [SplFileInfo](#) [http://php.net/manual/en/class.splfileinfo.php] class and provides some additional convenience methods. This class is the base class for [uploaded files](#) and [images](#).

- [Getting a File instance](#)
- [Taking Advantage of Spl](#)
- [New Features](#)
 - [Moving Files](#)

[Getting a File instance](#)

You create a new File instance by passing in the path to the file in the constructor. By default the file does not need to exist. However, you can pass an additional argument of “true” to check that the file exist and throw `FileNotFoundException()` if it does not.

```
$file = new \CodeIgniter\Files\File($path);
```

[Taking Advantage of Spl](#)

Once you have an instance, you have the full power of the `SplFileInfo` class at the ready, including:

```
// Get the file's basename
echo $file->getBasename();
// Get last modified time
echo $file->getMTime();
// Get the true realpath
echo $file->getRealpath();
// Get the file permissions
```

```

echo $file->getPerms();

// Write CSV rows to it.
if ($file->isWritable())
{
    $csv = $file->openFile('w');

    foreach ($rows as $row)
    {
        $csv->fputcsv($row);
    }
}

```

New Features

In addition to all of the methods in the SplFileInfo class, you get some new tools.

getRandomName()

You can generate a cryptographically secure random filename, with the current timestamp prepended, with the getRandomName() method. This is especially useful to rename files when moving it so that the filename is unguessable:

```

// Generates something like: 1465965676_385e33f741.jpg
$newName = $file->getRandomName();

```

getSize()

Returns the size of the uploaded file in bytes. You can pass in either 'kb' or 'mb' as the first parameter to get the results in kilobytes or megabytes, respectively:

```

$bytes      = $file->getSize();           // 256901
$kilobytes  = $file->getSize('kb');       // 250.880
$megabytes  = $file->getSize('mb');       // 0.245

```

getMimeType()

Retrieve the media type (mime type) of the file. Uses methods that are

considered as secure as possible when determining the type of file:

```
$type = $file->getMimeType();
```

```
echo $type; // image/png
```

guessExtension()

Attempts to determine the file extension based on the trusted `getMimeType()` method. If the mime type is unknown, will return null. This is often a more trusted source than simply using the extension provided by the filename.

Uses the values in **app/Config/Mimes.php** to determine extension:

```
// Returns 'jpg' (WITHOUT the period)
$ext = $file->guessExtension();
```

Moving Files

Each file can be moved to its new location with the aptly named `move()` method. This takes the directory to move the file to as the first parameter:

```
$file->move(WRITEPATH.'uploads');
```

By default, the original filename was used. You can specify a new filename by passing it as the second parameter:

```
$newName = $file->getRandomName();
$file->move(WRITEPATH.'uploads', $newName);
```

Honeypot Class

The Honeypot Class makes it possible to determine when a Bot makes a request to a CodeIgniter4 application, if it's enabled in `Application\Config\Filters.php` file. This is done by attaching form fields to any form, and this form field is hidden from a human but accessible to a Bot. When data is entered into the field, it's assumed the request is coming from a Bot, and you can throw a `HoneypotException`.

- [Enabling Honeypot](#)
- [Customizing Honeypot](#)

[Enabling Honeypot](#)

To enable a Honeypot, changes have to be made to the `app/Config/Filters.php`. Just uncomment `honeypot` from the `$globals` array, like...:

```
public $globals = [  
    'before' => [  
        'honeypot'  
        // 'csrf',  
    ],  
    'after' => [  
        'toolbar',  
        'honeypot'  
    ]  
];
```

A sample Honeypot filter is bundled, as `system/Filters/Honeypot.php`. If it is not suitable, make your own at `app/Filters/Honeypot.php`, and modify the `$aliases` in the configuration appropriately.

Customizing Honeypot

Honeypot can be customized. The fields below can be set either in `app/Config/Honeypot.php` or in `.env`.

- `hidden` - `true|false` to control visibility of the honeypot field; default is `true`
- `label` - HTML label for the honeypot field, default is 'Fill This Field'
- `name` - name of the HTML form field used for the template; default is 'honeypot'
- `template` - form field template used for the honeypot; default is '`<label>{label}</label><input type="text" name="{name}" value=""</>`'

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Image Manipulation Class

CodeIgniter's Image Manipulation class lets you perform the following actions:

- Image Resizing
- Thumbnail Creation
- Image Cropping
- Image Rotating
- Image Watermarking

The following image libraries are supported: GD/GD2, and ImageMagick.

- [Initializing the Class](#)
 - [Processing an Image](#)
 - [Processing Methods](#)

[Initializing the Class](#)

Like most other classes in CodeIgniter, the image class is initialized in your controller by calling the Services class:

```
$image = Config\Services::image();
```

You can pass the alias for the image library you wish to use into the Service function:

```
$image = Config\Services::image('imagick');
```

The available Handlers are as follows:

- gd The GD/GD2 image library
- imagick The ImageMagick library.

If using the ImageMagick library, you must set the path to the library on your server in **app/Config/Images.php**.

Note

The ImageMagick handler does NOT require the imagick extension to be loaded on the server. As long as your script can access the library and can run `exec()` on the server, it should work.

Processing an Image

Regardless of the type of processing you would like to perform (resizing, cropping, rotation, or watermarking), the general process is identical. You will set some preferences corresponding to the action you intend to perform, then call one of the available processing functions. For example, to create an image thumbnail you'll do this:

```
$image = Config\Services::image()  
    ->withFile('/path/to/image/mypic.jpg')  
    ->fit(100, 100, 'center')  
    ->save('/path/to/image/mypic_thumb.jpg');
```

The above code tells the library to look for an image called *mypic.jpg* located in the `source_image` folder, then create a new image from it that is 100 x 100 pixels using the GD2 image library, and save it to a new file (the thumb). Since it is using the `fit()` method, it will attempt to find the best portion of the image to crop based on the desired aspect ratio, and then crop and resize the result.

An image can be processed through as many of the available methods as needed before saving. The original image is left untouched, and a new image is used and passed through each method, applying the results on top of the previous results:

```
$image = Config\Services::image()  
    ->withFile('/path/to/image/mypic.jpg')  
    ->reorient()  
    ->rotate(90)
```

```
->crop(100, 100, 0, 0)  
->save('/path/to/image/mypic_thumb.jpg');
```

This example would take the same image and first fix any mobile phone orientation issues, rotate the image by 90 degrees, and then crop the result into a 100x100 pixel image, starting at the top left corner. The result would be saved as the thumbnail.

Note

In order for the image class to be allowed to do any processing, the folder containing the image files must have write permissions.

Note

Image processing can require a considerable amount of server memory for some operations. If you are experiencing out of memory errors while processing images you may need to limit their maximum size, and/or adjust PHP memory limits.

Processing Methods

There are six available processing methods:

- `$image->crop()`
- `$image->fit()`
- `$image->flatten()`
- `$image->flip()`
- `$image->resize()`
- `$image->rotate()`
- `$image->text()`

These methods return the class instance so they can be chained together, as shown above. If they fail they will throw a `CodeIgniter\Images\ImageException` that contains the error message. A good practice is to catch the exceptions, showing an error upon failure, like

this:

```
try {
$image = Config\Services::image()
    ->withFile('/path/to/image/mypic.jpg')
    ->fit(100, 100, 'center')
    ->save('/path/to/image/mypic_thumb.jpg');
}
catch (CodeIgniter\Images\ImageException $e)
{
    echo $e->getMessage();
}
```

Note

You can optionally specify the HTML formatting to be applied to the errors, by submitting the opening/closing tags in the function, like this:

```
$this->image_lib->display_errors('<p>', '</p>');
```

Cropping Images

Images can be cropped so that only a portion of the original image remains. This is often used when creating thumbnail images that should match a certain size/aspect ratio. This is handled with the `crop()` method:

```
crop(int $width = null, int $height = null, int $x = null, int $y
```

- **\$width** is the desired width of the resulting image, in pixels.
- **\$height** is the desired height of the resulting image, in pixels.
- **\$x** is the number of pixels from the left side of the image to start cropping.
- **\$y** is the number of pixels from the top of the image to start cropping.
- **\$maintainRatio** will, if true, adjust the final dimensions as needed to maintain the image's original aspect ratio.
- **\$masterDim** specifies which dimension should be left untouched when **\$maintainRatio** is true. Values can be: 'width', 'height', or 'auto'.

To take a 50x50 pixel square out of the center of an image, you would need to first calculate the appropriate x and y offset values:

```
$info = Services::image('imagick')
    ->withFile('/path/to/image/mypic.jpg')
    ->getFile()
    ->getProperties(true);

$xOffset = ($info['width'] / 2) - 25;
$yOffset = ($info['height'] / 2) - 25;

Services::image('imagick')
    ->withFile('/path/to/image/mypic.jpg')
    ->crop(50, 50, $xOffset, $yOffset)
    ->save('path/to/new/image.jpg');
```

Fitting Images

The `fit()` method aims to help simplify cropping a portion of an image in a “smart” way, by doing the following steps:

- Determine the correct portion of the original image to crop in order to maintain the desired aspect ratio.
- Crop the original image.
- Resize to the final dimensions.

```
fit(int $width, int $height = null, string $position = 'center')
```

- **\$width** is the desired final width of the image.
- **\$height** is the desired final height of the image.
- **\$position** determines the portion of the image to crop out. Allowed positions: ‘top-left’, ‘top’, ‘top-right’, ‘left’, ‘center’, ‘right’, ‘bottom-left’, ‘bottom’, ‘bottom-right’.

This provides a much simpler way to crop that will always maintain the aspect ratio:

```
Services::image('imagick')
    ->withFile('/path/to/image/mypic.jpg')
    ->fit(100, 150, 'left')
    ->save('path/to/new/image.jpg');
```

Flattening Images

The `flatten()` method aims to add a background color behind transparent images (PNG) and convert RGBA pixels to RGB pixels

- Specify a background color when converting from transparent images to jpgs.

```
flatten(int $red = 255, int $green = 255, int $blue = 255)
```

- **\$red** is the red value of the background.
- **\$green** is the green value of the background.
- **\$blue** is the blue value of the background.

```
Services::image('imagick')  
->withFile('/path/to/image/mypic.png')  
->flatten()  
->save('path/to/new/image.jpg');
```

```
Services::image('imagick')  
->withFile('/path/to/image/mypic.png')  
->flatten(25, 25, 112)  
->save('path/to/new/image.jpg');
```

Flipping Images

Images can be flipped along either their horizontal or vertical axis:

```
flip(string $dir)
```

- **\$dir** specifies the axis to flip along. Can be either 'vertical' or 'horizontal'.

```
Services::image('imagick')  
->withFile('/path/to/image/mypic.jpg')  
->flip('horizontal')  
->save('path/to/new/image.jpg');
```

Resizing Images

Images can be resized to fit any dimension you require with the `resize()` method:

```
resize(int $width, int $height, bool $maintainRatio = false, stri
```

- **\$width** is the desired width of the new image in pixels
- **\$height** is the desired height of the new image in pixels
- **\$maintainRatio** determines whether the image is stretched to fit the new dimensions, or the original aspect ratio is maintained.
- **\$masterDim** specifies which axis should have its dimension honored when maintaining ratio. Either 'width', 'height'.

When resizing images you can choose whether to maintain the ratio of the original image, or stretch/squash the new image to fit the desired dimensions. If `$maintainRatio` is true, the dimension specified by `$masterDim` will stay the same, while the other dimension will be altered to match the original image's aspect ratio.

```
Services::image('imagick')  
    ->withFile('/path/to/image/mypic.jpg')  
    ->resize(200, 100, true, 'height')  
    ->save('path/to/new/image.jpg');
```

Rotating Images

The `rotate()` method allows you to rotate an image in 90 degree increments:

```
rotate(float $angle)
```

- **\$angle** is the number of degrees to rotate. One of '90', '180', '270'.

Note

While the `$angle` parameter accepts a float, it will convert it to an integer during the process. If the value is any other than the three values listed above, it will throw a `CodeIgniterImagesImageException`.

Adding a Text Watermark

You can overlay a text watermark onto the image very simply with the `text()` method. This is useful for placing copyright notices, photographer names, or simply marking the images as a preview so they won't be used in other people's final products.

```
text(string $text, array $options = [])
```

The first parameter is the string of text that you wish to display. The second parameter is an array of options that allow you to specify how the text should be displayed:

```
Services::image('imagick')
    ->withFile('/path/to/image/mypic.jpg')
    ->text('Copyright 2017 My Photo Co', [
        'color'      => '#fff',
        'opacity'    => 0.5,
        'withShadow' => true,
        'hAlign'     => 'center',
        'vAlign'     => 'bottom',
        'fontSize'   => 20
    ])
    ->save('path/to/new/image.jpg');
```

The possible options that are recognized are as follows:

- **color** Text Color (hex number), i.e. #ff0000
- **opacity** A number between 0 and 1 that represents the opacity of the text.
- **withShadow** Boolean value whether to display a shadow or not.
- **shadowColor** Color of the shadow (hex number)
- **shadowOffset** How many pixels to offset the shadow. Applies to both the vertical and horizontal values.
- **hAlign** Horizontal alignment: left, center, right
- **vAlign** Vertical alignment: top, middle, bottom
- **hOffset** Additional offset on the x axis, in pixels
- **vOffset** Additional offset on the y axis, in pixels
- **fontPath** The full server path to the TTF font you wish to use. System font will be used if none is given.

- `fontSize` The font size to use. When using the GD handler with the system font, valid values are between 1-5.

Note

The ImageMagick driver does not recognize full server path for `fontPath`. Instead, simply provide the name of one of the installed system fonts that you wish to use, i.e. Calibri.

Pagination

CodeIgniter provides a very simple, but flexible pagination library that is simple to theme, works with the model, and capable of supporting multiple paginators on a single page.

Loading the Library

Like all services in CodeIgniter, it can be loaded via `Config\Services`, though you usually will not need to load it manually:

```
$pager = \Config\Services::pager();
```

Paginating Database Results

In most cases, you will be using the Pager library in order to paginate results that you retrieve from the database. When using the [Model](#) class, you can use its built-in `paginate()` method to automatically retrieve the current batch of results, as well as setup the Pager library so it's ready to use in your controllers. It even reads the current page it should display from the current URL via a `page=X` query variable.

To provide a paginated list of users in your application, your controller's method would look something like:

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class UserController extends Controller
{
    public function index()
    {
        $model = new \App\Models\UserModel();

        $data = [
```

```

        'users' => $model->paginate(10),
        'pager' => $model->pager
    ];

    echo view('users/index', $data);
}
}

```

In this example, we first create a new instance of our UserModel. Then we populate the data to send to the view. The first element is the results from the database, **users**, which is retrieved for the correct page, returning 10 users per page. The second item that must be sent to the view is the Pager instance itself. As a convenience, the Model will hold on to the instance it used and store it in the public class variable, **\$pager**. So, we grab that and assign it to the \$pager variable in the view.

Within the view, we then need to tell it where to display the resulting links:

```
<?= $pager->links() ?>
```

And that's all it takes. The Pager class will render a series of links that are compatible with the Bootstrap CSS framework by default. It will have First and Last page links, as well as Next and Previous links for any pages more than two pages on either side of the current page.

If you prefer a simpler output, you can use the `simpleLinks()` method, which only uses “Older” and “Newer” links, instead of the details pagination links:

```
<?= $pager->simpleLinks() ?>
```

Behind the scenes, the library loads a view file that determines how the links are formatted, making it simple to modify to your needs. See below for details on how to completely customize the output.

Paginating Multiple Results

If you need to provide links from two different result sets, you can pass group names to most of the pagination methods to keep the data separate:

```
// In the Controller
```

```

public function index()
{
    $userModel = new \App\Models\UserModel();
    $pageModel = new \App\Models\PageModel();

    $data = [
        'users' => $userModel->paginate(10, 'group1'),
        'pages' => $pageModel->paginate(15, 'group2'),
        'pager' => $userModel->pager
    ];

    echo view('users/index', $data);
}

// In the views:
<?= $pager->links('group1') ?>
<?= $pager->simpleLinks('group2') ?>

```

Manual Pagination

You may find times where you just need to create pagination based on known data. You can create links manually with the `makeLinks()` method, which takes the current page, the amount of results per page, and the total number of items as the first, second, and third parameters, respectively:

```
<?= $pager->makeLinks($page, $perPage, $total) ?>
```

This will, by default, display the links in the normal manner, as a series of links, but you can change the display template used by passing in the name of the template as the fourth parameter. More details can be found in the following sections.

```
<?= $pager->makeLinks($page, $perPage, $total, 'template_name') ?
```

It is also possible to use a URI segment for the page number, instead of the page query parameter. Simply specify the segment number to use as the fifth parameter to `makeLinks()`. URIs generated by the pager would then look like `https://domain.tld/model/[pageNumber]` instead of `https://domain.tld/model?page=[pageNumber]`:

```
<?= $pager->makeLinks($page, $perPage, $total, 'template_name', $
```

Please note: \$segment value can not be greater than the number of URI segments plus 1.

Paginating with Only Expected Queries

By default all GET queries are shown in the pagination links.

For example, when accessing the URL *http://domain.tld?search=foo&order=asc&hello=i+am+here&page=2*, the page 3 link can be generated, along with the other links, as follows:

```
echo $pager->links();  
// Page 3 link: http://domain.tld?search=foo&order=asc&hello=i+am
```

The `only()` method allows you to limit this just to queries already expected:

```
echo $pager->only(['search', 'order'])->links();  
// Page 3 link: http://domain.tld?search=foo&order=asc&page=3
```

The *page* query is enabled by default. And `only()` acts in all pagination links.

Customizing the Links

View Configuration

When the links are rendered out to the page, they use a view file to describe the HTML. You can easily change the view that is used by editing **app/Config/Pager.php**:

```
public $templates = [  
    'default_full' => 'CodeIgniter\Pager\Views\default_full',  
    'default_simple' => 'CodeIgniter\Pager\Views\default_simple'  
];
```

This setting stores the alias and [namespaced view paths](#) for the view that should be used. The *default_full* and *default_simple* views are used for the `links()` and `simpleLinks()` methods, respectively. To change the way those

are displayed application-wide, you could assign a new view here.

For example, say you create a new view file that works with the Foundation CSS framework, instead of Bootstrap, and you place that file at **app/Views/Pagers/foundation_full.php**. Since the **application** directory is namespaced as App, and all directories underneath it map directly to segments of the namespace, you can locate the view file through it's namespace:

```
'default_full' => 'App\Views\Pagers\foundation_full',
```

Since it is under the standard **app/Views** directory, though, you do not need to namespace it since the `view()` method can locate it by filename. In that case, you can simply give the sub-directory and file name:

```
'default_full' => 'Pagers/foundation_full',
```

Once you have created the view and set it in the configuration, it will automatically be used. You don't have to replace the existing templates. You can create as many additional templates as you need in the configuration file. A common situation would be needing different styles for the frontend and the backend of your application.

```
public $templates = [  
    'default_full' => 'CodeIgniter\Pager\Views\default_full',  
    'default_simple' => 'CodeIgniter\Pager\Views\default_simple',  
    'front_full' => 'App\Views\Pagers\foundation_full',  
];
```

Once configured, you can specify it as the last parameter in the `links()`, `simpleLinks()`, and `makeLinks()` methods:

```
<?= $pager->links('group1', 'front_full') ?>  
<?= $pager->simpleLinks('group2', 'front_full') ?>  
<?= $pager->makeLinks($page, $perPage, $total, 'front_full') ?>
```

Creating the View

When you create a new view, you only need to create the code that is needed for creating the pagination links themselves. You should not create unnecessary wrapping divs since it might be used in multiple places and you

only limit their usefulness. It is easiest to demonstrate creating a new view by showing you the existing default_full template:

```
<?php $pager->setSurroundCount(2) ?>

<nav aria-label="Page navigation">
  <ul class="pagination">
    <?php if ($pager->hasPrevious()) : ?>
      <li>
        <a href="<?= $pager->getFirst() ?>" aria-label="First"
          <span aria-hidden="true">First</span>
        </a>
      </li>
      <li>
        <a href="<?= $pager->getPrevious() ?>" aria-label="Pr
          <span aria-hidden="true">&laquo;</span>
        </a>
      </li>
    <?php endif ?>

    <?php foreach ($pager->links() as $link) : ?>
      <li <?= $link['active'] ? 'class="active"' : '' ?>>
        <a href="<?= $link['uri'] ?>"
          <?= $link['title'] ?>
        </a>
      </li>
    <?php endforeach ?>

    <?php if ($pager->hasNext()) : ?>
      <li>
        <a href="<?= $pager->getNext() ?>" aria-label="Previo
          <span aria-hidden="true">&raquo;</span>
        </a>
      </li>
      <li>
        <a href="<?= $pager->getLast() ?>" aria-label="Last">
          <span aria-hidden="true">Last</span>
        </a>
      </li>
    <?php endif ?>
  </ul>
</nav>
```

setSurroundCount()

In the first line, the `setSurroundCount()` method specifies that we want to show two links to either side of the current page link. The only parameter that it accepts is the number of links to show.

hasPrevious() & hasNext()

These methods return a boolean true if there are more links than can be displayed on either side of the current page, based on the value passed to `setSurroundCount`. For example, let's say we have 20 pages of data. The current page is page 3. If the surround count is 2, then the following links would show up in the list: 1, 2, 3, 4, and 5. Since the first link displayed is page one, `hasPrevious()` would return **false** since there is no page zero. However, `hasNext()` would return **true** since there are 15 additional pages of results after page five.

getPrevious() & getNext()

These methods return the URL for the previous or next pages of results on either side of the numbered links. See the previous paragraph for a full explanation.

getFirst() & getLast()

Much like `getPrevious()` and `getNext()`, these methods return links to the first and last pages in the result set.

links()

Returns an array of data about all of the numbered links. Each link's array contains the uri for the link, the title, which is just the number, and a boolean that tells whether the link is the current/active link or not:

```
$link = [  
    'active' => false,  
    'uri'    => 'http://example.com/foo?page=2',  
    'title'  => 1  
];
```

Security Class

The Security Class contains methods that help protect your site against Cross-Site Request Forgery attacks.

- [Loading the Library](#)
- [Cross-site request forgery \(CSRF\)](#)
- [Other Helpful Methods](#)

[Loading the Library](#)

If your only interest in loading the library is to handle CSRF protection, then you will never need to load it, as it runs as a filter and has no manual interaction.

If you find a case where you do need direct access though, you may load it through the Services file:

```
$security = \Config\Services::security();
```

[Cross-site request forgery \(CSRF\)](#)

You can enable CSRF protection by altering your **app/Config/Filters.php** and enabling the *csrf* filter globally:

```
public $globals = [  
    'before' => [  
        // 'honeypot'  
        'csrf'  
    ]  
];
```

Select URIs can be whitelisted from CSRF protection (for example API

endpoints expecting externally POSTed content). You can add these URIs by adding them as exceptions in the filter:

```
public $globals = [  
    'before' => [  
        'csrf' => ['except' => ['api/record/save']]  
    ]  
];
```

Regular expressions are also supported (case-insensitive):

```
public $globals = [  
    'before' => [  
        'csrf' => ['except' => ['api/record/[0-9]+']]  
    ]  
];
```

If you use the [form helper](#), then **form_open()** will automatically insert a hidden csrf field in your forms. If not, then you can use the always available `csrf_token()` and `csrf_hash()` functions

```
<input type="hidden" name="<?= csrf_token() ?>" value="<?= csrf_h  
< >
```

Additionally, you can use the `csrf_field()` method to generate this hidden input field for you:

```
// Generates: <input type="hidden" name="{csrf_token}" value="{cs  
<?= csrf_field() ?>  
< >
```

Tokens may be either regenerated on every submission (default) or kept the same throughout the life of the CSRF cookie. The default regeneration of tokens provides stricter security, but may result in usability concerns as other tokens become invalid (back/forward navigation, multiple tabs/windows, asynchronous actions, etc). You may alter this behavior by editing the following config parameter

```
public $CSRFRegenerate = true;
```

When a request fails the CSRF validation check, it will redirect to the previous page by default, setting an error flash message that you can display

to the end user. This provides a nicer experience than simply crashing. This can be turned off by editing the `$CSRFRedirect` value in **app/Config/App.php**:

```
public $CSRFRedirect = false;
```

Even when the redirect value is **true**, AJAX calls will not redirect, but will throw an error.

Other Helpful Methods

You will never need to use most of the methods in the Security class directly. The following are methods that you might find helpful that are not related to the CSRF protection.

sanitizeFilename()

Tries to sanitize filenames in order to prevent directory traversal attempts and other security threats, which is particularly useful for files that were supplied via user input. The first parameter is the path to sanitize.

If it is acceptable for the user input to include relative paths, e.g. `file/in/some/approved/folder.txt`, you can set the second optional parameter, `$relative_path` to `true`.

```
$path = $security->sanitizeFilename($request->getVar('filepath'))
```

Session Library

The Session class permits you maintain a user's "state" and track their activity while they browse your site.

CodeIgniter comes with a few session storage drivers, that you can see in the last section of the table of contents:

- [Using the Session Class](#)
 - [Initializing a Session](#)
 - [How do Sessions work?](#)
 - [What is Session Data?](#)
 - [Retrieving Session Data](#)
 - [Adding Session Data](#)
 - [Pushing new value to session data](#)
 - [Removing Session Data](#)
 - [Flashdata](#)
 - [Tempdata](#)
 - [Destroying a Session](#)
 - [Accessing session metadata](#)
- [Session Preferences](#)
- [Session Drivers](#)
 - [FileHandler Driver \(the default\)](#)
 - [DatabaseHandler Driver](#)
 - [RedisHandler Driver](#)
 - [MemcachedHandler Driver](#)

[Using the Session Class](#)

[Initializing a Session](#)

Sessions will typically run globally with each page load, so the Session class should be magically initialized.

To access and initialize the session:

```
$session = \Config\Services::session($config);
```

The `$config` parameter is optional - your application configuration. If not provided, the services register will instantiate your default one.

Once loaded, the Sessions library object will be available using:

```
$session
```

Alternatively, you can use the helper function that will use the default configuration options. This version is a little friendlier to read, but does not take any configuration options.

```
$session = session();
```

How do Sessions work?

When a page is loaded, the session class will check to see if a valid session cookie is sent by the user's browser. If a sessions cookie does **not** exist (or if it doesn't match one stored on the server or has expired) a new session will be created and saved.

If a valid session does exist, its information will be updated. With each update, the session ID may be regenerated if configured to do so.

It's important for you to understand that once initialized, the Session class runs automatically. There is nothing you need to do to cause the above behavior to happen. You can, as you'll see below, work with session data, but the process of reading, writing, and updating a session is automatic.

Note

Under CLI, the Session library will automatically halt itself, as this is a concept based entirely on the HTTP protocol.

A note about concurrency

Unless you're developing a website with heavy AJAX usage, you can skip this section. If you are, however, and if you're experiencing performance issues, then this note is exactly what you're looking for.

Sessions in previous versions of CodeIgniter didn't implement locking, which meant that two HTTP requests using the same session could run exactly at the same time. To use a more appropriate technical term - requests were non-blocking.

However, non-blocking requests in the context of sessions also means unsafe, because modifications to session data (or session ID regeneration) in one request can interfere with the execution of a second, concurrent request. This detail was at the root of many issues and the main reason why CodeIgniter 3.0 has a completely re-written Session library.

Why are we telling you this? Because it is likely that after trying to find the reason for your performance issues, you may conclude that locking is the issue and therefore look into how to remove the locks ...

DO NOT DO THAT! Removing locks would be **wrong** and it will cause you more problems!

Locking is not the issue, it is a solution. Your issue is that you still have the session open, while you've already processed it and therefore no longer need it. So, what you need is to close the session for the current request after you no longer need it.

```
$session->destroy();
```

What is Session Data?

Session data is simply an array associated with a particular session ID (cookie).

If you've used sessions in PHP before, you should be familiar with PHP's [`\$_SESSION` superglobal](http://php.net/manual/en/reserved.variables.session.php) (if not, please read the content on that link).

CodeIgniter gives access to its session data through the same means, as it uses the session handlers' mechanism provided by PHP. Using session data is as simple as manipulating (read, set and unset values) the `$_SESSION` array.

In addition, CodeIgniter also provides 2 special types of session data that are further explained below:flashdata and tempdata.

Retrieving Session Data

Any piece of information from the session array is available through the `$_SESSION` superglobal:

```
$_SESSION['item']
```

Or through the conventional accessor method:

```
$session->get('item');
```

Or through the magic getter:

```
$session->item
```

Or even through the session helper method:

```
session('item');
```

Where `item` is the array key corresponding to the item you wish to fetch. For example, to assign a previously stored 'name' item to the `$name` variable, you will do this:

```
$name = $_SESSION['name'];
```

```
// or:
```

```
$name = $session->name
```

```
// or:
```

```
$name = $session->get('name');
```

Note

The `get()` method returns NULL if the item you are trying to access does not exist.

If you want to retrieve all of the existing userdata, you can simply omit the item key (magic getter only works for single property values):

```
$_SESSION
```

```
// or:
```

```
$session->get();
```

Adding Session Data

Let's say a particular user logs into your site. Once authenticated, you could add their username and e-mail address to the session, making that data globally available to you without having to run a database query when you need it.

You can simply assign data to the `$_SESSION` array, as with any other variable. Or as a property of `$session`.

The former userdata method is deprecated, but you can pass an array containing your new session data to the `set()` method:

```
$session->set($array);
```

Where `$array` is an associative array containing your new data. Here's an example:

```
$newdata = [  
    'username' => 'johndoe',  
    'email'    => 'johndoe@some-site.com',  
    'logged_in' => TRUE  
];
```


```
$session->set($newdata);
```

If you want to add session data one value at a time, `set()` also supports this syntax:

```
$session->set('some_name', 'some_value');
```

If you want to verify that a session value exists, simply check with `isset()`:

```
// returns FALSE if the 'some_name' item doesn't exist or is NULL  
// TRUE otherwise:  
isset($_SESSION['some_name'])
```



Or you can call `has()`:

```
$session->has('some_name');
```

[Pushing new value to session data](#)

The push method is used to push a new value onto a session value that is an array. For instance, if the ‘hobbies’ key contains an array of hobbies, you can add a new value onto the array like so:

```
$session->push('hobbies', ['sport'=>'tennis']);
```

[Removing Session Data](#)

Just as with any other variable, unsetting a value in `$_SESSION` can be done through `unset()`:

```
unset($_SESSION['some_name']);  
  
// or multiple values:  
  
unset(  
    $_SESSION['some_name'],  
    $_SESSION['another_name']  
);
```

Also, just as `set()` can be used to add information to a session, `remove()` can

be used to remove it, by passing the session key. For example, if you wanted to remove 'some_name' from your session data array:

```
$session->remove('some_name');
```

This method also accepts an array of item keys to unset:

```
$array_items = ['username', 'email'];  
$session->remove($array_items);
```

Flashdata

CodeIgniter supports “flashdata”, or session data that will only be available for the next request, and is then automatically cleared.

This can be very useful, especially for one-time informational, error or status messages (for example: “Record 2 deleted”).

It should be noted that flashdata variables are regular session variables, managed inside the CodeIgniter session handler.

To mark an existing item as “flashdata”:

```
$session->markAsFlashdata('item');
```

If you want to mark multiple items as flashdata, simply pass the keys as an array:

```
$session->markAsFlashdata(['item', 'item2']);
```

To add flashdata:

```
$_SESSION['item'] = 'value';  
$session->markAsFlashdata('item');
```

Or alternatively, using the setFlashdata() method:

```
$session->setFlashdata('item', 'value');
```

You can also pass an array to setFlashdata(), in the same manner as set().

Readingflashdata variables is the same as reading regular session data through `$_SESSION`:

```
$_SESSION['item']
```

Important

The `get()` method WILL returnflashdata items when retrieving a single item by key. It will not returnflashdata when grabbing all userdata from the session, however.

However, if you want to be sure that you’re reading “flashdata” (and not any other kind), you can also use the `getFlashdata()` method:

```
$session->getFlashdata('item');
```

Or to get an array with allflashdata, simply omit the key parameter:

```
$session->getFlashdata();
```

Note

The `getFlashdata()` method returns NULL if the item cannot be found.

If you find that you need to preserve aflashdata variable through an additional request, you can do so using the `keepFlashdata()` method. You can either pass a single item or an array offlashdata items to keep.

```
$session->keepFlashdata('item');  
$session->keepFlashdata(['item1', 'item2', 'item3']);
```

Tempdata

CodeIgniter also supports “tempdata”, or session data with a specific expiration time. After the value expires, or the session expires or is deleted, the value is automatically removed.

Similarly to flashdata, tempdata variables are managed internally by the CodeIgniter session handler.

To mark an existing item as “tempdata”, simply pass its key and expiry time (in seconds!) to the `mark_as_temp()` method:

```
// 'item' will be erased after 300 seconds  
$session->markAsTempdata('item', 300);
```

You can mark multiple items as tempdata in two ways, depending on whether you want them all to have the same expiry time or not:

```
// Both 'item' and 'item2' will expire after 300 seconds  
$session->markAsTempdata(['item', 'item2'], 300);  
  
// 'item' will be erased after 300 seconds, while 'item2'  
// will do so after only 240 seconds  
$session->markAsTempdata([  
    'item' => 300,  
    'item2' => 240  
]);
```

To add tempdata:

```
$_SESSION['item'] = 'value';  
$session->markAsTempdata('item', 300); // Expire in 5 minutes
```

Or alternatively, using the `setTempdata()` method:

```
$session->setTempdata('item', 'value', 300);
```

You can also pass an array to `set_tempdata()`:

```
$tempdata = ['newuser' => TRUE, 'message' => 'Thanks for joining!'];  
$session->setTempdata($tempdata, NULL, $expire);
```

<  >

Note

If the expiration is omitted or set to 0, the default time-to-live value of 300 seconds (or 5 minutes) will be used.

To read a tempdata variable, again you can just access it through the `$_SESSION` superglobal array:

```
$_SESSION['item']
```

Important

The `get()` method WILL return tempdata items when retrieving a single item by key. It will not return tempdata when grabbing all userdata from the session, however.

Or if you want to be sure that you're reading "tempdata" (and not any other kind), you can also use the `getTempdata()` method:

```
$session->getTempdata('item');
```

And of course, if you want to retrieve all existing tempdata:

```
$session->getTempdata();
```

Note

The `getTempdata()` method returns NULL if the item cannot be found.

If you need to remove a tempdata value before it expires, you can directly unset it from the `$_SESSION` array:

```
unset($_SESSION['item']);
```

However, this won't remove the marker that makes this specific item to be tempdata (it will be invalidated on the next HTTP request), so if you intend to reuse that same key in the same request, you'd want to use `removeTempdata()`:

```
$session->removeTempdata('item');
```

Destroying a Session

To clear the current session (for example, during a logout), you may simply use either PHP's [session_destroy\(\)](http://php.net/session_destroy) [http://php.net/session_destroy] function, or the `sess_destroy()` method. Both will work in exactly the same way:

```
session_destroy();  
  
// or  
  
$session->destroy();
```

Note

This must be the last session-related operation that you do during the same request. All session data (including flashdata and tempdata) will be destroyed permanently and functions will be unusable during the same request after you destroy the session.

You may also use the `stop()` method to completely kill the session by removing the old `session_id`, destroying all data, and destroying the cookie that contained the session id:

```
$session->stop();
```

Accessing session metadata

In previous CodeIgniter versions, the session data array included 4 items by default: 'session_id', 'ip_address', 'user_agent', 'last_activity'.

This was due to the specifics of how sessions worked, but is now no longer necessary with our new implementation. However, it may happen that your application relied on these values, so here are alternative methods of accessing them:

- `session_id`: `session_id()`
- `ip_address`: `$_SERVER['REMOTE_ADDR']`
- `user_agent`: `$this->input->user_agent()` (unused by sessions)
- `last_activity`: Depends on the storage, no straightforward way.
Sorry!

Session Preferences

CodeIgniter will usually make everything work out of the box. However, Sessions are a very sensitive component of any application, so some careful configuration must be done. Please take your time to consider all of the options and their effects.

You'll find the following Session related preferences in your **app/Config/App.php** file:

Preference	Default	Options
sessionDriver	CodeIgniterSessionHandlersFileHandler	CodeIgniterSessionHandlersFileHandler CodeIgniterSessionHandlersDatabase CodeIgniterSessionHandlersDatabase CodeIgniterSessionHandlersDatabase
sessionCookieName	ci_session	[A-Z a-z 0-9 - _]
sessionExpiration	7200 (2 hours)	Time in seconds
sessionSavePath	NULL	None

sessionMatchIP	FALSE	TRUE
-----------------------	-------	------

sessionTimeToUpdate	300	Time
----------------------------	-----	------

sessionRegenerateDestroy FALSE

TRUE

Note

As a last resort, the Session library will try to fetch PHP's session related INI settings, as well as legacy CI settings such as 'sess_expire_on_close' when any of the above is not configured. However, you should never rely on this behavior as it can cause unexpected results or be changed in the future. Please configure everything properly.

In addition to the values above, the cookie and native drivers apply the following configuration values shared by the [IncomingRequest](#) and [Security](#) classes:

Preference	Default	Description
cookieDomain	'	The domain for which the session is applicable
cookiePath	/	The path to which the session is applicable
cookieSecure	FALSE	Whether to create the session cookie only on encrypted (HTTPS) connections

Note

The ‘cookieHTTPOnly’ setting doesn’t have an effect on sessions. Instead the HttpOnly parameter is always enabled, for security reasons. Additionally, the ‘cookiePrefix’ setting is completely ignored.

Session Drivers

As already mentioned, the Session library comes with 4 handlers, or storage engines, that you can use:

- CodeIgniterSessionHandlersFileHandler
- CodeIgniterSessionHandlersDatabaseHandler
- CodeIgniterSessionHandlersMemcachedHandler
- CodeIgniterSessionHandlersRedisHandler

By default, the FileHandler Driver will be used when a session is initialized, because it is the most safe choice and is expected to work everywhere (virtually every environment has a file system).

However, any other driver may be selected via the public `$sessionDriver` line in your **app/Config/App.php** file, if you chose to do so. Have it in mind though, every driver has different caveats, so be sure to get yourself familiar with them (below) before you make that choice.

FileHandler Driver (the default)

The ‘FileHandler’ driver uses your file system for storing session data.

It can safely be said that it works exactly like PHP’s own default session implementation, but in case this is an important detail for you, have it in mind that it is in fact not the same code and it has some limitations (and advantages).

To be more specific, it doesn’t support PHP’s [directory level and mode formats used in session.save_path](#)

[<http://php.net/manual/en/session.configuration.php#ini.session.save-path>], and it has most of the options hard-coded for safety. Instead, only absolute paths are supported

for public \$sessionSavePath.

Another important thing that you should know, is to make sure that you don't use a publicly-readable or shared directory for storing your session files. Make sure that *only you* have access to see the contents of your chosen *sessionSavePath* directory. Otherwise, anybody who can do that, can also steal any of the current sessions (also known as “session fixation” attack).

On UNIX-like operating systems, this is usually achieved by setting the 0700 mode permissions on that directory via the *chmod* command, which allows only the directory's owner to perform read and write operations on it. But be careful because the system user *running* the script is usually not your own, but something like 'www-data' instead, so only setting those permissions will probable break your application.

Instead, you should do something like this, depending on your environment

```
mkdir /<path to your application directory>/Writable/sessions/  
chmod 0700 /<path to your application directory>/Writable/session  
chown www-data /<path to your application directory>/Writable/ses  
< >
```

Bonus Tip

Some of you will probably opt to choose another session driver because file storage is usually slower. This is only half true.

A very basic test will probably trick you into believing that an SQL database is faster, but in 99% of the cases, this is only true while you only have a few current sessions. As the sessions count and server loads increase - which is the time when it matters - the file system will consistently outperform almost all relational database setups.

In addition, if performance is your only concern, you may want to look into using [tmpfs](http://eddmann.com/posts/storing-php-sessions-file-caches-in-memory-using-tmpfs/) [http://eddmann.com/posts/storing-php-sessions-file-caches-in-memory-using-tmpfs/], (warning: external resource), which can make your sessions blazing fast.

[DatabaseHandler Driver](#)

The 'DatabaseHandler' driver uses a relational database such as MySQL or PostgreSQL to store sessions. This is a popular choice among many users, because it allows the developer easy access to the session data within an application - it is just another table in your database.

However, there are some conditions that must be met:

- You can NOT use a persistent connection.
- You can NOT use a connection with the *cacheOn* setting enabled.

In order to use the 'DatabaseHandler' session driver, you must also create this table that we already mentioned and then set it as your `$sessionSavePath` value. For example, if you would like to use 'ci_sessions' as your table name, you would do this:

```
public $sessionDriver = 'CodeIgniter\Session\Handlers\DatabaseH
public $sessionSavePath = 'ci_sessions';
```

And then of course, create the database table ...

For MySQL:

```
CREATE TABLE IF NOT EXISTS `ci_sessions` (
  `id` varchar(128) NOT NULL,
  `ip_address` varchar(45) NOT NULL,
  `timestamp` int(10) unsigned DEFAULT 0 NOT NULL,
  `data` blob NOT NULL,
  KEY `ci_sessions_timestamp` (`timestamp`)
);
```

For PostgreSQL:

```
CREATE TABLE "ci_sessions" (
  "id" varchar(128) NOT NULL,
  "ip_address" varchar(45) NOT NULL,
  "timestamp" bigint DEFAULT 0 NOT NULL,
  "data" text DEFAULT '' NOT NULL
);
```


```
CREATE INDEX "ci_sessions_timestamp" ON "ci_sessions" ("timestamp"
```

You will also need to add a PRIMARY KEY **depending on your ‘sessionMatchIP’ setting**. The examples below work both on MySQL and PostgreSQL:

```
// When sessionMatchIP = TRUE
ALTER TABLE ci_sessions ADD PRIMARY KEY (id, ip_address);

// When sessionMatchIP = FALSE
ALTER TABLE ci_sessions ADD PRIMARY KEY (id);

// To drop a previously created primary key (use when changing th
ALTER TABLE ci_sessions DROP PRIMARY KEY;
```



You can choose the Database group to use by adding a new line to the **applicationConfigApp.php** file with the name of the group to use:

```
public $sessionDBGroup = 'groupName';
```

If you’d rather not do all of this by hand, you can use the `session:migration` command from the cli to generate a migration file for you:

```
> php spark session:migration
> php spark migrate
```

This command will take the **sessionSavePath** and **sessionMatchIP** settings into account when it generates the code.

Important

Only MySQL and PostgreSQL databases are officially supported, due to lack of advisory locking mechanisms on other platforms. Using sessions without locks can cause all sorts of problems, especially with heavy usage of AJAX, and we will not support such cases. Use `session_write_close()` after you’ve done processing session data if you’re having performance issues.

[RedisHandler Driver](#)

Note

Since Redis doesn't have a locking mechanism exposed, locks for this driver are emulated by a separate value that is kept for up to 300 seconds.

Redis is a storage engine typically used for caching and popular because of its high performance, which is also probably your reason to use the 'RedisHandler' session driver.

The downside is that it is not as ubiquitous as relational databases and requires the [phpredis](https://github.com/phpredis/phpredis) [https://github.com/phpredis/phpredis] PHP extension to be installed on your system, and that one doesn't come bundled with PHP. Chances are, you're only be using the RedisHandler driver only if you're already both familiar with Redis and using it for other purposes.

Just as with the 'FileHandler' and 'DatabaseHandler' drivers, you must also configure the storage location for your sessions via the `$sessionSavePath` setting. The format here is a bit different and complicated at the same time. It is best explained by the *phpredis* extension's README file, so we'll simply link you to it:

<https://github.com/phpredis/phpredis#php-session-handler>

Warning

CodeIgniter's Session library does NOT use the actual 'redis' `session.save_handler`. Take note **only** of the path format in the link above.

For the most common case however, a simple `host:port` pair should be sufficient:

```
public $sessionDriver      = 'CodeIgniter\Session\Handlers\RedisHand
public $sessionSavePath    = 'tcp://localhost:6379';
```

<

>

MemcachedHandler Driver

Note

Since Memcached doesn't have a locking mechanism exposed, locks for this driver are emulated by a separate value that is kept for up to 300 seconds.

The 'MemcachedHandler' driver is very similar to the 'RedisHandler' one in all of its properties, except perhaps for availability, because PHP's [Memcached](http://php.net/memcached) [http://php.net/memcached] extension is distributed via PECL and some Linux distributions make it available as an easy to install package.

Other than that, and without any intentional bias towards Redis, there's not much different to be said about Memcached - it is also a popular product that is usually used for caching and famed for its speed.

However, it is worth noting that the only guarantee given by Memcached is that setting value X to expire after Y seconds will result in it being deleted after Y seconds have passed (but not necessarily that it won't expire earlier than that time). This happens very rarely, but should be considered as it may result in loss of sessions.

The \$sessionSavePath format is fairly straightforward here, being just a host:port pair:

```
public $sessionDriver    = 'CodeIgniter\Session\Handlers\Memcached';  
public $sessionSavePath = 'localhost:11211';
```

Bonus Tip

Multi-server configuration with an optional *weight* parameter as the third colon-separated (:weight) value is also supported, but we have to note that we haven't tested if that is reliable.

If you want to experiment with this feature (on your own risk), simply

separate the multiple server paths with commas:

```
// localhost will be given higher priority (5) here,  
// compared to 192.0.2.1 with a weight of 1.  
public $sessionSavePath = 'localhost:11211:5,192.0.2.1:11211:1';
```

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Throttler

- [Overview](#)
- [Rate Limiting](#)
 - [The Code](#)
 - [Applying the Filter](#)
- [Class Reference](#)

The Throttler class provides a very simple way to limit an activity to be performed to a certain number of attempts within a set period of time. This is most often used for performing rate limiting on API's, or restricting the number of attempts a user can make against a form to help prevent brute force attacks. The class itself can be used for anything that you need to throttle based on actions within a set time interval.

[Overview](#)

The Throttler implements a simplified version of the [Token Bucket](#) [https://en.wikipedia.org/wiki/Token_bucket] algorithm. This basically treats each action that you want as a bucket. When you call the `check()` method, you tell it how large the bucket is, and how many tokens it can hold and the time interval. Each `check()` call uses 1 of the available tokens, by default. Let's walk through an example to make this clear.

Let's say we want an action to happen once every second. The first call to the Throttler would look like the following. The first parameter is the bucket name, the second parameter the number of tokens the bucket holds, and the third being the amount of time it takes the bucket to refill:

```
$throttler = \Config\Services::throttler();  
$throttler->check($name, 60, MINUTE);
```


Here we're using one of the [global constants](#) for the time, to make it a little more readable. This says that the bucket allows 60 actions every minute, or 1 action every second.

Let's say that a third-party script was trying to hit a URL repeatedly. At first, it would be able to use all 60 of those tokens in less than a second. However, after that the Throttler would only allow one action per second, potentially slowing down the requests enough that they attack is no longer worth it.

Note

For the Throttler class to work, the Cache library must be setup to use a handler other than dummy. For best performance, an in-memory cache, like Redis or Memcached, is recommended.

[Rate Limiting](#)

The Throttler class does not do any rate limiting or request throttling on its own, but is the key to making one work. An example [Filter](#) is provided that implements very simple rate limiting at one request per second per IP address. Here we will run through how it works, and how you could set it up and start using it in your application.

[The Code](#)

You could make your own Throttler filter, at **app/Filters/Throttle.php**, along the lines of:

```
<?php namespace App\Filters;

use CodeIgniter\Filters\FilterInterface;
use CodeIgniter\HTTP\RequestInterface;
use CodeIgniter\HTTP\ResponseInterface;
use Config\Services;

class Throttle implements FilterInterface
{
    /**
```

```

* This is a demo implementation of using the Throttler c
* to implement rate limiting for your application.
*
* @param RequestInterface|\CodeIgniter\HTTP\IncomingRequ
*
* @return mixed
*/
public function before(RequestInterface $request)
{
    $throttler = Services::throttler();

    // Restrict an IP address to no more
    // than 1 request per second across the
    // entire site.
    if ($throttler->check($request->getIPAddress(), 6
    {
        return Services::response()->setStatusCod
    }

    //-----

/**
 * We don't have anything to do here.
 *
 * @param RequestInterface|\CodeIgniter\HTTP\IncomingRequ
 * @param ResponseInterface|\CodeIgniter\HTTP\Response
 *
 * @return mixed
 */
public function after(RequestInterface $request, Response
{
}
}

```

When run, this method first grabs an instance of the throttler. Next it uses the IP address as the bucket name, and sets things to limit them to one request per second. If the throttler rejects the check, returning false, then we return a Response with the status code set to 429 - Too Many Attempts, and the script execution ends before it ever hits the controller. This example will throttle based on a single IP address across all requests made to the site, not per page.

[Applying the Filter](#)

We don't necessarily need to throttle every page on the site. For many web applications this makes the most sense to apply only to POST requests, though API's might want to limit every request made by a user. In order to apply this to incoming requests, you need to edit **/app/Config/Filters.php** and first add an alias to the filter:

```
public $aliases = [  
    ...  
    'throttle' => \App\Filters\Throttle::class  
];
```

Next, we assign it to all POST requests made on the site:

```
public $methods = [  
    'post' => ['throttle', 'CSRF']  
];
```

And that's all there is to it. Now all POST requests made on the site will have be rate limited.

[Class Reference](#)

check(*string \$key*, *int \$capacity*, *int \$seconds*[, *int \$cost* = 1])

- **\$key** (*string*) – The name of the bucket
- **\$capacity** (*int*) – The number of tokens the bucket holds

Parameters:

- **\$seconds** (*int*) – The number of seconds it takes for a bucket to completely fill
- **\$cost** (*int*) – The number of tokens that are spent for this action

Returns: TRUE if action can be performed, FALSE if not

Return type: bool

Checks to see if there are any tokens left within the bucket, or if too many have been used within the allotted time limit. During each check the available tokens are reduced by \$cost if successful.

getTokenTime()

Returns: The number of seconds until another token should be available.

Return type: integer

After `check()` has been run and returned `FALSE`, this method can be used to determine the time until a new token should be available and the action can be tried again. In this case the minimum enforced wait time is one second.

Dates and Times

CodeIgniter provides a fully-localized, immutable, date/time class that is built on PHP's `DateTime` object, but uses the `Intl` extension's features to convert times across timezones and display the output correctly for different locales. This class is the **Time** class and lives in the **CodeIgniter\I18n** namespace.

Note

Since the `Time` class extends `DateTime`, if there are features that you need that this class doesn't provide, you can likely find them within the `DateTime` class itself.

- [Instantiating](#)
- [Displaying the Value](#)
- [Working with Individual Values](#)

Instantiating

There are several ways that a new `Time` instance can be created. The first is simply to create a new instance like any other class. When you do it this way, you can pass in a string representing the desired time. This can be any string that PHP's `strtotime` function can parse:

```
use CodeIgniter\I18n\Time;

$myTime = new Time('+3 week');
$myTime = new Time('now');
```

You can pass in strings representing the timezone and the locale in the second and parameters, respectively. Timezones can be any supported by PHP's [DateTimeZone](http://php.net/manual/en/timezones.php) [http://php.net/manual/en/timezones.php] class. The locale can be any

supported by PHP's [Locale](http://php.net/manual/en/class.locale.php) [http://php.net/manual/en/class.locale.php] class. If no locale or timezone is provided, the application defaults will be used.

```
$myTime = new Time('now', 'America/Chicago', 'en_US');
```

now()

The Time class has several helper methods to instantiate the class. The first of these is the **now()** method that returns a new instance set to the current time. You can pass in strings representing the timezone and the locale in the second and third parameters, respectively. If no locale or timezone is provided, the application defaults will be used.

```
$myTime = Time::now('America/Chicago', 'en_US');
```

parse()

This helper method is a static version of the default constructor. It takes a string acceptable as DateTime's constructor as the first parameter, a timezone as the second parameter, and the locale as the third parameter.:

```
$myTime = Time::parse('next Tuesday', 'America/Chicago', 'en_US')
```

today()

Returns a new instance with the date set to the current date, and the time set to midnight. It accepts strings for the timezone and locale in the second and third parameters:

```
$myTime = Time::today('America/Chicago', 'en_US');
```

yesterday()

Returns a new instance with the date set to the yesterday's date and the time set to midnight. It accepts strings for the timezone and locale in the second and third parameters:

```
$myTime = Time::yesterday('America/Chicago', 'en_US');
```

tomorrow()

Returns a new instance with the date set to the tomorrow's date and the time set to midnight. It accepts strings for the timezone and locale in the second and third parameters:

```
$myTime = Time::tomorrow('America/Chicago', 'en_US');
```

createFromDate()

Given separate inputs for **year**, **month**, and **day**, will return a new instance. If any of these parameters are not provided, it will use the current value to fill it in. Accepts strings for the timezone and locale in the fourth and fifth parameters:

```
$today      = Time::createFromDate();           // Uses current  
$anniversary = Time::createFromDate(2018);      // Uses current month  
$date       = Time::createFromDate(2018, 3, 15, 'America/Chicago'  
< >
```

createFromTime()

Like **createFromDate** except it is only concerned with the **hours**, **minutes**, and **seconds**. Uses the current day for the date portion of the Time instance. Accepts strings for the timezone and locale in the fourth and fifth parameters:

```
$lunch  = Time::createFromTime(11, 30)          // 11:30 am today  
$dinner = Time::createFromTime(18, 00, 00)      // 6:00 pm today  
$time   = Time::createFromTime($hour, $minutes, $seconds, $timezo  
< >
```

create()

A combination of the previous two methods, takes **year**, **month**, **day**, **hour**, **minutes**, and **seconds** as separate parameters. Any value not provided will use the current date and time to determine. Accepts strings for the timezone and locale in the fourth and fifth parameters:

```
$time = Time::create($year, $month, $day, $hour, $minutes, $secon  
< >
```

createFromFormat()

This is a replacement for DateTime's method of the same name. This allows the timezone to be set at the same time, and returns a **Time** instance, instead of DateTime:

```
$time = Time::createFromFormat('j-M-Y', '15-Feb-2009', 'America/C
```



createFromTimestamp()

This method takes a UNIX timestamp and, optionally, the timezone and locale, to create a new Time instance:

```
$time = Time::createFromTimestamp(1501821586, 'America/Chicago',
```



instance()

When working with other libraries that provide a DateTime instance, you can use this method to convert that to a Time instance, optionally setting the locale. The timezone will be automatically determined from the DateTime instance passed in:

```
$dt    = new DateTime('now');  
$time = Time::instance($dt, 'en_US');
```

toDateTime()

While not an instantiator, this method is the opposite of the **instance** method, allowing you to convert a Time instance into a DateTime instance. This preserves the timezone setting, but loses the locale, since DateTime is not aware of locales:

```
$datetime = Time::toDateTime();
```

Displaying the Value

Since the Time class extends DateTime, you get all of the output methods that provides, including the format() method. However, the DateTime methods do not provide a localize result. The Time class does provide a number of helper methods to display localized versions of the value, though.

toLocalizedString()

This is the localized version of DateTime's format() method. Instead of using the values you might be familiar with, though, you must use values acceptable to the [IntlDateFormatter](http://php.net/manual/en/class.intldateformatter.php) class. A full listing of values can be found [here](http://www.icu-project.org/apiref/icu4c/classSimpleDateFormat.html#details).

```
$time = Time::parse('March 9, 2016 12:00:00', 'America/Chicago');  
echo $time->toLocalizedString('MMM d, yyyy');    // March 9, 2016
```

toDateTimeString()

This is the first of three helper methods to work with the IntlDateFormatter without having to remember their values. This will return a string formatted as you would commonly use for datetime columns in a database (Y-m-d H:i:s):

```
$time = Time::parse('March 9, 2016 12:00:00', 'America/Chicago');  
echo $time->toDateTimeString();    // 2016-03-09 12:00:00
```

toDateString()

Displays just the date portion of the Time:

```
$time = Time::parse('March 9, 2016 12:00:00', 'America/Chicago');  
echo $time->toDateTimeString();    // 2016-03-09
```

toTimeString()

Displays just the time portion of the value:

```
$time = Time::parse('March 9, 2016 12:00:00', 'America/Chicago');  
echo $time->toTimeString();    // 12:00:00
```

humanize()

This method returns a string that displays the difference between the current date/time and the instance in a human readable format that is geared towards being easily understood. It can create strings like '3 hours ago', 'in 1 month', etc:

```
// Assume current time is: March 10, 2017 (America/Chicago)
$time = Time::parse('March 9, 2016 12:00:00', 'America/Chicago');

echo $time->humanize();      // 1 year ago
```

The exact time displayed is determined in the following manner:

Time difference	Result
\$time > 1 year && < 2 years	in 1 year / 1 year ago
\$time > 1 month && < 1 year	in 6 months / 6 months ago
\$time > 7 days && < 1 month	in 3 weeks / 3 weeks ago
\$time > today && < 7 days	in 4 days / 4 days ago
\$time == tomorrow / yesterday	Tomorrow / Yesterday
\$time > 59 minutes && < 1 day	1:37pm
\$time > now && < 1 hour	in 35 minutes / 35 minutes ago
\$time == now	Now

The exact language used is controlled through the language file, Time.php.

Working with Individual Values

The Time object provides a number of methods to allow to get and set individual items, like the year, month, hour, etc, of an existing instance. All of the values retrieved through the following methods will be fully localized and respect the locale that the Time instance was created with.

All of the following *getX* and *setX* methods can also be used as if they were a class property. So, any calls to methods like *getYear* can also be accessed through *\$time->year*, and so on.

Getters

The following basic getters exist:

```
$time = Time::parse('August 12, 2016 4:15:23pm');

echo $time->getYear();           // 2016
echo $time->getMonth();          // 8
echo $time->getDay();            // 12
echo $time->getHour();           // 16
echo $time->getMinute();         // 15
echo $time->getSecond();         // 23

echo $time->year;                 // 2016
echo $time->month;                // 8
echo $time->day;                 // 12
echo $time->hour;                // 16
echo $time->minute;              // 15
echo $time->second;              // 23
```

In addition to these, a number of methods exist to provide additional information about the date:

```
$time = Time::parse('August 12, 2016 4:15:23pm');

echo $time->getDayOfWeek();      // 6 - but may vary based on locale
echo $time->getDayOfYear();      // 225
echo $time->getWeekOfMonth();    // 2
echo $time->getWeekOfYear();     // 33
echo $time->getTimestamp();      // 1471018523 - UNIX timestamp
echo $time->getQuarter();        // 3

echo $time->dayOfWeek;           // 6
echo $time->dayOfYear;           // 225
echo $time->weekOfMonth;         // 2
echo $time->weekOfYear;         // 33
echo $time->timestamp;           // 1471018523
echo $time->quarter;             // 3
```

getAge()

Returns the age, in years, of between the Time's instance and the current time. Perfect for checking the age of someone based on their birthday:

```
$time = Time::parse('5 years ago');
```

```
echo $time->getAge();    // 5  
echo $time->age;         // 5
```

getDST()

Returns boolean true/false based on whether the Time instance is currently observing Daylight Savings Time:

```
echo Time::createFromDate(2012, 1, 1)->getDst();    // false  
echo Time::createFromDate(2012, 9, 1)->dst;        // true
```

getLocal()

Returns boolean true if the Time instance is in the same timezone as the application is currently running in:

```
echo Time::now()->getLocal();    // true  
echo Time::now('Europe/London'); // false
```

getUtc()

Returns boolean true if the Time instance is in UTC time:

```
echo Time::now('America/Chicago')->getUtc();    // false  
echo Time::now('UTC')->utc;                      // true
```

getTimezone()

Returns a new [DateTimeZone](http://php.net/manual/en/class.datetimezone.php) [http://php.net/manual/en/class.datetimezone.php] object set the timezone of the Time instance:

```
$tz = Time::now()->getTimezone();  
$tz = Time::now()->timezone;
```

```
echo $tz->getName();  
echo $tz->getOffset();
```

getTimezoneName()

Returns the full [timezone string](http://php.net/manual/en/timezones.php) [http://php.net/manual/en/timezones.php] of the Time instance:

```
echo Time::now('America/Chicago')->getTimezoneName(); // Americ
echo Time::now('Europe/London')->timezoneName;         // Europe
< _____ >
```

Setters

The following basic setters exist. If any of the values set are out of range, an `InvalidArgumentException` will be thrown.

Note

All setters will return a new Time instance, leaving the original instance untouched.

Note

All setters will throw an `InvalidArgumentException` if the value is out of range.

```
$time = $time->setYear(2017);
$time = $time->setMonthNumber(4);           // April
$time = $time->setMonthLongName('April');
$time = $time->setMonthShortName('Feb');    // February
$time = $time->setDay(25);
$time = $time->setHour(14);                  // 2:00 pm
$time = $time->setMinute(30);
$time = $time->setSecond(54);
```

setTimezone()

Converts the time from it's current timezone into the new one:

```
$time  = Time::parse('May 10, 2017', 'America/Chicago');
$time2 = $time->setTimezone('Europe/London');           // Return
```

```
echo $time->timezoneName;    // American/Chicago
echo $time2->timezoneName;    // Europe/London
```

setTimestamp()

Returns a new instance with the date set to the new timestamp:

```
$time = Time::parse('May 10, 2017', 'America/Chicago');
$time2 = $time->setTimestamp(strtotime('April 1, 2017'));

echo $time->toDateTimeString();    // 2017-05-10 00:00:00
echo $time2->toDateTimeString();    // 2017-04-01 00:00:00
```

Modifying the Value

The following methods allow you to modify the date by adding or subtracting values to the current Time. This will not modify the existing Time instance, but will return a new instance.

```
$time = $time->addSeconds(23);
$time = $time->addMinutes(15);
$time = $time->addHours(12);
$time = $time->addDays(21);
$time = $time->addMonths(14);
$time = $time->addYears(5);

$time = $time->subSeconds(23);
$time = $time->subMinutes(15);
$time = $time->subHours(12);
$time = $time->subDays(21);
$time = $time->subMonths(14);
$time = $time->subYears(5);
```

Comparing Two Times

The following methods allow you to compare one Time instance with another. All comparisons are first converted to UTC before comparisons are done, to ensure that different timezones respond correctly.

equals()

Determines if the datetime passed in is equal to the current instance. Equal in this case means that they represent the same moment in time, and are not required to be in the same timezone, as both times are converted to UTC and compared that way:

```
$time1 = Time::parse('January 10, 2017 21:50:00', 'America/Chicago')
$time2 = Time::parse('January 11, 2017 03:50:00', 'Europe/London')

$time1->equals($time2);    // true
```

The value being tested against can be a Time instance, a DateTime instance, or a string with the full date time in a manner that a new DateTime instance can understand. When passing a string as the first parameter, you can pass a timezone string in as the second parameter. If no timezone is given, the system default will be used:

```
$time1->equals('January 11, 2017 03:50:00', 'Europe/London'); //
```

sameAs()

This is identical to the **equals** method, except that it only returns true when the date, time, AND timezone are all identical:

```
$time1 = Time::parse('January 10, 2017 21:50:00', 'America/Chicago')
$time2 = Time::parse('January 11, 2017 03:50:00', 'Europe/London')

$time1->sameAs($time2);    // false
$time2->sameAs('January 10, 2017 21:50:00', 'America/Chicago');
```

isBefore()

Checks if the passed in time is before the the current instance. The comparison is done against the UTC versions of both times:

```
$time1 = Time::parse('January 10, 2017 21:50:00', 'America/Chicago')
$time2 = Time::parse('January 11, 2017 03:50:00', 'America/Chicago')

$time1->isBefore($time2);    // true
$time2->isBefore($time1);    // false
```

The value being tested against can be a Time instance, a DateTime instance, or a string with the full date time in a manner that a new DateTime instance can understand. When passing a string as the first parameter, you can pass a timezone string in as the second parameter. If no timezone is given, the system default will be used:

```
$time1->isBefore('March 15, 2013', 'America/Chicago'); // false
```

isAfter()

Works exactly the same as **isBefore()** except checks if the time is after the time passed in:

```
$time1 = Time::parse('January 10, 2017 21:50:00', 'America/Chicago');  
$time2 = Time::parse('January 11, 2017 03:50:00', 'America/Chicago');  
  
$time1->isAfter($time2); // false  
$time2->isAfter($time1); // true
```

Viewing Differences

To compare two Times directly, you would use the **difference()** method, which returns a **CodeIgniterI18nTimeDifference** instance. The first parameter is either a Time instance, a DateTime instance, or a string with the date/time. If a string is passed in the first parameter, the second parameter can be a timezone string:

```
$time = Time::parse('March 10, 2017', 'America/Chicago');  
  
$diff = $time->difference(Time::now());  
$diff = $time->difference(new DateTime('July 4, 1975', 'America/Chicago'));  
$diff = $time->difference('July 4, 1975 13:32:05', 'America/Chicago');
```

Once you have the TimeDifference instance, you have several methods you can use to find information about the difference between the two times. The value returned will be negative if it was in the past, or positive if in the future

from the original time:

```
$current = Time::parse('March 10, 2017', 'America/Chicago');
$test    = Time::parse('March 10, 2010', 'America/Chicago');

$diff = $current->difference($test);

echo $diff->getYears();      // -7
echo $diff->getMonths();     // -84
echo $diff->getWeeks();      // -365
echo $diff->getDays();       // -2557
echo $diff->getHours();      // -61368
echo $diff->getMinutes();    // -3682080
echo $diff->getSeconds();    // -220924800
```

You can use either **getX()** methods, or access the calculate values as if they were properties:

```
echo $diff->years;          // -7
echo $diff->months;         // -84
echo $diff->weeks;          // -365
echo $diff->days;          // -2557
echo $diff->hours;          // -61368
echo $diff->minutes;        // -3682080
echo $diff->seconds;        // -220924800
```

humanize()

Much like Time's `humanize()` method, this returns a string that displays the difference between the times in a human readable format that is geared towards being easily understood. It can create strings like '3 hours ago', 'in 1 month', etc. The biggest differences are in how very recent dates are handled:

```
// Assume current time is: March 10, 2017 (America/Chicago)
$time = Time::parse('March 9, 2016 12:00:00', 'America/Chicago');

echo $time->humanize();      // 1 year ago
```

The exact time displayed is determined in the following manner:

Time difference	Result
\$time > 1 year && < 2 years	in 1 year / 1 year ago

\$time > 1 month && < 1 year	in 6 months / 6 months ago
\$time > 7 days && < 1 month	in 3 weeks / 3 weeks ago
\$time > today && < 7 days	in 4 days / 4 days ago
\$time > 1 hour && < 1 day	in 8 hours / 8 hours ago
\$time > 1 minute && < 1 hour	in 35 minutes / 35 minutes ago
\$time < 1 minute	Now

The exact language used is controlled through the language file, Time.php.

Typography

The Typography library contains methods that help you format text in semantically relevant ways.

Loading the Library

Like all services in CodeIgniter, it can be loaded via `Config\Services`, though you usually will not need to load it manually:

```
$typography = \Config\Services::typography();
```

Available static functions

The following functions are available:

autoTypography()

autoTypography(\$str[, \$reduce_linebreaks = FALSE])

- **\$str** (*string*) – Input string
 - **\$reduce_linebreaks** (*bool*) – Whether to reduce multiple instances of double newlines to two
- Parameters:**
- Returns:** HTML-formatted typography-safe string
- Return type:** string

Formats text so that it is semantically and typographically correct HTML.

Usage example:

```
$string = $typography->autoTypography($string);
```

Note

Typographic formatting can be processor intensive, particularly if you

have a lot of content being formatted. If you choose to use this function you may want to consider [caching](#) your pages.

formatCharacters()

formatCharacters(\$str)

Parameters: • **\$str** (*string*) – Input string
Returns: String with formatted characters.
Return type: string

This function mainly converts double and single quotes to curly entities, but it also converts em-dashes, double spaces, and ampersands.

Usage example:

```
$string = $typography->formatCharacters($string);
```

nl2brExceptPre()

nl2brExceptPre(\$str)

Parameters: • **\$str** (*string*) – Input string
Returns: String with HTML-formatted line breaks
Return type: string

Converts newlines to `
` tags unless they appear within `<pre>` tags. This function is identical to the native PHP `nl2br()` function, except that it ignores `<pre>` tags.

Usage example:

```
$string = $typography->nl2brExceptPre($string);
```

Working with Uploaded Files

CodeIgniter makes working with files uploaded through a form much simpler and more secure than using PHP's `$_FILES` array directly. This extends the [File class](#) and thus gains all of the features of that class.

Note

This is not the same as the File Uploading class in previous versions of CodeIgniter. This provides a raw interface to the uploaded files with a few small features.

- [Accessing Files](#)
 - [All Files](#)
 - [Single File](#)
- [Working With the File](#)
 - [Verify A File](#)
 - [File Names](#)
 - [Other File Info](#)
 - [Moving Files](#)
 - [Store Files](#)

[Accessing Files](#)

[All Files](#)

When you upload files they can be accessed natively in PHP through the `$_FILES` superglobal. This array has some major shortcomings when working with multiple files uploaded at once, and has potential security flaws many developers are not aware of. CodeIgniter helps with both of these situations

by standardizing your usage of files behind a common interface.

Files are accessed through the current `IncomingRequest` instance. To retrieve all files that were uploaded with this request, use `getFiles()`. This will return an array of files represented by instances of `CodeIgniter\HTTP\Files\UploadedFile`:

```
$files = $this->request->getFiles();
```

Of course, there are multiple ways to name the file input, and anything but the simplest can create strange results. The array returns in a manner that you would expect. With the simplest usage, a single file might be submitted like:

```
<input type="file" name="avatar" />
```

Which would return a simple array like:

```
[
    'avatar' => // UploadedFile instance
]
```

If you used an array notation for the name, the input would look something like:

```
<input type="file" name="my-form[details][avatar]" />
```

The array returned by `getFiles()` would look more like this:

```
[
    'my-form' => [
        'details' => [
            'avatar' => // UploadedFile instance
        ]
    ]
]
```

In some cases, you may specify an array of files to upload:

Upload an avatar: `<input type="file" name="my-form[details][avata`
Upload an avatar: `<input type="file" name="my-form[details][avata`

< >

In this case, the returned array of files would be more like:

```
[
    'my-form' => [
        'details' => [
            'avatar' => [
                0 => /* UploadedFile instance */,
                1 => /* UploadedFile instance */
            ]
        ]
    ]
]
```

Single File

If you just need to access a single file, you can use `getFile()` to retrieve the file instance directly. This will return an instance of `CodeIgniter\HTTP\Files\UploadedFile`:

Simplest usage

With the simplest usage, a single file might be submitted like:

```
<input type="file" name="userfile" />
```

Which would return a simple file instance like:

```
$file = $this->request->getFile('userfile');
```

Array notation

If you used an array notation for the name, the input would look something like:

```
<input type="file" name="my-form[details][avatar]" />
```

For get the file instance:

```
$file = $this->request->getFile('my-form.details.avatar');
```

Multiple files

```
<input type="file" name="images[]" multiple />
```

In controller:

```
if($imagefile = $this->request->getFiles())
{
    foreach($imagefile['images'] as $img)
    {
        if ($img->isValid() && ! $img->hasMoved())
        {
            $newName = $img->getRandomName();
            $img->move(WRITEPATH.'uploads', $newName);
        }
    }
}
```

where the **images** is loop is from the form field name

If there are multiple files with the same name you can use getFile() to retrieve every file individually:: In controller:

```
$file1 = $this->request->getFile('images.0');
$file2 = $this->request->getFile('images.1');
```

Another example:

Upload an avatar: <input type="file" name="my-form[details][avatars.0]" />
Upload an avatar: <input type="file" name="my-form[details][avatars.1]" />

In controller:

```
$file1 = $this->request->getFile('my-form.details.avatars.0');
$file2 = $this->request->getFile('my-form.details.avatars.1');
```

Note

using getFile() is more appropriate


Working With the File

Once you've retrieved the UploadedFile instance, you can retrieve information about the file in safe ways, as well as move the file to a new location.

Verify A File

You can check that a file was actually uploaded via HTTP with no errors by calling the `isValid()` method:

```
if (! $file->isValid())
{
    throw new RuntimeException($file->getErrorString(). '('.$f
}
```



As seen in this example, if a file had an upload error, you can retrieve the error code (an integer) and the error message with the `getError()` and `getErrorString()` methods. The following errors can be discovered through this method:

- The file exceeds your `upload_max_filesize` ini directive.
- The file exceeds the upload limit defined in your form.
- The file was only partially uploaded.
- No file was uploaded.
- The file could not be written on disk.
- File could not be uploaded: missing temporary directory.
- File upload was stopped by a PHP extension.

File Names

getName()

You can retrieve the original filename provided by the client with the `getName()` method. This will typically be the filename sent by the client, and should not be trusted. If the file has been moved, this will return the final name of the moved file:

```
$name = $file->getName();
```

getClientName()

Always returns the original name of the uploaded file as sent by the client, even if the file has been moved:

```
$originalName = $file->getClientName();
```

getTempName()

To get the full path of the temp file that was created during the upload, you can use the `getTempName()` method:

```
$tempfile = $file->getTempName();
```

Other File Info

getClientExtension()

Returns the original file extension, based on the file name that was uploaded. This is NOT a trusted source. For a trusted version, use `getExtension()` instead:

```
$ext = $file->getClientExtension();
```

getClientType()

Returns the mime type (mime type) of the file as provided by the client. This is NOT a trusted value. For a trusted version, use `getType()` instead:

```
$type = $file->getClientType();
```

```
echo $type; // image/png
```

Moving Files

Each file can be moved to its new location with the aptly named `move()` method. This takes the directory to move the file to as the first parameter:

```
$file->move(WRITEPATH, 'uploads');
```

By default, the original filename was used. You can specify a new filename by passing it as the second parameter:

```
$newName = $file->getRandomName();  
$file->move(WRITEPATH.'uploads', $newName);
```

Once the file has been removed the temporary file is deleted. You can check if a file has been moved already with the `hasMoved()` method, which returns a boolean:

```
if ($file->isValid() && ! $file->hasMoved())  
{  
    $file->move($path);  
}
```

Moving an uploaded file can fail, with an `HttpException`, under several circumstances:

- the file has already been moved
- the file did not upload successfully
- the file move operation fails (eg. improper permissions)

Store Files

Each file can be moved to its new location with the aptly named `store()` method.

With the simplest usage, a single file might be submitted like:

```
<input type="file" name="userfile" />
```

By default, Upload files are saved in `writable/uploads` directory. the `YYYYMMDD` folder and random file name will be created. return a file path:

```
$path = $this->request->getFile('userfile')->store();
```

You can specify directory to move the file to as the first parameter. a new filename by passing it as the second parameter:

```
$path = $this->request->getFile('userfile')->store('head_img/', '
< >
```

Moving an uploaded file can fail, with an HTTPException, under several circumstances:

- the file has already been moved
- the file did not upload successfully
- the file move operation fails (eg. improper permissions)

Working with URIs

CodeIgniter provides an object oriented solution for working with URI's in your application. Using this makes it simple to ensure that the structure is always correct, no matter how complex the URI might be, as well as adding relative URI to an existing one and have it resolved safely and correctly.

- [Creating URI instances](#)
 - [The Current URI](#)
- [URI Strings](#)
- [The URI Parts](#)
 - [Scheme](#)
 - [Authority](#)
 - [Userinfo](#)
 - [Host](#)
 - [Port](#)
 - [Path](#)
 - [Query](#)
 - [Fragment](#)
- [URI Segments](#)

[Creating URI instances](#)

Creating a URI instance is as simple as creating a new class instance:

```
$uri = new \CodeIgniter\HTTP\URI();
```

Alternatively, you can use the `service()` function to return an instance for you:

```
$uri = service('uri');
```

When you create the new instance, you can pass a full or partial URL in the constructor and it will be parsed into its appropriate sections:

```
$uri = new \CodeIgniter\HTTP\URI('http://www.example.com/some/path')
$uri = service('uri', 'http://www.example.com/some/path');
```

The Current URI

Many times, all you really want is an object representing the current URL of this request. This can be accessed in two different ways. The first, is to grab it directly from the current request object. Assuming that you're in a controller that extends `CodeIgniter\Controller` you can get it like:

```
$uri = $this->request->uri;
```

Second, you can use one of the functions available in the **url_helper**:

```
$uri = current_url(true);
```

You must pass `true` as the first parameter, otherwise it will return the string representation of the current URL.

URI Strings

Many times, all you really want is to get a string representation of a URI. This is easy to do by simply casting the URI as a string:

```
$uri = current_url(true);
echo (string)$uri; // http://example.com
```

If you know the pieces of the URI and just want to ensure it's all formatted correctly, you can generate a string using the URI class' static `createURIString()` method:

```
$uriString = URI::createURIString($scheme, $authority, $path, $query,
```

```
// Creates: http://example.com/some/path?foo=bar#first-heading
echo URI::createURIString('http', 'example.com', 'some/path', 'foo=bar', '#first-heading');
```

The URI Parts

Once you have a URI instance, you can set or retrieve any of the various parts of the URI. This section will provide details on what those parts are, and how to work with them.

Scheme

The scheme is frequently 'http' or 'https', but any scheme is supported, including 'file', 'mailto', etc.

```
$uri = new \CodeIgniter\HTTP\URI('http://www.example.com/some/pat  
echo $uri->getScheme(); // 'http'  
$uri->setScheme('https');
```

Authority

Many URIs contain several elements that are collectively known as the 'authority'. This includes any user info, the host and the port number. You can retrieve all of these pieces as one single string with the `getAuthority()` method, or you can manipulate the individual parts.

```
$uri = new \CodeIgniter\HTTP\URI('ftp://user:password@example.com  
echo $uri->getAuthority(); // user@example.com:21
```

By default, this will not display the password portion since you wouldn't want to show that to anyone. If you want to show the password, you can use the `showPassword()` method. This URI instance will continue to show that password until you turn it off again, so always make sure that you turn it off as soon as you are finished with it:

```
echo $uri->getAuthority(); // user@example.com:21  
echo $uri->showPassword()->getAuthority(); // user:password@exa  
  
// Turn password display off again.  
$uri->showPassword(false);
```

If you do not want to display the port, pass in `true` as the only parameter:

```
echo $uri->getAuthority(true); // user@example.com
```

Note

If the current port is the default port for the scheme it will never be displayed.

Userinfo

The userinfo section is simply the username and password that you might see with an FTP URI. While you can get this as part of the Authority, you can also retrieve it yourself:

```
echo $uri->getUserInfo(); // user
```

By default, it will not display the password, but you can override that with the `showPassword()` method:

```
echo $uri->showPassword()->getUserInfo(); // user:password
$uri->showPassword(false);
```

Host

The host portion of the URI is typically the domain name of the URL. This can be easily set and retrieved with the `getHost()` and `setHost()` methods:

```
$uri = new \CodeIgniter\HTTP\URI('http://www.example.com/some/path');
echo $uri->getHost(); // www.example.com
echo $uri->setHost('anotherexample.com')->getHost(); // anotherexample.com
```

Port

The port is an integer number between 0 and 65535. Each scheme has a

default value associated with it.

```
$uri = new \CodeIgniter\HTTP\URI('ftp://user:password@example.com')  
  
echo $uri->getPort(); // 21  
echo $uri->setPort(2201)->getPort(); // 2201
```

When using the `setPort()` method, the port will be checked that it is within the valid range and assigned.

Path

The path are all of the segments within the site itself. As expected, the `getPath()` and `setPath()` methods can be used to manipulate it:

```
$uri = new \CodeIgniter\HTTP\URI('http://www.example.com/some/path')  
  
echo $uri->getPath(); // 'some/path'  
echo $uri->setPath('another/path')->getPath(); // 'another/path'
```

Note

When setting the path this way, or any other way the class allows, it is sanitized to encode any dangerous characters, and remove dot segments for safety.

Query

The query variables can be manipulated through the class using simple string representations. Query values can only be set as a string currently.

```
$uri = new \CodeIgniter\HTTP\URI('http://www.example.com?foo=bar')  
  
echo $uri->getQuery(); // 'foo=bar'  
$uri->setQuery('foo=bar&bar=baz');
```

Note

Query values cannot contain fragments. An `InvalidArgumentException` will be thrown if it does.

You can set query values using an array:

```
$uri->setQueryArray(['foo' => 'bar', 'bar' => 'baz']);
```

The `setQuery()` and `setQueryArray()` methods overwrite any existing query variables. You can add a value to the query variables collection without destroying the existing query variables with the `addQuery()` method. The first parameter is the name of the variable, and the second parameter is the value:

```
$uri->addQuery('foo', 'bar');
```

Filtering Query Values

You can filter the query values returned by passing an options array to the `getQuery()` method, with either an *only* or an *except* key:

```
$uri = new \CodeIgniter\HTTP\URI('http://www.example.com?foo=bar&baz=foz');  
  
// Returns 'foo=bar'  
echo $uri->getQuery(['only' => ['foo']]);  
  
// Returns 'foo=bar&baz=foz'  
echo $uri->getQuery(['except' => ['bar']]);
```

This only changes the values returned during this one call. If you need to modify the URI's query values more permanently, you can use the `stripQuery()` and `keepQuery()` methods to change the actual object's query variable collection:

```
$uri = new \CodeIgniter\HTTP\URI('http://www.example.com?foo=bar&baz=foz');  
  
// Leaves just the 'baz' variable  
$uri->stripQuery('foo', 'bar');  
  
// Leaves just the 'foo' variable  
$uri->keepQuery('foo');
```

Fragment

Fragments are the portion at the end of the URL, preceded by the pound-sign (#). In HTML URL's these are links to an on-page anchor. Media URI's can make use of them in various other ways.

```
$uri = new \CodeIgniter\HTTP\URI('http://www.example.com/some/pat  
echo $uri->getFragment();    // 'first-heading'  
echo $uri->setFragment('second-heading')->getFragment();    // 's  
< >
```

URI Segments

Each section of the path between the slashes are a single segment. The URI class provides a simple way to determine what the values of the segments are. The segments start at 1 being the furthest left of the path.

```
// URI = http://example.com/users/15/profile  
  
// Prints '15'  
if ($request->uri->getSegment(1) == 'users')  
{  
    echo $request->uri->getSegment(2);  
}
```

You can get a count of the total segments:

```
$total = $request->uri->getTotalSegments(); // 3
```

Finally, you can retrieve an array of all of the segments:

```
$segments = $request->uri->getSegments();  
  
// $segments =  
[  
    0 => 'users',  
    1 => '15',  
    2 => 'profile'  
]
```


User Agent Class

The User Agent Class provides functions that help identify information about the browser, mobile device, or robot visiting your site.

- [Using the User Agent Class](#)
 - [Initializing the Class](#)
 - [User Agent Definitions](#)
 - [Example](#)
- [Class Reference](#)

[Using the User Agent Class](#)

[Initializing the Class](#)

The User Agent class is always available directly from the current [IncomingRequest](#) instance. By default, you will have a request instance in your controller that you can retrieve the User Agent class from:

```
$agent = $this->request->getUserAgent();
```

[User Agent Definitions](#)

The user agent name definitions are located in a config file located at: **app/Config/UserAgents.php**. You may add items to the various user agent arrays if needed.

[Example](#)

When the User Agent class is initialized it will attempt to determine whether the user agent browsing your site is a web browser, a mobile device, or a

robot. It will also gather the platform information if it is available:

```
$agent = $this->request->getUserAgent();

if ($agent->isBrowser())
{
    $currentAgent = $agent->getBrowser().' '.$agent->getVersi
}
elseif ($agent->isRobot())
{
    $currentAgent = $this->agent->robot();
}
elseif ($agent->isMobile())
{
    $currentAgent = $agent->getMobile();
}
else
{
    $currentAgent = 'Unidentified User Agent';
}

echo $currentAgent;

echo $agent->getPlatform(); // Platform info (Windows, Linux, Mac
< >
```

Class Reference

CodeIgniter\HTTP\UserAgent

isBrowser(*[\$key = NULL]*)

Parameters: • **\$key** (*string*) – Optional browser name

Returns: TRUE if the user agent is a (specified) browser,
FALSE if not

**Return
type:** bool

Returns TRUE/FALSE (boolean) if the user agent is a known web browser.

```
if ($agent->isBrowser('Safari'))
{
    echo 'You are using Safari.';
}
```

```

}
elseif ($agent->isBrowser())
{
    echo 'You are using a browser.';
}

```

Note

The string “Safari” in this example is an array key in the list of browser definitions. You can find this list in **app/Config/UserAgents.php** if you want to add new browsers or change the strings.

isMobile(*[\$key = NULL]*)

Parameters: • **\$key** (*string*) – Optional mobile device name

Returns: TRUE if the user agent is a (specified) mobile device, FALSE if not

Return type: bool

Returns TRUE/FALSE (boolean) if the user agent is a known mobile device.

```

if ($agent->isMobile('iphone'))
{
    echo view('iphone/home');
}
elseif ($agent->isMobile())
{
    echo view('mobile/home');
}
else
{
    echo view('web/home');
}

```

isRobot(*[\$key = NULL]*)

Parameters: • **\$key** (*string*) – Optional robot name

Returns: TRUE if the user agent is a (specified) robot, FALSE if not

Return bool
type:

Returns TRUE/FALSE (boolean) if the user agent is a known robot.

Note

The user agent library only contains the most common robot definitions. It is not a complete list of bots. There are hundreds of them so searching for each one would not be very efficient. If you find that some bots that commonly visit your site are missing from the list you can add them to your **app/Config/UserAgents.php** file.

isReferral()

Returns: TRUE if the user agent is a referral, FALSE if not
Return type: bool

Returns TRUE/FALSE (boolean) if the user agent was referred from another site.

getBrowser()

Returns: Detected browser or an empty string
Return type: string

Returns a string containing the name of the web browser viewing your site.

getVersion()

Returns: Detected browser version or an empty string
Return type: string

Returns a string containing the version number of the web browser viewing your site.

getMobile()

Returns: Detected mobile device brand or an empty string
Return type: string

Returns a string containing the name of the mobile device viewing your site.

getRobot()

Returns: Detected robot name or an empty string

Return type: string

Returns a string containing the name of the robot viewing your site.

getPlatform()

Returns: Detected operating system or an empty string

Return type: string

Returns a string containing the platform viewing your site (Linux, Windows, OS X, etc.).

getReferrer()

Returns: Detected referrer or an empty string

Return type: string

The referrer, if the user agent was referred from another site. Typically you'll test for this as follows:

```
if ($agent->isReferral())
{
    echo $agent->referrer();
}
```

getAgentString()

Returns: Full user agent string or an empty string

Return type: string

Returns a string containing the full user agent string. Typically it will be something like this:

Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.0.
< >

parse(*\$string*)

Parameters:

- **\$string** (*string*) – A custom user-agent string

Return type: void

Parses a custom user-agent string, different from the one reported by the current visitor.

Validation

CodeIgniter provides a comprehensive data validation class that helps minimize the amount of code you'll write.

- [Overview](#)
- [Form Validation Tutorial](#)
 - [The Form](#)
 - [The Success Page](#)
 - [The Controller](#)
 - [Try it!](#)
 - [Explanation](#)
 - [Loading the Library](#)
 - [Setting Validation Rules](#)
- [Working with Validation](#)
 - [Validating Keys that are Arrays](#)
 - [Validate 1 Value](#)
 - [Saving Sets of Validation Rules to the Config File](#)
- [Working With Errors](#)
 - [Setting Custom Error Messages](#)
 - [Getting All Errors](#)
 - [Getting a Single Error](#)
 - [Check If Error Exists](#)
- [Customizing Error Display](#)
 - [Creating the Views](#)
 - [Configuration](#)
 - [Specifying the Template](#)
- [Creating Custom Rules](#)
 - [Allowing Parameters](#)
- [Available Rules](#)
 - [Rules for File Uploads](#)

Overview

Before explaining CodeIgniter's approach to data validation, let's describe the ideal scenario:

1. A form is displayed.
2. You fill it in and submit it.
3. If you submitted something invalid, or perhaps missed a required item, the form is redisplayed containing your data along with an error message describing the problem.
4. This process continues until you have submitted a valid form.

On the receiving end, the script must:

1. Check for required data.
2. Verify that the data is of the correct type, and meets the correct criteria. For example, if a username is submitted it must be validated to contain only permitted characters. It must be of a minimum length, and not exceed a maximum length. The username can't be someone else's existing username, or perhaps even a reserved word. Etc.
3. Sanitize the data for security.
4. Pre-format the data if needed (Does the data need to be trimmed? HTML encoded? Etc.)
5. Prep the data for insertion in the database.

Although there is nothing terribly complex about the above process, it usually requires a significant amount of code, and to display error messages, various control structures are usually placed within the form HTML. Form validation, while simple to create, is generally very messy and tedious to implement.

Form Validation Tutorial

What follows is a "hands on" tutorial for implementing CodeIgniter's Form Validation.

In order to implement form validation you'll need three things:

1. A [View](#) file containing a form.
2. A View file containing a “success” message to be displayed upon successful submission.
3. A [controller](#) method to receive and process the submitted data.

Let’s create those three things, using a member sign-up form as the example.

[The Form](#)

Using a text editor, create a form called **Signup.php**. In it, place this code and save it to your **app/Views/** folder:

```
<html>
<head>
    <title>My Form</title>
</head>
<body>

<?= $validation->listErrors() ?>

<?= form_open('form') ?>

<h5>Username</h5>
<input type="text" name="username" value="" size="50" />

<h5>Password</h5>
<input type="text" name="password" value="" size="50" />

<h5>Password Confirm</h5>
<input type="text" name="passconf" value="" size="50" />

<h5>Email Address</h5>
<input type="text" name="email" value="" size="50" />

<div><input type="submit" value="Submit" /></div>

</form>

</body>
</html>
```

[The Success Page](#)

Using a text editor, create a form called **Success.php**. In it, place this code and save it to your **app/Views/** folder:

```
<html>
<head>
    <title>My Form</title>
</head>
<body>

<h3>Your form was successfully submitted!</h3>

<p><?= anchor('form', 'Try it again!') ?></p>

</body>
</html>
```

The Controller

Using a text editor, create a controller called **Form.php**. In it, place this code and save it to your **app/Controllers/** folder:

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Form extends Controller
{
    public function index()
    {
        helper(['form', 'url']);

        if (! $this->validate([]))
        {
            echo view('Signup', [
                'validation' => $this->validation
            ]);
        }
        else
        {
            echo view('Success');
        }
    }
}
```

[Try it!](#)

To try your form, visit your site using a URL similar to this one:

`example.com/index.php/form/`

If you submit the form you should simply see the form reload. That's because you haven't set up any validation rules yet.

Since you haven't told the Validation class to validate anything yet, it returns false (boolean false) by default. The `validate()` method only returns true if it has successfully applied your rules without any of them failing.

[Explanation](#)

You'll notice several things about the above pages:

The form (Signup.php) is a standard web form with a couple exceptions:

1. It uses a form helper to create the form opening. Technically, this isn't necessary. You could create the form using standard HTML. However, the benefit of using the helper is that it generates the action URL for you, based on the URL in your config file. This makes your application more portable in the event your URLs change.
2. At the top of the form you'll notice the following function call:

```
<?= $validation->listErrors() ?>
```

This function will return any error messages sent back by the validator. If there are no messages it returns an empty string.

The controller (Form.php) has one method: `index()`. This method uses the Controller-provided `validate` method and loads the form helper and URL helper used by your view files. It also runs the validation routine. Based on whether the validation was successful it either presents the form or the success page.

Loading the Library

The library is loaded as a service named **validation**:

```
$validation = \Config\Services::validation();
```

This automatically loads the `Config\Validation` file which contains settings for including multiple Rule sets, and collections of rules that can be easily reused.

Note

You may never need to use this method, as both the [Controller](#) and the [Model](#) provide methods to make validation even easier.

Setting Validation Rules

CodeIgniter lets you set as many validation rules as you need for a given field, cascading them in order. To set validation rules you will use the `setRule()`, `setRules()`, or `withRequest()` methods.

setRule()

This method sets a single rule. It takes the name of field as the first parameter, an optional label and a string with a pipe-delimited list of rules that should be applied:

```
$validation->setRule('username', 'Username', 'required');
```

The **field name** must match the key of any data array that is sent in. If the data is taken directly from `$_POST`, then it must be an exact match for the form input name.

setRules()

Like, `setRule()`, but accepts an array of field names and their rules:


```
$validation->setRules([
    'username' => 'required',
    'password' => 'required|min_length[10]'
]);
```

To give a labeled error message you can setup as:

```
$validation->setRules([
    'username' => ['label' => 'Username', 'rules' => 'required'],
    'password' => ['label' => 'Password', 'rules' => 'required|mi
]);
```

withRequest()

One of the most common times you will use the validation library is when validating data that was input from an HTTP Request. If desired, you can pass an instance of the current Request object and it will take all of the input data and set it as the data to be validated:

```
$validation->withRequest($this->request)
    ->run();
```

Working with Validation

Validating Keys that are Arrays

If your data is in a nested associative array, you can use “dot array syntax” to easily validate your data:

```
// The data to test:
'contacts' => [
    'name' => 'Joe Smith',
    'friends' => [
        [
            'name' => 'Fred Flinstone'
        ],
        [
            'name' => 'Wilma'
        ]
    ]
]
```

```
// Joe Smith
$validation->setRules([
    'contacts.name' => 'required'
]);

// Fred Flintstone & Wilma
$validation->setRules([
    'contacts.friends.name' => 'required'
]);
```

You can use the ‘*’ wildcard symbol to match any one level of the array:

```
// Fred Flintstone & Wilma
$validation->setRules([
    'contacts.*.name' => 'required'
]);
```

Validate 1 Value

Validate one value against a rule:

```
$validation->check($value, 'required');
```

Saving Sets of Validation Rules to the Config File

A nice feature of the Validation class is that it permits you to store all your validation rules for your entire application in a config file. You organize the rules into “groups”. You can specify a different group every time you run the validation.

How to save your rules

To store your validation rules, simply create a new public property in the Config\Validation class with the name of your group. This element will hold an array with your validation rules. As shown earlier, the validation array will have this prototype:

```
class Validation
{
    public $signup = [
```

```

        'username'      => 'required',
        'password'      => 'required',
        'pass_confirm' => 'required|matches[password]',
        'email'         => 'required|valid_email'
    ];
}

```

You can specify the group to use when you call the `run()` method:

```
$validation->run($data, 'signup');
```


You can also store custom error messages in this configuration file by naming the property the same as the group, and appended with `_errors`. These will automatically be used for any errors when this group is used:

```

class Validation
{
    public $signup = [
        'username'      => 'required',
        'password'      => 'required',
        'pass_confirm' => 'required|matches[password]',
        'email'         => 'required|valid_email'
    ];

    public $signup_errors = [
        'username' => [
            'required' => 'You must choose a username.',
        ],
        'email' => [
            'valid_email' => 'Please check the Email field. It do
        ]
    ];
}

```



Or pass all settings in an array:

```

class Validation
{
    public $signup = [
        'username' => [
            'label' => 'Username',
            'rules' => 'required',
            'errors' => [
                'required' => 'You must choose a {field}.'
            ]
        ]
    ];
}

```

```

        ],
        'email' => 'required|valid_email'
    ];

    public $signup_errors = [
        'email' => [
            'valid_email' => 'Please check the Email field. It do
        ]
    ];
}

```

See below for details on the formatting of the array.

Getting & Setting Rule Groups

Get Rule Group

This method gets a rule group from the validation configuration:

```
$validation->getRuleGroup('signup');
```

Set Rule Group

This method sets a rule group from the validation configuration to the validation service:

```
$validation->setRuleGroup('signup');
```

Working With Errors

The Validation library provides several methods to help you set error messages, provide custom error messages, and retrieve one or more errors to display.

By default, error messages are derived from language strings in `system/Language/en/Validation.php`, where each rule has an entry.

Setting Custom Error Messages

Both the `setRule()` and `setRules()` methods can accept an array of custom messages that will be used as errors specific to each field as their last parameter. This allows for a very pleasant experience for the user since the errors are tailored to each instance. If not custom error message is provided, the default value will be used.

These are two ways to provide custom error messages.

As the last parameter:

```
$validation->setRules([
    'username' => 'required|is_unique[users.username]',
    'password' => 'required|min_length[10]'
],
    // Errors
    [
        'username' => [
            'required' => 'All accounts must have usernames provi
        ],
        'password' => [
            'min_length' => 'Your password is too short. You want
        ]
    ]
);
```

Or as a labeled style:

```
$validation->setRules([
    'username' => [
        'label' => 'Username',
        'rules' => 'required|is_unique[users.username]',
        'errors' => [
            'required' => 'All accounts must have {field} pro
        ]
    ],
    'password' => [
        'label' => 'Password',
        'rules' => 'required|min_length[10]',
        'errors' => [
            'min_length' => 'Your {field} is too short. You w
        ]
    ]
]);
```

If you'd like to include a field's "human" name, or the optional parameter some rules allow for (such as `max_length`), you can add the `{field}` and `{param}` tags to your message, respectively:

```
'min_length' => '{field} must have at least {param} characters.'
```

On a field with the human name `Username` and a rule of `min_length[5]`, an error would display: "Username must have at least 5 characters."

Note

If you pass the last parameter the labeled style error messages will be ignored.

Getting All Errors

If you need to retrieve all error messages for failed fields, you can use the `getErrors()` method:

```
$errors = $validation->getErrors();
```

```
// Returns:
```

```
[
    'field1' => 'error message',
    'field2' => 'error message',
]
```

If no errors exist, an empty array will be returned.

Getting a Single Error

You can retrieve the error for a single field with the `getError()` method. The only parameter is the field name:

```
$error = $validation->getError('username');
```

If no error exists, an empty string will be returned.

Check If Error Exists

You can check to see if an error exists with the `hasError()` method. The only parameter is the field name:

```
if ($validation->hasError('username'))
{
    echo $validation->getError('username');
}
```

Customizing Error Display

When you call `$validation->listErrors()` or `$validation->showError()`, it loads a view file in the background that determines how the errors are displayed. By default, they display in a manner compatible with the [Bootstrap](http://getbootstrap.com/) [http://getbootstrap.com/] CSS framework. You can easily create new views and use them throughout your application.

Creating the Views

The first step is to create the custom views. These can be placed anywhere that the `view()` method can locate them, which means the standard View directory, or any namespaced View folder will work. For example, you could create a new view at **/app/Views/_errors_list.php**:

```
<div class="alert alert-danger" role="alert">
    <ul>
        <?php foreach ($errors as $error) : ?>
            <li><?= esc($error) ?></li>
        <?php endforeach ?>
    </ul>
</div>
```

An array named `$errors` is available within the view that contains a list of the errors, where the key is the name of the field that had the error, and the value is the error message, like this:

```
$errors = [
    'username' => 'The username field must be unique.',
    'email'    => 'You must provide a valid email address.'
```

```
];
```

There are actually two types of views that you can create. The first has an array of all of the errors, and is what we just looked at. The other type is simpler, and only contains a single variable, `$error` that contains the error message. This is used with the `showError()` method where a field must be specified:

```
<span class="help-block"><?= esc($error) ?></span>
```

Configuration

Once you have your views created, you need to let the Validation library know about them. Open `Config/Validation.php`. Inside, you'll find the `$templates` property where you can list as many custom views as you want, and provide an short alias they can be referenced by. If we were to add our example file from above, it would look something like:

```
public $templates = [  
    'list'      => 'CodeIgniter\Validation\Views\list',  
    'single'    => 'CodeIgniter\Validation\Views\single',  
    'my_list'   => '_errors_list'  
];
```

Specifying the Template

You can specify the template to use by passing it's alias as the first parameter in `listErrors`:

```
<?= $validation->listErrors('my_list') ?>
```

When showing field-specific errors, you can pass the alias as the second parameter to the `showError` method, right after the name of the field the error should belong to:

```
<?= $validation->showError('username', 'my_single') ?>
```

Creating Custom Rules

Rules are stored within simple, namespaced classes. They can be stored any location you would like, as long as the autoloader can find it. These files are called RuleSets. To add a new RuleSet, edit **Config/Validation.php** and add the new file to the \$ruleSets array:

```
public $ruleSets = [  
    \CodeIgniter\Validation\Rules::class,  
    \CodeIgniter\Validation\FileRules::class,  
    \CodeIgniter\Validation\CreditCardRules::class,  
];
```

You can add it as either a simple string with the fully qualified class name, or using the ::class suffix as shown above. The primary benefit here is that it provides some extra navigation capabilities in more advanced IDEs.

Within the file itself, each method is a rule and must accept a string as the first parameter, and must return a boolean true or false value signifying true if it passed the test or false if it did not:

```
class MyRules  
{  
    public function even(string $str): bool  
    {  
        return (int)$str % 2 == 0;  
    }  
}
```

By default, the system will look within CodeIgniter\Language\en\Validation.php for the language strings used within errors. In custom rules you may provide error messages by accepting an \$error variable by reference in the second parameter:

```
public function even(string $str, string &$error = null): bool  
{  
    if ((int)$str % 2 != 0)  
    {  
        $error = lang('myerrors.evenError');  
        return false;  
    }  
    return true;  
}
```

Your new custom rule could now be used just like any other rule:

```
$this->validate($request, [
    'foo' => 'required|even'
]);
```

Allowing Parameters

If your method needs to work with parameters, the function will need a minimum of three parameters: the string to validate, the parameter string, and an array with all of the data that was submitted the form. The `$data` array is especially handy for rules like `required_with` that needs to check the value of another submitted field to base its result on:

```
public function required_with($str, string $fields, array $data):
{
    $fields = explode(',', $fields);

    // If the field is present we can safely assume that
    // the field is here, no matter whether the corresponding
    // search field is present or not.
    $present = $this->required($str ?? '');

    if ($present)
    {
        return true;
    }

    // Still here? Then we fail this test if
    // any of the fields are present in $data
    // as $fields is the list
    $requiredFields = [];

    foreach ($fields as $field)
    {
        if (array_key_exists($field, $data))
        {
            $requiredFields[] = $field;
        }
    }

    // Remove any keys with empty values since, that means th
    // weren't truly there, as far as this is concerned.
    $requiredFields = array_filter($requiredFields, function
        return ! empty($data[$item]);
```

```

    });
    return empty($requiredFields);
}

```

Custom errors can be returned as the fourth parameter, just as described above.

Available Rules

The following is a list of all the native rules that are available to use:

Note

Rule is string; there must be no spaces between the parameters, especially the “is_unique” rule. There can be no spaces before and after “ignore_value”.

- “is_unique[supplier.name,uuid, \$uuid]” is not ok
- “is_unique[supplier.name,uuid,\$uuid]” is not ok
- “is_unique[supplier.name,uuid,\$uuid]” is ok

Rule	Parameter	Description	Example
alpha	No	Fails if field has anything other than alphabetic characters.	
alpha_space	No	Fails if field contains anything other than alphabetic characters or spaces.	
alpha_dash	No	Fails if field contains anything other than alpha-numeric characters, underscores or dashes.	
		Fails if field contains	

alpha_numeric	No	anything other than alpha-numeric characters or numbers.	
alpha_numeric_space	No	Fails if field contains anything other than alpha-numeric characters, numbers or space.	
decimal	No	Fails if field contains anything other than a decimal number.	
differs	Yes	Fails if field does not differ from the one in the parameter.	differs[fi
exact_length	Yes	Fails if field is not exactly the parameter value.	exact_lei
greater_than	Yes	Fails if field is less than or equal to the parameter value or not numeric.	greater_t
greater_than_equal_to	Yes	Fails if field is less than the parameter value, or not numeric.	greater_t
in_list	Yes	Fails if field is not within a predetermined list.	in_list[re
integer	No	Fails if field contains anything other than an integer.	
is_natural	No	Fails if field contains anything other than a natural number: 0, 1, 2, 3, etc.	
is_natural_no_zero	No	Fails if field contains anything other than a natural number, except zero: 1, 2, 3, etc.	
less_than	Yes	Fails if field is greater than or equal to the parameter value or not numeric.	less_thar

less_than_equal_to	Yes	Fails if field is greater than the parameter value or not numeric.	less_than
matches	Yes	The value must match the value of the field in the parameter.	matches
max_length	Yes	Fails if field is longer than the parameter value.	max_len
min_length	Yes	Fails if field is shorter than the parameter value.	min_len
numeric	No	Fails if field contains anything other than numeric characters.	
regex_match	Yes	Fails if field does not match the regular expression.	regex_m
if_exist	No	If this rule is present, validation will only return possible errors if the field key exists, regardless of its value.	
permit_empty	No	Allows the field to receive an empty array, empty string, null or false.	
required	No	Fails if the field is an empty array, empty string, null or false.	
required_with	Yes	The field is required if any of the fields in the parameter are set.	required.
required_without	Yes	The field is required when any of the fields in the parameter are not set.	required.
is_unique	Yes	Checks if this field value exists in the database. Optionally set a column and value to ignore, useful when updating records to ignore	is_uniqu

		itself.	
timezone	No	Fails if field does match a timezone per <code>timezone_identifiers_list</code>	
valid_base64	No	Fails if field contains anything other than valid Base64 characters.	
valid_json	No	Fails if field does not contain a valid JSON string.	
valid_email	No	Fails if field does not contain a valid email address.	
valid_emails	No	Fails if any value provided in a comma separated list is not a valid email.	
valid_ip	No	Fails if the supplied IP is not valid. Accepts an optional parameter of 'ipv4' or 'ipv6' to specify an IP format.	<code>valid_ip </code>
valid_url	No	Fails if field does not contain a valid URL.	
valid_date	No	Fails if field does not contain a valid date. Accepts an optional parameter to matches a date format.	<code>valid_da</code>
		Verifies that the credit card number matches the format used by the specified provider. Current supported providers are: American Express (amex), China Unionpay (unionpay), Diners Club CarteBlance (carteblanche), Diners Club (dinersclub), Discover Card (discover), Interpayment (interpayment), JCB (jcb),	

valid_cc_number	Yes	Maestro (maestro), Dankort (dankort), NSPK MIR (mir), Troy (troy), MasterCard (mastercard), Visa (visa), UATP (uatp), Verve (verve), CIBC Convenience Card (cibc), Royal Bank of Canada Client Card (rbc), TD Canada Trust Access Card (tdtrust), Scotiabank Scotia Card (scotia), BMO ABM Card (bmoabm), HSBC Canada Card (hsbc)	valid_cc.
-----------------	-----	---	-----------

Rules for File Uploads

These validation rules enable you to do the basic checks you might need to verify that uploaded files meet your business needs. Since the value of a file upload HTML field doesn't exist, and is stored in the `$_FILES` global, the name of the input field will need to be used twice. Once to specify the field name as you would for any other rule, but again as the first parameter of all file upload related rules:

```
// In the HTML
<input type="file" name="avatar">

// In the controller
$this->validate([
    'avatar' => 'uploaded[avatar]|max_size[avatar,1024]'
]);
```

Rule	Parameter	Description	Example
uploaded	Yes	Fails if the name of the parameter does not match the name of any uploaded	uploaded[field_name]

		files.	
max_size	Yes	<p>Fails if the uploaded file named in the parameter is larger than the second parameter in kilobytes (kb).</p>	max_size[field_name,2048]
max_dims	Yes	<p>Fails if the maximum width and height of an uploaded image exceed values. The first parameter is the field name. The second is the width, and the third is the height. Will also fail if the file cannot be determined to be an image.</p>	max_dims[field_name,300,150]
		Fails if the file's mime type is not	

mime_in	Yes	one listed in the parameters.	mime_in[field_name,image/png,image/
ext_in	Yes	Fails if the file's extension is not one listed in the parameters.	ext_in[field_name,png,jpg,gif]
is_image	Yes	Fails if the file cannot be determined to be an image based on the mime type.	is_image[field_name]

Note

You can also use any native PHP functions that permit up to two parameters, where at least one is required (to pass the field data).

Helpers

Helpers are collections of useful procedural functions.

- [Array Helper](#)
- [Cookie Helper](#)
- [Date Helper](#)
- [Filesystem Helper](#)
- [Form Helper](#)
- [HTML Helper](#)
- [Inflector Helper](#)
- [Number Helper](#)
- [Security Helper](#)
- [Text Helper](#)
- [URL Helper](#)
- [XML Helper](#)

Array Helper

The array helper provides several functions to simplify more complex usages of arrays. It is not intended to duplicate any of the existing functionality that PHP provides - unless it is to vastly simplify their usage.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('array');
```

[Available Functions](#)

The following functions are available:

dot_array_search(*string* \$search, *array* \$values)

- | | |
|---------------------|---|
| Parameters: | <ul style="list-style-type: none">• \$search (<i>string</i>) – The dot-notation string describing how to search the array• \$values (<i>array</i>) – The array to search |
| Returns: | The value found within the array, or null |
| Return type: | mixed |

This method allows you to use dot-notation to search through an array for a specific-key, and allows the use of a the ‘*’ wildcard. Given the following array:

```
$data = [
```

```

    'foo' => [
        'buzz' => [
            'fizz' => 11
        ],
        'bar' => [
            'baz' => 23
        ]
    ]
]

```

We can locate the value of ‘fizz’ by using the search string “foo.buzz.fizz”. Likewise, the value of baz can be found with “foo.bar.baz”:

```

// Returns: 11
$fizz = dot_array_search('foo.buzz.fizz', $data);

// Returns: 23
$baz = dot_array_search('foo.bar.baz', $data);

```

You can use the asterisk as a wildcard to replace any of the segments. When found, it will search through all of the child nodes until it finds it. This is handy if you don’t know the values, or if your values have a numeric index:

```

// Returns: 23
$baz = dot_array_search('foo.*.baz', $data);

```

Cookie Helper

The Cookie Helper file contains functions that assist in working with cookies.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('cookie');
```

[Available Functions](#)

The following functions are available:

set_cookie(\$name[, \$value = "[, \$expire = "[, \$domain = "[, \$path = '/'[, \$prefix = "[, \$secure = false[, \$httpOnly = false]]]]]])

- **\$name** (*mixed*) – Cookie name or associative array of all of the parameters available to this function
 - **\$value** (*string*) – Cookie value
 - **\$expire** (*int*) – Number of seconds until expiration
 - **\$domain** (*string*) – Cookie domain (usually: .yourdomain.com)
- Parameters:**
- **\$path** (*string*) – Cookie path
 - **\$prefix** (*string*) – Cookie name prefix
 - **\$secure** (*bool*) – Whether to only send the cookie through HTTPS
 - **\$httpOnly** (*bool*) – Whether to hide the cookie from

JavaScript

Return type: void

This helper function gives you friendlier syntax to set browser cookies. Refer to the [Response Library](#) for a description of its use, as this function is an alias for `Response::setCookie()`.

get_cookie(\$index[, \$xssClean = false])

Parameters:

- **\$index** (*string*) – Cookie name
- **\$xss_clean** (*bool*) – Whether to apply XSS filtering to the returned value

Returns: The cookie value or NULL if not found

Return type: mixed

This helper function gives you friendlier syntax to get browser cookies. Refer to the [IncomingRequest Library](#) for detailed description of its use, as this function acts very similarly to `IncomingRequest::getCookie()`, except it will also prepend the `$cookiePrefix` that you might've set in your `app/Config/App.php` file.

delete_cookie(\$name[, \$domain = "[, \$path = '[, \$prefix = "]]])

Parameters:

- **\$name** (*string*) – Cookie name
- **\$domain** (*string*) – Cookie domain (usually: `.yourdomain.com`)
- **\$path** (*string*) – Cookie path
- **\$prefix** (*string*) – Cookie name prefix

Return type: void

Lets you delete a cookie. Unless you've set a custom path or other values, only the name of the cookie is needed.

```
delete_cookie('name');
```

This function is otherwise identical to `set_cookie()`, except that it does not have the value and expiration parameters. You can submit an array of

values in the first parameter or you can set discrete parameters.

```
delete_cookie($name, $domain, $path, $prefix);
```

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Date Helper

The Date Helper file contains functions that assist in working with dates.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('date');
```

[Available Functions](#)

The following functions are available:

```
now([$timezone = NULL])
```

Parameters: • **\$timezone** (*string*) – Timezone

Returns: UNIX timestamp

Return type: int

Returns the current time as a UNIX timestamp, referenced either to your server's local time or any PHP supported timezone, based on the “time reference” setting in your config file. If you do not intend to set your master time reference to any other PHP supported timezone (which you'll typically do if you run a site that lets each user set their own timezone settings) there is no benefit to using this function over PHP's `time()` function.

```
echo now('Australia/Victoria');
```


If a timezone is not provided, it will return `time()` based on the **time_reference** setting.

Many functions previously found in the CodeIgniter 3 `date_helper` have been moved to the `Time` module in CodeIgniter 4.

Filesystem Helper

The Directory Helper file contains functions that assist in working with directories.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('filesystem');
```

[Available Functions](#)

The following functions are available:

directory_map(\$source_dir[, \$directory_depth = 0[, \$hidden = FALSE]])

- Parameters:**
- **\$source_dir** (*string*) – Path to the source directory
 - **\$directory_depth** (*int*) – Depth of directories to traverse (0 = fully recursive, 1 = current dir, etc)
 - **\$hidden** (*bool*) – Whether to include hidden directories

Returns: An array of files

Return type: array

Examples:

```
$map = directory_map('./mydirectory/');
```

Note

Paths are almost always relative to your main index.php file.

Sub-folders contained within the directory will be mapped as well. If you wish to control the recursion depth, you can do so using the second parameter (integer). A depth of 1 will only map the top level directory:

```
$map = directory_map('./mydirectory/', 1);
```

By default, hidden files will not be included in the returned array. To override this behavior, you may set a third parameter to true (boolean):

```
$map = directory_map('./mydirectory/', FALSE, TRUE);
```

Each folder name will be an array index, while its contained files will be numerically indexed. Here is an example of a typical array:

```
Array (
    [libraries] => Array
        (
            [0] => benchmark.html
            [1] => config.html
            ["database/"] => Array
                (
                    [0] => query_builder.h
                    [1] => binds.html
                    [2] => configuration.h
                    [3] => connecting.html
                    [4] => examples.html
                    [5] => fields.html
                    [6] => index.html
                    [7] => queries.html
                )
            [2] => email.html
            [3] => file_uploading.html
            [4] => image_lib.html
            [5] => input.html
            [6] => language.html
            [7] => loader.html
            [8] => pagination.html
            [9] => uri.html
        )
)
```

)



If no results are found, this will return an empty array.

write_file(\$path, \$data[, \$mode = 'wb'])

Parameters:

- **\$path** (*string*) – File path
- **\$data** (*string*) – Data to write to file
- **\$mode** (*string*) – fopen() mode

Returns: TRUE if the write was successful, FALSE in case of an error

Return type: bool

Writes data to the file specified in the path. If the file does not exist then the function will create it.

Example:

```
$data = 'Some file data';
if ( ! write_file('./path/to/file.php', $data) )
{
    echo 'Unable to write the file';
}
else
{
    echo 'File written!';
}
```

You can optionally set the write mode via the third parameter:

```
write_file('./path/to/file.php', $data, 'r+');
```

The default mode is 'wb'. Please see the [PHP user guide](http://php.net/manual/en/function.fopen.php) [http://php.net/manual/en/function.fopen.php] for mode options.

Note

In order for this function to write data to a file, its permissions must be set such that it is writable. If the file does not already exist, then the directory containing it must be writable.

Note

The path is relative to your main site index.php file, NOT your controller or view files. CodeIgniter uses a front controller so paths are always relative to the main site index.

Note

This function acquires an exclusive lock on the file while writing to it.

delete_files(*\$path*[, *\$del_dir* = FALSE[, *\$htdocs* = FALSE]])

Parameters:

- **\$path** (*string*) – Directory path
- **\$del_dir** (*bool*) – Whether to also delete directories
- **\$htdocs** (*bool*) – Whether to skip deleting .htaccess and index page files

Returns: TRUE on success, FALSE in case of an error

Return type: bool

Deletes ALL files contained in the supplied path.

Example:

```
delete_files('./path/to/directory/');
```

If the second parameter is set to TRUE, any directories contained within the supplied root path will be deleted as well.

Example:

```
delete_files('./path/to/directory/', TRUE);
```

Note

The files must be writable or owned by the system in order to be deleted.

get_filenames(\$source_dir[, \$include_path = FALSE])

Parameters:

- **\$source_dir** (*string*) – Directory path
- **\$include_path** (*bool*) – Whether to include the path as part of the filenames

Returns: An array of file names

Return type: array

Takes a server path as input and returns an array containing the names of all files contained within it. The file path can optionally be added to the file names by setting the second parameter to TRUE.

Example:

```
$controllers = get_filenames(APPPATH.'controllers/');
```

get_dir_file_info(\$source_dir, \$stop_level_only)

Parameters:

- **\$source_dir** (*string*) – Directory path
- **\$stop_level_only** (*bool*) – Whether to look only at the specified directory (excluding sub-directories)

Returns: An array containing info on the supplied directory's contents

Return type: array

Reads the specified directory and builds an array containing the filenames, filesize, dates, and permissions. Sub-folders contained within the specified path are only read if forced by sending the second parameter to FALSE, as this can be an intensive operation.

Example:

```
$models_info = get_dir_file_info(APPPATH.'models/');
```

get_file_info(\$file[, \$returned_values = ['name', 'server_path', 'size', 'date']])

Parameters:

- **\$file** (*string*) – File path
- **\$returned_values** (*array*) – What type of info to

return

Returns: An array containing info on the specified file or FALSE on failure

Return type: array

Given a file and path, returns (optionally) the *name*, *path*, *size* and *date modified* information attributes for a file. Second parameter allows you to explicitly declare what information you want returned.

Valid \$returned_values options are: *name*, *size*, *date*, *readable*, *writable*, *executable* and *fileperms*.

symbolic_permissions(\$perms)

Parameters: • **\$perms** (*int*) – Permissions

Returns: Symbolic permissions string

Return type: string

Takes numeric permissions (such as is returned by `fileperms()`) and returns standard symbolic notation of file permissions.

```
echo symbolic_permissions(fileperms('./index.php')); // -rw-r
```



octal_permissions(\$perms)

Parameters: • **\$perms** (*int*) – Permissions

Returns: Octal permissions string

Return type: string

Takes numeric permissions (such as is returned by `fileperms()`) and returns a three character octal notation of file permissions.

```
echo octal_permissions(fileperms('./index.php')); // 644
```

set_realpath(\$path[, \$check_existance = FALSE])

• **\$path** (*string*) – Path

Parameters: • **\$check_existance** (*bool*) – Whether to check if the path actually exists

Returns: An absolute path

**Return
type:** string

This function will return a server path without symbolic links or relative directory structures. An optional second argument will cause an error to be triggered if the path cannot be resolved.

Examples:

```
$file = '/etc/php5/apache2/php.ini';
echo set_realpath($file); // Prints '/etc/php5/apache2/php.ini'

$non_existent_file = '/path/to/non-exist-file.txt';
echo set_realpath($non_existent_file, TRUE); // Shows an er
echo set_realpath($non_existent_file, FALSE); // Prints '/pa

$directory = '/etc/php5';
echo set_realpath($directory); // Prints '/etc/php5/'

$non_existent_directory = '/path/to/nowhere';
echo set_realpath($non_existent_directory, TRUE); // Sho
echo set_realpath($non_existent_directory, FALSE); // Pri
```


Form Helper

The Form Helper file contains functions that assist in working with forms.

- [Loading this Helper](#)
- [Escaping field values](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('form');
```

[Escaping field values](#)

You may need to use HTML and characters such as quotes within your form elements. In order to do that safely, you'll need to use [common function](#) `esc()`.

Consider the following example:

```
$string = 'Here is a string containing "quoted" text.';

<input type="text" name="myfield" value="<?= $string; ?>" />
```

Since the above string contains a set of quotes, it will cause the form to break. The [esc\(\)](#) function converts HTML special characters so that it can be used safely:

```
<input type="text" name="myfield" value="<?= esc($string); ?>" />
```

Note

If you use any of the form helper functions listed on this page, and you pass values as an associative array, the form values will be automatically escaped, so there is no need to call this function. Use it only if you are creating your own form elements, which you would pass as strings.

Available Functions

The following functions are available:

form_open([*\$action* = "[", *\$attributes* = "[", *\$hidden* = []]])

Parameters:

- **\$action** (*string*) – Form action/target URI string
- **\$attributes** (*mixed*) – HTML attributes, as an array or escaped string
- **\$hidden** (*array*) – An array of hidden fields' definitions

Returns: An HTML form opening tag

Return type: string

Creates an opening form tag with a base URL **built from your config preferences**. It will optionally let you add form attributes and hidden input fields, and will always add the *accept-charset* attribute based on the charset value in your config file.

The main benefit of using this tag rather than hard coding your own HTML is that it permits your site to be more portable in the event your URLs ever change.

Here's a simple example:

```
echo form_open('email/send');
```

The above example would create a form that points to your base URL plus the “email/send” URI segments, like this:

```
<form method="post" accept-charset="utf-8" action="http://exam
```

< >

Adding Attributes

Attributes can be added by passing an associative array to the second parameter, like this:

```
$attributes = ['class' => 'email', 'id' => 'myform'];  
echo form_open('email/send', $attributes);
```

Alternatively, you can specify the second parameter as a string:

```
echo form_open('email/send', 'class="email" id="myform"');
```

The above examples would create a form similar to this:

```
<form method="post" accept-charset="utf-8" action="http://
```

If CSRF filter is turned on *form_open()* will generate CSRF field at the beginning of the form. You can specify ID of this field by passing *csrf_id* as one of the \$attribute array:

```
form_open('/u/sign-up', ['csrf_id' => 'my-id']);
```

will return:

```
<form action="/u/sign-up" method="post" accept-charset="utf-8"> <input type="hidden" id="my-id" name="csrf_field" value="964ede6e0ae8a680f7b8eab69136717d" />
```

Adding Hidden Input Fields

Hidden fields can be added by passing an associative array to the third parameter, like this:

```
$hidden = ['username' => 'Joe', 'member_id' => '234'];  
echo form_open('email/send', '', $hidden);
```

You can skip the second parameter by passing any false value to it.

The above example would create a form similar to this:

```
<form method="post" accept-charset="utf-8" action="http://
    <input type="hidden" name="username" value="Joe" /
    <input type="hidden" name="member_id" value="234"
< >
```

form_open_multipart([\$action = "[", \$attributes = "[", \$hidden = []]))

- **\$action** (*string*) – Form action/target URI string
- **\$attributes** (*mixed*) – HTML attributes, as an array or escaped string
- **\$hidden** (*array*) – An array of hidden fields' definitions

Returns: An HTML multipart form opening tag

Return type: string

This function is identical to [form_open\(\)](#) above, except that it adds a *multipart* attribute, which is necessary if you would like to use the form to upload files with.

form_hidden(\$name[, \$value = "])

- **\$name** (*string*) – Field name
- **\$value** (*string*) – Field value

Returns: An HTML hidden input field tag

Return type: string

Lets you generate hidden input fields. You can either submit a name/value string to create one field:

```
form_hidden('username', 'johndoe');
// Would produce: <input type="hidden" name="username" value="
< >
```

... or you can submit an associative array to create multiple fields:

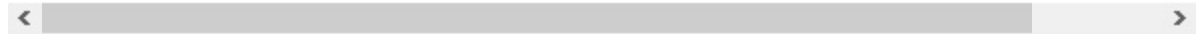
```
$data = [
    'name' => 'John Doe',
    'email' => 'john@example.com',
    'url' => 'http://example.com'
];

echo form_hidden($data);
```

```

/*
    Would produce:
    <input type="hidden" name="name" value="John Doe" />
    <input type="hidden" name="email" value="john@example."
    <input type="hidden" name="url" value="http://example.
*/

```



You can also pass an associative array to the value field:

```

$data = [
    'name' => 'John Doe',
    'email' => 'john@example.com',
    'url' => 'http://example.com'
];

```

```


echo form_hidden('my_array', $data);

```

```

/*
    Would produce:
    <input type="hidden" name="my_array[name]" value="John
    <input type="hidden" name="my_array[email]" value="joh
    <input type="hidden" name="my_array[url]" value="http:
*/

```



If you want to create hidden input fields with extra attributes:

```

$data = [
    'type' => 'hidden',
    'name' => 'email',
    'id' => 'hiddenemail',
    'value' => 'john@example.com',
    'class' => 'hiddenemail'
];

```

```


echo form_input($data);

```

```

/*
    Would produce:
    <input type="hidden" name="email" value="john@example.
*/

```



```
form_input([$data = "[, $value = "[, $extra = "[, $type = 'text']]]])
```

Parameters:

- **\$data** (*array*) – Field attributes data
- **\$value** (*string*) – Field value
- **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string
- **\$type** (*string*) – The type of input field. i.e. 'text', 'email', 'number', etc.

Returns: An HTML text input field tag

Return type: string

Lets you generate a standard text input field. You can minimally pass the field name and value in the first and second parameter:

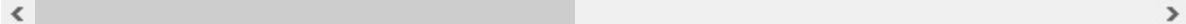
```
echo form_input('username', 'johndoe');
```

Or you can pass an associative array containing any data you wish your form to contain:

```
$data = [  
    'name'      => 'username',  
    'id'        => 'username',  
    'value'     => 'johndoe',  
    'maxlength' => '100',  
    'size'      => '50',  
    'style'     => 'width:50%'  
];
```

```
echo form_input($data);
```

```
/*  
    Would produce:  
  
    <input type="text" name="username" value="johndoe" id=  
*/
```



If you would like your form to contain some additional data, like JavaScript, you can pass it as a string in the third parameter:

```
$js = 'onClick="some_function()";'  
echo form_input('username', 'johndoe', $js);
```

Or you can pass it as an array:

```
$js = ['onClick' => 'some_function();'];  
echo form_input('username', 'johndoe', $js);
```

To support the expanded range of HTML5 input fields, you can pass an input type in as the fourth parameter:

```
echo form_input('email', 'joe@example.com', ['placeholder' =>  
/*  
    would produce:  
  
    <input type="email" name="email" value="joe@example.co  
*/  
< >
```

form_password(*[\$data = "[, \$value = "[, \$extra = "]]]*)

Parameters:

- **\$data** (*array*) – Field attributes data
- **\$value** (*string*) – Field value
- **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string

Returns: An HTML password input field tag

Return type: string

This function is identical in all respects to the [form_input\(\)](#) function above except that it uses the “password” input type.

form_upload(*[\$data = "[, \$value = "[, \$extra = "]]]*)

Parameters:

- **\$data** (*array*) – Field attributes data
- **\$value** (*string*) – Field value
- **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string

Returns: An HTML file upload input field tag

Return type: string

This function is identical in all respects to the [form_input\(\)](#) function above except that it uses the “file” input type, allowing it to be used to

upload files.

form_textarea([\$data = "[", \$value = "[", \$extra = "]]])

- Parameters:**
- **\$data** (*array*) – Field attributes data
 - **\$value** (*string*) – Field value
 - **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string
- Returns:** An HTML textarea tag
- Return type:** string

This function is identical in all respects to the [form_input\(\)](#) function above except that it generates a “textarea” type.

Note

Instead of the *maxlength* and *size* attributes in the above example, you will instead specify *rows* and *cols*.

form_dropdown([\$name = "[", \$options = [], \$selected = [], \$extra = "]]]])

- **\$name** (*string*) – Field name
 - **\$options** (*array*) – An associative array of options to be listed
- Parameters:**
- **\$selected** (*array*) – List of fields to mark with the *selected* attribute
 - **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string
- Returns:** An HTML dropdown select field tag
- Return type:** string

Lets you create a standard drop-down field. The first parameter will contain the name of the field, the second parameter will contain an associative array of options, and the third parameter will contain the value you wish to be selected. You can also pass an array of multiple items

through the third parameter, and the helper will create a multiple select for you.

Example:

```
$options = [
    'small' => 'Small Shirt',
    'med'   => 'Medium Shirt',
    'large' => 'Large Shirt',
    'xlarge' => 'Extra Large Shirt',
];

$shirts_on_sale = ['small', 'large'];
echo form_dropdown('shirts', $options, 'large');

/*
    Would produce:

    <select name="shirts">
        <option value="small">Small Shirt</option>
        <option value="med">Medium Shirt</option>
        <option value="large" selected="selected">Larg
        <option value="xlarge">Extra Large Shirt</opti
    </select>
*/

echo form_dropdown('shirts', $options, $shirts_on_sale);

/*
    Would produce:

    <select name="shirts" multiple="multiple">
        <option value="small" selected="selected">Smal
        <option value="med">Medium Shirt</option>
        <option value="large" selected="selected">Larg
        <option value="xlarge">Extra Large Shirt</opti
    </select>
*/
```



If you would like the opening `<select>` to contain additional data, like an id attribute or JavaScript, you can pass it as a string in the fourth parameter:

```
$js = 'id="shirts" onChange="some_function();"';
echo form_dropdown('shirts', $options, 'large', $js);
```

Or you can pass it as an array:

```
$js = [
    'id' => 'shirts',
    'onChange' => 'some_function()';
];
echo form_dropdown('shirts', $options, 'large', $js);
```

If the array passed as `$options` is a multidimensional array, then `form_dropdown()` will produce an `<optgroup>` with the array key as the label.

form_multiselect(`[$name = "", $options = [], $selected = [], $extra = ""]`)

- **\$name** (*string*) – Field name
 - **\$options** (*array*) – An associative array of options to be listed
- Parameters:**
- **\$selected** (*array*) – List of fields to mark with the *selected* attribute
 - **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string
- Returns:** An HTML dropdown multiselect field tag
- Return type:** string

Lets you create a standard multiselect field. The first parameter will contain the name of the field, the second parameter will contain an associative array of options, and the third parameter will contain the value or values you wish to be selected.

The parameter usage is identical to using [form_dropdown\(\)](#) above, except of course that the name of the field will need to use POST array syntax, e.g. `foo[]`.

form_fieldset(`[$legend_text = "", $attributes = []]`)

- **\$legend_text** (*string*) – Text to put in the `<legend>` tag
- Parameters:**
- **\$attributes** (*array*) – Attributes to be set on the `<fieldset>` tag

Returns: An HTML fieldset opening tag

Return type: string

Lets you generate fieldset/legend fields.

Example:

```
echo form_fieldset('Address Information');  
echo "<p>fieldset content here</p>\n";  
echo form_fieldset_close();
```

```
/*  
    Produces:  
  
        <fieldset>  
            <legend>Address Information</legend>  
            <p>form content here</p>  
        </fieldset>  
*/
```

Similar to other functions, you can submit an associative array in the second parameter if you prefer to set additional attributes:

```
$attributes = [  
    'id' => 'address_info',  
    'class' => 'address_info'  
];  
  
echo form_fieldset('Address Information', $attributes);  
echo "<p>fieldset content here</p>\n";  
echo form_fieldset_close();  
  
/*  
    Produces:  
  
        <fieldset id="address_info" class="address_info">  
            <legend>Address Information</legend>  
            <p>form content here</p>  
        </fieldset>  
*/
```

form_fieldset_close([*\$extra* = "])

- **\$extra** (*string*) – Anything to append after the closing

Parameters: tag, *as is*

Returns: An HTML fieldset closing tag

Return type: string

Produces a closing `</fieldset>` tag. The only advantage to using this function is it permits you to pass data to it which will be added below the tag. For example

```
$string = '</div></div>';  
echo form_fieldset_close($string);  
// Would produce: </fieldset></div></div>
```

form_checkbox(`[$data = "[, $value = "[, $checked = FALSE[, $extra = "]]])`

Parameters:

- **\$data** (*array*) – Field attributes data
- **\$value** (*string*) – Field value
- **\$checked** (*bool*) – Whether to mark the checkbox as being *checked*
- **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string

Returns: An HTML checkbox input tag

Return type: string

Lets you generate a checkbox field. Simple example:

```
echo form_checkbox('newsletter', 'accept', TRUE);  
// Would produce: <input type="checkbox" name="newsletter" va  
< >
```

The third parameter contains a boolean TRUE/FALSE to determine whether the box should be checked or not.

Similar to the other form functions in this helper, you can also pass an array of attributes to the function:

```
$data = [  
    'name' => 'newsletter',
```

```

        'id'      => 'newsletter',
        'value'   => 'accept',
        'checked' => TRUE,
        'style'   => 'margin:10px'
    ];

    echo form_checkbox($data);
    // Would produce: <input type="checkbox" name="newsletter" id=
    <

```

Also as with other functions, if you would like the tag to contain additional data like JavaScript, you can pass it as a string in the fourth parameter:

```

$js = 'onClick="some_function()";';
echo form_checkbox('newsletter', 'accept', TRUE, $js);

```

Or you can pass it as an array:

```

$js = ['onClick' => 'some_function();'];
echo form_checkbox('newsletter', 'accept', TRUE, $js);

```

```

form_radio([$data = "[, $value = "[, $checked = FALSE[, $extra =
"]]]])

```

- Parameters:**
- **\$data** (*array*) – Field attributes data
 - **\$value** (*string*) – Field value
 - **\$checked** (*bool*) – Whether to mark the radio button as being *checked*
 - **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string
- Returns:** An HTML radio input tag
- Return type:** string

This function is identical in all respects to the [form_checkbox\(\)](#) function above except that it uses the “radio” input type.

```

form_label([$label_text = "[, $id = "[, $attributes = []]])

```

- Parameters:**
- **\$label_text** (*string*) – Text to put in the <label> tag
 - **\$id** (*string*) – ID of the form element that we’re

making a label for

- **\$attributes** (*string*) – HTML attributes

Returns: An HTML field label tag

Return type: string

Lets you generate a <label>. Simple example:

```
echo form_label('What is your Name', 'username');  
// Would produce: <label for="username">What is your Name</la  
< >
```

Similar to other functions, you can submit an associative array in the third parameter if you prefer to set additional attributes.

Example:

```
$attributes = [  
    'class' => 'mycustomclass',  
    'style' => 'color: #000;',  
];  
  
echo form_label('What is your Name', 'username', $attributes);  
// Would produce: <label for="username" class="mycustomclass"  
< >
```

form_submit([*\$data* = "[, \$value = "[, \$extra = "]]])

Parameters:

- **\$data** (*string*) – Button name
- **\$value** (*string*) – Button value
- **\$extra** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string

Returns: An HTML input submit tag

Return type: string

Lets you generate a standard submit button. Simple example:

```
echo form_submit('mysubmit', 'Submit Post!');  
// Would produce: <input type="submit" name="mysubmit" value=  
< >
```

Similar to other functions, you can submit an associative array in the first

parameter if you prefer to set your own attributes. The third parameter lets you add extra data to your form, like JavaScript.

```
form_reset([$data = "[, $value = "[, $extra = "]]])
```

Parameters:

- ***\$data*** (*string*) – Button name
- ***\$value*** (*string*) – Button value
- ***\$extra*** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string

Returns: An HTML input reset button tag

Return type: string

Lets you generate a standard reset button. Use is identical to **form_submit()**.

```
form_button([$data = "[, $content = "[, $extra = "]]])
```

Parameters:

- ***\$data*** (*string*) – Button name
- ***\$content*** (*string*) – Button label
- ***\$extra*** (*mixed*) – Extra attributes to be added to the tag either as an array or a literal string

Returns: An HTML button tag

Return type: string

Lets you generate a standard button element. You can minimally pass the button name and content in the first and second parameter:

```
echo form_button('name', 'content');  
// Would produce: <button name="name" type="button">Content</b>
```

Or you can pass an associative array containing any data you wish your form to contain:

```
$data = [  
    'name'    => 'button',  
    'id'      => 'button',  
    'value'   => 'true',  
    'type'    => 'reset',
```

```

        'content' => 'Reset'
    ];

    echo form_button($data);
    // Would produce: <button name="button" id="button" value="tru
    <
    >

```

If you would like your form to contain some additional data, like JavaScript, you can pass it as a string in the third parameter:

```

$js = 'onClick="some_function()"';
echo form_button('mybutton', 'Click Me', $js);

```

form_close(*[\$extra = "]*)

Parameters:

- **\$extra** (*string*) – Anything to append after the closing tag, as is

Returns: An HTML form closing tag

Return type: string

Produces a closing `</form>` tag. The only advantage to using this function is it permits you to pass data to it which will be added below the tag. For example:

```

$string = '</div></div>';
echo form_close($string);
// Would produce: </form> </div></div>

```

set_value(*\$field[, \$default = "[, \$html_escape = TRUE]*)

Parameters:

- **\$field** (*string*) – Field name
- **\$default** (*string*) – Default value
- **\$html_escape** (*bool*) – Whether to turn off HTML escaping of the value

Returns: Field value

Return type: string

Permits you to set the value of an input form or textarea. You must supply the field name via the first parameter of the function. The second (optional) parameter allows you to set a default value for the form. The

third (optional) parameter allows you to turn off HTML escaping of the value, in case you need to use this function in combination with i.e. [form_input\(\)](#) and avoid double-escaping.

Example:

```
<input type="text" name="quantity" value="<?php echo set_value
< >
```

The above form will show “0” when loaded for the first time.

set_select(\$field[, \$value = "[, \$default = FALSE]]])

Parameters:

- **\$field** (*string*) – Field name
- **\$value** (*string*) – Value to check for
- **\$default** (*string*) – Whether the value is also a default one

Returns: ‘selected’ attribute or an empty string

Return type: string

If you use a <select> menu, this function permits you to display the menu item that was selected.

The first parameter must contain the name of the select menu, the second parameter must contain the value of each item, and the third (optional) parameter lets you set an item as the default (use boolean TRUE/FALSE).

Example:

```
<select name="myselect">
  <option value="one" <?php echo set_select('myselect',
  <option value="two" <?php echo set_select('myselect',
  <option value="three" <?php echo set_select('myselect
</select>
< >
```

set_checkbox(\$field[, \$value = "[, \$default = FALSE]]])

- **\$field** (*string*) – Field name
- **\$value** (*string*) – Value to check for

Parameters: • **\$default** (*string*) – Whether the value is also a default one

Returns: 'checked' attribute or an empty string

Return type: string

Permits you to display a checkbox in the state it was submitted.

The first parameter must contain the name of the checkbox, the second parameter must contain its value, and the third (optional) parameter lets you set an item as the default (use boolean TRUE/FALSE).

Example:

```
<input type="checkbox" name="mycheck" value="1" <?php echo set  
<input type="checkbox" name="mycheck" value="2" <?php echo set
```

set_radio(\$field[, \$value = "[, \$default = FALSE]])

Parameters:

- **\$field** (*string*) – Field name
- **\$value** (*string*) – Value to check for
- **\$default** (*string*) – Whether the value is also a default one

Returns: 'checked' attribute or an empty string

Return type: string

Permits you to display radio buttons in the state they were submitted. This function is identical to the [set_checkbox\(\)](#) function above.

Example:

```
<input type="radio" name="myradio" value="1" <?php echo set_r  
<input type="radio" name="myradio" value="2" <?php echo set_r
```

Note

If you are using the Form Validation class, you must always specify a

rule for your field, even if empty, in order for the `set_*` () functions to work. This is because if a Form Validation object is defined, the control for `set_*` () is handed over to a method of the class instead of the generic helper function.

HTML Helper

The HTML Helper file contains functions that assist in working with HTML.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('html');
```

[Available Functions](#)

The following functions are available:

```
img([$src = "[, $indexPage = false[, $attributes = "]])
```

- | | |
|---------------------|---|
| Parameters: | <ul style="list-style-type: none">• \$src (<i>mixed</i>) – Image source data• \$indexPage (<i>bool</i>) – Whether to treat \$src as a routed URI string• \$attributes (<i>mixed</i>) – HTML attributes |
| Returns: | HTML image tag |
| Return type: | string |

Lets you create HTML tags. The first parameter contains the image source. Example:

```
echo img('images/picture.jpg');  
// 
```

There is an optional second parameter that is a true/false value that specifies if the *src* should have the page specified by `$config['indexPage']` added to the address it creates. Presumably, this would be if you were using a media controller:

```
echo img('images/picture.jpg', true);  
// 
```

Additionally, an associative array can be passed as the first parameter, for complete control over all attributes and values. If an *alt* attribute is not provided, CodeIgniter will generate an empty string.

Example:

```
$imageProperties = [  
    'src'      => 'images/picture.jpg',  
    'alt'      => 'Me, demonstrating how to eat 4 slices of pizz  
    'class'    => 'post_images',  
    'width'    => '200',  
    'height'   => '200',  
    'title'    => 'That was quite a night',  
    'rel'      => 'lightbox'  
];  
  
img($imageProperties);  
// 
```

link_tag(`[$href = "", $rel = 'stylesheet', $type = 'text/css', $title = "", $media = "", $indexPage = false]`)

- **\$href** (*string*) – The source of the link file
- **\$rel** (*string*) – Relation type
- **\$type** (*string*) – Type of the related document

Parameters:

- **\$title** (*string*) – Link title
- **\$media** (*string*) – Media type
- **\$indexPage** (*bool*) – Whether to treat *\$src* as a routed URI string

Returns: HTML link tag

Return

type: string

Lets you create HTML `<link />` tags. This is useful for stylesheet links, as well as other links. The parameters are *href*, with optional *rel*, *type*, *title*, *media* and *indexPath*.

indexPath is a boolean value that specifies if the *href* should have the page specified by `$config['indexPath']` added to the address it creates.

Example:

```
echo link_tag('css/mystyles.css');  
// <link href="http://site.com/css/mystyles.css" rel="styleshe  
< >
```

Further examples:

```
echo link_tag('favicon.ico', 'shortcut icon', 'image/ico');  
// <link href="http://site.com/favicon.ico" rel="shortcut icon  
  
echo link_tag('feed', 'alternate', 'application/rss+xml', 'My  
// <link href="http://site.com/feed" rel="alternate" type="app  
< >
```

Alternately, an associative array can be passed to the `link_tag()` function for complete control over all attributes and values:

```
$link = [  
    'href' => 'css/printer.css',  
    'rel'   => 'stylesheet',  
    'type'  => 'text/css',  
    'media' => 'print'  
];  
  
echo link_tag($link);  
// <link href="http://site.com/css/printer.css" rel="styleshee  
< >
```

script_tag(`[$src = "[, $indexPath = false]]`)

- Parameters:**
- **\$src** (*mixed*) – The source name of a JavaScript file
 - **\$indexPath** (*bool*) – Whether to treat \$src as a routed URI string

Returns: HTML script tag

Return type: string

Lets you create HTML `<script></script>` tags. The parameters is *src*, with optional *indexPage*.

indexPage is a boolean value that specifies if the *src* should have the page specified by `$config['indexPage']` added to the address it creates.

Example:

```
echo script_tag('js/mystyles.js');  
// <script src="http://site.com/js/mystyles.js" type="text/jav  
< >
```

Alternately, an associative array can be passed to the `script_tag()` function for complete control over all attributes and values:

```
$script = ['src' => 'js/printer.js'];  
echo script_tag($script);  
// <script src="http://site.com/js/printer.js" type="text/java  
< >
```

`ul($list[, $attributes = ""])`

Parameters:

- **\$list** (*array*) – List entries
- **\$attributes** (*array*) – HTML attributes

Returns: HTML-formatted unordered list

Return type: string

Permits you to generate unordered HTML lists from simple or multi-dimensional arrays. Example:

```
$list = [  
    'red',  
    'blue',  
    'green',  
    'yellow'  
];  
  
$attributes = [  

```

```

        'class' => 'boldlist',
        'id'     => 'mylist'
    ];

    echo ul($list, $attributes);

```

The above code will produce this:

```

<ul class="boldlist" id="mylist">
  <li>red</li>
  <li>blue</li>
  <li>green</li>
  <li>yellow</li>
</ul>

```

Here is a more complex example, using a multi-dimensional array:

```

$attributes = [
    'class' => 'boldlist',
    'id'     => 'mylist'
];

$list = [
    'colors' => [
        'red',
        'blue',
        'green'
    ],
    'shapes' => [
        'round',
        'square',
        'circles' => [
            'ellipse',
            'oval',
            'sphere'
        ]
    ],
    'moods' => [
        'happy',
        'upset' => [
            'defeated' => [
                'dejected',
                'disheartened',
                'depressed'
            ],
            'annoyed',
            'cross',
        ]
    ]
];

```



```

        'angry'
    ]
];

echo ul($list, $attributes);

```

The above code will produce this:

```

<ul class="boldlist" id="mylist">
  <li>colors
    <ul>
      <li>red</li>
      <li>blue</li>
      <li>green</li>
    </ul>
  </li>
  <li>shapes
    <ul>
      <li>round</li>
      <li>square</li>
      <li>circles
        <ul>
          <li>ellipse</li>
          <li>oval</li>
          <li>sphere</li>
        </ul>
      </li>
    </ul>
  </li>
  <li>moods
    <ul>
      <li>happy</li>
      <li>upset
        <ul>
          <li>defeated
            <ul>
              <li>dejected</li>
              <li>disheartened</li>
              <li>depressed</li>
            </ul>
          </li>
          <li>annoyed</li>
          <li>cross</li>
          <li>angry</li>
        </ul>
      </li>
    </ul>
  </li>

```

```

        </ul>
    </li>
</ul>

```

ol(\$list, \$attributes = "")

Parameters:

- **\$list** (*array*) – List entries
- **\$attributes** (*array*) – HTML attributes

Returns: HTML-formatted ordered list

Return type: string

Identical to [ul\(\)](#), only it produces the tag for ordered lists instead of .

video(\$src[, \$unsupportedMessage = "[, \$attributes = "[, \$tracks = [], \$indexPage = false]]])

Parameters:

- **\$src** (*mixed*) – Either a source string or an array of sources. See [source\(\)](#) function
- **\$unsupportedMessage** (*string*) – The message to display if the media tag is not supported by the browser
- **\$attributes** (*string*) – HTML attributes
- **\$tracks** (*array*) – Use the track function inside an array. See [track\(\)](#) function
- **\$indexPage** (*bool*) –

Returns: HTML-formatted video element

Return type: string

Permits you to generate HTML video element from simple or source arrays. Example:

```

$tracks =
[
    track('subtitles_no.vtt', 'subtitles', 'no', 'Norwegian No
    track('subtitles_yes.vtt', 'subtitles', 'yes', 'Norwegian
];

```

```
echo video('test.mp4', 'Your browser does not support the vide
```

```

echo video
(
    'http://www.codeigniter.com/test.mp4',
    'Your browser does not support the video tag.',
    'controls',
    $tracks
);

echo video
(
    [
        source('movie.mp4', 'video/mp4', 'class="test"'),
        source('movie.ogg', 'video/ogg'),
        source('movie.mov', 'video/quicktime'),
        source('movie.ogv', 'video/ogv; codecs=dirac, speex')
    ],
    'Your browser does not support the video tag.',
    'class="test" controls',
    $tracks
);

```

The above code will produce this:

```

<video src="test.mp4" controls>
  Your browser does not support the video tag.
</video>

<video src="http://www.codeigniter.com/test.mp4" controls>
  <track src="subtitles_no.vtt" kind="subtitles" srclang="no"
  <track src="subtitles_yes.vtt" kind="subtitles" srclang="yes"
  Your browser does not support the video tag.
</video>

<video class="test" controls>
  <source src="movie.mp4" type="video/mp4" class="test" />
  <source src="movie.ogg" type="video/ogg" />
  <source src="movie.mov" type="video/quicktime" />
  <source src="movie.ogv" type="video/ogv; codecs=dirac, speex" />
  <track src="subtitles_no.vtt" kind="subtitles" srclang="no"
  <track src="subtitles_yes.vtt" kind="subtitles" srclang="yes"
  Your browser does not support the video tag.
</video>

```

```

audio($src[, $unsupportedMessage = "[, $attributes = "[, $tracks = [][,

```

`$indexPath = false]]]])`

- **\$src** (*mixed*) – Either a source string or an array of sources. See [source\(\)](#) function
 - **\$unsupportedMessage** (*string*) – The message to display if the media tag is not supported by the browser
- Parameters:**
- **\$attributes** (*string*) –
 - **\$tracks** (*array*) – Use the track function inside an array. See [track\(\)](#) function
 - **\$indexPath** (*bool*) –
- Returns:** HTML-formatted audio element
- Return type:** string

Identical to [video\(\)](#), only it produces the <audio> tag instead of <video>.

`source($src = "[, $type = false[, $attributes = "])`

- Parameters:**
- **\$src** (*string*) – The path of the media resource
 - **\$type** (*bool*) – The MIME-type of the resource with optional codecs parameters
 - **\$attributes** (*array*) – HTML attributes
- Returns:** HTML source tag
- Return type:** string

Lets you create HTML <source /> tags. The first parameter contains the source source. Example:

```
echo source('movie.mp4', 'video/mp4', 'class="test"');  
// <source src="movie.mp4" type="video/mp4" class="test" />
```

`embed($src = "[, $type = false[, $attributes = "[, $indexPath = false]]]])`

- Parameters:**
- **\$src** (*string*) – The path of the resource to embed
 - **\$type** (*bool*) – MIME-type
 - **\$attributes** (*array*) – HTML attributes
 - **\$indexPath** (*bool*) –
- Returns:** HTML embed tag

Return type: string

Lets you create HTML <embed /> tags. The first parameter contains the embed source. Example:

```
echo embed('movie.mov', 'video/quicktime', 'class="test"');  
// <embed src="movie.mov" type="video/quicktime" class="test"/>
```

object(\$data = "[, \$type = false[, \$attributes = "]])

- **\$data** (*string*) – A resource URL
- **\$type** (*bool*) – Content-type of the resource
- Parameters:** • **\$attributes** (*array*) – HTML attributes
- **\$params** (*array*) – Use the param function inside an array. See [param\(\)](#) function

Returns: HTML object tag

Return type: string

Lets you create HTML <object /> tags. The first parameter contains the object data. Example:

```
echo object('movie.swf', 'application/x-shockwave-flash', 'cla  
echo object  
(  
    'movie.swf',  
    'application/x-shockwave-flash',  
    'class="test"',  
    [  
        param('foo', 'bar', 'ref', 'class="test"'),  
        param('hello', 'world', 'ref', 'class="test"')  
    ]  
);
```

The above code will produce this:

```
<object data="movie.swf" class="test"></object>
```

```
<object data="movie.swf" class="test">  
  <param name="foo" type="ref" value="bar" class="test" />  
  <param name="hello" type="ref" value="world" class="test" />
```

</object>

param(\$name = "[, \$type = false[, \$attributes = "])

Parameters:

- **\$name** (*string*) – The name of the parameter
- **\$value** (*string*) – The value of the parameter
- **\$attributes** (*array*) – HTML attributes

Returns: HTML param tag

Return type: string

Lets you create HTML <param /> tags. The first parameter contains the param source. Example:

```
echo param('movie.mov', 'video/quicktime', 'class="test"');  
// <param src="movie.mov" type="video/quicktime" class="test"/
```

track(\$name = "[, \$type = false[, \$attributes = "])

Parameters:

- **\$name** (*string*) – The name of the parameter
- **\$value** (*string*) – The value of the parameter
- **\$attributes** (*array*) – HTML attributes

Returns: HTML track tag

Return type: string

Generates a track element to specify timed tracks. The tracks are formatted in WebVTT format. Example:

```
echo track('subtitles_no.vtt', 'subtitles', 'no', 'Norwegian N  
// <track src="subtitles_no.vtt" kind="subtitles" srclang="no"
```

doctype([\$type = 'html5'])

Parameters:

- **\$type** (*string*) – Doctype name

Returns: HTML DocType tag

Return type: string

Helps you generate document type declarations, or DTD's. HTML 5 is used by default, but many doctypes are available.

Example:

```
echo doctype();  
// <!DOCTYPE html>
```

```
echo doctype('html4-trans');  
// <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://w  
< >
```

The following is a list of the pre-defined doctype choices. These are configurable, pulled from *application/Config/DocTypes.php*, or they could be over-ridden in your *.env* configuration.

Document type	Option	Result
XHTML 1.1	xhtml11	<!DOCTYPE html PUBLIC "-//W3 1.1//EN" " http://www.w3.org/TR/xhtml11/D
XHTML 1.0 Strict	xhtml1- strict	<!DOCTYPE html PUBLIC "-//W3 Strict//EN" " http://www.w3.org/TR/strict.dtd ">
XHTML 1.0 Transitional	xhtml1- trans	<!DOCTYPE html PUBLIC "-//W3 Transitional//EN" " http://www.w3.org/TR/xhtml1/DItransitional.dtd ">
XHTML 1.0 Frameset	xhtml1- frame	<!DOCTYPE html PUBLIC "-//W3 Frameset//EN" " http://www.w3.org/TR/xhtml1/DIframeset.dtd ">
XHTML Basic 1.1	xhtml- basic11	<!DOCTYPE html PUBLIC "-//W3 Basic 1.1//EN" " http://www.w3.org/basic/xhtml-basic11.dtd ">
HTML 5	html5	<!DOCTYPE html>
HTML 4 Strict	html4- strict	<!DOCTYPE HTML PUBLIC "-// 4.01//EN" " http://www.w3.org/TR/
HTML 4 Transitional	html4- trans	<!DOCTYPE HTML PUBLIC "-// 4.01 Transitional//EN" " http://www.w3.org/TR/html4/loos
		<!DOCTYPE HTML PUBLIC "-//

HTML 4 Frameset	html4-frame	4.01 Frameset//EN” “ http://www.w3.org/TR/html4/frameset.dtd ”>
MathML 1.01	mathml1	<!DOCTYPE math SYSTEM “ http://www.w3.org/Math/DTD/mathml1.dtd ”>
MathML 2.0	mathml2	<!DOCTYPE math PUBLIC “-//W3C 2.0//EN” “ http://www.w3.org/Math/DTD/mathml2.dtd ”>
SVG 1.0	svg10	<!DOCTYPE svg PUBLIC “-//W3C 1.0//EN” “ http://www.w3.org/TR/2001/09/20010904/DTD/svg10.dtd ”>
SVG 1.1 Full	svg11	<!DOCTYPE svg PUBLIC “-//W3C 1.1//EN” “ http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd ”>
SVG 1.1 Basic	svg11-basic	<!DOCTYPE svg PUBLIC “-//W3C Basic//EN” “ http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-basic.dtd ”>
SVG 1.1 Tiny	svg11-tiny	<!DOCTYPE svg PUBLIC “-//W3C Tiny//EN” “ http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-tiny.dtd ”>
XHTML+MathML+SVG (XHTML host)	xhtml-math-svg-xh	<!DOCTYPE html PUBLIC “-//W3C plus MathML 2.0 plus SVG 1.1//EN” “ http://www.w3.org/2002/04/xhtml-math-svg.dtd ”>
XHTML+MathML+SVG (SVG host)	xhtml-math-svg-sh	<!DOCTYPE svg:svg PUBLIC “-//W3C 1.1 plus MathML 2.0 plus SVG 1.1//EN” “ http://www.w3.org/2002/04/xhtml-math-svg.dtd ”>
XHTML+RDFa 1.0	xhtml-rdfa-1	<!DOCTYPE html PUBLIC “-//W3C XHTML+RDFa 1.0//EN” “ http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd ”>
XHTML+RDFa 1.1	xhtml-rdfa-2	<!DOCTYPE html PUBLIC “-//W3C XHTML+RDFa 1.1//EN” “ http://www.w3.org/MarkUp/DTD/xhtml-rdfa-2.dtd ”>

Inflector Helper

The Inflector Helper file contains functions that permits you to change **English** words to plural, singular, camel case, etc.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('inflector');
```

[Available Functions](#)

The following functions are available:

singular(\$string)

Parameters: • **\$string** (*string*) – Input string

Returns: A singular word

Return type: string

Changes a plural word to singular. Example:

```
echo singular('dogs'); // Prints 'dog'
```

plural(\$string)

Parameters: • **\$string** (*string*) – Input string

Returns: A plural word

Return type: string

Changes a singular word to plural. Example:

```
echo plural('dog'); // Prints 'dogs'
```

camelize(\$string)

Parameters: • **\$string** (*string*) – Input string

Returns: Camelized string

Return type: string

Changes a string of words separated by spaces or underscores to camel case. Example:

```
echo camelize('my_dog_spot'); // Prints 'myDogSpot'
```

underscore(\$string)

Parameters: • **\$string** (*string*) – Input string

Returns: String containing underscores instead of spaces

Return type: string

Takes multiple words separated by spaces and underscores them. Example:

```
echo underscore('my dog spot'); // Prints 'my_dog_spot'
```

humanize(\$string[, \$separator = '_'])

Parameters: • **\$string** (*string*) – Input string

• **\$separator** (*string*) – Input separator

Returns: Humanized string

Return type: string

Takes multiple words separated by underscores and adds spaces between them. Each word is capitalized.

Example:

```
echo humanize('my_dog_spot'); // Prints 'My Dog Spot'
```

To use dashes instead of underscores:

```
echo humanize('my-dog-spot', '-'); // Prints 'My Dog Spot'
```

is_pluralizable(\$word)

Parameters: • **\$word** (*string*) – Input string

Returns: TRUE if the word is countable or FALSE if not

Return type: bool

Checks if the given word has a plural version. Example:

```
is_pluralizable('equipment'); // Returns FALSE
```

dasherize(\$string)

Parameters: • **\$string** (*string*) – Input string

Returns: Dasherized string

Return type: string

Replaces underscores with dashes in the string. Example:

```
dasherize('hello_world'); // Returns 'hello-world'
```

ordinal(\$integer)

Parameters: • **\$integer** (*int*) – The integer to determine the suffix

Returns: Ordinal suffix

Return type: string

Returns the suffix that should be added to a number to denote the position such as 1st, 2nd, 3rd, 4th. Example:

```
ordinal(1); // Returns 'st'
```

ordinalize(\$integer)

Parameters: • **\$integer** (*int*) – The integer to ordinalize

Returns: Ordinalized integer

Return type: string

Turns a number into an ordinal string used to denote the position such as 1st, 2nd, 3rd, 4th. Example:

```
ordinalize(1); // Returns '1st'
```


Number Helper

The Number Helper file contains functions that help you work with numeric data in a locale-aware manner.

- [Loading this Helper](#)
- [When Things Go Wrong](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('number');
```

[When Things Go Wrong](#)

If PHP's internationalization and localization logic cannot handle a value provided, for the given locale and options, then a `BadFunctionCallException()` will be thrown.

[Available Functions](#)

The following functions are available:


number_to_size(\$num[, \$precision = 1[, \$locale = null])

- Parameters:**
- **\$num** (*mixed*) – Number of bytes
 - **\$precision** (*int*) – Floating point precision
- Returns:** Formatted data size string, or false if the provided value is not numeric

Return string
type:

Formats numbers as bytes, based on size, and adds the appropriate suffix.
Examples:

```
echo number_to_size(456); // Returns 456 Bytes
echo number_to_size(4567); // Returns 4.5 KB
echo number_to_size(45678); // Returns 44.6 KB
echo number_to_size(456789); // Returns 447.8 KB
echo number_to_size(3456789); // Returns 3.3 MB
echo number_to_size(12345678912345); // Returns 1.8 GB
echo number_to_size(123456789123456789); // Returns 11,228.3 T
```



An optional second parameter allows you to set the precision of the result:

```
echo number_to_size(45678, 2); // Returns 44.61 KB
```

An optional third parameter allows you to specify the locale that should be used when generating the number, and can affect the formatting. If no locale is specified, the Request will be analyzed and an appropriate locale taken from the headers, or the app-default:

```
// Generates 11.2 TB
echo number_to_size(12345678912345, 1, 'en_US');
// Generates 11,2 TB
echo number_to_size(12345678912345, 1, 'fr_FR');
```

Note

The text generated by this function is found in the following language file: `language/<your_lang>/Number.php`

number_to_amount(\$num[, \$precision = 1[, \$locale = null])

Parameters:

- **\$num** (*mixed*) – Number to format
- **\$precision** (*int*) – Floating point precision
- **\$locale** (*string*) – The locale to use for formatting

Returns: A human-readable version of the string, or false if the provided value is not numeric

Return type: string

Converts a number into a human-readable version, like **123.4 trillion** for numbers up to the quadrillions. Examples:

```
echo number_to_amount(123456); // Returns 123 thousand
echo number_to_amount(123456789); // Returns 123 million
echo number_to_amount(1234567890123, 2); // Returns 1.23 trill
echo number_to_amount('123,456,789,012', 2); // Returns 123.46
```

An optional second parameter allows you to set the precision of the result:

```
echo number_to_amount(45678, 2); // Returns 45.68 thousand
```

An optional third parameter allows the locale to be specified:

```
echo number_to_amount('123,456,789,012', 2, 'de_DE'); // Retur
```

number_to_currency(\$num, \$currency[, \$locale = null])

Parameters:

- **\$num** (*mixed*) – Number to format
- **\$currency** (*string*) – The currency type, i.e. USD, EUR, etc
- **\$locale** (*string*) – The locale to use for formatting

Returns: The number as the appropriate currency for the locale

Return type: string

Converts a number in common currency formats, like USD, EUR, GBP, etc:

```
echo number_to_currency(1234.56, 'USD'); // Returns $1,234.56
echo number_to_currency(1234.56, 'EUR'); // Returns €1,234.56
echo number_to_currency(1234.56, 'GBP'); // Returns £1,234.56
echo number_to_currency(1234.56, 'YEN'); // Returns YEN1,234.
```

number_to_roman(\$num)

Parameters: • **\$num** (*string*) – The number want to convert
Returns: The roman number converted from given parameter
Return type: string|null

Converts a number into roman:

```
echo number_to_roman(23); // Returns XXIII
echo number_to_roman(324); // Returns CCCXXIV
echo number_to_roman(2534); // Returns MMDXXXIV
```

This function only handles numbers in the range 1 through 3999. It will return null for any value outside that range .

Security Helper

The Security Helper file contains security related functions.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('security');
```

[Available Functions](#)

The following functions are available:

sanitize_filename(\$filename)

Parameters: • **\$filename** (*string*) – Filename

Returns: Sanitized file name

Return type: string

Provides protection against directory traversal.

This function is an alias for

`\CodeIgniter\Security::sanitize_filename()`. For more info, please see the [Security Library](#) documentation.

strip_image_tags(\$str)

Parameters: • **\$str** (*string*) – Input string

Returns: The input string with no image tags

Return type: string

This is a security function that will strip image tags from a string. It leaves the image URL as plain text.

Example:

```
$string = strip_image_tags($string);
```

encode_php_tags(\$str)

Parameters: • **\$str** (*string*) – Input string

Returns: Safely formatted string

Return type: string

This is a security function that converts PHP tags to entities.

Example:

```
$string = encode_php_tags($string);
```

Text Helper

The Text Helper file contains functions that assist in working with Text.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code:

```
helper('text');
```

[Available Functions](#)

The following functions are available:

random_string(*[\$type = 'alnum'[, \$len = 8]]*)

- Parameters:**
- **\$type** (*string*) – Randomization type
 - **\$len** (*int*) – Output string length

Returns: A random string

Return type: string

Generates a random string based on the type and length you specify. Useful for creating passwords or generating random hashes.

The first parameter specifies the type of string, the second parameter specifies the length. The following choices are available:

- **alpha:** A string with lower and uppercase letters only.
- **alnum:** Alpha-numeric string with lower and uppercase characters.

- **basic**: A random number based on `mt_rand()` (length ignored).
- **numeric**: Numeric string.
- **nozero**: Numeric string with no zeros.
- **md5**: An encrypted random number based on `md5()` (fixed length of 32).
- **sha1**: An encrypted random number based on `sha1()` (fixed length of 40).
- **crypto**: A random string based on `random_bytes()`.

Usage example:

```
echo random_string('alnum', 16);
```

increment_string(\$str[, \$separator = '_'[, \$first = 1]])

Parameters:

- **\$str** (*string*) – Input string
- **\$separator** (*string*) – Separator to append a duplicate number with
- **\$first** (*int*) – Starting number

Returns: An incremented string

Return type: string

Increments a string by appending a number to it or increasing the number. Useful for creating “copies” or a file or duplicating database content which has unique titles or slugs.

Usage example:

```
echo increment_string('file', '_'); // "file_1"
echo increment_string('file', '-', 2); // "file-2"
echo increment_string('file_4'); // "file_5"
```

alternator(\$args)

Parameters: • **\$args** (*mixed*) – A variable number of arguments

Returns: Alternated string(s)


Return type: mixed

Allows two or more items to be alternated between, when cycling through a loop. Example:

```
for ($i = 0; $i < 10; $i++)
{
    echo alternator('string one', 'string two');
}
```

You can add as many parameters as you want, and with each iteration of your loop the next item will be returned.

```
for ($i = 0; $i < 10; $i++)
{
    echo alternator('one', 'two', 'three', 'four', 'five')
}
```



Note

To use multiple separate calls to this function simply call the function with no arguments to re-initialize.


reduce_double_slashes(\$str)

Parameters: • **\$str** (*string*) – Input string
Returns: A string with normalized slashes
Return type: string

Converts double slashes in a string to a single slash, except those found in URL protocol prefixes (e.g. http://).

Example:

```
$string = "http://example.com//index.php";
echo reduce_double_slashes($string); // results in "http://exa
```



strip_slashes(\$data)

Parameters: • **\$data** (*mixed*) – Input string or an array of strings
Returns: String(s) with stripped slashes
Return type: mixed

Removes any slashes from an array of strings.

Example:

```
$str = [  
    'question' => 'Is your name O'reilly?',  
    'answer'    => 'No, my name is O'connor.'  
];  
  
$str = strip_slashes($str);
```

The above will return the following array:

```
[  
    'question' => "Is your name O'reilly?",  
    'answer'    => "No, my name is O'connor."  
];
```

Note

For historical reasons, this function will also accept and handle string inputs. This however makes it just an alias for `stripslashes()`.

reduce_multiples(\$str[, \$character = "[, \$trim = FALSE])

Parameters:	• \$str (<i>string</i>) – Text to search in
	• \$character (<i>string</i>) – Character to reduce
	• \$trim (<i>bool</i>) – Whether to also trim the specified character
Returns:	Reduced string
Return type:	string

Reduces multiple instances of a particular character occurring directly after each other. Example:

```
$string = "Fred, Bill,, Joe, Jimmy";  
$string = reduce_multiples($string, ","); //results in "Fred, B  
< >
```

If the third parameter is set to TRUE it will remove occurrences of the character at the beginning and the end of the string. Example:

```
$string = ",Fred, Bill,, Joe, Jimmy,";
```

```
$string = reduce_multiples($string, ",", ",", TRUE); //results in
```



quotes_to_entities(\$str)


Parameters: • **\$str** (*string*) – Input string

Returns: String with quotes converted to HTML entities

Return type: string

Converts single and double quotes in a string to the corresponding HTML entities. Example:

```
$string = "Joe's \"dinner\"";  
$string = quotes_to_entities($string); //results in "Joe&#39;s
```



strip_quotes(\$str)

Parameters: • **\$str** (*string*) – Input string

Returns: String with quotes stripped

Return type: string

Removes single and double quotes from a string. Example:

```
$string = "Joe's \"dinner\"";  
$string = strip_quotes($string); //results in "Joes dinner"
```

word_limiter(\$str[, \$limit = 100[, \$end_char = '…']])

Parameters:


- **\$str** (*string*) – Input string
- **\$limit** (*int*) – Limit
- **\$end_char** (*string*) – End character (usually an ellipsis)

Returns: Word-limited string

Return type: string

Truncates a string to the number of *words* specified. Example:

```
$string = "Here is a nice text string consisting of eleven wor  
$string = word_limiter($string, 4);  
// Returns: Here is a nice
```



The third parameter is an optional suffix added to the string. By default it adds an ellipsis.

character_limiter(\$str[, \$n = 500[, \$end_char = '…']])

Parameters:

- **\$str** (*string*) – Input string
- **\$n** (*int*) – Number of characters
- **\$end_char** (*string*) – End character (usually an ellipsis)

Returns: Character-limited string

Return type: string

Truncates a string to the number of *characters* specified. It maintains the integrity of words so the character count may be slightly more or less than what you specify.

Example:

```
$string = "Here is a nice text string consisting of eleven wor  
$string = character_limiter($string, 20);  
// Returns: Here is a nice text string
```



The third parameter is an optional suffix added to the string, if undeclared this helper uses an ellipsis.

Note

If you need to truncate to an exact number of characters please see the [ellipsize\(\)](#) function below.

ascii_to_entities(\$str)

Parameters: • **\$str** (*string*) – Input string

Returns: A string with ASCII values converted to entities

Return type: string

Converts ASCII values to character entities, including high ASCII and MS Word characters that can cause problems when used in a web page,

so that they can be shown consistently regardless of browser settings or stored reliably in a database. There is some dependence on your server's supported character sets, so it may not be 100% reliable in all cases, but for the most part it should correctly identify characters outside the normal range (like accented characters).

Example:

```
$string = ascii_to_entities($string);
```

entities_to_ascii(\$str[, \$all = TRUE])

Parameters:

- **\$str** (*string*) – Input string
- **\$all** (*bool*) – Whether to convert unsafe entities as well

Returns: A string with HTML entities converted to ASCII characters

Return type: string

This function does the opposite of [ascii_to_entities\(\)](#). It turns character entities back into ASCII.

convert_accented_characters(\$str)

Parameters:

- **\$str** (*string*) – Input string

Returns: A string with accented characters converted

Return type: string

Transliterates high ASCII characters to low ASCII equivalents. Useful when non-English characters need to be used where only standard ASCII characters are safely used, for instance, in URLs.

Example:

```
$string = convert_accented_characters($string);
```

Note

This function uses a companion config file *app/Config/ForeignCharacters.php* to define the to and from array for transliteration.

word_censor(\$str, \$censored[, \$replacement = "])

Parameters:

- **\$str** (*string*) – Input string
- **\$censored** (*array*) – List of bad words to censor
- **\$replacement** (*string*) – What to replace bad words with

Returns: Censored string

Return type: string

Enables you to censor words within a text string. The first parameter will contain the original string. The second will contain an array of words which you disallow. The third (optional) parameter can contain a replacement value for the words. If not specified they are replaced with pound signs: #####.

Example:

```
$disallowed = ['darn', 'shucks', 'golly', 'phooey'];  
$string      = word_censor($string, $disallowed, 'Beep!');
```

highlight_code(\$str)

Parameters: • **\$str** (*string*) – Input string

Returns: String with code highlighted via HTML

Return type: string

Colorizes a string of code (PHP, HTML, etc.). Example:

```
$string = highlight_code($string);
```

The function uses PHP's `highlight_string()` function, so the colors used are the ones specified in your `php.ini` file.

highlight_phrase(\$str, \$phrase[, \$tag_open = '<mark>', \$tag_close = '</mark>'])

- **\$str** (*string*) – Input string
- **\$phrase** (*string*) – Phrase to highlight

Parameters: • **\$tag_open** (*string*) – Opening tag used for the highlight
• **\$tag_close** (*string*) – Closing tag for the highlight

Returns: String with a phrase highlighted via HTML

Return type: string

Will highlight a phrase within a text string. The first parameter will contain the original string, the second will contain the phrase you wish to highlight. The third and fourth parameters will contain the opening/closing HTML tags you would like the phrase wrapped in.

Example:

```
$string = "Here is a nice text string about nothing in particu  
echo highlight_phrase($string, "nice text", '<span style="colo  
< >
```

The above code prints:

```
Here is a <span style="color:#990000;">nice text</span> string  
< >
```

Note

This function used to use the `` tag by default. Older browsers might not support the new HTML5 mark tag, so it is recommended that you insert the following CSS code into your stylesheet if you need to support such browsers:

```
mark {  
    background: #ff0;  
    color: #000;  
};
```

word_wrap(\$str[, \$charlim = 76])

Parameters: • **\$str** (*string*) – Input string
• **\$charlim** (*int*) – Character limit

Returns: Word-wrapped string

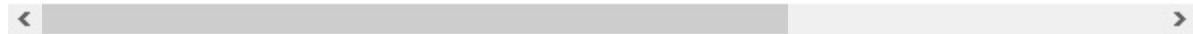
Return type: string

Wraps text at the specified *character* count while maintaining complete words.

Example:

```
$string = "Here is a simple string of text that will help us d  
echo word_wrap($string, 25);
```

```
// Would produce:  
// Here is a simple string  
// of text that will help us  
// demonstrate this  
// function.
```



Excessively long words will be split, but URLs will not be.

ellipsize(\$str, \$max_length[, \$position = 1[, \$ellipsis = '…']])

- **\$str** (*string*) – Input string
- **\$max_length** (*int*) – String length limit
- Parameters:** • **\$position** (*mixed*) – Position to split at (int or float)
- **\$ellipsis** (*string*) – What to use as the ellipsis character

Returns: Ellipsized string

Return type: string

This function will strip tags from a string, split it at a defined maximum length, and insert an ellipsis.

The first parameter is the string to ellipsize, the second is the number of characters in the final string. The third parameter is where in the string the ellipsis should appear from 0 - 1, left to right. For example. a value of 1 will place the ellipsis at the right of the string, .5 in the middle, and 0 at the left.

An optional forth parameter is the kind of ellipsis. By default, … will be inserted.

Example:

```
$str = 'this_string_is_entirely_too_long_and_might_break_my_de  
echo ellipsize($str, 32, .5);
```

<  >

Produces:

this_string_is_e…ak_my_design.jpg

excerpt(\$text, \$phrase = false, \$radius = 100, \$ellipsis = '...')

- Parameters:**
- **\$text** (*string*) – Text to extract an excerpt
 - **\$phrase** (*string*) – Phrase or word to extract the text around
 - **\$radius** (*int*) – Number of characters before and after \$phrase
 - **\$ellipsis** (*string*) – What to use as the ellipsis character

Returns: Excerpt.

Return type: string

This function will extract \$radius number of characters before and after the central \$phrase with an ellipsis before and after.

The first parameter is the text to extract an excerpt from, the second is the central word or phrase to count before and after. The third parameter is the number of characters to count before and after the central phrase. If no phrase passed, the excerpt will include the first \$radius characters with the ellipsis at the end.

Example:

```
$text = 'Ut vel faucibus odio. Quisque quis congue libero. Et  
eros lorem, eget porttitor augue dignissim tincidunt. In eget  
mauris faucibus molestie vitae ultricies odio. Vestibulum id u  
Curabitur non mauris lectus. Phasellus eu sodales sem. Integer  
ac enim hendrerit gravida. Donec ac magna vel nunc tincidunt m  
vitae nisl. Cras sed auctor mauris, non dictum tortor. Nulla v  
arcu. Cras ac ipsum sit amet augue laoreet laoreet. Aenean a r  
Sed ut tortor diam.';
```

```
echo excerpt($str, 'Donec');
```

Produces:

```
... non mauris lectus. Phasellus eu sodales sem. Integer dictu  
enim hendrerit gravida. Donec ac magna vel nunc tincidunt mole  
vitae nisl. Cras sed auctor mauris, non dictum ...
```

URL Helper

The URL Helper file contains functions that assist in working with URLs.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is automatically loaded by the framework on every request.

[Available Functions](#)

The following functions are available:

site_url(*[\$uri = ''[, \$protocol = NULL[, \$altConfig = NULL]]*)

Parameters:

- **\$uri** (*string*) – URI string
- **\$protocol** (*string*) – Protocol, e.g. 'http' or 'https'
- **\$altConfig** (*\Config\App*) – Alternate configuration to use

Returns: Site URL

Return type: string

Returns your site URL, as specified in your config file. The `index.php` file (or whatever you have set as your site **index_page** in your config file) will be added to the URL, as will any URI segments you pass to the function, plus the **url_suffix** as set in your config file.

You are encouraged to use this function any time you need to generate a local URL so that your pages become more portable in the event your

URL changes.

Segments can be optionally passed to the function as a string or an array. Here is a string example:

```
echo site_url('news/local/123');
```

The above example would return something like:

http://example.com/index.php/news/local/123

Here is an example of segments passed as an array:

```
$segments = ['news', 'local', '123'];  
echo site_url($segments);
```

You may find the alternate configuration useful if generating URLs for a different site than yours, which contains different configuration preferences. We use this for unit testing the framework itself.

base_url([*\$uri* = "[, *\$protocol* = *NULL*]])

Parameters:

- **\$uri** (*string*) – URI string
- **\$protocol** (*string*) – Protocol, e.g. 'http' or 'https'

Returns: Base URL

Return type: string

Returns your site base URL, as specified in your config file. Example:

```
echo base_url();
```

This function returns the same thing as [site_url\(\)](#), without the *index_page* or *url_suffix* being appended.

Also like [site_url\(\)](#), you can supply segments as a string or an array. Here is a string example:

```
echo base_url("blog/post/123");
```

The above example would return something like:

http://example.com/blog/post/123

This is useful because unlike `site_url()`, you can supply a string to a file, such as an image or stylesheet. For example:

```
echo base_url("images/icons/edit.png");
```

This would give you something like:

`http://example.com/images/icons/edit.png`

current_url(*[\$returnObject = false]*)

Parameters: • **\$returnObject** (*boolean*) – True if you would like a URI instance returned, instead of a string.

Returns: The current URL

Return type: string|URI

Returns the full URL (including segments) of the page being currently viewed.

Note

Calling this function is the same as doing this:: `base_url(uri_string());`

previous_url(*[\$returnObject = false]*)

Parameters: • **\$returnObject** (*boolean*) – True if you would like a URI instance returned instead of a string.

Returns: The URL the user was previously on

Return type: string|URI

Returns the full URL (including segments) of the page the user was previously on.

Due to security issues of blindly trusting the HTTP_REFERER system variable, CodeIgniter will store previously visited pages in the session if it's available. This ensures that we always use a known and trusted source. If the session hasn't been loaded, or is otherwise unavailable, then a sanitized version of HTTP_REFERER will be used.

uri_string()

Returns: An URI string

Return type: string

Returns the path part of your current URL. For example, if your URL was this:

`http://some-site.com/blog/comments/123`

The function would return:

`blog/comments/123`

index_page([*\$saltConfig* = NULL])

Parameters:

- ***\$saltConfig*** (*ConfigApp*) – Alternate configuration to use

Returns: 'index_page' value

Return type: mixed

Returns your site **index_page**, as specified in your config file. Example:

```
echo index_page();
```

As with [site_url\(\)](#), you may specify an alternate configuration. You may find the alternate configuration useful if generating URLs for a different site than yours, which contains different configuration preferences. We use this for unit testing the framework itself.

anchor([*\$uri* = "", *\$title* = "", *\$attributes* = "", *\$saltConfig* = NULL])])])])

Parameters:

- ***\$uri*** (*mixed*) – URI string or array of URI segments
- ***\$title*** (*string*) – Anchor title
- ***\$attributes*** (*mixed*) – HTML attributes
- ***\$saltConfig*** (*ConfigApp*) – Alternate configuration to use

Returns: HTML hyperlink (anchor tag)

Return type: string

Creates a standard HTML anchor link based on your local site URL.

The first parameter can contain any segments you wish appended to the URL. As with the [site_url\(\)](#) function above, segments can be a string or an array.

Note

If you are building links that are internal to your application do not include the base URL (<http://...>). This will be added automatically from the information specified in your config file. Include only the URI segments you wish appended to the URL.

The second segment is the text you would like the link to say. If you leave it blank, the URL will be used.

The third parameter can contain a list of attributes you would like added to the link. The attributes can be a simple string or an associative array.

Here are some examples:

```
echo anchor('news/local/123', 'My News', 'title="News title"')
// Prints: <a href="http://example.com/index.php/news/local/12
```

```
echo anchor('news/local/123', 'My News', ['title' => 'The best
// Prints: <a href="http://example.com/index.php/news/local/12
```

```
echo anchor('', 'Click here');
// Prints: <a href="http://example.com/index.php">Click here</
< >
```

As above, you may specify an alternate configuration. You may find the alternate configuration useful if generating links for a different site than yours, which contains different configuration preferences. We use this for unit testing the framework itself.

Note

Attributes passed into the anchor function are automatically escaped to protected against XSS attacks.

```
anchor_popup([$uri = "[, $title = "[, $attributes = FALSE[, $altConfig =  
NULL]]]])
```

Parameters:

- ***\$uri*** (*string*) – URI string
- ***\$title*** (*string*) – Anchor title
- ***\$attributes*** (*mixed*) – HTML attributes
- ***\$altConfig*** (*ConfigApp*) – Alternate configuration to use

Returns: Pop-up hyperlink

Return type: string

Nearly identical to the [anchor\(\)](#) function except that it opens the URL in a new window. You can specify JavaScript window attributes in the third parameter to control how the window is opened. If the third parameter is not set it will simply open a new window with your own browser settings.

Here is an example with attributes:

```
$atts = [  
    'width'      => 800,  
    'height'     => 600,  
    'scrollbars' => 'yes',  
    'status'     => 'yes',  
    'resizable'  => 'yes',  
    'screenx'    => 0,  
    'screeny'    => 0,  
    'window_name' => '_blank'  
];  
  
echo anchor_popup('news/local/123', 'Click Me!', $atts);
```

Note

The above attributes are the function defaults so you only need to set the ones that are different from what you need. If you want the function to use all of its defaults simply pass an empty array in the third parameter:

```
echo anchor_popup('news/local/123', 'Click Me!', []);
```

Note

The **window_name** is not really an attribute, but an argument to the JavaScript [window.open\(\)](http://www.w3schools.com/jsref/met_win_open.asp) [http://www.w3schools.com/jsref/met_win_open.asp] method, which accepts either a window name or a window target.

Note

Any other attribute than the listed above will be parsed as an HTML attribute to the anchor tag.

As above, you may specify an alternate configuration. You may find the alternate configuration useful if generating links for a different site than yours, which contains different configuration preferences. We use this for unit testing the framework itself.

Note

Attributes passed into the anchor_popup function are automatically escaped to protected against XSS attacks.

mailto(\$email[, \$title = "[, \$attributes = "]])

Parameters:

- **\$email** (*string*) – E-mail address
- **\$title** (*string*) – Anchor title
- **\$attributes** (*mixed*) – HTML attributes

Returns: A “mail to” hyperlink

Return type: string

Creates a standard HTML e-mail link. Usage example:

```
echo mailto('me@my-site.com', 'Click Here to Contact Me');
```

As with the [anchor\(\)](#) tab above, you can set attributes using the third parameter:

```
$attributes = ['title' => 'Mail me'];  
echo mailto('me@my-site.com', 'Contact Me', $attributes);
```

Note

Attributes passed into the `mailto` function are automatically escaped to protected against XSS attacks.

`safe_mailto($email[, $title = "[, $attributes = "]])`

Parameters:

- **`$email`** (*string*) – E-mail address
- **`$title`** (*string*) – Anchor title
- **`$attributes`** (*mixed*) – HTML attributes

Returns: A spam-safe “mail to” hyperlink

Return type: string

Identical to the [`mailto\(\)`](#) function except it writes an obfuscated version of the `mailto` tag using ordinal numbers written with JavaScript to help prevent the e-mail address from being harvested by spam bots.

`auto_link($str[, $type = 'both'[, $popup = FALSE]])`

Parameters:

- **`$str`** (*string*) – Input string
- **`$type`** (*string*) – Link type (‘email’, ‘url’ or ‘both’)
- **`$popup`** (*bool*) – Whether to create popup links

Returns: Linkified string

Return type: string

Automatically turns URLs and e-mail addresses contained in a string into links. Example:

```
$string = auto_link($string);
```

The second parameter determines whether URLs and e-mails are converted or just one or the other. Default behavior is both if the parameter is not specified. E-mail links are encoded as [`safe_mailto\(\)`](#) as shown above.

Converts only URLs:

```
$string = auto_link($string, 'url');
```

Converts only e-mail addresses:

```
$string = auto_link($string, 'email');
```

The third parameter determines whether links are shown in a new window. The value can be TRUE or FALSE (boolean):

```
$string = auto_link($string, 'both', TRUE);
```

Note

The only URLs recognized are those that start with “www.” or with “://”.

```
url_title($str[, $separator = '-', $lowercase = FALSE])
```

Parameters:

- **\$str** (*string*) – Input string
- **\$separator** (*string*) – Word separator
- **\$lowercase** (*bool*) – Whether to transform the output string to lower-case

Returns: URL-formatted string

Return type: string

Takes a string as input and creates a human-friendly URL string. This is useful if, for example, you have a blog in which you’d like to use the title of your entries in the URL. Example:

```
$title      = "What's wrong with CSS?";  
$url_title = url_title($title);  
// Produces: Whats-wrong-with-CSS
```

The second parameter determines the word delimiter. By default dashes are used. Preferred options are: - (dash) or _ (underscore).

Example:

```
$title      = "What's wrong with CSS?";  
$url_title = url_title($title, 'underscore');  
// Produces: Whats_wrong_with_CSS
```

The third parameter determines whether or not lowercase characters are forced. By default they are not. Options are boolean TRUE/FALSE.

Example:

```
$title      = "What's wrong with CSS?";  
$url_title = url_title($title, 'underscore', TRUE);  
// Produces: whats_wrong_with_css
```

prep_url(\$str = "")

Parameters: • \$str (*string*) – URL string

Returns: Protocol-prefixed URL string

Return type: string

This function will add *http://* in the event that a protocol prefix is missing from a URL.

Pass the URL string to the function like this:

```
$url = prep_url('example.com');
```

XML Helper

The XML Helper file contains functions that assist in working with XML data.

- [Loading this Helper](#)
- [Available Functions](#)

[Loading this Helper](#)

This helper is loaded using the following code

```
helper('xml');
```

[Available Functions](#)

The following functions are available:

xml_convert(\$str[, \$protect_all = FALSE])

- | | |
|---------------------|--|
| Parameters: | <ul style="list-style-type: none">• \$str (<i>string</i>) – the text string to convert• \$protect_all (<i>bool</i>) – Whether to protect all content that looks like a potential entity instead of just numbered entities, e.g. &foo; |
| Returns: | XML-converted string |
| Return type: | string |

Takes a string as input and converts the following reserved XML characters to entities:

- Ampersands: &

- Less than and greater than characters: < >
- Single and double quotes: ‘ “
- Dashes: -

This function ignores ampersands if they are part of existing numbered character entities, e.g. {. Example:

```
$string = '<p>Here is a paragraph & an entity (&#123;).</p>';  
$string = xml_convert($string);  
echo $string;
```

outputs:

```
&lt;p&gt;Here is a paragraph &amp; an entity (&#123;).&lt;/p&g  
< >
```

Testing

CodeIgniter ships with a number of tools to help you test and debug your application thoroughly. The following sections should get you quickly testing your applications.

- [Getting Started](#)
- [Database](#)
- [Controller Testing](#)
- [HTTP Testing](#)
- [Benchmarking](#)
- [Debugging Your Application](#)

Testing

CodeIgniter has been built to make testing both the framework and your application as simple as possible. Support for PHPUnit is built in, and the framework provides a number of convenient helper methods to make testing every aspect of your application as painless as possible.

- [System Setup](#)
 - [Installing phpUnit](#)
- [Testing Your Application](#)
 - [PHPUnit Configuration](#)
 - [The Test Class](#)
 - [Mocking Services](#)
 - [Stream Filters](#)

[System Setup](#)

[Installing phpUnit](#)

CodeIgniter uses [phpUnit](https://phpunit.de/) [https://phpunit.de/] as the basis for all of its testing. There are two ways to install phpUnit to use within your system.

Composer

The recommended method is to install it in your project using [Composer](https://getcomposer.org/) [https://getcomposer.org/]. While it's possible to install it globally we do not recommend it, since it can cause compatibility issues with other projects on your system as time goes on.

Ensure that you have Composer installed on your system. From the project root (the directory that contains the application and system directories) type

the following from the command line:

```
> composer require --dev phpunit/phpunit
```

This will install the correct version for your current PHP version. Once that is done, you can run all of the tests for this project by typing:

```
> ./vendor/bin/phpunit
```

Phar

The other option is to download the .phar file from the [phpUnit](https://phpunit.de/getting-started/phpunit-7.html) [https://phpunit.de/getting-started/phpunit-7.html] site. This is standalone file that should be placed within your project root.

Testing Your Application

PHPUnit Configuration

The framework has a `phpunit.xml.dist` file in the project root. This controls unit testing of the framework itself. If you provide your own `phpunit.xml`, it will over-ride this.

Your `phpunit.xml` should exclude the system folder, as well as any vendor or ThirdParty folders, if you are unit testing your application.

The Test Class

In order to take advantage of the additional tools provided, your tests must extend `\PHPUnit_TestCase`. All tests are expected to be located in the **tests/** directory by default.

To test a new library, **Foo**, you would create a new file at **tests/TestFoo.php**:

```
<?php namespace Tests;  
  
class MyTests extends \PHPUnit_TestCase  
{
```

```
    public function testFooNotBar()  
    {  
        . . .  
    }  
}
```

You can create any directory structure that fits your testing style/needs. When namespacing the test classes, remember that the **tests** directory is the root of the Tests namespace, so any classes you use must have the correct namespace relative to Tests.

Note

Namespaces are not required for test classes, but they are helpful to ensure no class names collide.

When testing database results, you must use the [CIDatabaseTestClass](#) class.

Additional Assertions

CIUnitTestCase provides additional unit testing assertions that you might find useful.

assertLogged(\$level, \$expectedMessage)

Ensure that something you expected to be logged actually was:

```
$config = new LoggerConfig();  
$logger = new Logger($config);  
  
... do something that you expect a log entry from  
$logger->log('error', "That's no moon");  
  
$this->assertLogged('error', "That's no moon");
```

assertEventTriggered(\$eventName)

Ensure that an event you expected to be triggered actually was:

```
Events::on('foo', function($arg) use(&$result) {  
    $result = $arg;  
});
```

```
Events::trigger('foo', 'bar');
```

```
$this->assertEventTriggered('foo');
```

assertHeaderEmitted(\$header, \$ignoreCase=false)

Ensure that a header or cookie was actually emitted:

```
$response->setCookie('foo', 'bar');
```

```
ob_start();
```

```
$this->response->send();
```

```
$output = ob_get_clean(); // in case you want to check the actual
```

```
$this->assertHeaderEmitted("Set-Cookie: foo=bar");
```

<  >

Note: the test case with this should be [run as a separate process in PHPunit](https://phpunit.readthedocs.io/en/7.4/annotations.html#runinseparateprocess)

[<https://phpunit.readthedocs.io/en/7.4/annotations.html#runinseparateprocess>].

assertHeaderNotEmitted(\$header, \$ignoreCase=false)

Ensure that a header or cookie was actually emitted:

```
$response->setCookie('foo', 'bar');
```

```
ob_start();
```

```
$this->response->send();
```

```
$output = ob_get_clean(); // in case you want to check the actual
```

```
$this->assertHeaderNotEmitted("Set-Cookie: banana");
```

<  >

Note: the test case with this should be [run as a separate process in PHPunit](https://phpunit.readthedocs.io/en/7.4/annotations.html#runinseparateprocess)

[<https://phpunit.readthedocs.io/en/7.4/annotations.html#runinseparateprocess>].

assertCloseEnough(\$expected, \$actual, \$message='', \$tolerance=1)

For extended execution time testing, tests that the absolute difference

between expected and actual time is within the prescribed tolerance.:

```
$timer = new Timer();  
$timer->start('longjohn', strtotime('-11 minutes'));  
$this->assertCloseEnough(11 * 60, $timer->getElapsedTime('longjoh  
< >
```

The above test will allow the actual time to be either 660 or 661 seconds.

assertCloseEnoughString(\$expected, \$actual, \$message='', \$tolerance=1)

For extended execution time testing, tests that the absolute difference between expected and actual time, formatted as strings, is within the prescribed tolerance.:

```
$timer = new Timer();  
$timer->start('longjohn', strtotime('-11 minutes'));  
$this->assertCloseEnoughString(11 * 60, $timer->getElapsedTime('l  
< >
```

The above test will allow the actual time to be either 660 or 661 seconds.

Accessing Protected/Private Properties

When testing, you can use the following setter and getter methods to access protected and private methods and properties in the classes that you are testing.

getPrivateMethodInvoker(\$instance, \$method)

Enables you to call private methods from outside the class. This returns a function that can be called. The first parameter is an instance of the class to test. The second parameter is the name of the method you want to call.

```
// Create an instance of the class to test  
$obj = new Foo();  
  
// Get the invoker for the 'privateMethod' method.  
$method = $this->getPrivateMethodInvoker($obj, 'privateMethod'  
  
// Test the results
```

```
$this->assertEquals('bar', $method('param1', 'param2'));
```

getPrivateProperty(\$instance, \$property)

Retrieves the value of a private/protected class property from an instance of a class. The first parameter is an instance of the class to test. The second parameter is the name of the property.

```
// Create an instance of the class to test
$obj = new Foo();

// Test the value
$this->assertEquals('bar', $this->getPrivateProperty($obj, 'baz'))
```

setPrivateProperty(\$instance, \$property, \$value)

Set a protected value within a class instance. The first parameter is an instance of the class to test. The second parameter is the name of the property to set the value of. The third parameter is the value to set it to:

```
// Create an instance of the class to test
$obj = new Foo();

// Set the value
$this->setPrivateProperty($obj, 'baz', 'oops!');

// Do normal testing...
```

Mocking Services

You will often find that you need to mock one of the services defined in **app/Config/Services.php** to limit your tests to only the code in question, while simulating various responses from the services. This is especially true when testing controllers and other integration testing. The **Services** class provides two methods to make this simple: `injectMock()`, and `reset()`.

injectMock()

This method allows you to define the exact instance that will be returned by

the Services class. You can use this to set properties of a service so that it behaves in a certain way, or replace a service with a mocked class.

```
public function testSomething()  
{  
    $curlrequest = $this->getMockBuilder('CodeIgniter\HTTP\CURLRe  
        ->setMethods(['request'])  
        ->getMock();  
    Services::injectMock('curlrequest', $curlrequest);  
  
    // Do normal testing here....  
}  
< >
```

The first parameter is the service that you are replacing. The name must match the function name in the Services class exactly. The second parameter is the instance to replace it with.

reset()

Removes all mocked classes from the Services class, bringing it back to its original state.

Stream Filters

CITestStreamFilter provides an alternate to these helper methods.

You may need to test things that are difficult to test. Sometimes, capturing a stream, like PHP's own STDOUT, or STDERR, might be helpful. The **CITestStreamFilter** helps you capture the output from the stream of your choice.


An example demonstrating this inside one of your test cases:

```
public function setUp()  
{  
    CITestStreamFilter::$buffer = '';  
    $this->stream_filter = stream_filter_append(STDOUT, 'CITestSt  
}  
  
public function tearDown()  
{
```

```
        stream_filter_remove($this->stream_filter);
    }

    public function testSomeOutput()
    {
        CLI::write('first. ');
        $expected = "first.\n";
        $this->assertEquals($expected, CITestStreamFilter::$buffer);
    }

```



Testing Your Database

- [The Test Class](#)
- [Test Database Setup](#)
 - [Migrations and Seeds](#)
- [Helper Methods](#)

[The Test Class](#)

In order to take advantage of the built-in database tools that CodeIgniter provides for testing, your tests must extend `CIDatabaseTestCase`:

```
<?php namespace App\Database;

use CodeIgniter\Test\CIDatabaseTestCase;

class MyTests extends CIDatabaseTestCase
{
    . . .
}
```

Because special functionality executed during the `setUp()` and `tearDown()` phases, you must ensure that you call the parent's methods if you need to use those methods, otherwise you will lose much of the functionality described here:

```
<?php namespace App\Database;

use CodeIgniter\Test\CIDatabaseTestCase;

class MyTests extends CIDatabaseTestCase
{
    public function setUp()
    {
        parent::setUp();
    }
}
```

```

        // Do something here....
    }

    public function tearDown()
    {
        parent::tearDown();

        // Do something here....
    }
}

```

Test Database Setup

When running database tests, you need to provide a database that can be used during testing. Instead of using the PHPUnit built-in database features, the framework provides tools specific to CodeIgniter. The first step is to ensure that you have a tests database group setup in **app/Config/Database.php**. This specifies a database connection that is only used while running tests, to keep your other data safe.

If you have multiple developers on your team, you will likely want to keep your credentials store in the **.env** file. To do so, edit the file to ensure the following lines are present, and have the correct information:

```

database.tests.dbdriver = 'MySQLi';
database.tests.username = 'root';
database.tests.password = '';
database.tests.database = '';

```

Migrations and Seeds

When running tests you need to ensure that your database has the correct schema setup, and that it is in a known state for every test. You can use migrations and seeds to setup your database, by adding a couple of class properties to your test.

```

<?php namespace App\Database;

use CodeIgniter\Test\CIDatabaseTestCase;

class MyTests extends CIDatabaseTestCase

```

```
{  
    protected $refresh = true;  
    protected $seed     = 'TestSeeder';  
    protected $basePath = 'path/to/database/files';  
}
```

\$refresh

This boolean value determines whether the database is completely refreshed before every test. If true, all migrations are rolled back to version 0, then the database is migrated to the latest available migration.

\$seed

If present and not empty, this specifies the name of a Seed file that is used to populate the database with test data prior to every test running.

\$basePath

By default, CodeIgniter will look in **tests/_support/database/migrations** and **tests/_support_database/seeds** to locate the migrations and seeds that it should run during testing. You can change this directory by specifying the path in the \$basePath property. This should not include the **migrations** or **seeds** directories, but the path to the single directory that holds both of those sub-directories.

Helper Methods

The **CIDatabaseTestCase** class provides several helper methods to aid in testing your database.

seed(\$name)

Allows you to manually load a Seed into the database. The only parameter is the name of the seed to run. The seed must be present within the path specified in \$basePath.

dontSeeInDatabase(\$table, \$criteria)

Asserts that a row with criteria matching the key/value pairs in `$criteria` DOES NOT exist in the database.

```
$criteria = [
    'email' => 'joe@example.com',
    'active' => 1
];
$this->dontSeeInDatabase('users', $criteria);
```

seeInDatabase(\$table, \$criteria)

Asserts that a row with criteria matching the key/value pairs in `$criteria` DOES exist in the database.

```
$criteria = [
    'email' => 'joe@example.com',
    'active' => 1
];
$this->seeInDatabase('users', $criteria);
```

grabFromDatabase(\$table, \$column, \$criteria)

Returns the value of `$column` from the specified table where the row matches `$criteria`. If more than one row is found, it will only test against the first one.

```
$username = $this->grabFromDatabase('users', 'username', ['email'
```



hasInDatabase(\$table, \$data)

Inserts a new row into the database. This row is removed after the current test runs. `$data` is an associative array with the data to insert into the table.

```
$data = [
    'email' => 'joe@example.com',
    'name' => 'Joe Cool'
];
$this->hasInDatabase('users', $data);
```

seeNumRecords(\$expected, \$table, \$criteria)

Asserts that a number of matching rows are found in the database that match \$criteria.

```
$criteria = [  
    'deleted' => 1  
];  
$this->seeNumRecords(2, 'users', $criteria);
```

Testing Controllers

Testing your controllers is made convenient with a couple of new helper classes and traits. When testing controllers, you can execute the code within a controller, without first running through the entire application bootstrap process. Often times, using the [Feature Testing tools](#) will be simpler, but this functionality is here in case you need it.

Note

Because the entire framework has not been bootstrapped, there will be times when you cannot test a controller this way.

The Helper Trait

You can use either of the base test classes, but you do need to use the `ControllerTester` trait within your tests:

```
<?php namespace CodeIgniter;

use CodeIgniter\Test\ControllerTester;

class TestControllerA extends \CIDatabaseTestCase
{
    use ControllerTester;
}
```

Once the trait has been included, you can start setting up the environment, including the request and response classes, the request body, URI, and more. You specify the controller to use with the `controller()` method, passing in the fully qualified class name of your controller. Finally, call the `execute()` method with the name of the method to run as the parameter:

```
<?php namespace CodeIgniter;
```

```

use CodeIgniter\Test\ControllerTester;

class TestControllerA extends \CIDatabaseTestCase
{
    use ControllerTester;

    public function testShowCategories()
    {
        $result = $this->withURI('http://example.com/categories')
            ->controller(\App\Controllers\ForumContro
            ->execute('showCategories');

        $this->assertTrue($result->isOk());
    }
}

```

Helper Methods

controller(\$class)

Specifies the class name of the controller to test. The first parameter must be a fully qualified class name (i.e. include the namespace):

```
$this->controller(\App\Controllers\ForumController::class);
```

execute(\$method)

Executes the specified method within the controller. The only parameter is the name of the method to run:

```
$results = $this->controller(\App\Controllers\ForumController::cl
    ->execute('showCategories');
```

This returns a new helper class that provides a number of routines for checking the response itself. See below for details.

withConfig(\$config)

Allows you to pass in a modified version of **ConfigApp.php** to test with different settings:

```
$config = new Config\App();  
$config->appTimezone = 'America/Chicago';  
  
$results = $this->withConfig($config)  
    ->controller(\App\Controllers\ForumController::c  
    ->execute('showCategories');
```

If you do not provide one, the application's App config file will be used.

withRequest(\$request)

Allows you to provide an **IncomingRequest** instance tailored to your testing needs:

```
$request = new CodeIgniter\HTTP\IncomingRequest(new Config\App(),  
$request->setLocale($locale);  
  
$results = $this->withRequest($request)  
    ->controller(\App\Controllers\ForumController::c  
    ->execute('showCategories');
```

If you do not provide one, a new IncomingRequest instance with the default application values will be passed into your controller.

withResponse(\$response)

Allows you to provide a **Response** instance:

```
$response = new CodeIgniter\HTTP\Response(new Config\App());  
  
$results = $this->withResponse($response)  
    ->controller(\App\Controllers\ForumController::c  
    ->execute('showCategories');
```

If you do not provide one, a new Response instance with the default application values will be passed into your controller.

withURI(\$uri)

Allows you to provide a new URI that simulates the URL the client was

visiting when this controller was run. This is helpful if you need to check URI segments within your controller. The only parameter is a string representing a valid URI:

```
$results = $this->withURI('http://example.com/forums/categories')
    ->controller(\App\Controllers\ForumController::c
    ->execute('showCategories');
```

It is a good practice to always provide the URI during testing to avoid surprises.

withBody(\$body)

Allows you to provide a custom body for the request. This can be helpful when testing API controllers where you need to set a JSON value as the body. The only parameter is a string that represents the body of the request:

```
$body = json_encode(['foo' => 'bar']);

$results = $this->withBody($body)
    ->controller(\App\Controllers\ForumController::c
    ->execute('showCategories');
```

Checking the Response

When the controller is executed, a new **ControllerResponse** instance will be returned that provides a number of helpful methods, as well as direct access to the Request and Response that were generated.

isOK()

This provides a simple check that the response would be considered a “successful” response. This primarily checks that the HTTP status code is within the 200 or 300 ranges:

```
$results = $this->withBody($body)
    ->controller(\App\Controllers\ForumController::c
    ->execute('showCategories');
```

```
if ($results->isOk())  
{  
    . . .  
}
```

isRedirect()

Checks to see if the final response was a redirection of some sort:

```
$results = $this->withBody($body)  
            ->controller(\App\Controllers\ForumController::c  
            ->execute('showCategories');  
  
if ($results->isRedirect())  
{  
    . . .  
}
```

request()

You can access the Request object that was generated with this method:

```
$results = $this->withBody($body)  
            ->controller(\App\Controllers\ForumController::c  
            ->execute('showCategories');  
  
$request = $results->request();
```

response()

This allows you access to the response object that was generated, if any:

```
$results = $this->withBody($body)  
            ->controller(\App\Controllers\ForumController::c  
            ->execute('showCategories');  
  
$response = $results->response();
```

getBody()

You can access the body of the response that would have been sent to the client with the **getBody()** method. This could be generated HTML, or a JSON response, etc.:

```
$results = $this->withBody($body)
            ->controller(\App\Controllers\ForumController::c
            ->execute('showCategories');

$body = $results->getBody();
```

Response Helper methods

The response you get back contains a number of helper methods to inspect the HTML output within the response. These are useful for using within assertions in your tests.

The **see()** method checks the text on the page to see if it exists either by itself, or more specifically within a tag, as specified by type, class, or id:

```
// Check that "Hello World" is on the page
$results->see('Hello World');
// Check that "Hello World" is within an h1 tag
$results->see('Hello World', 'h1');
// Check that "Hello World" is within an element with the "notice
$results->see('Hello World', '.notice');
// Check that "Hello World" is within an element with id of "titl
$results->see('Hello World', '#title');
```

The **dontSee()** method is the exact opposite:

```
// Checks that "Hello World" does NOT exist on the page
$results->dontSee('Hello World');
// Checks that "Hello World" does NOT exist within any h1 tag
$results->dontSee('Hello World', 'h1');
```


The **seeElement()** and **dontSeeElement()** are very similar to the previous methods, but do not look at the values of the elements. Instead, they simply check that the elements exist on the page:

```
// Check that an element with class 'notice' exists
$results->seeElement('.notice');
```

```
// Check that an element with id 'title' exists
$results->seeElement('#title')
// Verify that an element with id 'title' does NOT exist
$results->dontSeeElement('#title');
```


You can use **seeLink()** to ensure that a link appears on the page with the specified text:

```
// Check that a link exists with 'Upgrade Account' as the text::
$results->seeLink('Upgrade Account');
// Check that a link exists with 'Upgrade Account' as the text, A
$results->seeLink('Upgrade Account', '.upsell');
```



The **seeInField()** method checks for any input tags exist with the name and value:

```
// Check that an input exists named 'user' with the value 'John S
$results->seeInField('user', 'John Snow');
// Check a multi-dimensional input
$results->seeInField('user[name]', 'John Snow');
```



Finally, you can check if a checkbox exists and is checked with the **seeCheckboxIsChecked()** method:

```
// Check if checkbox is checked with class of 'foo'
$results->seeCheckboxIsChecked('.foo');
// Check if checkbox with id of 'bar' is checked
$results->seeCheckboxIsChecked('#bar');
```


HTTP Feature Testing

Feature testing allows you to view the results of a single call to your application. This might be returning the results of a single web form, hitting an API endpoint, and more. This is handy because it allows you to test the entire life-cycle of a single request, ensuring that the routing works, the response is the correct format, analyze the results, and more.

- [The Test Class](#)
- [Requesting A Page](#)
 - [Setting Different Routes](#)
 - [Setting Session Values](#)
 - [Bypassing Events](#)
- [Testing the Response](#)
 - [Checking Response Status](#)
 - [Session Assertions](#)
 - [Header Assertions](#)
 - [Cookie Assertions](#)
 - [DOM Assertions](#)
 - [Working With JSON](#)
 - [Working With XML](#)

[The Test Class](#)

Feature testing requires that all of your test classes extend the `CodeIgniter\Test\FeatureTestCase` class. Since this extends [CIDatabaseTestCase](#) you must always ensure that `parent::setUp()` and `parent::tearDown()` are called before you take your actions.

```
<?php namespace App;  
  
use CodeIgniter\Test\FeatureTestCase;
```

```

class TestFoo extends FeatureTestCase
{
    public function setUp()
    {
        parent::setUp();
    }

    public function tearDown()
    {
        parent::tearDown();
    }
}

```

Requesting A Page

Essentially, the FeatureTestCase simply allows you to call an endpoint on your application and get the results back. to do this, you use the `call()` method. The first parameter is the HTTP method to use (most frequently either GET or POST). The second parameter is the path on your site to test. The third parameter accepts an array that is used to populate the the superglobal variables for the HTTP verb you are using. So, a method of **GET** would have the `$_GET` variable populated, while a **post** request would have the `$_POST` array populated.

```

// Get a simple page
$result = $this->call('get', site_url());

// Submit a form
$result = $this->call('post', site_url('contact'), [
    'name' => 'Fred Flintstone',
    'email' => 'flintyfred@example.com'
]);

```

Shorthand methods for each of the HTTP verbs exist to ease typing and make things clearer:

```

$this->get($path, $params);
$this->post($path, $params);
$this->put($path, $params);
$this->patch($path, $params);
$this->delete($path, $params);
$this->options($path, $params);

```

Note

The \$params array does not make sense for every HTTP verb, but is included for consistency.

Setting Different Routes

You can use a custom collection of routes by passing an array of routes into the withRoutes() method. This will override any existing routes in the system:

```
$routes = [  
    'users' => 'UserController::list'  
];  
  
$result = $this->withRoutes($routes)  
    ->get('users');
```

Setting Session Values

You can set custom session values to use during a single test with the withSession() method. This takes an array of key/value pairs that should exist within the \$_SESSION variable when this request is made. This is handy for testing authentication and more.

```
$values = [  
    'logged_in' => 123  
];  
  
$result = $this->withSession($values)  
    ->get('admin');
```

Bypassing Events

Events are handy to use in your application, but can be problematic during testing. Especially events that are used to send out emails. You can tell the system to skip any event handling with the skipEvents() method:

```
$result = $this->skipEvents()
```

```
->post('users', $userInfo);
```

Testing the Response

Once you've performed a `call()` and have results, there are a number of new assertions that you can use in your tests.

Note

The Response object is publicly available at `$result->response`. You can use that instance to perform other assertions against, if needed.

Checking Response Status

isOK()

Returns a boolean true/false based on whether the response is perceived to be "ok". This is primarily determined by a response status code in the 200 or 300's.

```
if ($result->isOK())  
{  
    ...  
}
```

assertOK()

This assertion simply uses the **isOK()** method to test a response.

```
$this->assertOK();
```

isRedirect()

Returns a boolean true/false based on whether the response is a redirected response.

```
if ($result->isRedirect())  
{
```

```
    ...  
}
```

assertRedirect()

Asserts that the Response is an instance of RedirectResponse.

```
$this->assertRedirect();
```

assertStatus(int \$code)

Asserts that the HTTP status code returned matches \$code.

```
$this->assertStatus(403);
```

Session Assertions

assertSessionHas(string \$key, \$value = null)

Asserts that a value exists in the resulting session. If \$value is passed, will also assert that the variable's value matches what was specified.

```
$this->assertSessionHas('logged_in', 123);
```

assertSessionMissing(string \$key)

Asserts that the resulting session does not include the specified \$key.

```
$this->assertSessionMissin('logged_in');
```

Header Assertions

assertHeader(string \$key, \$value = null)

Asserts that a header named \$key exists in the response. If \$value is not empty, will also assert that the values match.

```
$this->assertHeader('Content-Type', 'text/html');
```

assertHeaderMissing(string \$key)

Asserts that a header name **\$key** does not exist in the response.

```
$this->assertHeader('Accepts');
```

Cookie Assertions

assertCookie(string \$key, \$value = null, string \$prefix = “)

Asserts that a cookie named **\$key** exists in the response. If **\$value** is not empty, will also assert that the values match. You can set the cookie prefix, if needed, by passing it in as the third parameter.

```
$this->assertCookie('foo', 'bar');
```

assertCookieMissing(string \$key)

Asserts that a cookie named **\$key** does not exist in the response.

```
$this->assertCookieMissing('ci_session');
```

assertCookieExpired(string \$key, string \$prefix = “)

Asserts that a cookie named **\$key** exists, but has expired. You can set the cookie prefix, if needed, by passing it in as the second parameter.

```
$this->assertCookieExpired('foo');
```

DOM Assertions

You can perform tests to see if specific elements/text/etc exist with the body of the response with the following assertions.

assertSee(string \$search = null, string \$element = null)

Asserts that text/HTML is on the page, either by itself or - more specifically - within a tag, as specified by type, class, or id:

```
// Check that "Hello World" is on the page  
$this->assertSee('Hello World');  
// Check that "Hello World" is within an h1 tag
```

```
$this->assertS('Hello World', 'h1');  
// Check that "Hello World" is within an element with the "notice"  
$this->assertS('Hello World', '.notice');  
// Check that "Hello World" is within an element with id of "title"  
$this->assertS('Hello World', '#title');
```

assertDontSee(string \$search = null, string \$element = null)

Asserts the exact opposite of the **assertSee()** method:

```
// Checks that "Hello World" does NOT exist on the page  
$results->dontSee('Hello World');  
// Checks that "Hello World" does NOT exist within any h1 tag  
$results->dontSee('Hello World', 'h1');
```

assertSeeElement(string \$search)

Similar to **assertSee()**, however this only checks for an existing element. It does not check for specific text:

```
// Check that an element with class 'notice' exists  
$results->seeElement('.notice');  
// Check that an element with id 'title' exists  
$results->seeElement('#title')
```

assertDontSeeElement(string \$search)

Similar to **assertSee()**, however this only checks for an existing element that is missing. It does not check for specific text:

```
// Verify that an element with id 'title' does NOT exist  
$results->dontSeeElement('#title');
```

assertSeeLink(string \$text, string \$details=null)

Asserts that an anchor tag is found with matching **\$text** as the body of the tag:

```
// Check that a link exists with 'Upgrade Account' as the text::  
$results->seeLink('Upgrade Account');  
// Check that a link exists with 'Upgrade Account' as the text, A  
$results->seeLink('Upgrade Account', '.upsell');
```

assertSeeInField(string \$field, string \$value=null)

Asserts that an input tag exists with the name and value:

```
// Check that an input exists named 'user' with the value 'John S  
$results->seeInField('user', 'John Snow');  
// Check a multi-dimensional input  
$results->seeInField('user[name]', 'John Snow');
```

Working With JSON

Responses will frequently contain JSON responses, especially when working with API methods. The following methods can help to test the responses.

getJSON()

This method will return the body of the response as a JSON string:

```
// Response body is this:  
['foo' => 'bar']  
  
$json = $result->getJSON();  
  
// $json is this:  
{  
    "foo": "bar"  
}
```

Note

Be aware that the JSON string will be pretty-printed in the result.

assertJSONFragment(array \$fragment)

Asserts that \$fragment is found within the JSON response. It does not need to match the entire JSON value.

```
// Response body is this:
```



```
[
    'config' => ['key-a', 'key-b']
]

// Is true
$this->assertJSONFragment(['config' => ['key-a']]);
```

Note

This simply uses phpUnit's own [assertArraySubset\(\)](https://phpunit.readthedocs.io/en/7.2/assertions.html#assertarraysubset) [https://phpunit.readthedocs.io/en/7.2/assertions.html#assertarraysubset] method to do the comparison.

assertJSONExact(\$test)

Similar to **assertJSONFragment()**, but checks the entire JSON response to ensure exact matches.

[Working With XML](#)

getXML()

If your application returns XML, you can retrieve it through this method.

Benchmarking

CodeIgniter provides two separate tools to help you benchmark your code and test different options: the Timer and the Iterator. The Timer allows you to easily calculate the time between two points in the execution of your script. The Iterator allows you to setup several variations and run those tests, recording performance and memory statistics to help you decide which version is the best.

The Timer class is always active, being started from the moment the framework is invoked until right before sending the output to the user, enabling a very accurate timing of the entire system execution.

- [Using the Timer](#)
 - [Viewing Your Benchmark Points](#)
 - [Displaying Execution Time](#)
- [Using the Iterator](#)
 - [Creating Tasks To Run](#)
 - [Running the Tasks](#)

[Using the Timer](#)

With the Timer, you can measure the time between two moments in the execution of your application. This makes it simple to measure the performance of different aspects of your application. All measurement is done using the `start()` and `stop()` methods.

The `start()` methods takes a single parameter: the name of this timer. You can use any string as the name of the timer. It is only used for you to reference later to know which measurement is which:

```
$benchmark = \Config\Services::timer();
```

```
$benchmark->start('render view');
```

The `stop()` method takes the name of the timer that you want to stop as the only parameter, also:

```
$benchmark->stop('render view');
```

The name is not case-sensitive, but otherwise must match the name you gave it when you started the timer.

Alternatively, you can use the [global function](#) `timer()` to start and stop timers:

```
// Start the timer
timer('render view');
// Stop a running timer,
// if one of this name has been started
timer('render view');
```

Viewing Your Benchmark Points

When your application runs, all of the timers that you have set are collected by the Timer class. It does not automatically display them, though. You can retrieve all of your timers by calling the `getTimers()` method. This returns an array of benchmark information, including start, end, and duration:

```
$timers = $benchmark->getTimers();

// Timers =
[
    'render view' => [
        'start'    => 1234567890,
        'end'      => 1345678920,
        'duration' => 15.4315      // number of seconds
    ]
]
```

You can change the precision of the calculated duration by passing in the number of decimal places you want shown as the only parameter. The default value is 4 numbers behind the decimal point:

```
$timers = $benchmark->getTimers(6);
```

The timers are automatically displayed in the [Debug Toolbar](#).

Displaying Execution Time

While the `getTimers()` method will give you the raw data for all of the timers in your project, you can retrieve the duration of a single timer, in seconds, with the `getElapsedTime()` method. The first parameter is the name of the timer to display. The second is the number of decimal places to display. This defaults to 4:

```
echo timer()->getElapsedTime('render view');  
// Displays: 0.0234
```

Using the Iterator

The Iterator is a simple tool that is designed to allow you to try out multiple variations on a solution to see the speed differences and different memory usage patterns. You can add any number of “tasks” for it to run and the class will run the task hundreds or thousands of times to get a clearer picture of performance. The results can then be retrieved and used by your script, or displayed as an HTML table.

Creating Tasks To Run

Tasks are defined within Closures. Any output the task creates will be discarded automatically. They are added to the Iterator class through the `add()` method. The first parameter is a name you want to refer to this test by. The second parameter is the Closure, itself:

```
$iterator = new \CodeIgniter\Benchmark\Iterator();  
  
// Add a new task  
$iterator->add('single_concat', function()  
{  
    $str = 'Some basic'. 'little'. 'string concatenatio  
});  
  
// Add another task
```

```
$iterator->add('double', function($a='little')
{
    $str = "Some basic {$little} string test.";
})
};
```

Running the Tasks

Once you've added the tasks to run, you can use the `run()` method to loop over the tasks many times. By default, it will run each task 1000 times. This is probably sufficient for most simple tests. If you need to run the tests more times than that, you can pass the number as the first parameter:

```
// Run the tests 3000 times.
$iterator->run(3000);
```

Once it has run, it will return an HTML table with the results of the test. If you don't want the results displayed, you can pass in *false* as the second parameter:

```
// Don't display the results.
$iterator->run(1000, false);
```

Debugging Your Application

- [Replace var_dump](#)
 - [Enabling Kint](#)
 - [Using Kint](#)
- [The Debug Toolbar](#)
 - [Enabling the Toolbar](#)
 - [Setting Benchmark Points](#)
 - [Creating Custom Collectors](#)

[Replace var_dump](#)

While using XDebug and a good IDE can be indispensable to debug your application, sometimes a quick `var_dump()` is all you need. CodeIgniter makes that even better by bundling in the excellent [Kint](#)

[<https://raveren.github.io/kint/>] debugging tool for PHP. This goes way beyond your usual tool, providing many alternate pieces of data, like formatting timestamps into recognizable dates, showing you hexcodes as colors, display array data like a table for easy reading, and much, much more.

[Enabling Kint](#)

By default, Kint is enabled in **development** and **testing** environments only. This can be altered by modifying the `$useKint` value in the environment configuration section of the main **index.php** file:

```
$useKint = true;
```

[Using Kint](#)

`d()`

The `d()` method dumps all of the data it knows about the contents passed as the only parameter to the screen, and allows the script to continue executing:

```
d($_SERVER);
```

dd()

This method is identical to `d()`, except that it also `dies()` and no further code is executed this request.

trace()

This provides a backtrace to the current execution point, with Kint's own unique spin:

```
Kint::trace();
```

For more information, see [Kint's page](https://kint-php.github.io/kint/) [https://kint-php.github.io/kint/].

The Debug Toolbar

The Debug Toolbar provides at-a-glance information about the current page request, including benchmark results, queries you have run, request and response data, and more. This can all prove very useful during development to help you debug and optimize.

Note

The Debug Toolbar is still under construction with several planned features not yet implemented.

Enabling the Toolbar

The toolbar is enabled by default in any environment *except* production. It will be shown whenever the constant `CI_DEBUG` is defined and its value is positive. This is defined in the boot files (i.e. `app/Config/Boot/development.php`) and can be modified there to determine

what environments it shows itself in.

The toolbar itself is displayed as an [After Filter](#). You can stop it from ever running by removing it from the `$globals` property of **app/Config/Filters.php**.

Choosing What to Show

CodeIgniter ships with several Collectors that, as the name implies, collect data to display on the toolbar. You can easily make your own to customize the toolbar. To determine which collectors are shown, again head over to the App configuration file:

```
public $toolbarCollectors = [  
    'CodeIgniter\Debug\Toolbar\Collectors\Timers',  
    'CodeIgniter\Debug\Toolbar\Collectors\Database',  
    'CodeIgniter\Debug\Toolbar\Collectors\Logs',  
    'CodeIgniter\Debug\Toolbar\Collectors\Views',  
    'CodeIgniter\Debug\Toolbar\Collectors\Cache',  
    'CodeIgniter\Debug\Toolbar\Collectors\Files',  
    'CodeIgniter\Debug\Toolbar\Collectors\Routes',  
];
```

Comment out any collectors that you do not want to show. Add custom Collectors here by providing the fully-qualified class name. The exact collectors that appear here will affect which tabs are shown, as well as what information is shown on the Timeline.

Note

Some tabs, like Database and Logs, will only display when they have content to show. Otherwise, they are removed to help out on smaller displays.

The Collectors that ship with CodeIgniter are:

- **Timers** collects all of the benchmark data, both by the system and by your application.

- **Database** Displays a list of queries that all database connections have performed, and their execution time.
- **Logs** Any information that was logged will be displayed here. In long-running systems, or systems with many items being logged, this can cause memory issues and should be disabled.
- **Views** Displays render time for views on the timeline, and shows any data passed to the views on a separate tab.
- **Cache** Will display information about cache hits and misses, and execution times.
- **Files** displays a list of all files that have been loaded during this request.
- **Routes** displays information about the current route and all routes defined in the system.

Setting Benchmark Points

In order for the Profiler to compile and display your benchmark data you must name your mark points using specific syntax.

Please read the information on setting Benchmark points in the [Benchmark Library](#) page.

Creating Custom Collectors

Creating custom collectors is a straightforward task. You create a new class, fully-namespaced so that the autoloader can locate it, that extends `CodeIgniter\Debug\Toolbar\Collectors\BaseCollector`. This provides a number of methods that you can override, and has four required class properties that you must correctly set depending on how you want the Collector to work

```
<?php namespace MyNamespace;

use CodeIgniter\Debug\Toolbar\Collectors\BaseCollector;

class MyCollector extends BaseCollector
{
    protected $hasTimeline    = false;

    protected $hasTabContent = false;
```

```
        protected $hasVarData    = false;

        protected $title         = '';
    }
```

\$hasTimeline should be set to true for any Collector that wants to display information in the toolbar's timeline. If this is true, you will need to implement the `formatTimelineData()` method to format and return the data for display.

\$hasTabContent should be true if the Collector wants to display its own tab with custom content. If this is true, you will need to provide a `$title`, implement the `display()` method to render out tab's contents, and might need to implement the `getTitleDetails()` method if you want to display additional information just to the right of the tab content's title.

\$hasVarData should be true if this Collector wants to add additional data to the Vars tab. If this is true, you will need to implement the `getVarData()` method.

\$title is displayed on open tabs.

Displaying a Toolbar Tab

To display a toolbar tab you must:

1. Fill in `$title` with the text displayed as both the toolbar title and the tab header.
2. Set `$hasTabContent` to true.
3. Implement the `display()` method.
4. Optionally, implement the `getTitleDetails()` method.

The `display()` creates the HTML that is displayed within the tab itself. It does not need to worry about the title of the tab, as that is automatically handled by the toolbar. It should return a string of HTML.

The `getTitleDetails()` method should return a string that is displayed just to the right of the tab's title. it can be used to provide additional overview

information. For example, the Database tab displays the total number of queries across all connections, while the Files tab displays the total number of files.


Providing Timeline Data

To provide information to be displayed in the Timeline you must:

1. Set `$hasTimeline` to `true`.
2. Implement the `formatTimelineData()` method.

The `formatTimelineData()` method must return an array of arrays formatted in a way that the timeline can use it to sort it correctly and display the correct information. The inner arrays must include the following information:

```
$data[] = [
    'name'      => '',           // Name displayed on the left of t
    'component' => '',           // Name of the Component listed in
    'start'     => 0.00,         // start time, like microtime(true)
    'duration'  => 0.00         // duration, like microtime(true)
];
```



Providing Vars

To add data to the Vars tab you must:

1. Set `$hasVarData` to `true`
2. Implement `getVarData()` method.

The `getVarData()` method should return an array containing arrays of key/value pairs to display. The name of the outer array's key is the name of the section on the Vars tab:

```
$data = [
    'section 1' => [
        'foo' => 'bar',
        'bar' => 'baz'
    ],
    'section 2' => [
```

```
        'foo' => 'bar',  
        'bar' => 'baz'  
    ]  
];
```

Command Line Usage

CodeIgniter 4 can also be used with command line programs.

- [Running via the Command Line](#)
- [Custom CLI Commands](#)
- [CLI Library](#)
- [CLIRequest Class](#)

Running via the Command Line

As well as calling an applications [Controllers](#) via the URL in a browser they can also be loaded via the command-line interface (CLI).

- [What is the CLI?](#)
- [Why run via the command-line?](#)
- [Let's try it: Hello World!](#)
- [That's the basics!](#)
 - [CLI-Only Routing](#)
 - [The CLI Library](#)

[What is the CLI?](#)

The command-line interface is a text-based method of interacting with computers. For more information, check the [Wikipedia article](#)

[http://en.wikipedia.org/wiki/Command-line_interface].

[Why run via the command-line?](#)

There are many reasons for running CodeIgniter from the command-line, but they are not always obvious.

- Run your cron-jobs without needing to use *wget* or *curl*.
- Make your cron-jobs inaccessible from being loaded in the URL by checking the return value of `is_cli()`.
- Make interactive “tasks” that can do things like set permissions, prune cache folders, run backups, etc.
- Integrate with other applications in other languages. For example, a random C++ script could call one command and run code in your models!

Let's try it: Hello World!

Let's create a simple controller so you can see it in action. Using your text editor, create a file called Tools.php, and put the following code in it:

```
<?php namespace App\Controller;

use CodeIgniter\Controller;

class Tools extends Controller {

    public function message($to = 'World')
    {
        echo "Hello {$to}!".PHP_EOL;
    }
}
```

Then save the file to your **app/Controllers/** directory.

Now normally you would visit the your site using a URL similar to this:

example.com/index.php/tools/message/to

Instead, we are going to open Terminal in Mac/Linux or go to Run > “cmd” in Windows and navigate to our CodeIgniter project's web root.

```
$ cd /path/to/project/public
$ php index.php tools message
```

If you did it right, you should see *Hello World!* printed.

```
$ php index.php tools message "John Smith"
```

Here we are passing it a argument in the same way that URL parameters work. “John Smith” is passed as a argument and output is:

Hello John Smith!

That's the basics!

That, in a nutshell, is all there is to know about controllers on the command

line. Remember that this is just a normal controller, so routing and `_remap()` works fine.

However, CodeIgniter provides additional tools to make creating CLI-accessible scripts even more pleasant, include CLI-only routing, and a library that helps you with CLI-only tools.

[CLI-Only Routing](#)

In your **Routes.php** file you can create routes that are only accessible from the CLI as easily as you would create any other route. Instead of using the `get()`, `post()`, or similar method, you would use the `cli()` method. Everything else works exactly like a normal route definition:

```
$routes->cli('tools/message/(:segment)', 'Tools::message/$1');
```

For more information, see the [Routes](#) page.

[The CLI Library](#)

The CLI library makes working with the CLI interface simple. It provides easy ways to output text in multiple colors to the terminal window. It also allows you to prompt a user for information, making it easy to build flexible, smart tools.

See the [CLI Library](#) page for detailed information.

Custom CLI Commands

While the ability to use cli commands like any other route is convenient, you might find times where you need a little something different. That's where CLI Commands come in. They are simple classes that do not need to have routes defined for, making them perfect for building tools that developers can use to make their jobs simpler, whether by handling migrations or database seeding, checking cronjob status, or even building out custom code generators for your company.

- [Running Commands](#)
- [Using Help Command](#)
- [Creating New Commands](#)
 - [File Location](#)
 - [An Example Command](#)
- [BaseCommand](#)

[Running Commands](#)

Commands are run from the command line, in the root directory. The same one that holds the **/app** and **/system** directories. A custom script, **spark** has been provided that is used to run any of the cli commands:

```
> php spark
```

When called without specifying a command, a simple help page is displayed that also provides a list of available commands. You should pass the name of the command as the first argument to run that command:

```
> php spark migrate
```

Some commands take additional arguments, which should be provided

directly after the command, separated by spaces:

```
> php spark db:seed DevUserSeeder
```

For all of the commands CodeIgniter provides, if you do not provide the required arguments, you will be prompted for the information it needs to run correctly:

```
> php spark migrate:version  
> Version?
```

Using Help Command

You can get help about any CLI command using the help command as follows:

```
> php spark help db:seed
```

Creating New Commands

You can very easily create new commands to use in your own development. Each class must be in its own file, and must extend `CodeIgniter\CLI\BaseCommand`, and implement the `run()` method.

The following properties should be used in order to get listed in CLI commands and to add help functionality to your command:

- (`$group`): a string to describe the group the command is lumped under when listing commands. For example (Database)
- (`$name`): a string to describe the command's name. For example (migrate:create)
- (`$description`): a string to describe the command. For example (Creates a new migration file.)
- (`$usage`): a string to describe the command usage. For example (migrate:create [migration_name] [Options])
- (`$arguments`): an array of strings to describe each command argument. For example ('migration_name' => 'The migration file name')
- (`$options`): an array of strings to describe each command option. For

example ('-n' => 'Set migration namespace')

Help description will be automatically generated according to the above parameters.

[File Location](#)

Commands must be stored within a directory named **Commands**. However, that directory can be located anywhere that the [Autoloader](#) can locate it. This could be in **/app/Commands**, or a directory that you keep commands in to use in all of your project development, like **Acme/Commands**.

Note

When the commands are executed, the full CodeIgniter cli environment has been loaded, making it possible to get environment information, path information, and to use any of the tools you would use when making a Controller.

[An Example Command](#)

Let's step through an example command whose only function is to report basic information about the application itself, for demonstration purposes. Start by creating a new file at **/app/Commands/AppInfo.php**. It should contain the following code:

```
<?php namespace App\Commands;

use CodeIgniter\CLI\BaseCommand;

class AppInfo extends BaseCommand
{
    protected $group          = 'demo';
    protected $name           = 'app:info';
    protected $description    = 'Displays basic application informat

    public function run(array $params)
    {
```

```
}  
}
```

If you run the **list** command, you will see the new command listed under its own demo group. If you take a close look, you should see how this works fairly easily. The `$group` property simply tells it how to organize this command with all of the other commands that exist, telling it what heading to list it under.

The `$name` property is the name this command can be called by. The only requirement is that it must not contain a space, and all characters must be valid on the command line itself. By convention, though, commands are lowercase, with further grouping of commands being done by using a colon with the command name itself. This helps keep multiple commands from having naming collisions.

The final property, `$description` is a short string that is displayed in the **list** command and should describe what the command does.

run()

The `run()` method is the method that is called when the command is being run. The `$params` array is a list of any cli arguments after the command name for your use. If the cli string was:

```
> php spark foo bar baz
```

Then **foo** is the command name, and the `$params` array would be:

```
$params = ['bar', 'baz'];
```

This can also be accessed through the [CLI](#) library, but this already has your command removed from the string. These parameters can be used to customize how your scripts behave.

Our demo command might have a run method something like:

```
public function run(array $params)
```

```

{
    CLI::write('PHP Version: '. CLI::color(PHP_VERSION(), 'yellow')
    CLI::write('CI Version: '. CLI::color(CodeIgniter::CI_VERSION
    CLI::write('APPPATH: '. CLI::color(APPPATH, 'yellow'));
    CLI::write('SYSTEMPATH: '. CLI::color(SYSTEMPATH, 'yellow'));
    CLI::write('ROOTPATH: '. CLI::color(ROOTPATH, 'yellow'));
    CLI::write('Included files: '. CLI::color(count(get_included_
}

```

BaseCommand

The BaseCommand class that all commands must extend have a couple of helpful utility methods that you should be familiar with when creating your own commands. It also has a [Logger](#) available at **\$this->logger**.

class **CodeIgniterCLIBaseCommand**

call(string \$command[, array \$params=[]])

- Parameters:**
- **\$command** (string) – The name of another command to call.
 - **\$params** (array) – Additional cli arguments to make available to that command.

This method allows you to run other commands during the execution of your current command:

```

$this->call('command_one');
$this->call('command_two', $params);

```

showError(Exception \$e)

- Parameters:**
- **\$e** (Exception) – The exception to use for error reporting.

A convenience method to maintain a consistent and clear error output to the cli:

```

try
{
    . . .
}
catch (\Exception $e)

```

```
{
    $this->showError($e);
}
```

showHelp()

A method to show command help:
(usage,arguments,description,options)

getPad(\$array, \$pad)

Parameters:

- **\$array** (*Exception*) – The \$key => \$value array.
- **\$pad** (*Exception*) – The pad spaces.

A method to calculate padding for \$key => \$value array output. The padding can be used to output a will formatted table in CLI:

```
$pad = $this->getPad($this->options, 6);
foreach ($this->options as $option => $description)
{
    CLI::write($tab . CLI::color(str_pad($option, $pad)
```

// Output will be

```
-n          Set migration namespace
-r          override file
```

<

>

CLI Library

CodeIgniter's CLI library makes creating interactive command-line scripts simple, including:

- Prompting the user for more information
- Writing multi-colored text the terminal
- Beeping (be nice!)
- Showing progress bars during long tasks
- Wrapping long text lines to fit the window.

- [Initializing the Class](#)
- [Getting Input from the User](#)
- [Providing Feedback](#)

[Initializing the Class](#)

You do not need to create an instance of the CLI library, since all of it's methods are static. Instead, you simply need to ensure your controller can locate it via a use statement above your class:

```
<?php namespace App\Controllers;

use CodeIgniter\CLI\CLI;

class MyController extends \CodeIgniter\Controller
{
    . . .
}
```

The class is automatically initialized when the file is loaded the first time.

[Getting Input from the User](#)

Sometimes you need to ask the user for more information. They might not have provided optional command-line arguments, or the script may have encountered an existing file and needs confirmation before overwriting. This is handled with the `prompt()` method.

You can provide a question by passing it in as the first parameter:

```
$color = CLI::prompt('What is your favorite color?');
```

You can provide a default answer that will be used if the user just hits enter by passing the default in the second parameter:

```
$color = CLI::prompt('What is your favorite color?', 'blue');
```

You can restrict the acceptable answers by passing in an array of allowed answers as the second parameter:

```
$overwrite = CLI::prompt('File exists. Overwrite?', ['y', 'n']);
```

Finally, you can pass validation rules to the answer input as the third parameter:

```
$email = CLI::prompt('What is your email?', null, 'required|valid  
< >
```

Providing Feedback

write()

Several methods are provided for you to provide feedback to your users. This can be as simple as a single status update or a complex table of information that wraps to the user's terminal window. At the core of this is the `write()` method which takes the string to output as the first parameter:

```
CLI::write('The rain in Spain falls mainly on the plains.');
```

You can change the color of the text by passing in a color name as the first parameter:

```
CLI::write('File created.', 'green');
```


This could be used to differentiate messages by status, or create ‘headers’ by using a different color. You can even set background colors by passing the color name in as the third parameter:

```
CLI::write('File overwritten.', 'light_red', 'dark_gray');
```

The following colors are available:

- black
- dark_gray
- blue
- dark_blue
- light_blue
- green
- light_green
- cyan
- light_cyan
- red
- light_red
- purple
- light_purple
- light_yellow
- yellow
- light_gray
- white

color()

While the `write()` command will write a single line to the terminal, ending it with a EOL character, you can use the `color()` method to make a string fragment that can be used in the same way, except that it will not force an EOL after printing. This allows you to create multiple outputs on the same row. Or, more commonly, you can use it inside of a `write()` method to create a string of a different color inside:

```
CLI::write("fileA \t". CLI::color('/path/to/file', 'white'), 'yel
```



This example would write a single line to the window, with `fileA` in yellow,

followed by a tab, and then `/path/to/file` in white text.

error()


If you need to output errors, you should use the appropriately named `error()` method. This writes light-red text to `STDERR`, instead of `STDOUT`, like `write()` and `color()` do. This can be useful if you have scripts watching for errors so they don't have to sift through all of the information, only the actual error messages. You use it exactly as you would the `write()` method:

```
CLI::error('Cannot write to file: '. $file);
```

wrap()

This command will take a string, start printing it on the current line, and wrap it to a set length on new lines. This might be useful when displaying a list of options with descriptions that you want to wrap in the current window and not go off screen:

```
CLI::color("task1\t", 'yellow');
CLI::wrap("Some long description goes here that might be longer t
```



By default the string will wrap at the terminal width. Windows currently doesn't provide a way to determine the window size, so we default to 80 characters. If you want to restrict the width to something shorter that you can be pretty sure fits within the window, pass the maximum line-length as the second parameter. This will break the string at the nearest word barrier so that words are not broken.

```
// Wrap the text at max 20 characters wide
CLI::wrap($description, 20);
```

You may find that you want a column on the left of titles, files, or tasks, while you want a column of text on the right with their descriptions. By default, this will wrap back to the left edge of the window, which doesn't allow things to line up in columns. In cases like this, you can pass in a number of spaces to pad every line after the first line, so that you will have a crisp column edge on the left:

```

// Determine the maximum length of all titles
// to determine the width of the left column
$maxlen = max(array_map('strlen', $titles));

for ($i=0; $i <= count($titles); $i++)
{
    CLI::write(
        // Display the title on the left of the row
        $title[$i].' '.
        // Wrap the descriptions in a right-hand column
        // with its left side 3 characters wider than
        // the longest item on the left.
        CLI::wrap($descriptions[$i], 40, $maxlen+3)
    );
}

```

Would create something like this:

```

task1a      Lorem Ipsum is simply dummy
             text of the printing and typesetting
             industry.
task1abc    Lorem Ipsum has been the industry's
             standard dummy text ever since the

```

newLine()

The `newLine()` method displays a blank line to the user. It does not take any parameters:

```
CLI::newLine();
```

clearScreen()

You can clear the current terminal window with the `clearScreen()` method. In most versions of Windows, this will simply insert 40 blank lines since Windows doesn't support this feature. Windows 10 bash integration should change this:

```
CLI::clearScreen();
```

showProgress()

If you have a long-running task that you would like to keep the user updated

with the progress, you can use the `showProgress()` method which displays something like the following:

```
[####.....] 40% Complete
```

This block is animated in place for a very nice effect.

To use it, pass in the current step as the first parameter, and the total number of steps as the second parameter. The percent complete and the length of the display will be determined based on that number. When you are done, pass `false` as the first parameter and the progress bar will be removed.

```
$totalSteps = count($tasks);  
$currStep   = 1;
```

```
foreach ($tasks as $task)  
{  
    CLI::showProgress($currStep++, $totalSteps);  
    $task->run();  
}
```

```
// Done, so erase it...  
CLI::showProgress(false);
```

table()

```
$thead = ['ID', 'Title', 'Updated At', 'Active'];  
$tbody = [  
    [7, 'A great item title', '2017-11-15 10:35:02', 1],  
    [8, 'Another great item title', '2017-11-16 13:46:54', 0]  
];
```

```
CLI::table($tbody, $thead);
```

```
+-----+-----+-----+-----+  
| ID | Title                               | Updated At           | Active |  
+-----+-----+-----+-----+  
| 7  | A great item title                 | 2017-11-16 10:35:02 | 1      |  
| 8  | Another great item title           | 2017-11-16 13:46:54 | 0      |  
+-----+-----+-----+-----+
```

wait()

Waits a certain number of seconds, optionally showing a wait message and

waiting for a key press.

```
// wait for specified interval, with countdown displayed  
CLI::wait($seconds, true);  
  
// show continuation message and wait for input  
CLI::wait(0, false);  
  
// wait for specified interval  
CLI::wait($seconds, false);
```

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

CLIRequest Class

If a request comes from a command line invocation, the request object is actually a `CLIRequest`. It behaves the same as a [conventional request](#) but adds some accessor methods for convenience.

Additional Accessors

`getSegments()`

Returns an array of the command line arguments deemed to be part of a path:

```
// command line: php index.php users 21 profile -foo bar  
echo $request->getSegments(); // ['users', '21', 'profile']
```

`getPath()`

Returns the reconstructed path as a string:

```
// command line: php index.php users 21 profile -foo bar  
echo $request->getPath(); // users/21/profile
```

`getOptions()`

Returns an array of the command line arguments deemed to be options:

```
// command line: php index.php users 21 profile -foo bar  
echo $request->getOptions(); // ['foo' => 'bar']
```

`getOption($which)`

Returns the value of a specific command line argument deemed to be an option:

```
// command line: php index.php users 21 profile -foo bar  
echo $request->getOption('foo'); // bar  
echo $request->getOption('notthere'); // NULL
```

getOptionString()

Returns the reconstructed command line string for the options:

```
// command line: php index.php users 21 profile -foo bar  
echo $request->getOptionPath(); // -foo bar
```

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Extending CodeIgniter

CodeIgniter 4 has been designed to be easy to extend or build upon.

- [Creating Core System Classes](#)
- [Events](#)
- [Contributing to CodeIgniter](#)

Creating Core System Classes

Every time CodeIgniter runs there are several base classes that are initialized automatically as part of the core framework. It is possible, however, to swap any of the core system classes with your own version or even just extend the core versions.

Most users will never have any need to do this, but the option to replace or extend them does exist for those who would like to significantly alter the CodeIgniter core.

Note

Messing with a core system class has a lot of implications, so make sure you know what you are doing before attempting it.

System Class List

The following is a list of the core system files that are invoked every time CodeIgniter runs:

- Config\Services
- CodeIgniter\Autoloader\Autoloader
- CodeIgniter\Config\DotEnv
- CodeIgniter\Controller
- CodeIgniter\Debug\Exceptions
- CodeIgniter\Debug\Timer
- CodeIgniter\Events\Events
- CodeIgniter\HTTP\CLIRequest (if launched from command line only)
- CodeIgniter\HTTP\IncomingRequest (if launched over HTTP)
- CodeIgniter\HTTP\Request
- CodeIgniter\HTTP\Response

- CodeIgniter\HTTP\Message
- CodeIgniter\Log\Logger
- CodeIgniter\Log\Handlers\BaseHandler
- CodeIgniter\Log\Handlers\FileHandler
- CodeIgniter\Router\RouteCollection
- CodeIgniter\Router\Router
- CodeIgniter\Security\Security
- CodeIgniter\View\View
- CodeIgniter\View\Escaper

Replacing Core Classes

To use one of your own system classes instead of a default one, ensure that the [Autoloader](#) can find your class, that your new class extends the appropriate interface, and modify the appropriate [Service](#) to load your class in place of the core class.

For example, if you have a new App\Libraries\RouteCollection class that you would like to use in place of the core system class, you would create your class like this:

```
<?php namespace App\Libraries;

use CodeIgniter\Router\RouteCollectionInterface;

class RouteCollection implements RouteCollectionInterface
{
}
```

Then you would modify the routes service to load your class instead:

```
public static function routes($getShared = false)
{
    if (! $getShared)
    {
        return new \App\Libraries\RouteCollection();
    }

    return static::getSharedInstance('routes');
}
```

Extending Core Classes

If all you need to is add some functionality to an existing library - perhaps add a method or two - then it's overkill to recreate the entire library. In this case it's better to simply extend the class. Extending the class is nearly identical to replacing a class with a one exception:

- The class declaration must extend the parent class.

For example, to extend the native RouteCollection class, you would declare your class with:

```
<?php namespace App\Libraries;

use CodeIgniter\Router\RouteCollection;

class RouteCollection extends RouteCollection
{
}
```

If you need to use a constructor in your class make sure you extend the parent constructor:

```
<?php namespace App\Libraries;

use CodeIgniter\Router\RouteCollection;

class RouteCollection extends RouteCollection
{
    public function __construct()
    {
        parent::__construct();
    }
}
```

Tip: Any functions in your class that are named identically to the methods in the parent class will be used instead of the native ones (this is known as “method overriding”). This allows you to substantially alter the CodeIgniter core.

If you are extending the Controller core class, then be sure to extend your

new class in your application controller's constructors:

```
<?php namespace App\Controllers;  
  
use App\BaseController;  
  
class Home extends BaseController {  
  
}
```

Events

CodeIgniter's Events feature provides a means to tap into and modify the inner workings of the framework without hacking core files. When CodeIgniter runs it follows a specific execution process. There may be instances, however, when you'd like to cause some action to take place at a particular stage in the execution process. For example, you might want to run a script right before your controllers get loaded, or right after, or you might want to trigger one of your own scripts in some other location.

Events work on a *publish/subscribe* pattern, where an event, is triggered at some point during the script execution. Other scripts can “subscribe” to that event by registering with the Events class to let it know they want to perform an action when that event is triggered.

Enabling Events

Events are always enabled, and are available globally.

Defining an Event

Most events are defined within the **app/Config/Events.php** file. You can subscribe an action to an event with the Events class' `on()` method. The first parameter is the name of the event to subscribe to. The second parameter is a callable that will be run when that event is triggered:

```
use CodeIgniter\Events\Events;

Events::on('pre_system', ['MyClass', 'MyFunction']);
```

In this example, whenever the **pre_controller** event is executed, an instance of `MyClass` is created and the `MyFunction` method is run. Note that the second parameter can be *any* form of [callable](http://php.net/manual/en/function.is-callable.php) [http://php.net/manual/en/function.is-callable.php] that PHP recognizes:

```

// Call a standalone function
Events::on('pre_system', 'some_function');

// Call on an instance method
$user = new User();
Events::on('pre_system', [$user, 'some_method']);

// Call on a static method
Events::on('pre_system', 'SomeClass::someMethod');

// Use a Closure
Events::on('pre_system', function(...$params)
{
    . . .
});

```

Setting Priorities

Since multiple methods can be subscribed to a single event, you will need a way to define in what order those methods are called. You can do this by passing a priority value as the third parameter of the `on()` method. Lower values are executed first, with a value of 1 having the highest priority, and there being no limit on the lower values:

```
Events::on('post_controller_constructor', 'some_function', 25);
```

Any subscribers with the same priority will be executed in the order they were defined.

Three constants are defined for your use, that set some helpful ranges on the values. You are not required to use these but you might find they aid readability:

```

define('EVENT_PRIORITY_LOW', 200);
define('EVENT_PRIORITY_NORMAL', 100);
define('EVENT_PRIORITY_HIGH', 10);

```

Once sorted, all subscribers are executed in order. If any subscriber returns a boolean false value, then execution of the subscribers will stop.


Publishing your own Events

The Events library makes it simple for you to create events in your own code, also. To use this feature, you would simply need to call the `trigger()` method on the **Events** class with the name of the event:

```
\CodeIgniter\Events\Events::trigger('some_event');
```

You can pass any number of arguments to the subscribers by adding them as additional parameters. Subscribers will be given the arguments in the same order as defined:

```
\CodeIgniter\Events\Events::trigger('some_events', $foo, $bar, $baz);  
  
Events::on('some_event', function($foo, $bar, $baz) {  
    ...  
});
```



Simulating Events

During testing, you might not want the events to actually fire, as sending out hundreds of emails a day is both slow and counter-productive. You can tell the Events class to only simulate running the events with the `simulate()` method. When **true**, all events will be skipped over during the trigger method. Everything else will work as normal, though.

```
Events::simulate(true);
```

You can stop simulation by passing false:

```
Events::simulate(false);
```

Event Points

The following is a list of available event points within the CodeIgniter core code:

- **pre_system** Called very early during system execution. Only the benchmark and events class have been loaded at this point. No routing or other processes have happened.

- **post_controller_constructor** Called immediately after your controller is instantiated, but prior to any method calls happening.
- **post_system** Called after the final rendered page is sent to the browser, at the end of system execution after the finalized data is sent to the browser.

Contributing to CodeIgniter

CodeIgniter is a community driven project and accepts contributions of code and documentation from the community. These contributions are made in the form of Issues or [Pull Requests](https://help.github.com/articles/using-pull-requests/) on the [CodeIgniter4 repository](https://github.com/codeigniter4/CodeIgniter4) on GitHub.

Issues are a quick way to point out a bug. If you find a bug or documentation error in CodeIgniter then please check a few things first:

- There is not already an open Issue
- The issue has already been fixed (check the develop branch, or look for closed Issues)
- Is it something really obvious that you fix it yourself?

Reporting issues is helpful but an even better approach is to send a Pull Request, which is done by “Forking” the main repository and committing to your own copy. This will require you to use the version control system called Git.

Please see the [Contributing to CodeIgniter4](https://github.com/codeigniter4/CodeIgniter4/tree/develop/contributing)

[<https://github.com/codeigniter4/CodeIgniter4/tree/develop/contributing>] section of our code repository.

Support

Please note that GitHub is not for general support questions! If you are having trouble using a feature of CodeIgniter, ask for help on our [forums](http://forum.codeigniter.com/) instead.

If you are not sure whether you are using something correctly or if you have found a bug, again - please ask on the forums first.

Security

Did you find a security issue in CodeIgniter?

Please *don't* disclose it publicly, but e-mail us at security@codeigniter.com, or report it via our page on [HackerOne](https://hackerone.com/codeigniter) [https://hackerone.com/codeigniter].

If you've found a critical vulnerability, we'd be happy to credit you in our [ChangeLog](https://codeigniter4.github.io/userguide/changelog.html) [https://codeigniter4.github.io/userguide/changelog.html].

Tips for a Good Issue Report

Use a descriptive subject line (eg parser library chokes on commas) rather than a vague one (eg. your code broke).

Address a single issue in a report.

Identify the CodeIgniter version (eg 4.0.1) and the component if you know it (eg. parser library)

Explain what you expected to happen, and what did happen. Include error messages and stacktrace, if any.

Include short code segments if they help to explain. Use a pastebin or dropbox facility to include longer segments of code or screenshots - do not include them in the issue report itself. This means setting a reasonable expiry for those, until the issue is resolved or closed.

If you know how to fix the issue, you can do so in your own fork & branch, and submit a pull request. The issue report information above should be part of that.

If your issue report can describe the steps to reproduce the problem, that is great. If you can include a unit test that reproduces the problem, that is even better, as it gives whoever is fixing it a clearer target!

The MIT License (MIT)

Copyright (c) 2014-2019 British Columbia Institute of Technology

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Change Logs

Version 4.0.0-beta.1

Release Date: Unreleased

Highlights:

- New View Layouts provide simple way to create site site view templates.
- Fixed user guide CSS for proper wide table display
- Converted UploadedFile to use system messages
- Numerous database, migration & model bugs fixed
- Refactored unit testing for appstarter & framework distributions

New messages:

- Database.tableNotFound
- HTTP.uploadErr...

App changes:

- app/Config/Cache has new setting: database
- app/Views/welcome_message has logo tinted
- composer.json has a case correction
- env adds CI_ENVIRONMENT suggestion

[See all the changes.](#)

Version 4.0.0-alpha.5

Release Date: January 30, 2019

Alpha 5

Highlights:

- updated PHP dependency to 7.2
- new feature branches have been created for the email and queue modules,
so they don't impact the release of 4.0.0
- dropped several language messages that were unused (eg Migrations.missingTable)
and added some new (eg Migrations.invalidType)
- lots of bug fixes
- code coverage is up to 78%

[See all the changes.](#)

Version 4.0.0-alpha.4

Release Date: December 15, 2018

Next release of CodeIgniter4

Highlights:

- Refactor for consistency: folder application renamed to app;
constant BASEPATH renamed to SYSTEMPATH
- Debug toolbar gets its own config, history collector
- Numerous corrections and enhancements

[See all the changes.](#)

Version 4.0.0-alpha.3

Release Date: November 30, 2018

Next alpha release of CodeIgniter4

- Numerous bug fixes, across the framework
- Many missing features implemented, across the framework
- Code coverage is up to 72%
- CodeIgniter4 has been promoted to its own github organization. That is reflected in docs and comments.
- We have integrated a git pre-commit hook, which will apply the CI4 code sniffer rules, and attempt to fix them. We have run all the source files through it, and any “funny” code formatting is temporary until the rules are updated.
- We welcome Natan Felles, from Brazil, to the code developer team. He has proven to be passionate, dedicated and thorough :)

[See all the changes.](#)

Version 4.0.0-alpha.2

Release Date: Oct 26, 2018

Second alpha release of CodeIgniter4

- bug fixes
- features implemented
- tutorial revised

[See all the changes.](#)

Version 4.0.0-alpha.1

Release Date: September 28, 2018

Rewrite of the CodeIgniter framework

Non-code changes:

- User Guide adapted or rewritten
- [System message translations repository](https://github.com/bcit-ci/CodeIgniter4-translations) [https://github.com/bcit-ci/CodeIgniter4-translations]
- [Roadmap subforum](https://forum.codeigniter.com/forum-33.html) [https://forum.codeigniter.com/forum-33.html] for more

transparent planning

New core classes:

- CodeIgniter (bootstrap)
- Common (shared functions)
- ComposerScripts (integrate third party tools)
- Controller (base controller)
- Model (base model)
- Entity (entity encapsulation)

Some new, some old & some borrowed packages, all namespaced.

[See all the changes.](#)

Version 4.0-dev

Release Date: Not released

Next alpha release of CodeIgniter4

The list of changed files follows, with PR numbers shown.

PRs merged:

© Copyright 2014-2019 British Columbia Institute of Technology. Last updated on Mar 01, 2019. Created using [Sphinx](#) 1.4.5.

Version 4.0.0-alpha.5

Release Date: Jan 30, 2019

Next alpha release of CodeIgniter4

Highlights:

- added \$maxQueries setting to app/Config/Toolbar.php
- updated PHP dependency to 7.2
- new feature branches have been created for the email and queue modules, so they don't impact the release of 4.0.0
- dropped several language messages that were unused (eg Migrations.missingTable) and added some new (eg Migrations.invalidType)
- lots of bug fixes, especially for the database support
- provided filters (CSRF, Honeypot, DebugToolbar) have been moved from app/Filters/ to system/Filters/
- revisited the installation and tutorial sections of the user guide
- code coverage is at 77% ... getting ever closer to our target of 80% :)

We hope this will be the last alpha, and that the next pre-release will be our first beta ... fingers crossed!

The list of changed files follows, with PR numbers shown.

- admin/
 - starter/
 - README.md #1637
 - app/Config/Paths.php #1685
 - release-appstarter #1685
- app/
 - Config/

- Filters #1686
 - Modules #1665
 - Services #614216
 - Toolbar
- contributing/
 - guidelines.rst #1671, #1673
 - internals.rst #1671
- public/
 - index.php #1648, #1670
- system/
 - Autoloader/
 - Autoloader #1665, #1672
 - FileLocator #1665
 - Commands/
 - Database/MigrationRollback #1683
 - Config/
 - BaseConfig #1635
 - BaseService #1635, #1665
 - Paths #1626
 - Services #614216, #3a4ade, #1643
 - View #1616
 - Database/
 - BaseBuilder #1640, #1663, #1677
 - BaseConnection #1677
 - Config #6b8b8b, #1660
 - MigrationRunner #81d371, #1660
 - Query #1677
 - Database/Postgre/
 - Builder #d2b377
 - Debug/Toolbar/Collectors/
 - Logs #1654

- Views #3a4ade
- Events/
 - Events #1635
- Exceptions/
 - ConfigException #1660
- Files/
 - Exceptions/FileException #1636
 - File #1636
- Filters/
 - Filters #1635, #1625, #6dab8f
 - CSRF #1686
 - DebugToolbar #1686
 - Honeypot #1686
- Helpers/
 - form_helper #1633
 - html_helper #1538
 - xml_helper #1641
- HTTP/
 - ContentSecurityPolicy #1641, #1642
 - URI #2e698a
- Language/
 - /en/Files #1636
 - Language #1641
- Log/
 - Handlers/FileHandler #1641
- Router/
 - RouteCollection #1665, #5951c3
 - Router #9e435c, #7993a7, #1678
- Session/

- Handlers/BaseHandler #1684
 - Handlers/FileHandler #1684
 - Handlers/MemcachedHandler #1679
 - Session #1679
- bootstrap #81d371, #1665
- Common #1660
- Entity #1623, #1622
- Model #1617, #1632, #1656, #1689
- tests/
 - README.md #1671
- tests/system/
 - API/
 - ResponseTraitTest #1635
 - Autoloader/
 - AutoloaderTest #1665
 - FileLocatorTest #1665, #1686
 - CLI/
 - CommandRunnerTest #1635
 - CommandsTest #1635
 - Config/
 - BaseConfigTest #1635
 - ConfigTest #1643
 - ServicesTest #1635, #1643
 - Database/Builder/
 - AliasTest #bea1dd
 - DeleteTest #1677
 - GroupTest #1640
 - InsertTest #1640, #1677
 - LikeTest #1640, #1677
 - SelectTest #1663
 - UpdateTest #1640, #1677
 - WhereTest #1640, #1677

- Database/Live/
 - AliasTest #1675
 - ConnectTest #1660, #1675
 - ForgeTest #6b8b8b
 - InsertTest #1677
 - Migrations/MigrationRunnerTest #1660, #1675
 - ModelTest #1617, #1689
- Events/
 - EventTest #1635
- Filters/
 - CSRFTest #1686
 - DebugToolbarTest #1686
 - FiltersTest #1635, #6dab8f, #1686
 - HoneypotTest #1686
- Helpers/
 - FormHelperTest #1633
 - XMLHelperTest #1641
- Honeypot/
 - HoneypotTest #1686
- HTTP/
 - ContentSecurityPolicyTest #1641
 - IncomingRequestTest #1641
- Language/
 - LanguageTest #1643
- Router/
 - RouteCollectionTest #5951c3
 - RouterTest #9e435c
- Validation/
 - RulesTest #1689
- View/

- ParserPluginTest #1669
 - ParserTest #1669
- user_guide_src/
 - concepts/
 - autoloader #1665
 - structure #1648
 - database/
 - connecting #1660
 - transactions #1645
 - general/
 - configuration #1643
 - managing_apps #5f305a, #1648
 - modules #1613, #1665
 - helpers/
 - form_helper #1633
 - incoming/
 - filters #1686
 - index #4a1886
 - methodspoofing #4a1886
 - installation/
 - index #1690, #1693
 - installing_composer #1673, #1690
 - installing_git #1673, #1690
 - installing_manual #1673, #1690
 - repositories #1673, #1690
 - running #1690, #1691
 - troubleshooting #1690, #1693
 - libraries/
 - honeypot #1686
 - index #1643, #1690
 - throttler #1686

- tutorial/
 - create_news_item #1693
 - index #1693
 - news_section #1693
 - static_pages #1693
- composer.json #1670
- contributing.md #1670
- README.md #1670
- spark #1648
- .travis.yml #1649, #1670

PRs merged:

- #1693 Docs/tutorial
- #5951c3 Allow domain/sub-domain routes to overwrite existing routes
- #1691 Update the running docs
- #1690 Rework install docs
- #bea1dd Additional AliasTests for potential LeftJoin issue
- #1689 Model Validation Fix
- #1687 Add copyright blocks to filters
- #1686 Refactor/filters
- #1685 Fix admin - app starter creation
- #1684 Updating session id cleanup for filehandler
- #1683 Fix migrate:refresh bug
- #d2b377 Fix Postgres replace command to work new way of storing binds
- #4a1886 Document method spoofing
- #2e698a urldecode URI keys as well as values.
- #1679 save_path - for memcached
- #1678 fix route not replacing forward slashes
- #1677 Implement Don't Escape feature for db engine
- #1675 Add missing test group directives

- #1674 Update changelog
- #1673 Updated download & installation docs
- #1672 Update Autoloader.php
- #1670 Update PHP dependency to 7.2
- #1671 Update docs
- #1669 Enhance Parser & Plugin testing
- #1665 Composer PSR4 namespaces are now part of the modules auto-discovery
- #6dab8f Filters match case-insensitively
- #1663 Fix bind issue that occurred when using whereIn
- #1660 Migrations Tests and database tweaks
- #1656 DBGroup in __get(), allows to validate “database” data outside the model
- #1654 Toolbar - Return Logger::\$logCache items
- #1649 remove php 7.3 from “allow_failures” in travis config
- #1648 Update “managing apps” docs
- #1645 Fix transaction enabling confusing (docu)
- #1643 Remove email module
- #1642 CSP nonce attribute value in “”
- #81d371 Safety checks for config files during autoload and migrations
- #1641 More unit testing tweaks
- #1640 Update getCompiledX methods in BaseBuilder
- #1637 Fix starter README
- #1636 Refactor Files module
- #5f305a UG - Typo in managing apps
- #1635 Unit testing enhancements
- #1633 Uses csrf_field and form_hidden
- #1632 DBGroup should be passed to ->run instead of ->setRules
- #1631 move use statement after License doc at UploadedFile class
- #1630 Update copyright to 2019
- #1629 “application” to “app” directory doc and comments
- #3a4ade view() now properly reads the app config again
- #7993a7 Final piece to get translateURIDashes working appropriately
- #9e435c TranslateURIDashes fix
- #1626 clean up Paths::\$viewDirectory property
- #1625 After matches is not set empty
- #1623 Property was not cast if was defined as nullable

- #1622 Nullable support for __set
- #1617 countAllResults() should respect soft deletes
- #1616 Fix View config merge order
- #614216 Moved honeypot service out of the app Services file to the system Services where it belongs
- #6b8b8b Allow db forge and utils to take an array of connection info instead of a group name
- #1613 Typo in documentation
- #1538 img fix(?) - html_helper

Version 4.0.0-alpha.4

Release Date: Dec 15, 2018

Next alpha release of CodeIgniter4

Highlights:

- Refactor for consistency: folder application renamed to app; constant BASEPATH renamed to SYSTEMPATH
- Debug toolbar gets its own config, history collector
- Numerous corrections and enhancements

The list of changed files follows, with PR numbers shown.

- admin/
 - docbot #1573
 - framework/composer.json #1555
 - release #1573
 - release-deploy #1573
 - starter/composer.json #1573, #1600
- app/
 - Config/
 - App #1571
 - Autoload #1579
 - ContentSecurityPolicy #1581
 - Events #1571, #1595
 - Paths #1579
 - Routes #1579
 - Services #1579
 - Toolbar #1571, #1579
 - Filters/

- Toolbar #1571
- Views/
 - errors/* #1579
- public/
 - index #1579
- system/
 - Autoloader/
 - Autoloader #1562
 - FileLocator #1562, #1579
 - CLI/
 - CommandRunner #1562
 - Config/
 - AutoloadConfig #1555, #1579
 - BaseConfig #1562
 - Services #1571, #1562
 - Database/
 - BaseBuilder #a0fc68
 - MigrationRunner #1585
 - MySQLi/Connection #1561, #8f205a
 - Debug/
 - Collectors/* #1571, #1589, #1579
 - Exceptions #1579
 - Toolbar #1571
 - Views/toolbar.tpl #1571
 - Views/toolbarloader.js #1594
 - Helpers/
 - form_helper #1548
 - url_helper #1588
 - HTTP/
 - ContentSecurityPolicy #1581

- DownloadResponse
 - I18n/
 - Time #1603
 - Language/
 - Language #1587, #1562, #1610
 - en/
 - CLI #1562
 - HTTP #d7dfc5
 - Log/
 - Handlers/FileHandler #1579
 - Logger #1562, #1579
 - Session/
 - Handlers/DatabaseHandler #1598
 - Test/
 - CIUnitTest #1581, #1593, #1579
 - FeatureResponse #1593
 - FeatureTestCase #1593
 - View/
 - View #1571, #1579
 - bootstrap #1579
 - CodeIgniter #ab8b5b, #1579
 - Common #1569, #1563, #1562, #1601, #1579
 - Entity #4c7bfe, #1575
 - Model #1602, #a0fc68
- tests/
 - Autoloader/
 - AutolaoderTest #1562, #1579
 - FileLocatorTest #1562, #1579
 - Config/
 - ServicesTest #1562

- Database/
 - Live/ModelTest #1602, #a0fc68
- Files/
 - FileTest #1579
- Helpers/
 - FormHelperTest #1548
 - URLHelperTest #1588
- HTTP/
 - ContentSecurityPolicyTest #1581
 - DownloadResponseTest #1576, #1579
 - IncomingRequestDetectingTest #1576
 - IncomingRequestTest #1576
 - RedirectResponseTest #1562
 - ResponseTest #1576
- I18n/
 - TimeDifferenceTest #1603
 - TimeTest #1603
- Language/
 - LanguageTest #1587, #1610
- Log/
 - FileHandlerTest #1579
- Router/
 - RouterCollectionTest #1562
 - RouterTest #1562
- Test/
 - FeatureResponseTest #1593
 - FeatureTestCaseTest #1593
 - TestCaseTest #1593
- Validation/
 - ValidationTest #1562

- View/
 - ParserPluginTest #1562
 - ParserTest #1562
 - ViewTest #1562
 - CodeIgniterTest #1562
 - CommonFunctionsTest #1569, #1562
 - EntityTest #4c7bfe, #1575
- user_guide_src/source/
 - cli/
 - cli #1579
 - cli_commands #1579
 - concepts/
 - autoloader #1579
 - mvc #1579
 - services #1579
 - structure #1579
 - database/
 - configuration #1579
 - dbmgt/
 - migration #1579
 - seeds #1579
 - general/
 - common_functions #d7dfc5, #1579
 - configuration #1608
 - errors #1579
 - installation/
 - downloads #1579
 - models/
 - entities #547792, #1575
 - outgoing/

- localization #1610
 - response #1581, #1579
 - view_parser #1579
- testing/
 - debugging #1579
 - overview #1593, #1579
- tutorial/
 - news_section #1586
 - static_pages #1579
- composer.json #1555
- ComposerScripts #1551
- spark #1579
- Vagrantfile.dist #1459

PRs merged:

- #1610 Test, fix & enhance Language
- #a0fc68 Clear binds after inserts, updates, and find queries
- #1608 Note about environment configuration in UG
- #1606 release framework script clean up
- #1603 Flesh out I18n testing
- #8f305a Catch mysql connection errors and sanitize username and password
- #1602 Model's first and update didn't work primary key-less tables
- #1601 clean up ConfigServices in Common.php
- #1600 admin/starter/composer.json clean up
- #1598 use \$defaultGroup as default value for database session DBGroup
- #1595 handle fatal error via pre_system
- #1594 Fix Toolbar invalid css
- #1593 Flesh out the Test package testing
- #1589 Fix Toolbar file loading throw exception
- #1588 Fix site_url generate invalid url
- #1587 Add Language fallback
- #1586 Fix model namespace in tutorial

- #1585 Type hint MigrationRunner methods
- #4c7bfe Entity fill() now respects mapped properties
- #547792 Add _get and _set notes for Entity class
- #1582 Fix changelog index & common functions UG indent
- #1581 ContentSecurityPolicy testing & enhancement
- #1579 Use Absolute Paths
- #1576 Testing13/http
- #1575 Adds ?integer, ?double, ?string, etc. cast types
- #ab8b5b Set baseURL to example.com during testing by default.
- #d7dfc5 Doc tweaks for redirects
- #1573 Lessons learned
- #1571 Toolbar updates
- #1569 Test esc() with different encodings and ignore app-only helpers
- #1563 id attribute support added for csrf_field
- #1562 Integrates Autoloader and FileLocator
- #1561 Update Connection.php
- #1557 remove prefix on use statements
- #1556 using protected instead of public modifier for setUp() function in tests
- #1555 autoload clean up: remove PsrLog namespace from composer.json
- #1551 remove manual define “system/” directory prefix at ComposerScripts
- #1548 allows to set empty html attr
- #1459 Add Vagrantfile

Version 4.0.0-alpha.3

Release Date: November 30, 2018

Next alpha release of CodeIgniter4

The list of changed files follows, with PR numbers shown.

- admin/
 - framework/* #1553
 - starter/* #1553
 - docbot #1553
 - release* #1484,
 - pre-commit #1388
 - README.md #1553
 - setup.sh #1388
- application /
 - Config/
 - Autoload #1396, #1416
 - Mimes #1368, #1465
 - Pager #622
 - Services #1469
 - Filters/Honeypot #1376
 - Views/
 - errors/* #1415, #1413, #1469
 - form.php removed #1442
- public /
 - index.php #1388, #1457
- system /
 - Autoloader/
 - Autoloader #1547

- FileLocator #1547, #1550
- Cache/
 - Exceptions/CacheException #1525
 - Handlers/FileHandler #1547, #1525
 - Handlers/MemcachedHandler #1383
- CLI/
 - CLI #1432, #1489
- Commands/
 - Database/
 - CreateMigration #1374, #1422, #1431
 - MigrateCurrent #1431
 - MigrateLatest #1431
 - MigrateRollback #1431
 - MigrateStatus #1431
 - MigrateVersion #1431
 - Sessions/CrateMigration #1357
- Config/
 - AutoloadConfig #1416
 - BaseService #1469
 - Mimes #1453
 - Services #1180, #1469
- Database/
 - BaseBuilder #1335, #1491, #1522
 - BaseConnection #1335, #1407, #1491, #1522
 - BaseResult #1426
 - Config #1465, #1469, #1554
 - Forge #1343, #1449, #1470, #1530
 - MigrationRunner #1371
 - MySQLi/Connection #1335, #1449
 - MySQLi/Forge #1343, #1344, #1530
 - MySQLi/Result #1530
 - Postgre/Connection #1335, #1449

- Postgre/Forge #1530
- SQLite3/Connection #1335, #1449
- SQLite3/Forge #1470, #1547
- Debug
 - Exceptions #1500
 - Toolbar #1370, #1465, #1469, #1547
 - Toolbar/Views/toolbar.tpl #1469
- Email/
 - Email #1389, #1413, #1438, #1454, #1465, #1469, #1547
- Events/
 - Events #1465, #1469, #1547
- Files/
 - File #1399, #1547
- Format/
 - XMLFormatter #1471
- Helpers/
 - array_helper #1412
 - filesystem_helper #1547
- Honeypot/
 - Honeypot #1460
- HTTP/
 - CURLRequest #1547, #1498
 - DownloadResponse #1375
 - Exceptions/DownloadException #1405
 - Files/FileCollection #1506
 - Files/UploadedFile #1335, #1399, #1500, #1506, #1547
 - IncomingRequest #1445, #1469, #1496
 - Message #1497
 - RedirectResponse #1387, #1451, #1464

- Response #1456, #1472, #1477, #1486, #1504, #1505, #1497, #622
- ResponseInterface #1384
- UploadedFile #1368, #1456
- URI #1213, #1469, #1508
- Images/Handlers/
 - ImageMagickHandler #1546
- Language/
 - en/Cache #1525
 - en/Database #1335
 - en/Filters #1378
 - en/Migrations #1374
 - Language #1480, #1489
- Log/
 - Handlers/FileHandler #1547
- Pager/
 - Pager #1213, #622
 - PagerInterface #622
 - PagerRenderer #1213, #622
 - Views/default_full #622
 - Views/default_head #622
 - Views/default_simple #622
- Router/
 - RouteCollection #1464, #1524
 - RouteCollectionInterface #1406, #1410
 - Router #1523, #1547
- Session/Handlers/
 - BaseHandler #1180, #1483
 - DatabaseHandler #1180
 - FileHandler #1180, #1547
 - MemcachedHandler #1180
 - RedisHandler #1180

- Test/
 - CIUnitTestCase #1467
 - FeatureTestCase #1427, #1468
 - Filters/CITestStreamFilter #1465
 - Validation /
 - CreditCardRules #1447, #1529
 - FormatRules #1507
 - Rules #1345
 - Validation #1345
 - View/
 - Filters #1469
 - Parser #1417, #1547
 - View #1357, #1377, #1410, #1547
 - bootstrap #1547
 - CodeIgniter #1465, #1505, #1523, 2047b5a, #1547
 - Common #1486, #1496, #1504, #1513
 - ComposerScripts #1469, #1547
 - Controller #1423
 - Entity #1369, #1373
 - Model #1345, #1380, #1373, #1440
- tests /
 - _support/
 - HTTP/MockResponse #1456
 - _bootstrap.php #1397, #1443
 - Cache/Handlers/
 - FileHandlerTest #1547, #1525
 - MemcachedHandlerTest #1180, #1383
 - RedisHandlerTest #1180, #1481
 - CLI/
 - CLITest #1467, #1489
 - Commands/

- SessionCommandsTest #1455
- Database/Live/
 - ConnectTest #1554
 - ForgeTest #1449, #1470
- HTTP/
 - CURLRequestTest#1498
 - Files/FileCollectionTest #1506
 - Files/FileMovingTest #1424
 - DownloadResponseTest #1375
 - IncomingRequestTest #1496
 - RedirectResponseTest #1387, #1456
 - ResponseCookieTest #1472, #1509
 - ResponseSendTest #1477, #1486, #1509
 - ResponseTest #1375, #1456, #1472, #1486, #622
 - URITest #1456, #1495
- Helpers/
 - DateHelperTest #1479
- I18n/
 - TimeTest #1467, #1473
- Language/
 - LanguageTest #1480
- Log/
 - FileHandlerTest #1425
- Pager/
 - PagerRendererTest #1213, #622
 - PagerTest #622
- Router/
 - RouteCollectionTest #1438, #1524
 - RouterTest #1438, #1523
- Session/

- SessionTest #1180
- Test/
 - BootstrapFCPATHTest #1397
 - FeatureTestCase #1468
 - TestCaseEmissionsTest #1477
 - TestCaseTest #1390
- Throttle/
 - ThrottleTest #1398
- Validation/
 - FormatRulesTest #1507
- View/
 - ParserTest #1335
- CodeIgniterTest #1500
- CommonFunctionsSendTest #1486, #1509
- CommonFunctionsTest #1180, #1486, #1496
- user_guide_src /source/
 - changelogs/ #1385, #1490, #1553
 - concepts/
 - autoloader #1547
 - security #1540
 - services #1469
 - structure #1448
 - database/
 - queries #1407
 - dbmgmt/
 - forge #1470
 - migration #1374, #1385, #1431
 - seeds #1482
 - extending/
 - core_classes #1469

- helpers/
 - form_helper #1499
- installation/
 - index #1388
- libraries/
 - caching #1525
 - pagination #1213
 - validation #27868b, #1540
- models/
 - entities #1518, #1540
- outgoing/
 - response #1472, #1494
- testing/
 - overview #1467
- tutorial/
 - create_news_item #1442
 - static_pages #1547
- /
 - composer.json #1388, #1418, #1536, #1553
 - README.md #1553
 - spark 2047b5a
 - .travis.yml #1394

PRs merged:

- #1554 Serviceinstances
- #1553 Admin/scripts
- #1550 remove commented CLI::newLine(\$tempFiles) at FileLocator
- #1549 use .gitkeep instead of .gitignore in Database/Seeds directory
- #1547 Change file exists to is file
- #1546 ImageMagickHandler::__construct ...

- #1540 Update validation class User Guide
- #1530 database performance improvement : use foreach() when possible
- 2047b5a Don't run filters when using spark.
- #1539 remove mb_* (mb string usage) in CreditCardRules
- #1536 ext-json in composer.json
- #1525 remove unneeded try {} catch {}
- #1524 Test routes resource with 'websafe' option
- #1523 Check if the matched route regex is filtered
- #1522 add property_exists check on BaseBuilder
- #1521 .gitignore clean up
- #1518 Small typo: changed setCreatedOn to setCreatedAt
- #1517 move .htaccess from per-directory in writable/{directory} to writable/
- #1513 More secure redirection
- #1509 remove unused use statements
- #1508 remove duplicate strtolower() call in URI::setScheme() call
- #1507 Fix multi "empty" string separated by "," marked as valid emails
- #1506 Flesh out HTTP/File unit testing
- #1505 Do not exit until all Response is completed
- 27868b Add missing docs for {field} and {param} placeholders
- #1504 Revert RedirectResponse changes
- #1500 Ignoring errors suppressed by @
- #1499 Fix form_helper's set_value writeup
- #1498 Add CURLRequest helper methods
- #1497 Remove unused RedirectException
- #1496 Fix Common::old()
- #1495 Add URI segment test
- #1494 Method naming in user guide
- #1491 Error logging
- #1490 Changelog(s) restructure
- #1489 Add CLI::strlen()
- #1488 Load Language strings from other locations
- #1486 Test RedirectResponse problem report
- #1484 missing slash
- #1483 Small typo in SessionHandlersBaseHandler.php
- #1482 doc fix: query binding fix in Seeds documentation
- #1481 RedisHandler test clean up

- #1480 Fix Language Key-File confusion
- #1479 Yet another time test to fix
- #1477 Add Response send testing
- #1475 Correct phpdocs for Forge::addField()
- #1473 Fuzzify another time test
- #1472 HTTPResponse cookie testing & missing functionality
- #1471 remove unused local variable \$result in XMLFormatter::format()
- #1470 Allow create table with array field constraints
- #1469 use static:: instead of self:: for call protected/public functions as well
- #1468 Fix FeatureTestCaseTest output buffer
- #1467 Provide time testing within tolerance
- #1466 Fix phpdocs for BaseBuilder
- #1465 use static:: instead of self:: for protected and public properties
- #1464 remove unused use statements
- #1463 Fix the remaining bcit-ci references
- #1461 Typo fix: donload -> download
- #1460 remove unneeded ternary check at HoneyPot
- #1457 use \$paths->systemDirectory in public/index.php
- #1456 Beef up HTTP URI & Response testing
- #1455 un-ignore app/Database/Migrations directory
- #1454 add missing break; in loop at Email::getEncoding()
- #1453 BugFix if there extension has only one mime type
- #1451 remove unneeded \$session->start(); check on RedirectResponse
- #1450 phpcbf: fix all at once
- #1449 Simplify how to get indexData from mysql/mariadb
- #1448 documentation: add missing application structures
- #1447 add missing break; on loop cards to get card info at CreditCardRules
- #1445 using existing is_cli() function in HTTPIncomingRequest
- #1444 Dox for reorganized repo admin (4 of 4)
- #1443 Fixes unit test output not captured
- #1442 remove form view in app/View/ and form helper usage in create new items tutorial
- #1440 Access to model's last inserted ID
- #1438 Tailor the last few repo org names (3 of 4)
- #1437 Replace repo org name in MOST php docs (2 of 4)

- #1436 Change github organization name in docs (1 of 4)
- #1432 Use mb_strlen to get length of columns
- #1431 can't call run() method with params from commands migrations
- #1427 Fixes "options" request call parameter in FeatureTestCase
- #1416 performance improvement in DatabaseBaseResult
- #1425 Ensure FileHandlerTest uses MockFileHandler
- #1424 Fix FileMovingTest leaving cruft
- #1423 Fix Controller use validate bug
- #1422 fix Migrations.classNotFound
- #1418 normalize composer.json
- #1417 fix Parser::parsePairs always escapes
- #1416 remove \$psr4['TestsSupport'] definition in applicationConfigAutoload
- #1415 remove unneded "defined('BASEPATH') ...
- #1413 set more_entropy = true in all uniqid() usage
- #1412 function_exists() typo fixes on array_helper
- #1411 add missing break; in loop in View::render()
- #1410 Fix spark serve not working from commit 2d0b325
- #1407 Database: add missing call initialize() check on BaseConnection->prepare()
- #1406 Add missing parameter to RouteCollectionInterface
- #1405 Fix language string used in DownloadException
- #1402 Correct class namespacing in the user guide
- #1399 optional type hinting in guessExtension
- #1398 Tweak throttle testing
- #1397 Correcting FCPATH setting in tests/_support/_bootstrap.php
- #1396 only register PSR4 "TestsSupport" namespace in "testing" environment
- #1395 short array syntax in docs
- #1394 add php 7.3 to travis config
- #1390 Fixed not to output "Hello" at test execution
- #1389 Capitalize email filename
- #1388 Phpcs Auto-fix on commit
- #1387 Redirect to named route
- #1385 Fix migration page; update changelog
- #1384 add missing ResponseInterface constants
- #1383 fix TypeError in MemcachedHandler::__construct()

- #1381 Remove unused use statements
- #1380 count() improvement, use truthy check
- #1378 Update Filters language file
- #1377 fix monolog will cause an error
- #1376 Fix cannot use class HoneyPot because already in use in AppFiltersHoneyPot
- #1375 Give download a header conforming to RFC 6266
- #1374 Missing feature migration.
- #1373 Turning off casting for db insert/save
- #1371 update method name in coding style
- #1370 Toolbar needs logging. Fixes #1258
- #1369 Remove invisible character
- #1368 UploadedFile->guessExtension()...
- #1360 rm -cached php_errors.log file
- #1357 Update template file is not .php compatibility
- #1345 is_unique tried to connect to default database instead of defined in DBGroup
- #1344 Not to quote unnecessary table options
- #1343 Avoid add two single quote to constraint
- #1335 Review and improvements in databases drivers MySQLi, PostgreSQL and SQLite
- #1213 URI segment as page number in Pagination
- #1180 using HTTPRequest instance to pull ip address
- #622 Add Header Link Pagination

Version 4.0.0-alpha.2

Release Date: Oct 26, 2018

Second alpha release of CodeIgniter4

The list of changed files follows, with PR numbers shown.

application /

- composer.json #1312
- Config/Boot/development, production, testing #1312
- Config/Paths #1341
- Config/Routes #1281
- Filters/Honeypot #1314
- Views/errors/cli/error_404 #1272
- Views/welcome_message #1342

public /

- .htaccess #1281
- index #1295, #1313

system /

- CLI/
 - CommandRunner #1350, #1356
- Commands/
 - Server/Serve #1313
- Config/
 - AutoloadConfig #1271
 - Services #1341
- Database/
 - BaseBuilder #1217
 - BaseUtils #1209, #1329
 - Database #1339

- MySQLi/Utils #1209
- Debug/Toolbar/
 - Views/toolbar.css #1342
- Exceptions/
 - CastException #1283
 - DownloadException #1239
 - FrameworkException #1313
- Filters/
 - Filters #1239
- Helpers/
 - cookie_helper #1286
 - form_helper #1244, #1327
 - url_helper #1321
 - xml_helper #1209
- Honeypot/
 - Honeypot #1314
- HTTP/
 - CliRequest #1303
 - CURLRequest #1303
 - DownloadResponse #1239
 - Exceptions/HttpException #1303
 - IncomingRequest #1304, #1313
 - Negotiate #1306
 - RedirectResponse #1300, #1306, #1329
 - Response #1239, #1286
 - ResponseInterface #1239
 - URI #1300
- Language/en/
 - Cast #1283
 - HTTP #1239
- Router/

- RouteCollection #1285, #1355
- Test/
 - CIUnitTestCase #1312, #1361
 - FeatureTestCase #1282
- CodeIgniter #1239 #1337
- Common #1291
- Entity #1283, #1311
- Model #1311

tests /

- API/
 - ResponseTraitTest #1302
- Commands/
 - CommandsTest #1356
- Database/
 - BaseBuilderTest #1217
 - Live/ModelTest #1311
- Debug/
 - TimerTest #1273
- Helpers/
 - CookieHelperTest #1286
- Honeypot/
 - HoneypotTest #1314
- HTTP/
 - Files/
 - FileMovingTest #1302
 - UploadedFileTest #1302
 - CLIRequestTest #1303
 - CURLRequestTest #1303
 - DownloadResponseTest #1239

- NegotiateTest #1306
 - RedirectResponseTest #1300, #1306, #1329
 - ResponseTest #1239
- I18n/
 - TimeTest #1273, #1316
- Router/
 - RouteTest #1285, #1355
- Test/
 - TestCaseEmissionsTest #1312
 - TestCaseTest #1312
- View/
 - ParserTest #1311
- EntityTest #1319

user_guide_src /source/

- cli/
 - cli_request #1303
- database/
 - query_builder #1217
 - utilities #1209
- extending/
 - contributing #1280
- general/
 - common_functions #1300, #1329
 - helpers #1291
 - managing_apps #1341
- helpers/
 - xml_helper #1321
- incoming/
 - controllers #1323

- routing #1337
- intro/
 - requirements #1280, #1303
- installation/ #1280, #1303
 - troubleshooting #1265
- libraries/
 - curlrequest #1303
 - honeypot #1314
 - sessions #1333
 - uploaded_files #1302
- models/
 - entities #1283
- outgoing/
 - response #1340
- testing/
 - overview #1312
- tutorial... #1265, #1281, #1294

/

- spark #1305

PRs merged:

- #1361 Add timing assertion to CIUnitTestCase
- #1312 Add headerEmitted assertions to CIUnitTestCase
- #1356 Testing/commands
- #1355 Handle duplicate HTTP verb and generic rules properly
- #1350 Checks if class is instantiable and is a command
- #1348 Fix sphinx formatting in sessions
- #1347 Fix sphinx formatting in sessions
- #1342 Toolbar Styles

- #1341 Make viewpath configurable in Paths.php. Fixes #1296
- #1340 Update docs for downloads to reflect the need to return it. Fixes #1331
- #1339 Fix error where Forge class might not be returned. Fixes #1225
- #1337 Filter in the router Fixes #1315
- #1336 Revert alpha.2
- #1334 Proposed changelog for alpha.2
- #1333 Error in user guide for session config. Fixes #1330
- #1329 Tweaks
- #1327 FIX form_hidden and form_open - value escaping as is in form_input.
- #1323 Fix doc error : show_404() doesn't exist any more
- #1321 Added missing xml_helper UG page
- #1319 Testing/entity
- #1316 Refactor TimeTest
- #1314 Fix & expand Honeypot & its tests
- #1313 Clean exception
- #1311 Entities store an original stack of values to compare against so we d...
- #1306 Testing3/http
- #1305 Change chdir('public') to chdir(\$public)
- #1304 Refactor script name stripping in parseRequestURI()
- #1303 Testing/http
- #1302 Exception: No Formatter defined for mime type ''
- #1300 Allow redirect with Query Vars from the current request.
- #1295 Fix grammar in front controller comment.
- #1294 Updated final tutorial page. Fixes #1292
- #1291 Allows extending of helpers. Fixes #1264
- #1286 Cookies
- #1285 Ensure current HTTP verb routes are matched prior to any * matched ro...
- #1283 Entities
- #1282 system/Test/FeatureTestCase::setUpRequest(), minor fixes phpdoc block...
- #1281 Tut
- #1280 Add contributing reference to user guide
- #1273 Fix/timing

- #1272 Fix undefined variable “heading” in cli 404
- #1271 remove inexistent “CodeIgniterLoader” from AutoloadConfig::classmap
- #1269 Release notes & process
- #1266 Adjusting the release build scripts
- #1265 WIP Fix docs re PHP server
- #1245 Fix #1244 (form_hidden declaration)
- #1239 **【Unsolicited PR】**I changed the download method to testable.
- #1217 Optional parameter for resetSelect() call in Builder’s countAll();
- #1209 Fix undefined function xml_convert at DatabaseBaseUtils

Version 4.0.0-alpha.1

Release Date: September 28, 2018

Rewrite of the CodeIgniter framework

New packages list:

- API
 - \ ResponseTrait
- Autoloader
 - \ AutoLoader, FileLocator
- CLI
 - \ BaseCommand, CLI, CommandRunner, Console
- Cache
 - \ CacheFactory, CacheInterface
 - \ Handlers ... Dummy, File, Memcached, Predis, Redis, Wincache
- Commands
 - \ Help, ListCommands
 - \ Database \ CreateMigration, MigrateCurrent, MigrateLatest, MigrateRefresh, MigrateRollback, MigrateStatus, MigrateVersion, Seed
 - \ Server \ Serve
 - \ Sessions \ CreateMigration
 - \ Utilities \ Namespaces, Routes
- Config
 - \ AutoloadConfig, BaseConfig, BaseService, Config, DotEnv, ForeignCharacters, Routes, Services, View
- Database

- \ BaseBuilder, BaseConnection, BasePreparedQuery, BaseResult, BaseUtils, Config, ConnectionInterface, Database, Forge, Migration, MigrationRunner, PreparedQueryInterface, Query, QueryInterface, ResultInterface, Seeder
 - \ MySQLi \ Builder, Connection, Forge, PreparedQuery, Result
 - \ Postgre \ Builder, Connection, Forge, PreparedQuery, Result, Utils
 - \ SQLite3 \ Builder, Connection, Forge, PreparedQuery, Result, Utils
- Debug
 - \ Exceptions, Iterator, Timer, Toolbar
 - \ Toolbar \ Collectors...
- Email
 - \ Email
- Events
 - \ Events
- Files
 - \ File
- Filters
 - \ FilterInterface, Filters
- Format
 - \ FormatterInterface, JSONFormatter, XMLFormatter
- HTTP
 - \ CLIRequest, CURLRequest, ContentSecurityPolicy, Header, IncomingRequest, Message, Negotiate, Request, RequestInterface, Response, ResponseInterface, URI, UserAgent
 - \ Files \ FileCollection, UploadedFile, UploadedFileInterface

- Helpers
 - ... array, cookie, date, filesystem, form, html, inflector, number, security, text, url
- Honeypot
 - \ Honeypot
- I18n
 - \ Time, TimeDifference
- Images
 - \ Image, ImageHandlerInterface
 - \ Handlers ... Base, GD, ImageMagick
- Language
 - \ Language
- Log
 - Logger, LoggerAwareTrait
 - \ Handlers ... Base, ChromeLogger, File, HandlerInterface
- Pager
 - \ Pager, PagerInterface, PagerRenderer
- Router
 - \ RouteCollection, RouteCollectionInterface, Router, RouterInterface
- Security
 - \ Security
- Session
 - \ Session, SessionInterface
 - \ Handlers ... Base, File, Memcached, Redis
- Test
 - \ CIDatabaseTestCase, CIUnitTestCase, FeatureResponse, FeatureTestCase, ReflectionHelper
 - \ Filters \ CITestStreamFilter

- ThirdParty (bundled)
 - \ Kint (for \Debug)
 - \ PSR \ Log (for \Log)
 - \ ZendEscaper \ Escaper (for \View)
- Throttle
 - \ Throttler, ThrottlerInterface
- Typography
 - \ Typography
- Validation
 - \ CreditCardRules, FileRules, FormatRules, Rules, Validation, ValidationInterface
- View
 - \ Cell, Filters, Parser, Plugins, RendererInterface, View

Index

[Symbols](#) | [A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#)

Symbols

[\(\)](#) ([method](#)), [\[1\]](#), [\[2\]](#), [\[3\]](#), [\[4\]](#), [\[5\]](#), [\[6\]](#), [\[7\]](#), [\[8\]](#), [\[9\]](#), [\[10\]](#), [\[11\]](#), [\[12\]](#), [\[13\]](#), [\[14\]](#), [\[15\]](#), [\[16\]](#), [\[17\]](#), [\[18\]](#), [\[19\]](#), [\[20\]](#), [\[21\]](#), [\[22\]](#), [\[23\]](#), [\[24\]](#), [\[25\]](#), [\[26\]](#), [\[27\]](#), [\[28\]](#), [\[29\]](#), [\[30\]](#), [\[31\]](#), [\[32\]](#), [\[33\]](#), [\[34\]](#), [\[35\]](#), [\[36\]](#), [\[37\]](#), [\[38\]](#), [\[39\]](#), [\[40\]](#), [\[41\]](#), [\[42\]](#), [\[43\]](#), [\[44\]](#), [\[45\]](#), [\[46\]](#), [\[47\]](#), [\[48\]](#), [\[49\]](#), [\[50\]](#), [\[51\]](#), [\[52\]](#), [\[53\]](#), [\[54\]](#), [\[55\]](#), [\[56\]](#), [\[57\]](#), [\[58\]](#), [\[59\]](#), [\[60\]](#), [\[61\]](#), [\[62\]](#), [\[63\]](#), [\[64\]](#), [\[65\]](#), [\[66\]](#), [\[67\]](#), [\[68\]](#), [\[69\]](#), [\[70\]](#), [\[71\]](#), [\[72\]](#), [\[73\]](#), [\[74\]](#), [\[75\]](#), [\[76\]](#), [\[77\]](#), [\[78\]](#)

A

addColumn() (CodeIgniterDatabaseForge method)	anchor_popup() (global function)
addField() (CodeIgniterDatabaseForge method)	APPPATH (global constant)
addKey() (CodeIgniterDatabaseForge method)	ascii_to_entities() (global function)
addPrimaryKey() (CodeIgniterDatabaseForge method)	audio() (global function)
addUniqueKey() (CodeIgniterDatabaseForge method)	auto_link() (global function)
alternator() (global function)	autoTypography() (global function)
anchor() (global function)	

B

[base_url\(\)](#) ([global function](#))

C

<u>cache()</u> (global function)	<u>countAll()</u>
<u>call()</u> (CodeIgniterCLIBaseCommand method)	<u>(CodeIgniterDatabaseBaseBuilder method)</u>
<u>camelize()</u> (global function)	<u>countAllResults()</u>
<u>character_limiter()</u> (global function)	<u>(CodeIgniterDatabaseBaseBuilder method)</u>
<u>CodeIgniterCLIBaseCommand</u> (class)	<u>createDatabase()</u>
<u>CodeIgniterDatabaseBaseBuilder</u> (class)	<u>(CodeIgniterDatabaseForge method)</u>
<u>CodeIgniterDatabaseBaseResult</u> (class)	<u>createTable()</u>
<u>CodeIgniterDatabaseForge</u> (class)	<u>(CodeIgniterDatabaseForge method)</u>
<u>CodeIgniterDatabaseMigrationRunner</u> (class)	<u>csrf_field()</u> (global function)
<u>convert_accented_characters()</u> (global function)	<u>csrf_hash()</u> (global function)
	<u>csrf_token()</u> (global function)
	<u>current()</u>
	<u>(CodeIgniterDatabaseMigrationRunner method)</u>
	<u>current_url()</u> (global function)

D

<u>dasherize()</u> (global function)	<u>directory_map()</u> (global function)
<u>dataSeek()</u>	<u>distinct()</u>
<u>(CodeIgniterDatabaseBaseResult method)</u>	<u>(CodeIgniterDatabaseBaseBuilder method)</u>
<u>DAY</u> (global constant)	<u>doctype()</u> (global function)
<u>DECADE</u> (global constant)	<u>dot_array_search()</u> (global function)
<u>decrement()</u>	<u>dropColumn()</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>(CodeIgniterDatabaseForge method)</u>
<u>delete()</u>	<u>dropDatabase()</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>(CodeIgniterDatabaseForge method)</u>
<u>delete_cookie()</u> (global function)	<u>dropTable()</u>
<u>delete_files()</u> (global function)	

[\(CodeIgniterDatabaseForge
method\)](#)

E

[ellipse\(\) \(global function\)](#)

[embed\(\) \(global function\)](#)

[emptyTable\(\)](#)

[\(CodeIgniterDatabaseBaseBuilder
method\)](#)

[encode_php_tags\(\) \(global function\)](#)

[entities_to_ascii\(\) \(global
function\)](#)

[env\(\) \(global function\)](#)

[esc\(\) \(global function\)](#)

[excerpt\(\) \(global function\)](#)

F

[FCPATH \(global constant\)](#)

[findMigrations\(\)](#)

[\(CodeIgniterDatabaseMigrationRunner
method\)](#)

[force_https\(\) \(global function\)](#)

[form_button\(\) \(global function\)](#)

[form_checkbox\(\) \(global function\)](#)

[form_close\(\) \(global function\)](#)

[form_dropdown\(\) \(global function\)](#)

[form_fieldset\(\) \(global function\)](#)

[form_fieldset_close\(\) \(global function\)](#)

[form_hidden\(\) \(global function\)](#)

[form_input\(\) \(global function\)](#)

[form_label\(\) \(global function\)](#)

[form_multiselect\(\) \(global
function\)](#)

[form_open\(\) \(global function\)](#)

[form_open_multipart\(\) \(global
function\)](#)

[form_password\(\) \(global functio](#)

[form_radio\(\) \(global function\)](#)

[form_reset\(\) \(global function\)](#)

[form_submit\(\) \(global function\)](#)

[form_textarea\(\) \(global function\)](#)

[form_upload\(\) \(global function\)](#)

[formatCharacters\(\) \(global
function\)](#)

[freeResult\(\)](#)

[\(CodeIgniterDatabaseBaseResul
method\)](#)

[from\(\)](#)

[\(CodeIgniterDatabaseBaseBuild
method\)](#)

G

<u>get()</u>	<u>getLastRow()</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>get_cookie() (global function)</u>	<u>getNextRow()</u>
<u>get_dir_file_info() (global function)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>get_file_info() (global function)</u>	<u>getPad()</u>
<u>get_filenames() (global function)</u>	<u>(CodeIgniterCLIBaseCommand method)</u>
<u>getCompiledDelete()</u>	<u>getPreviousRow()</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>getCompiledInsert()</u>	<u>getResult()</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>getCompiledSelect()</u>	<u>getResultArray()</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>getCompiledUpdate()</u>	<u>getResultObject()</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>getCustomResultObject()</u>	<u>getRow()</u>
<u>(CodeIgniterDatabaseBaseResult method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>getCustomRowObject()</u>	<u>getRowArray()</u>
<u>(CodeIgniterDatabaseBaseResult method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>getFieldCount()</u>	<u>getRowObject()</u>
<u>(CodeIgniterDatabaseBaseResult method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>getFieldData()</u>	<u>getUnbufferedRow()</u>
<u>(CodeIgniterDatabaseBaseResult method)</u>	<u>(CodeIgniterDatabaseBaseResult method)</u>
<u>getFieldNames()</u>	<u>getWhere()</u>
<u>(CodeIgniterDatabaseBaseResult method)</u>	<u>(CodeIgniterDatabaseBaseBuilder method)</u>
<u>getFirstRow()</u>	<u>groupBy()</u>
<u>(CodeIgniterDatabaseBaseResult method)</u>	

[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[groupEnd\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[groupStart\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)

H

[having\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[helper\(\) \(global function\)](#)
[highlight_code\(\) \(global function\)](#)

[highlight_phrase\(\) \(global function\)](#)
[HOURL \(global constant\)](#)
[humanize\(\) \(global function\)](#)

I

[img\(\) \(global function\)](#)
[increment\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[increment_string\(\) \(global function\)](#)
[index_page\(\) \(global function\)](#)
[insert\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[insertBatch\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[is_cli\(\) \(global function\)](#)
[is_pluralizable\(\) \(global function\)](#)

J

[join\(\) \(CodeIgniterDatabaseBaseBuilder method\)](#)

L

<u>latest()</u>	<u>limit()</u>
<u>(CodeIgniterDatabaseMigrationRunner method)</u>	<u>(CodeIgniterDatabaseBaseBuilder method)</u>
<u>latestAll()</u>	<u>link_tag() (global function)</u>
<u>(CodeIgniterDatabaseMigrationRunner method)</u>	<u>log_message() (global function)</u>
<u>like()</u>	
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	

M

<u>mailto() (global function)</u>	<u>modifyColumn()</u>
<u>MINUTE (global constant)</u>	<u>(CodeIgniterDatabaseForge method)</u>
	<u>MONTH (global constant)</u>

N

<u>nl2brExceptPre() (global function)</u>	<u>number_to_amount() (global function)</u>
<u>notGroupStart()</u>	<u>number_to_currency() (global function)</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>number_to_roman() (global function)</u>
<u>notLike()</u>	<u>number_to_size() (global function)</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	
<u>now() (global function)</u>	

O

<u>object() (global function)</u>	<u>orGroupStart()</u>
<u>octal_permissions() (global function)</u>	<u>(CodeIgniterDatabaseBaseBuilder method)</u>
<u>offset()</u>	<u>orHaving()</u>
<u>(CodeIgniterDatabaseBaseBuilder method)</u>	<u>(CodeIgniterDatabaseBaseBuilder method)</u>

[ol\(\) \(global function\)](#)
[old\(\) \(global function\)](#)
[orderBy\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[ordinal\(\) \(global function\)](#)
[ordinalize\(\) \(global function\)](#)

[orLike\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[orNotGroupStart\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[orNotLike\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[orWhere\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[orWhereIn\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)
[orWhereNotIn\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)

P

[param\(\) \(global function\)](#)
[plural\(\) \(global function\)](#)

[prep_url\(\) \(global function\)](#)
[previous_url\(\) \(global function\)](#)

Q

[quotes_to_entities\(\) \(global function\)](#)

R

[random_string\(\) \(global function\)](#)
[redirect\(\) \(global function\)](#)
[reduce_double_slashes\(\) \(global function\)](#)
[reduce_multiples\(\) \(global](#)

[renameTable\(\)](#)
[\(CodeIgniterDatabaseForge method\)](#)
[replace\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder method\)](#)

[function\)](#)
[remove_invisible_characters\(\)](#)
[\(global function\)](#)

[resetQuery\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[ROOTPATH \(global constant\)](#)
[route_to\(\) \(global function\)](#)

S

[safe_mailto\(\) \(global function\)](#)
[sanitize_filename\(\) \(global](#)
[function\)](#)
[script_tag\(\) \(global function\)](#)
[SECOND \(global constant\)](#)
[select\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[selectAvg\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[selectMax\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[selectMin\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[selectSum\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[service\(\) \(global function\)](#)
[session\(\) \(global function\)](#)
[set\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[set_checkbox\(\) \(global function\)](#)
[set_cookie\(\) \(global function\)](#)
[set_radio\(\) \(global function\)](#)

[setGroup\(\)](#)
[\(CodeIgniterDatabaseMigrationRunn](#)
[method\)](#)
[setInsertBatch\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[setNamespace\(\)](#)
[\(CodeIgniterDatabaseMigrationRunn](#)
[method\)](#)
[setRow\(\)](#)
[\(CodeIgniterDatabaseBaseResult](#)
[method\)](#)
[setUpdateBatch\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[showError\(\)](#)
[\(CodeIgniterCLIBaseCommand](#)
[method\)](#)
[showHelp\(\)](#)
[\(CodeIgniterCLIBaseCommand](#)
[method\)](#)
[single_service\(\) \(global function\)](#)
[singular\(\) \(global function\)](#)
[site_url\(\) \(global function\)](#)
[source\(\) \(global function\)](#)
[stringify_attributes\(\) \(global function\)](#)
[strip_image_tags\(\) \(global function\)](#)
[strip_quotes\(\) \(global function\)](#)

[set_realpath\(\) \(global function\)](#)
[set_select\(\) \(global function\)](#)
[set_value\(\) \(global function\)](#)

[strip_slashes\(\) \(global function\)](#)
[symbolic_permissions\(\) \(global function\)](#)
[SYSTEMPATH \(global constant\)](#)

T

[timer\(\) \(global function\)](#)
[track\(\) \(global function\)](#)

[truncate\(\) \(CodeIgniterDatabaseBaseBuilder method\)](#)

U

[ul\(\) \(global function\)](#)
[underscore\(\) \(global function\)](#)
[update\(\) \(CodeIgniterDatabaseBaseBuilder method\)](#)

[updateBatch\(\) \(CodeIgniterDatabaseBaseBuilder method\)](#)
[uri_string\(\) \(global function\)](#)
[url_title\(\) \(global function\)](#)

V

[version\(\) \(CodeIgniterDatabaseMigrationRunner method\)](#)
[video\(\) \(global function\)](#)

[view\(\) \(global function\)](#)

W

[WEEK \(global constant\)](#)
[where\(\) \(CodeIgniterDatabaseBaseBuilder method\)](#)
[whereIn\(\) \(CodeIgniterDatabaseBaseBuilder method\)](#)

[word_limiter\(\) \(global function\)](#)
[word_wrap\(\) \(global function\)](#)
[write_file\(\) \(global function\)](#)
[WRITEPATH \(global constant\)](#)

[method\)](#)
[whereNotIn\(\)](#)
[\(CodeIgniterDatabaseBaseBuilder](#)
[method\)](#)
[word_censor\(\) \(global function\)](#)

X

[xml_convert\(\) \(global function\)](#)

Y

[YEAR \(global constant\)](#)