Laravel v3.2

A Framework For Web Artisans

Laravel is a clean and classy framework for PHP web development. Freeing you from spaghetti code, Laravel helps you

create wonderful applications using simple, expressive syntax. Development should be a creative experience

that you enjoy, not something that is painful. Enjoy the fresh air.

Versi PDF oleh Wuri Nugrahadi http://empu.web.id

TABLE OF CONTENTS

GENERAL

LARAVEL DOCUMENTATION

The Basics

Welcome to the Laravel documentation. These documents were designed to function both as a getting-started guide and as a feature reference. Even though you may jump into any section and start learning, we recommend reading the documentation in order as it allows us to progressively establish concepts that will be used in later documents.

Who Will Enjoy Laravel?

Laravel is a powerful framework that emphasizes flexibility and expressiveness. Users new to Laravel will enjoy the same ease of development that is found in the most popular and lightweight PHP frameworks. More experienced users will appreciate the opportunity to modularize their code in ways that are not possible with other frameworks. Laravel's flexibility will allow your organization to update and mold the application over time as is needed and its expressiveness will allow you and your team to develop code that is both concise and easily read.

What Makes Laravel Different?

There are many ways in which Laravel differentiates itself from other frameworks. Here are a few examples that we think make good bullet points:

- **Bundles** are Laravel's modular packaging system. The Laravel Bundle Repository is already populated with quite a few features that can be easily added to your application. You can either download a bundle repository to your bundles directory or use the "Artisan" command-line tool to automatically install them.
- **The Eloquent ORM** is the most advanced PHP ActiveRecord implementation available. With the capacity to easily apply constraints to both relationships and nested eager-loading you'll have complete control over your data with all of the

conveniences of ActiveRecord. Eloquent natively supports all of the methods from Laravel's Fluent query-builder.

- **Application Logic** can be implemented within your application either using controllers (which many web-developers are already familiar with) or directly into route declarations using syntax similar to the Sinatra framework. Laravel is designed with the philosophy of giving a developer the flexibility that they need to create everything from very small sites to massive enterprise applications.
- **Reverse Routing** allows you to create links to named routes. When creating links just use the route's name and Laravel will automatically insert the correct URI. This allows you to change your routes at a later time and Laravel will update all of the relevant links site-wide.
- **Restful Controllers** are an optional way to separate your GET and POST request logic. In a login example your controller's get_login() action would serve up the form and your controller's post_login() action would accept the posted form, validate, and either redirect to the login form with an error message or redirect your user to their dashboard.
- **Class Auto Loading** keeps you from having to maintain an autoloader configuration and from loading unnecessary components when they won't be used. Want to use a library or model? Don't bother loading it, just use it. Laravel will handle the rest.
- **View Composers** are blocks of code that can be run when a view is loaded. A good example of this would be a blog side-navigation view that contains a list of random blog posts. Your composer would contain the logic to load the blog posts so that all you have to do i load the view and it's all ready for you. This keeps you from having to make sure that your controllers load the a bunch of data from your models for views that are unrelated to that method's page content.
- **The IoC container** (Inversion of Control) gives you a method for generating new objects and optionally instantiating and referencing singletons. IoC means that you'll rarely ever need to bootstrap any external libraries. It also means that you can access these objects from anywhere in your code without needing to deal with an inflexible monolithic structure.
- **Migrations** are version control for your database schemas and they are directly integrated into Laravel. You can both generate and run migrations using the "Artisan" command-line utility. Once another member makes schema changes you can update your local copy from the repository and run migrations. Now you're up to date, too!
- **Unit-Testing** is an important part of Laravel. Laravel itself sports hundreds of tests to help ensure that new changes don't unexpectedly break anything. This is one of the reasons why Laravel is widely considered to have some of the most stable releases in the industry. Laravel also makes it easy for you to write unit-tests for your own code. You can then run tests with the "Artisan" command-line utility.
- **Automatic Pagination** prevents your application logic from being cluttered up with a bunch of pagination configuration. Instead of pulling in the current page, getting a count of db records, and selected your data using a limit/offset just call 'paginate' and tell Laravel where to output the paging links in your view. Laravel automatically does the rest. Laravel's pagination system was designed to be easy to implement and easy to change. It's also important to note that just because Laravel can handle these things automatically doesn't mean that you can't call and configure these systems manually if you prefer.

These are just a few ways in which Laravel differentiates itself from other PHP frameworks. All of these features and many more are discussed thoroughly in this documentation.

Application Structure

Laravel's directory structure is designed to be familiar to users of other popular PHP frameworks. Web applications of any shape or size can easily be created using this structure similarly to the way that they would be created in other frameworks.

However due to Laravel's unique architecture, it is possible for developers to create their own infrastructure that is specifically designed for their application. This may be most beneficial to large projects such as content-management-systems. This kind of architectural flexibility is unique to Laravel.

Throughout the documentation we'll specify the default locations for declarations where appropriate.

Laravel's Community

Laravel is lucky to be supported by rapidly growing, friendly and enthusiastic community. The Laravel Forumsare a great place to find help, make a suggestion, or just see what other people are saying.

Many of us hang out every day in the #laravel IRC channel on FreeNode. Here's a forum post explaining how you can join us. Hanging out in the IRC channel is a really great way to learn more about web-development using Laravel. You're welcome to ask questions, answer other people's questions, or just hang out and learn from other people's questions being answered. We love Laravel and would love to talk to you about it, so don't be a stranger!

License Information

Laravel is open-sourced software licensed under the MIT License.

LARAVEL CHANGE LOG

Laravel 3.2.8

- Fix double slash bug in URLs when using languages and no "index.php".
- Fix possible security issue in Auth "remember me" cookies.

Upgrading From 3.2.7

- Replace the **laravel** folder.

Laravel 3.2.7

- Fix bug in Eloquent `to_array` method.
- Fix bug in displaying of generic error page.

Upgrading From 3.2.6

- Replace the **laravel** folder.

Laravel 3.2.6

- Revert Blade code back to 3.2.3 tag.

Upgrading From 3.2.5

- Replace the **laravel** folder.

Laravel 3.2.5

- Revert nested where code back to 3.2.3 tag.

Upgrading From 3.2.4

- Replace the **laravel** folder.

Laravel 3.2.4

- Speed up many to many eager loading mapping.
- Tweak the Eloquent::changed() method.
- Various bug fixes and improvements.

Upgrading From 3.2.3

- Replace the **laravel** folder.

Laravel 3.2.3

- Fixed eager loading bug in Eloquent.
- Added `laravel.resolving` event for all IoC resolutions.

Upgrading From 3.2.2

- Replace the **laravel** folder.

Laravel 3.2.2

- Overall improvement of Postgres support.
- Fix issue in SQL Server Schema grammar.
- Fix issue with eager loading and `first` or `find`.
- Fix bug causing parameters to not be passed to `IoC::resolve`.
- Allow the specification of hostnames in environment setup.
- Added `DB::last_query` method.
- Added `password` option to Auth configuration.

Upgrading From 3.2.1

- Replace the **laravel** folder.

Laravel 3.2.1

- Fixed bug in cookie retrieval when cookie is set on same request.
- Fixed bug in SQL Server grammar for primary keys.
- Fixed bug in Validator on PHP 5.4.
- If HTTP / HTTPS is not specified for generated links, current protocol is used.
- Fix bug in Eloquent auth driver.
- Added `format` method to message container.

Upgrading From 3.2

- Replace the **laravel** folder.

Laravel 3.2

- Added `to_array` method to the base Eloquent model.
- Added `$hidden` static variable to the base Eloquent model.
- Added `sync` method to has_many_and_belongs_to Eloquent relationship.
- Added `save` method to has_many Eloquent relationship.
- Added `unless` structure to Blade template engine.
- Added Blade comments.
- Added simpler environment management.
- Added `Blade::extend()` method to define custom blade compilers.
- Added `View::exists` method.
- Use Memcached API instead of older Memcache API.
- Added support for bundles outside of the bundle directory.
- Added support for DateTime database query bindings.

- Migrated to the Symfony HttpFoundation component for core request / response handling.
- Fixed the passing of strings into the `Input::except` method.
- Fixed replacement of optional parameters in `URL::transpose` method.
- Improved `update` handling on `Has_Many` and `Has_One` relationships.
- Improved View performance by only loading contents from file once.
- Fix handling of URLs beginning with hashes in `URL::to`.
- Fix the resolution of unset Eloquent attributes.
- Allows pivot table timestamps to be disabled.
- Made the `get_timestamp` Eloquent method static.
- `Request::secure` now takes `application.ssl` configuration option into consideration.
- Simplified the `paths.php` file.
- Only write file caches if number of minutes is greater than zero.
- Added `$default` parameter to Bundle::option method.
- Fixed bug present when using Eloquent models with Twig.
- Allow multiple views to be registered for a single composer.
- Added `Request::set_env` method.
- `Schema::drop` now accepts `$connection` as second parameter.
- Added `Input::merge` method.
- Added `Input::replace` method.
- Added saving, saved, updating, creating, deleting, and deleted events to Eloquent.
- Added new `Sectionable` interface to allow cache drivers to simulate namespacing.
- Added support for `HAVING` SQL clauses.
- Added `array_pluck` helper, similar to pluck method in Underscore.js.
- Allow the registration of custom cache and session drivers.
- Allow the specification of a separate asset base URL for using CDNs.
- Allow a `starter` Closure to be defined in `bundles.php` to be run on Bundle::start.
- Allow the registration of custom database drivers.
- New, driver based authentication system.
- Added Input::json() method for working with applications using Backbone.js or similar.
- Added Response::json method for creating JSON responses.
- Added Response::eloquent method for creating Eloquent responses.
- Fixed bug when using many-to-many relationships on non-default database connection.
- Added true reflection based IoC to container.
- Added `Request::route()->controller` and `Request::route()->controller_action`.
- Added `Event::queue`, `Event::flusher`, and `Event::flush` methods to Event class.

- Added `array_except` and `array_only` helpers, similar to `Input::except` and `Input::only` but for arbitrary arrays.

Upgrading From 3.1

- Add new `asset_url` and `profiler` options to application configuration.
- Replace **auth** configuration file.

Add the following entry to the `aliases` array in `config/application.php`..

```
'Profiler'    => 'Laravel\\Profiling\\Profiler',
```

Add the following code above `Blade::sharpen()` in `application/start.php`..

```
if (Config::get('application.profiler'))
{
    Profiler::attach();
}
```

- Upgrade the **paths.php** file.
- Replace the **laravel** folder.

Laravel 3.1.9

- Fixes cookie session driver bug that caused infinite loop on some occasions.

Upgrading From 3.1.8

- Replace the **laravel** folder.

Laravel 3.1.8

- Fixes possible WSOD when using Blade's @include expression.

Upgrading From 3.1.7

- Replace the **laravel** folder.

Laravel 3.1.7

- Fixes custom validation language line loading from bundles.
- Fixes double-loading of classes when overriding the core.
- Classify migration names.

Upgrading From 3.1.6

- Replace the **laravel** folder.

Laravel 3.1.6

- Fixes many-to-many eager loading in Eloquent.

Upgrading From 3.1.5

- Replace the **laravel** folder.

Laravel 3.1.5

- Fixes bug that could allow secure cookies to be sent over HTTP.

Upgrading From 3.1.4

- Replace the **laravel** folder.

Laravel 3.1.4

- Fixes Response header casing bug.
- Fixes SQL "where in" (...) short-cut bug.

Upgrading From 3.1.3

- Replace the **laravel** folder.

Laravel 3.1.3

- Fixes **delete** method in Eloquent models.

Upgrade From 3.1.2

- Replace the **laravel** folder.

Laravel 3.1.2

- Fixes Eloquent query method constructor conflict.

Upgrade From 3.1.1

- Replace the **laravel** folder.

Laravel 3.1.1

- Fixes Eloquent model hydration bug involving custom setters.

Upgrading From 3.1

- Replace the **laravel** folder.

Laravel 3.1

- Added events to logger for more flexibility.
- Added **database.fetch** configuration option.
- Added controller factories for injecting any IoC.
- Added **link_to_action** HTML helpers.
- Added ability to set default value on Config::get.
- Added the ability to add pattern based filters.
- Improved session ID assignment.
- Added support for "unsigned" integers in schema builder.
- Added config, view, and lang loaders.
- Added more logic to **application/start.php** for more flexibility.
- Added foreign key support to schema builder.
- Postgres "unique" indexes are now added with ADD CONSTRAINT.
- Added "Event::until" method.
- Added "memory" cache and session drivers.
- Added Controller::detect method.
- Added Cache::forever method.
- Controller layouts now resolved in Laravel\Controller __construct.
- Rewrote Eloquent and included in core.
- Added "match" validation rule.
- Fixed table prefix bug.
- Added Form::macro method.
- Added HTML::macro method.
- Added Route::forward method.
- Prepend table name to default index names in schema.
- Added "forelse" to Blade.
- Added View::render_each.
- Able to specify full path to view (path:).
- Added support for Blade template inheritance.
- Added "before" and "after" validation checks for dates.

Upgrading From 3.0

Replace your application/start.php file.

The default **start.php** file has been expanded in order to give you more flexibility over the loading of your language, configuration, and view files. To upgrade your file, copy your current file and paste it at the bottom of a copy of the new Laravel 3.1 start file. Next, scroll up in the **start** file until you see the default Autoloader registrations (line 61 and line 76).

Delete both of these sections since you just pasted your previous auto-loader registrations at the bottom of the file.

Remove the display option from your errors configuration file.

This option is now set at the beginning of your **application/start** file.

Call the parent controller's constructor from your controller.

Simply add a **parent::__construct();** to to any of your controllers that have a constructor.

Prefix Laravel migration created indexes with their table name.

If you have created indexes on tables using the Laravel migration system and you used to the default index naming scheme provided by Laravel, prefix the index names with their table name on your database. So, if the current index name is "id_unique" on the "users" table, make the index name "users_id_unique".

Add alias for Eloquent in your application configuration.

Add the following to the **aliases** array in your **application/config/application.php** file:

```
'Eloquent' => 'Laravel\\Database\\Eloquent\\Model',
'Blade' => 'Laravel\\Blade',
```

Update Eloquent many-to-many tables.

Eloquent now maintains **created_at** and **updated_at** column on many-to-many intermediate tables by default. Simply add these columns to your tables. Also, many-to-many tables are now the singular model names concatenated with an underscore. For example, if the relationship is between User and Role, the intermediate table name should be **role_user**.

Remove Eloquent bundle.

If you are using the Eloquent bundle with your installation, you can remove it from your bundles directory and your **application/bundles.php** file. Eloquent version 2 is included in the core in Laravel 3.1. Your models can also now extend simply **Eloquent** instead of **Eloquent\Model**.

Update your config/strings.php file.

English pluralization and singularization is now automatic. Just completely replace your**application/config/strings.php** file.

Add the fetch option to your database configuration file.

A new **fetch** option allows you to specify in which format you receive your database results. Just copy and paste the option from the new **application/config/database.php** file.

Add database option to your Redis configuration.

If you are using Redis, add the "database" option to your Redis connection configurations. The "database" value can be zero by default.

```
'redis' => array(
    'default' => array(
        'host' => '127.0.0.1',
        'port' => 6379,
        'database' => 0
    ),
),
```

INSTALLATION & SETUP

Requirements

- Apache, nginx, or another compatible web server.
- Laravel takes advantage of the powerful features that have become available in PHP 5.3. Consequently, PHP 5.3 is a requirement.
- Laravel uses the FileInfo library to detect files' mime-types. This is included by default with PHP 5.3. However, Windows users may need to add a line to their php.ini file before the Fileinfo module is enabled. For more information check out the installation / configuration details on PHP.net.
- Laravel uses the Mcrypt library for encryption and hash generation. Mcrypt typically comes pre-installed. If you can't find Mcrypt in the output of phpinfo() then check the vendor site of your LAMP installation or check out the installation / configuration details on PHP.net.

Installation

1. Download Laravel
2. Extract the Laravel archive and upload the contents to your web server.
3. Set the value of the **key** option in the **config/application.php** file to a random, 32 character string.
4. Verify that the `storage/views` directory is writable.
5. Navigate to your application in a web browser.

If all is well, you should see a pretty Laravel splash page. Get ready, there is lots more to learn!

Extra Goodies

Installing the following goodies will help you take full advantage of Laravel, but they are not required:

- SQLite, MySQL, PostgreSQL, or SQL Server PDO drivers.
- Memcached or APC.

Problems?

If you are having problems installing, try the following:

- Make sure the **public** directory is the document root of your web server. (see: Server Configuration below)
- If you are using mod_rewrite, set the **index** option in **application/config/application.php** to an empty string.
- Verify that your storage folder and the folders within are writable by your web server.

Server Configuration

Like most web-development frameworks, Laravel is designed to protect your application code, bundles, and local storage by placing only files that are necessarily public in the web server's DocumentRoot. This prevents some types of server misconfiguration from making your code (including database passwords and other configuration data) accessible through the web server. It's best to be safe.

In this example let's imagine that we installed Laravel to the directory **/Users/JonSnow/Sites/MySite**.

A very basic example of an Apache VirtualHost configuration for MySite might look like this.

```
<VirtualHost *:80>
    DocumentRoot /Users/JonSnow/Sites/MySite/public
    ServerName mysite.dev
</VirtualHost>
```

Notice that while we installed to **/Users/JonSnow/Sites/MySite** our DocumentRoot points to**/Users/JonSnow/Sites/MySite/public**.

While pointing the DocumentRoot to the public folder is a commonly used best-practice, it's possible that you may need to use Laravel on a host that does not allow you to update your DocumentRoot. A collection of algorithms to circumvent this need can be found on the Laravel forums.

Basic Configuration

All of the configuration provided are located in your applications config/ directory. We recommend that you read through these files just to get a basic understanding of the options available to you. Pay special attention to the**application/config/application.php** file as it contains the basic configuration options for your application.

It's **extremely** important that you change the **application key** option before working on your site. This key is used throughout the framework for encryption, hashing, etc. It lives in the **config/application.php** file and should be set to a random, 32 character string. A standards-compliant application key can be automatically generated using the Artisan command-line utility. More information can be found in the Artisan command index.

> **Note:** If you are using mod_rewrite, you should set the index option to an empty string.

Environments

Most likely, the configuration options you need for local development are not the same as the options you need on your production server. Laravel's default environment handling mechanism is URL based, which will make setting up environments a breeze. Pop open the `paths.php` file in the root of your Laravel installation. You should see an array like this:

```
$environments = array(

    'local' => array('http://localhost*', '*.dev'),

);
```

This tells Laravel that any URLs beginning with "localhost" or ending with ".dev" should be considered part of the "local" environment.

Next, create an **application/config/local** directory. Any files and options you place in this directory will override the options in the base **application/config** directory. For example, you may wish to create an **application.php**file within your new **local** configuration directory:

```
return array(

    'url' => 'http://localhost/laravel/public',

);
```

In this example, the local **URL** option will override the **URL** option in **application/config/application.php**. Notice that you only need to specify the options you wish to override.

Isn't it easy? Of course, you are free to create as many environments as you wish!

Cleaner URLs

Most likely, you do not want your application URLs to contain "index.php". You can remove it using HTTP rewrite rules. If you are using Apache to serve your application, make sure to enable mod_rewrite and create a**.htaccess** file like this one in your **public** directory:

```
<IfModule mod_rewrite.c>
     RewriteEngine on

     RewriteCond %{REQUEST_FILENAME} !-f
     RewriteCond %{REQUEST_FILENAME} !-d

     RewriteRule ^(.*)$ index.php/$1 [L]
</IfModule>
```

Is the .htaccess file above not working for you? Try this one:

```
Options +FollowSymLinks
RewriteEngine on

RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

RewriteRule . index.php [L]
```

After setting up HTTP rewriting, you should set the **index** configuration option in**application/config/application.php** to an empty string.

> **Note:** Each web server has a different method of doing HTTP rewrites, and may require a slightly different .htaccess file.

ROUTING

The Basics

Laravel uses the latest features of PHP 5.3 to make routing simple and expressive. It's important that building everything from APIs to complex web applications is as easy as possible. Routes are typically defined in**application/routes.php**.

Unlike many other frameworks with Laravel it's possible to embed application logic in two ways. While controllers are the most common way to implement application logic it's also possible to embed your logic directly into routes. This is **especially** nice for small sites that contain only a few pages as you don't have to create a bunch of controllers just to expose half a dozen methods or put a handful of unrelated methods into the same controller and then have to manually designate routes that point to them.

In the following example the first parameter is the route that you're "registering" with the router. The second parameter is the function containing the logic for that route. Routes are defined without a front-slash. The only exception to this is the default route which is represented with **only** a front-slash.

> **Note:** Routes are evaluated in the order that they are registered, so register any "catch-all" routes at the bottom of your **routes.php** file.

Registering a route that responds to "GET /":

```
Route::get('/', function()
{
    return "Hello World!";
});
```

Registering a route that is valid for any HTTP verb (GET, POST, PUT, and DELETE):

```
Route::any('/', function()
{
    return "Hello World!";
});
```

Registering routes for other request methods:

```
Route::post('user', function()
{
    //
});

Route::put('user/(:num)', function($id)
{
    //
});

Route::delete('user/(:num)', function($id)
{
    //
});
```

Registering a single URI for multiple HTTP verbs:

```
Router::register(array('GET', 'POST'), $uri, $callback);
```

Wildcards

Forcing a URI segment to be any digit:

```
Route::get('user/(:num)', function($id)
{
    //
});
```

Allowing a URI segment to be any alpha-numeric string:

```
Route::get('post/(:any)', function($title)
{
    //
});
```

Catching the remaining URI without limitations:

```
Route::get('files/(:all)', function($path)
{
    //
});
```

Allowing a URI segment to be optional:

```
Route::get('page/(:any?)', function($page = 'index')
{
    //
});
```

The 404 Event

If a request enters your application but does not match any existing route, the 404 event will be raised. You can find the default event handler in your **application/routes.php** file.

The default 404 event handler:

```
Event::listen('404', function()
{
    return Response::error('404');
});
```

You are free to change this to fit the needs of your application!

Further Reading:

- *Events*

Filters

Route filters may be run before or after a route is executed. If a "before" filter returns a value, that value is considered the response to the request and the route is not executed, which is convenient when implementing authentication filters, etc. Filters are typically defined in **application/routes.php**.

Registering a filter:

```
Route::filter('filter', function()
{
    return Redirect::to('home');
});
```

Attaching a filter to a route:

```
Route::get('blocked', array('before' => 'filter', function()
{
    return View::make('blocked');
}));
```

Attaching an "after" filter to a route:

```
Route::get('download', array('after' => 'log', function()
{
    //
}));
```

Attaching multiple filters to a route:

```
Route::get('create', array('before' => 'auth|csrf', function()
{
    //
}));
```

Passing parameters to filters:

```
Route::get('panel', array('before' => 'role:admin', function()
{
    //
}));
```

Pattern Filters

Sometimes you may want to attach a filter to all requests that begin with a given URI. For example, you may want to attach the "auth" filter to all requests with URIs that begin with "admin". Here's how to do it:

Defining a URI pattern based filter:

```
Route::filter('pattern: admin/*', 'auth');
```

Optionally you can register filters directly when attaching filters to a given URI by supplying an array with the name of the filter and a callback.

Defining a filter and URI pattern based filter in one:

```
Route::filter('pattern: admin/*', array('name' => 'auth', function()
{
    //
}));
```

Global Filters

Laravel has two "global" filters that run **before** and **after** every request to your application. You can find them both in the **application/routes.php** file. These filters make great places to start common bundles or add global assets.

> **Note:** The **after** filter receives the **Response** object for the current request.

Route Groups

Route groups allow you to attach a set of attributes to a group of routes, allowing you to keep your code neat and tidy.

```
Route::group(array('before' => 'auth'), function()
{
    Route::get('panel', function()
    {
        //
    });

    Route::get('dashboard', function()
    {
        //
    });
});
```

Named Routes

Constantly generating URLs or redirects using a route's URI can cause problems when routes are later changed. Assigning the route a name gives you a convenient way to refer to the route throughout your application. When a route change occurs the generated links will point to the new route with no further configuration needed.

Registering a named route:

```
Route::get('/', array('as' => 'home', function()
{
    return "Hello World";
}));
```

Generating a URL to a named route:

```
$url = URL::to_route('home');
```

Redirecting to the named route:

```
return Redirect::to_route('home');
```

Once you have named a route, you may easily check if the route handling the current request has a given name.

Determine if the route handling the request has a given name:

```
if (Request::route()->is('home'))
{
    // The "home" route is handling the request!
}
```

HTTPS Routes

When defining routes, you may use the "https" attribute to indicate that the HTTPS protocol should be used when generating a URL or Redirect to that route.

Defining an HTTPS route:

```
Route::get('login', array('https' => true, function()
{
    return View::make('login');
}));
```

Using the "secure" short-cut method:

```
Route::secure('GET', 'login', function()
{
    return View::make('login');
});
```

Bundle Routes

Bundles are Laravel's modular package system. Bundles can easily be configured to handle requests to your application. We'll be going over bundles in more detail in another document. For now, read through this section and just be aware that not only can routes be used to expose functionality in bundles, but they can also be registered from within bundles.

Let's open the **application/bundles.php** file and add something:

Registering a bundle to handle routes:

```
return array(

    'admin' => array('handles' => 'admin'),

);
```

Notice the new **handles** option in our bundle configuration array? This tells Laravel to load the Admin bundle on any requests where the URI begins with "admin".

Now you're ready to register some routes for your bundle, so create a **routes.php** file within the root directory of your bundle and add the following:

Registering a root route for a bundle:

```
Route::get('(:bundle)', function()
{
    return 'Welcome to the Admin bundle!';
});
```

Let's explore this example. Notice the **(:bundle)** place-holder? That will be replaced with the value of the**handles** clause that you used to register your bundle. This keeps your code D.R.Y. and allows those who use your bundle to change it's root URI without breaking your routes! Nice, right?

Of course, you can use the **(:bundle)** place-holder for all of your routes, not just your root route.

Registering bundle routes:

```
Route::get('(:bundle)/panel', function()
{
    return "I handle requests to admin/panel!";
});
```

Controller Routing

Controllers provide another way to manage your application logic. If you're unfamiliar with controllers you may want to read about controllers and return to this section.

It is important to be aware that all routes in Laravel must be explicitly defined, including routes to controllers. This means that controller methods that have not been exposed through route registration **cannot** be accessed. It's possible to automatically expose all methods within a controller using controller route registration. Controller route registrations are typically defined in **application/routes.php**.

Most likely, you just want to register all of the controllers in your application's "controllers" directory. You can do it in one simple statement. Here's how:

Register all controllers for the application:

```
Route::controller(Controller::detect());
```

The **Controller::detect** method simply returns an array of all of the controllers defined for the application.

If you wish to automatically detect the controllers in a bundle, just pass the bundle name to the method. If no bundle is specified, the application folder's controller directory will be searched.

> **Note:** It is important to note that this method gives you no control over the order in which controllers are loaded. Controller::detect() should only be used to Route controllers in very small sites. "Manually" routing controllers gives you much more control, is more self-documenting, and is certainly advised.

Register all controllers for the "admin" bundle:

```
Route::controller(Controller::detect('admin'));
```

Registering the "home" controller with the Router:

```
Route::controller('home');
```

Registering several controllers with the router:

```
Route::controller(array('dashboard.panel', 'admin'));
```

Once a controller is registered, you may access its methods using a simple URI convention:

```
http://localhost/controller/method/arguments
```

This convention is similar to that employed by CodeIgniter and other popular frameworks, where the first segment is the controller name, the second is the method, and the remaining segments are passed to the method as arguments. If no method segment is present, the "index" method will be used.

This routing convention may not be desirable for every situation, so you may also explicitly route URIs to controller actions using a simple, intuitive syntax.

Registering a route that points to a controller action:

```
Route::get('welcome', 'home@index');
```

Registering a filtered route that points to a controller action:

```
Route::get('welcome', array('after' => 'log', 'uses' => 'home@index'));
```

Registering a named route that points to a controller action:

```
Route::get('welcome', array('as' => 'home.welcome', 'uses' => 'home@index'));
```

CLI Route Testing

You may test your routes using Laravel's "Artisan" CLI. Simple specify the request method and URI you want to use. The route response will be var_dump'd back to the CLI.

Calling a route via the Artisan CLI:

```
php artisan route:call get api/user/1
```

CONTROLLERS

The Basics

Controllers are classes that are responsible for accepting user input and managing interactions between models, libraries, and views. Typically, they will ask a model for data, and then return a view that presents that data to the user.

The usage of controllers is the most common method of implementing application logic in modern web-development. However, Laravel also empowers developers to implement their application logic within routing declarations. This is explored in detail in the routing document. New users are encouraged to start with controllers. There is nothing that route-based application logic can do that controllers can't.

Controller classes should be stored in **application/controllers** and should extend the Base_Controller class. A Home_Controller class is included with Laravel.

Creating a simple controller:

```
class Admin_Controller extends Base_Controller
{
    public function action_index()
    {
        //
    }

}
```

Actions are the name of controller methods that are intended to be web-accessible. Actions should be prefixed with "action_". All other methods, regardless of scope, will not be web-accessible.

> **Note:** The Base_Controller class extends the main Laravel Controller class, and gives you a convenient place to put methods that are common to many controllers.

Controller Routing

It is important to be aware that all routes in Laravel must be explicitly defined, including routes to controllers.

This means that controller methods that have not been exposed through route registration **cannot** be accessed. It's possible to automatically expose all methods within a controller using controller route registration. Controller route registrations are typically defined in **application/routes.php**.

Check the routing page for more information on routing to controllers.

Bundle Controllers

Bundles are Laravel's modular package system. Bundles can be easily configured to handle requests to your application. We'll be going over bundles in more detail in another document.

Creating controllers that belong to bundles is almost identical to creating your application controllers. Just prefix the controller class name with the name of the bundle, so if your bundle is named "admin", your controller classes would look like this:

Creating a bundle controller class:

```
class Admin_Home_Controller extends Base_Controller
{

    public function action_index()
    {
        return "Hello Admin!";
    }

}
```

But, how do you register a bundle controller with the router? It's simple. Here's what it looks like:

Registering a bundle's controller with the router:

```
Route::controller('admin::home');
```

Great! Now we can access our "admin" bundle's home controller from the web!

> **Note:** Throughout Laravel the double-colon syntax is used to denote bundles. More information on bundles can be found in the bundle documentation.

Action Filters

Action filters are methods that can be run before or after a controller action. With Laravel you don't only have control over which filters are assigned to which actions. But, you can also choose which http verbs (post, get, put, and delete) will activate a filter.

You can assign "before" and "after" filters to controller actions within the controller's constructor.

Attaching a filter to all actions:

```
$this->filter('before', 'auth');
```

In this example the 'auth' filter will be run before every action within this controller. The auth action comes out-of-the-box with Laravel and can be found in **application/routes.php**. The auth filter verifies that a user is logged in and redirects them to 'login' if they are not.

Attaching a filter to only some actions:

```
$this->filter('before', 'auth')->only(array('index', 'list'));
```

In this example the auth filter will be run before the action_index() or action_list() methods are run. Users must be logged in before having access to these pages. However, no other actions within this controller require an authenticated session.

Attaching a filter to all except a few actions:

```
$this->filter('before', 'auth')->except(array('add', 'posts'));
```

Much like the previous example, this declaration ensures that the auth filter is run on only some of this controller's actions. Instead of declaring to which actions the filter applies we are instead declaring the actions that will not require authenticated sessions. It can sometimes be safer to use the 'except' method as it's possible to add new actions to this controller and to forget to add them to only(). This could potentially lead your controller's action being unintentionally accessible by users who haven't been authenticated.

Attaching a filter to run on POST:

```
$this->filter('before', 'csrf')->on('post');
```

This example shows how a filter can be run only on a specific http verb. In this case we're running the csrf filter only when a form post is made. The csrf filter is designed to prevent form posts from other systems (spam bots for example) and comes by default with Laravel. You can find the csrf filter in **application/routes.php**.

Further Reading:

- *Route Filters*

Nested Controllers

Controllers may be located within any number of sub-directories within the main **application/controllers** folder.

Define the controller class and store it in **controllers/admin/panel.php**.

```
class Admin_Panel_Controller extends Base_Controller
{

    public function action_index()
    {
        //
    }

}
```

Register the nested controller with the router using "dot" syntax:

```
Route::controller('admin.panel');
```

> **Note:** When using nested controllers, always register your controllers from most nested to least nested in order to avoid shadowing controller routes.

Access the "index" action of the controller:

```
http://localhost/admin/panel
```

Controller Layouts

Full documentation on using layouts with Controllers can be found on the Templating page.

RESTful Controllers

Instead of prefixing controller actions with "action_", you may prefix them with the HTTP verb they should respond to.

Adding the RESTful property to the controller:

```
class Home_Controller extends Base_Controller
{

    public $restful = true;

}
```

Building RESTful controller actions:

```
class Home_Controller extends Base_Controller
{

    public $restful = true;

    public function get_index()
    {
        //
    }

    public function post_index()
    {
        //
    }

}
```

This is particularly useful when building CRUD methods as you can separate the logic which populates and renders a form from the logic that validates and stores the results.

Dependency Injection

If you are focusing on writing testable code, you will probably want to inject dependencies into the constructor of your controller. No problem. Just register your controller in the IoC container. When registering the controller with the container, prefix the key with **controller**. So, in our **application/start.php** file, we could register our user controller like so:

```
IoC::register('controller: user', function()
{
    return new User_Controller;
});
```

When a request to a controller enters your application, Laravel will automatically determine if the controller is registered in the container, and if it is, will use the container to resolve an instance of the controller.

> **Note:** Before diving into controller dependency injection, you may wish to read the documentation on Laravel's beautiful IoC container.

Controller Factory

If you want even more control over the instantiation of your controllers, such as using a third-party IoC container, you'll need to use the Laravel controller factory.

Register an event to handle controller instantiation:

```
Event::listen(Controller::factory, function($controller)
{
    return new $controller;
});
```

The event will receive the class name of the controller that needs to be resolved. All you need to do is return an instance of the controller.

MODELS & LIBRARIES

Models

Models are the heart of your application. Your application logic (controllers / routes) and views (html) are just the mediums with which users interact with your models. The most typical type of logic contained within a model isBusiness Logic.

Some examples of functionality that would exist within a model are:

- Database Interactions
- File I/O
- Interactions with Web Services

For instance, perhaps you are writing a blog. You will likely want to have a "Post" model. Users may want to comment on posts so you'd also have a "Comment" model. If users are going to be commenting then we'll also need a "User" model. Get the idea?

Libraries

Libraries are classes that perform tasks that aren't specific to your application. For instance, consider a PDF generation library that converts HTML. That task, although complicated, is not specific to your application, so it is considered a "library".

Creating a library is as easy as creating a class and storing it in the libraries folder. In the following example, we will create a simple library with a method that echos the text that is passed to it. We create the **printer.php** file in the libraries folder with the following code.

```
<?php

class Printer {

    public static function write($text) {
        echo $text;
    }
}
```

You can now call Printer::write('this text is being echod from the write method!') from anywhere within your application.

Auto Loading

Libraries and Models are very easy to use thanks to the Laravel auto-loader. To learn more about the auto-loader check out the documentation on Auto-Loading.

Best Practices

We've all head the mantra: "controllers should be thin!" But, how do we apply that in real life? It's possible that part of the problem is the word "model". What does it even mean? Is it even a useful term? Many associate "model" with "database", which leads to having very bloated controllers, with light models that access the database. Let's explore some alternatives.

What if we just totally scrapped the "models" directory? Let's name it something more useful. In fact, let's just give it the same as our application. Perhaps are our satellite tracking site is named "Trackler", so let's create a "trackler" directory within the application folder.

Great! Next, let's break our classes into "entities", "services", and "repositories". So, we'll create each of those three directories within our "trackler" folder. Let's explore each one:

Entities

Think of entities as the data containers of your application. They primarily just contain properties. So, in our application, we may have a "Location" entity which has "latitude" and "longitude" properties. It could look something like this:

```php
<?php namespace Trackler\Entities;

class Location {

    public $latitude;
    public $longitude;

    public function __construct($latitude, $longitude)
    {
        $this->latitude = $latitude;
        $this->longitude = $longitude;
    }

}
```

Looking good. Now that we have an entity, let's explore our other two folders.

Services

Services contain the *processes* of your application. So, let's keep using our Trackler example. Our application might have a form on which a user may enter their GPS location. However, we need to validate that the coordinates are correctly formatted. We need to *validate* the *location entity*. So, within our "services" directory, we could create a "validators" folder with the following class:

```php
<?php namespace Trackler\Services\Validators;

use Trackler\Entities\Location;

class Location_Validator {

    public static function validate(Location $location)
    {
        // Validate the location instance...
    }

}
```

Great! Now we have a great way to test our validation in isolation from our controllers and routes! So, we've validated the location and we're ready to store it. What do we do now?

Repositories

Repositories are the data access layer of your application. They are responsible for storing and retrieving the*entities* of your application. So, let's continue using our *location* entity in this example. We need a location repository that can store them. We could store them using any mechanism we want, whether that is a relational database, Redis, or the next storage hotness. Let's look at an example:

```
<?php namespace Trackler\Repositories;

use Trackler\Entities\Location;

class Location_Repository {

    public function save(Location $location, $user_id)
    {
        // Store the location for the given user ID...
    }

}
```

Now we have a clean separation of concerns between our application's entities, services, and repositories. This means we can inject stub repositories into our services or controllers, and test those pieces of our application in isolation from the database. Also, we can entirely switch data store technologies without affecting our services, entities, or controllers. We've achieved a good *separation of concerns*.

Further Reading:

- IoC Container

VIEWS & RESPONSES

The Basics

Views contain the HTML that is sent to the person using your application. By separating your view from the business logic of your application, your code will be cleaner and easier to maintain.

All views are stored within the **application/views** directory and use the PHP file extension. The **View** class provides a simple way to retrieve your views and return them to the client. Let's look at an example!

Creating the view:

```
<html>
    I'm stored in views/home/index.php!
</html>
```

Returning the view from a route:

```
Route::get('/', function()
{
    return View::make('home.index');
});
```

Returning the view from a controller:

```
public function action_index()
{
    return View::make('home.index');
});
```

Determining if a view exists:

```
$exists = View::exists('home.index');
```

Sometimes you will need a little more control over the response sent to the browser. For example, you may need to set a custom header on the response, or change the HTTP status code. Here's how:

Returning a custom response:

```
Route::get('/', function()
{
    $headers = array('foo' => 'bar');

    return Response::make('Hello World!', 200, $headers);
});
```

Returning a custom response containing a view, with binding data:

```
return Response::view('home', array('foo' => 'bar'));
```

Returning a JSON response:

```
return Response::json(array('name' => 'Batman'));
```

Returning Eloquent models as JSON:

```
return Response::eloquent(User::find(1));
```

Binding Data To Views

Typically, a route or controller will request data from a model that the view needs to display. So, we need a way to pass the data to the view. There are several ways to accomplish this, so just pick the way that you like best!

Binding data to a view:

```
Route::get('/', function()
{
    return View::make('home')->with('name', 'James');
});
```

Accessing the bound data within a view:

```
<html>
    Hello, <?php echo $name; ?>.
</html>
```

Chaining the binding of data to a view:

```
View::make('home')
    ->with('name', 'James')
    ->with('votes', 25);
```

Passing an array of data to bind data:

```
View::make('home', array('name' => 'James'));
```

Using magic methods to bind data:

```
$view->name  = 'James';
$view->email = 'example@example.com';
```

Using the ArrayAccess interface methods to bind data:

```
$view['name']  = 'James';
$view['email'] = 'example@example.com';
```

Nesting Views

Often you will want to nest views within views. Nested views are sometimes called "partials", and help you keep views small and modular.

Binding a nested view using the "nest" method:

```
View::make('home')->nest('footer', 'partials.footer');
```

Passing data to a nested view:

```
$view = View::make('home');

$view->nest('content', 'orders', array('orders' => $orders));
```

Sometimes you may wish to directly include a view from within another view. You can use the **render** helper function:

Using the "render" helper to display a view:

```
<div class="content">
    <?php echo render('user.profile'); ?>
</div>
```

It is also very common to have a partial view that is responsible for display an instance of data in a list. For example, you may create a partial view responsible for displaying the details about a single order. Then, for example, you may loop through an array of orders, rendering the partial view for each order. This is made simpler using the **render_each** helper:

Rendering a partial view for each item in an array:

```
<div class="orders">
    <?php echo render_each('partials.order', $orders, 'order');
</div>
```

The first argument is the name of the partial view, the second is the array of data, and the third is the variable name that should be used when each array item is passed to the partial view.

Named Views

Named views can help to make your code more expressive and organized. Using them is simple:

Registering a named view:

```
View::name('layouts.default', 'layout');
```

Getting an instance of the named view:

```
return View::of('layout');
```

Binding data to a named view:

```
return View::of('layout', array('orders' => $orders));
```

View Composers

Each time a view is created, its "composer" event will be fired. You can listen for this event and use it to bind assets and common data to the view each time it is created. A common use-case for this functionality is a side-navigation partial that shows a list of random blog posts. You can nest your partial view by loading it in your layout view. Then, define a composer for that partial. The composer can then query the posts table and gather all of the necessary data to render your view. No more random logic strewn about! Composers are typically defined in **application/routes.php**. Here's an example:

Register a view composer for the "home" view:

```
View::composer('home', function($view)
{
    $view->nest('footer', 'partials.footer');
});
```

Now each time the "home" view is created, an instance of the View will be passed to the registered Closure, allowing you to prepare the view however you wish.

Register a composer that handles multiple views:

```
View::composer(array('home', 'profile'), function($view)
{
    //
});
```

> **Note:** A view can have more than one composer. Go wild!

Redirects

It's important to note that both routes and controllers require responses to be returned with the 'return' directive. Instead of calling "Redirect::to()"" where you'd like to redirect the user. You'd instead use "return Redirect::to()". This distinction is important as it's different than most other PHP frameworks and it could be easy to accidentally overlook the importance of this practice.

Redirecting to another URI:

```
return Redirect::to('user/profile');
```

Redirecting with a specific status:

```
return Redirect::to('user/profile', 301);
```

Redirecting to a secure URI:

```
return Redirect::to_secure('user/profile');
```

Redirecting to the root of your application:

```
return Redirect::home();
```

Redirecting back to the previous action:

```
return Redirect::back();
```

Redirecting to a named route:

```
return Redirect::to_route('profile');
```

Redirecting to a controller action:

```
return Redirect::to_action('home@index');
```

Sometimes you may need to redirect to a named route, but also need to specify the values that should be used instead of the route's URI wildcards. It's easy to replace the wildcards with proper values:

Redirecting to a named route with wildcard values:

```
return Redirect::to_route('profile', array($username));
```

Redirecting to an action with wildcard values:

```
return Redirect::to_action('user@profile', array($username));
```

Redirecting With Flash Data

After a user creates an account or signs into your application, it is common to display a welcome or status message. But, how can you set the status message so it is available for the next request? Use the with() method to send flash data along with the redirect response.

```
return Redirect::to('profile')->with('status', 'Welcome Back!');
```

You can access your message from the view with the Session get method:

```
$status = Session::get('status');
```

Further Reading:

- *Sessions*

Downloads

Sending a file download response:

```
return Response::download('file/path.jpg');
```

Sending a file download and assigning a file name:

```
return Response::download('file/path.jpg', 'photo.jpg');
```

Errors

To generating proper error responses simply specify the response code that you wish to return. The corresponding view stored in **views/error** will automatically be returned.

Generating a 404 error response:

```
return Response::error('404');
```

Generating a 500 error response:

```
return Response::error('500');
```

INPUT & COOKIES

Input

The **Input** class handles input that comes into your application via GET, POST, PUT, or DELETE requests. Here are some examples of how to access input data using the Input class:

Retrieve a value from the input array:

```
$email = Input::get('email');
```

Note: The "get" method is used for all request types (GET, POST, PUT, and DELETE), not just GET requests.

Retrieve all input from the input array:

```
$input = Input::get();
```

Retrieve all input including the $_FILES array:

```
$input = Input::all();
```

By default, *null* will be returned if the input item does not exist. However, you may pass a different default value as a second parameter to the method:

Returning a default value if the requested input item doesn't exist:

```
$name = Input::get('name', 'Fred');
```

Using a Closure to return a default value:

```
$name = Input::get('name', function() {return 'Fred';});
```

Determining if the input contains a given item:

```
if (Input::has('name')) ...
```

Note: The "has" method will return *false* if the input item is an empty string.

JSON Input

When working with JavaScript MVC frameworks like Backbone.js, you will need to get the JSON posted by the application. To make your life easier, we've included the `Input::json` method:

Get JSON input to the application:

```
$data = Input::json();
```

Files

Retrieving all items from the $_FILES array:

```
$files = Input::file();
```

Retrieving an item from the $_FILES array:

```
$picture = Input::file('picture');
```

Retrieving a specific item from a $_FILES array:

```
$size = Input::file('picture.size');
```

Old Input

You'll commonly need to re-populate forms after invalid form submissions. Laravel's Input class was designed with this problem in mind. Here's an example of how you can easily retrieve the input from the previous request. First, you need to flash the input data to the session:

Flashing input to the session:

```
Input::flash();
```

Flashing selected input to the session:

```
Input::flash('only', array('username', 'email'));

Input::flash('except', array('password', 'credit_card'));
```

Retrieving a flashed input item from the previous request:

```
$name = Input::old('name');
```

Note: You must specify a session driver before using the "old" method.

Further Reading:

- *Sessions*

Redirecting With Old Input

Now that you know how to flash input to the session. Here's a shortcut that you can use when redirecting that prevents you from having to micro-manage your old input in that way:

Flashing input from a Redirect instance:

```
return Redirect::to('login')->with_input();
```

Flashing selected input from a Redirect instance:

```
return Redirect::to('login')->with_input('only', array('username'));

return Redirect::to('login')->with_input('except', array('password'));
```

Cookies

Laravel provides a nice wrapper around the $_COOKIE array. However, there are a few things you should be aware of before using it. First, all Laravel cookies contain a "signature hash". This allows the framework to verify that the cookie has not been modified on the client. Secondly, when setting cookies, the cookies are not immediately sent to the browser, but are pooled until the end of the request and then sent together. This means that you will not be able to both set a cookie and retrieve the value that you set in the same request.

Retrieving a cookie value:

```
$name = Cookie::get('name');
```

Returning a default value if the requested cookie doesn't exist:

```
$name = Cookie::get('name', 'Fred');
```

Setting a cookie that lasts for 60 minutes:

```
Cookie::put('name', 'Fred', 60);
```

Creating a "permanent" cookie that lasts five years:

```
Cookie::forever('name', 'Fred');
```

Deleting a cookie:

```
Cookie::forget('name');
```

Merging & Replacing

Sometimes you may wish to merge or replace the current input. Here's how:

Merging new data into the current input:

```
Input::merge(array('name' => 'Spock'));
```

Replacing the entire input array with new data:

```
Input::replace(array('doctor' => 'Bones', 'captain' => 'Kirk'));
```

Clearing Input

To clear all input data for the current request, you may use the `clear` method:

```
Input::clear();
```

BUNDLES

The Basics

Bundles are the heart of the improvements that were made in Laravel 3.0. They are a simple way to group code into convenient "bundles". A bundle can have it's own views, configuration, routes, migrations, tasks, and more. A bundle could be everything from a database ORM to a robust authentication system. Modularity of this scope is an important aspect that has driven virtually all design decisions within Laravel. In many ways you can actually think of the application folder as the special default bundle with which Laravel is pre-programmed to load and use.

Creating Bundles

The first step in creating a bundle is to create a folder for the bundle within your **bundles** directory. For this example, let's create an "admin" bundle, which could house the administrator back-end to our application. The**application/start.php** file provides some basic configuration that helps to define how our application will run. Likewise we'll create a **start.php** file within our new bundle folder for the same purpose. It is run every time the bundle is loaded. Let's create it:

Creating a bundle start.php file:

```
<?php

Autoloader::namespaces(array(
    'Admin' => Bundle::path('admin').'models',
));
```

In this start file we've told the auto-loader that classes that are namespaced to "Admin" should be loaded out of our bundle's models directory. You can do anything you want in your start file, but typically it is used for registering classes with the auto-loader. **In fact, you aren't required to create a start file for your bundle.**

Next, we'll look at how to register this bundle with our application!

Registering Bundles

Now that we have our admin bundle, we need to register it with Laravel. Pull open your**application/bundles.php** file. This is where you register all bundles used by your application. Let's add ours:

Registering a simple bundle:

```
return array('admin'),
```

By convention, Laravel will assume that the Admin bundle is located at the root level of the bundle directory, but we can specify another location if we wish:

Registering a bundle with a custom location:

```
return array(

    'admin' => array('location' => 'userscape/admin'),

);
```

Now Laravel will look for our bundle in **bundles/userscape/admin**.

Bundles & Class Loading

Typically, a bundle's **start.php** file only contains auto-loader registrations. So, you may want to just skip**start.php** and declare your bundle's mappings right in its registration array. Here's how:

Defining auto-loader mappings in a bundle registration:

```
return array(

    'admin' => array(
        'autoloads' => array(
            'map' => array(
                'Admin' => '(:bundle)/admin.php',
            ),
            'namespaces' => array(
                'Admin' => '(:bundle)/lib',
            ),
            'directories' => array(
                '(:bundle)/models',
            ),
        ),
    ),

);
```

Notice that each of these options corresponds to a function on the Laravel auto-loader. In fact, the value of the option will automatically be passed to the corresponding function on the auto-loader.

You may have also noticed the **(:bundle)** place-holder. For convenience, this will automatically be replaced with the path to the bundle. It's a piece of cake.

Starting Bundles

So our bundle is created and registered, but we can't use it yet. First, we need to start it:

Starting a bundle:

```
Bundle::start('admin');
```

This tells Laravel to run the **start.php** file for the bundle, which will register its classes in the auto-loader. The start method will also load the **routes.php** file for the bundle if it is present.

Note: The bundle will only be started once. Subsequent calls to the start method will be ignored.

If you use a bundle throughout your application, you may want it to start on every request. If this is the case, you can configure the bundle to auto-start in your **application/bundles.php** file:

Configuration a bundle to auto-start:

```
return array(

    'admin' => array('auto' => true),

);
```

You do not always need to explicitly start a bundle. In fact, you can usually code as if the bundle was auto-started and Laravel will take care of the rest. For example, if you attempt to use a bundle views, configurations, languages, routes or filters, the bundle will automatically be started!

Each time a bundle is started, it fires an event. You can listen for the starting of bundles like so:

Listen for a bundle's start event:

```
Event::listen('laravel.started: admin', function()
{
    // The "admin" bundle has started...
});
```

It is also possible to "disable" a bundle so that it will never be started.

Disabling a bundle so it can't be started:

```
Bundle::disable('admin');
```

Routing To Bundles

Refer to the documentation on bundle routing and bundle controllers for more information on routing and bundles.

Using Bundles

As mentioned previously, bundles can have views, configuration, language files and more. Laravel uses a double-colon syntax for loading these items. So, let's look at some examples:

Loading a bundle view:

```
return View::make('bundle::view');
```

Loading a bundle configuration item:

```
return Config::get('bundle::file.option');
```

Loading a bundle language line:

```
return Lang::line('bundle::file.line');
```

Sometimes you may need to gather more "meta" information about a bundle, such as whether it exists, its location, or perhaps its entire configuration array. Here's how:

Determine whether a bundle exists:

```
Bundle::exists('admin');
```

Retrieving the installation location of a bundle:

```
$location = Bundle::path('admin');
```

Retrieving the configuration array for a bundle:

```
$config = Bundle::get('admin');
```

Retrieving the names of all installed bundles:

```
$names = Bundle::names();
```

Bundle Assets

If your bundle contains views, it is likely you have assets such as JavaScript and images that need to be available in the **public** directory of the application. No problem. Just create **public** folder within your bundle and place all of your assets in this folder.

Great! But, how do they get into the application's **public** folder. The Laravel "Artisan" command-line provides a simple command to copy all of your bundle's assets to the public directory. Here it is:

Publish bundle assets into the public directory:

```
php artisan bundle:publish
```

This command will create a folder for the bundle's assets within the application's **public/bundles** directory. For example, if your bundle is named "admin", a **public/bundles/admin** folder will be created, which will contain all of the files in your bundle's public folder.

For more information on conveniently getting the path to your bundle assets once they are in the public directory, refer to the documentation on asset management.

Installing Bundles

Of course, you may always install bundles manually; however, the "Artisan" CLI provides an awesome method of installing and upgrading your bundle. The framework uses simple Zip extraction to install the bundle. Here's how it works.

Installing a bundle via Artisan:

```
php artisan bundle:install eloquent
```

Great! Now that you're bundle is installed, you're ready to register it and publish its assets.

Need a list of available bundles? Check out the Laravel bundle directory

Upgrading Bundles

When you upgrade a bundle, Laravel will automatically remove the old bundle and install a fresh copy.

Upgrading a bundle via Artisan:

```
php artisan bundle:upgrade eloquent
```

> **Note:** After upgrading the bundle, you may need to re-publish its assets.

Important: Since the bundle is totally removed on an upgrade, you must be aware of any changes you have made to the bundle code before upgrading. You may need to change some configuration options in a bundle. Instead of modifying the bundle code directly, use the bundle start events to set them. Place something like this in your **application/start.php** file.

Listening for a bundle's start event:

```
Event::listen('laravel.started: admin', function()
{
    Config::set('admin::file.option', true);
});
```

CLASS AUTO LOADING

The Basics

Auto-loading allows you to lazily load class files when they are needed without explicitly *requiring* or *including*them. So, only the classes you actually need are loaded for any given request to your application, and you can just jump right in and start using any class without loading it's related file.

By default, the **models** and **libraries** directories are registered with the auto-loader in the **application/start.php**file. The loader uses a class to file name loading convention, where all file names are lower-cased. So for instance, a "User" class within the models directory should have a file name of "user.php". You may also nest classes within sub-directories. Just namespace the classes to match the directory structure. So, a "Entities\User" class would have a file name of "entities/user.php" within the models directory.

Registering Directories

As noted above, the models and libraries directories are registered with the auto-loader by default; however, you may register any directories you like to use the same class to file name loading conventions:

Registering directories with the auto-loader:

```
Autoloader::directories(array(
    path('app').'entities',
    path('app').'repositories',
));
```

Registering Mappings

Sometimes you may wish to manually map a class to its related file. This is the most performant way of loading classes:

Registering a class to file mapping with the auto-loader:

```
Autoloader::map(array(
    'User'    => path('app').'models/user.php',
    'Contact' => path('app').'models/contact.php',
));
```

Registering Namespaces

Many third-party libraries use the PSR-0 standard for their structure. PSR-0 states that class names should match their file names, and directory structure is indicated by namespaces. If you are using a PSR-0 library, just register it's root namespace and directory with the auto-loader:

Registering a namespace with the auto-loader:

```
Autoloader::namespaces(array(
    'Doctrine' => path('libraries').'Doctrine',
));
```

Before namespaces were available in PHP, many projects used underscores to indicate directory structure. If you are using one of these legacy libraries, you can still easily register it with the auto-loader. For example, if you are using SwiftMailer, you may have noticed all classes begin with "Swift_". So, we'll register "Swift" with the auto-loader as the root of an underscored project.

Registering an "underscored" library with the auto-loader:

```
Autoloader::underscored(array(
    'Swift' => path('libraries').'SwiftMailer',
));
```

ERRORS & LOGGING

Basic Configuration

All of the configuration options regarding errors and logging live in the **application/config/errors.php** file. Let's jump right in.

Ignored Errors

The **ignore** option contains an array of error levels that should be ignored by Laravel. By "ignored", we mean that we won't stop execution of the script on these errors. However, they will be logged when logging is enabled.

Error Detail

The **detail** option indicates if the framework should display the error message and stack trace when an error occurs. For development, you will want this to be **true**. However, in a production environment, set this to **false**. When disabled, the view located in **application/views/error/500.php** will be displayed, which contains a generic error message.

Logging

To enable logging, set the **log** option in the error configuration to "true". When enabled, the Closure defined by the **logger** configuration item will be executed when an error occurs. This gives you total flexibility in how the error should be logged. You can even e-mail the errors to your development team!

By default, logs are stored in the **storage/logs** directory, and a new log file is created for each day. This keeps your log files from getting crowded with too many messages.

The Logger Class

Sometimes you may wish to use Laravel's **Log** class for debugging, or just to log informational messages. Here's how to use it:

Writing a message to the logs:

```
Log::write('info', 'This is just an informational message!');
```

Using magic methods to specify the log message type:

```
Log::info('This is just an informational message!');
```

RUNTIME CONFIGURATION

The Basics

Sometimes you may need to get and set configuration options at runtime. For this you'll use the **Config** class, which utilizes Laravel's "dot" syntax for accessing configuration files and items.

Retrieving Options

Retrieve a configuration option:

```
$value = Config::get('application.url');
```

Return a default value if the option doesn't exist:

```
$value = Config::get('application.timezone', 'UTC');
```

Retrieve an entire configuration array:

```
$options = Config::get('database');
```

Setting Options

Set a configuration option:

```
Config::set('cache.driver', 'apc');
```

EXAMINING REQUESTS

Working With The URI

Getting the current URI for the request:

```
echo URI::current();
```

Getting a specific segment from the URI:

```
echo URI::segment(1);
```

Returning a default value if the segment doesn't exist:

```
echo URI::segment(10, 'Foo');
```

Getting the full request URI, including query string:

```
echo URI::full();
```

Sometimes you may need to determine if the current URI is a given string, or begins with a given string. Here's an example of how you can use the is() method to accomplish this:

Determine if the URI is "home":

```
if (URI::is('home'))
{
    // The current URI is "home"!
}
```

Determine if the current URI begins with "docs/":

```
if URI::is('docs/*'))
{
    // The current URI begins with "docs/"!
}
```

Other Request Helpers

Getting the current request method:

```
echo Request::method();
```

Accessing the $_SERVER global array:

```
echo Request::server('http_referer');
```

Retrieving the requester's IP address:

```
echo Request::ip();
```

Determining if the current request is using HTTPS:

```
if (Request::secure())
{
    // This request is over HTTPS!
}
```

Determining if the current request is an AJAX request:

```
if (Request::ajax())
{
    // This request is using AJAX!
}
```

Determining if the current requst is via the Artisan CLI:

```
if (Request::cli())
{
    // This request came from the CLI!
}
```

GENERATING URLS

The Basics

Retrieving the application's base URL:

```
$url = URL::base();
```

Generating a URL relative to the base URL:

```
$url = URL::to('user/profile');
```

Generating a HTTPS URL:

```
$url = URL::to_secure('user/login');
```

Retrieving the current URL:

```
$url = URL::current();
```

Retrieving the current URL including query string:

```
$url = URL::full();
```

URLs To Routes

Generating a URL to a named route:

```
$url = URL::to_route('profile');
```

Sometimes you may need to generate a URL to a named route, but also need to specify the values that should be used instead of the route's URI wildcards. It's easy to replace the wildcards with proper values:

Generating a URL to a named route with wildcard values:

```
$url = URL::to_route('profile', array($username));
```

Further Reading:

- Named Routes

URLs To Controller Actions

Generating a URL to a controller action:

```
$url = URL::to_action('user@profile');
```

Generating a URL to an action with wildcard values:

```
$url = URL::to_action('user@profile', array($username));
```

URLs To Assets

URLs generated for assets will not contain the "application.index" configuration option.

Generating a URL to an asset:

```
$url = URL::to_asset('js/jquery.js');
```

URL Helpers

There are several global functions for generating URLs designed to make your life easier and your code cleaner:

Generating a URL relative to the base URL:

```
$url = url('user/profile');
```

Generating a URL to an asset:

```
$url = asset('js/jquery.js');
```

Generating a URL to a named route:

```
$url = route('profile');
```

Generating a URL to a named route with wildcard values:

```
$url = route('profile', array($username));
```

Generating a URL to a controller action:

```
$url = action('user@profile');
```

Generating a URL to an action with wildcard values:

```
$url = action('user@profile', array($username));
```

EVENTS

The Basics

Events can provide a great away to build de-coupled applications, and allow plug-ins to tap into the core of your application without modifying its code.

Firing Events

To fire an event, just tell the **Event** class the name of the event you want to fire:

Firing an event:

```
$responses = Event::fire('loaded');
```

Notice that we assigned the result of the **fire** method to a variable. This method will return an array containing the responses of all the event's listeners.

Sometimes you may want to fire an event, but just get the first response. Here's how:

Firing an event and retrieving the first response:

```
$response = Event::first('loaded');
```

Note: The **first** method will still fire all of the handlers listening to the event, but will only return the first response.

The **Event::until** method will execute the event handlers until the first non-null response is returned.

Firing an event until the first non-null response:

```
$response = Event::until('loaded');
```

Listening To Events

So, what good are events if nobody is listening? Register an event handler that will be called when an event fires:

Registering an event handler:

```
Event::listen('loaded', function()
{
    // I'm executed on the "loaded" event!
});
```

The Closure we provided to the method will be executed each time the "loaded" event is fired.

Queued Events

Sometimes you may wish to "queue" an event for firing, but not fire it immediately. This is possible using the `queue` and `flush` methods. First, throw an event on a given queue with a unique identifier:

Registering a queued event:

```
Event::queue('foo', $user->id, array($user));
```

This method accepts three parameters. The first is the name of the queue, the second is a unique identifier for this item on the queue, and the third is an array of data to pass to the queue flusher.

Next, we'll register a flusher for the `foo` queue:

Registering an event flusher:

```
Event::flusher('foo', function($key, $user)
{
    //
});
```

Note that the event flusher receives two arguments. The first, is the unique identifier for the queued event, which in this case would be the user's ID. The second (and any remaining) parameters would be the payload items for the queued event.

Finally, we can run our flusher and flush all queued events using the `flush` method:

```
Event::flush('foo');
```

Laravel Events

There are several events that are fired by the Laravel core. Here they are:

Event fired when a bundle is started:

```
Event::listen('laravel.started: bundle', function() {});
```

Event fired when a database query is executed:

```
Event::listen('laravel.query', function($sql, $bindings, $time) {});
```

Event fired right before response is sent to browser:

```
Event::listen('laravel.done', function($response) {});
```

Event fired when a messaged is logged using the Log class:

```
Event::listen('laravel.log', function($type, $message) {});
```

VALIDATION

The Basics

Almost every interactive web application needs to validate data. For instance, a registration form probably requires the password to be confirmed. Maybe the e-mail address must be unique. Validating data can be a cumbersome process. Thankfully, it isn't in Laravel. The Validator class provides an awesome array of validation helpers to make validating your data a breeze. Let's walk through an example:

Get an array of data you want to validate:

```
$input = Input::all();
```

Define the validation rules for your data:

```
$rules = array(
    'name'  => 'required|max:50',
    'email' => 'required|email|unique:users',
);
```

Create a Validator instance and validate the data:

```
$validation = Validator::make($input, $rules);

if ($validation->fails())
{
    return $validation->errors;
}
```

With the *errors* property, you can access a simple message collector class that makes working with your error messages a piece of cake. Of course, default error messages have been setup for all validation rules. The default messages live at **language/en/validation.php**.

Now you are familiar with the basic usage of the Validator class. You're ready to dig in and learn about the rules you can use to validate your data!

Validation Rules

- Required
- Alpha, Alpha Numeric, & Alpha Dash
- Size
- Numeric
- Inclusion & Exclusion
- Confirmation
- Acceptance
- Same & Different
- Regular Expression Match
- Uniqueness & Existence
- Dates
- E-Mail Addresses
- URLs
- Uploads

Required

Validate that an attribute is present and is not an empty string:

```
'name' => 'required'
```

Validate that an attribute is present, when another attribute is present:

```
'last_name' => 'required_with:first_name'
```

Alpha, Alpha Numeric, & Alpha Dash

Validate that an attribute consists solely of letters:

```
'name' => 'alpha'
```

Validate that an attribute consists of letters and numbers:

```
'username' => 'alpha_num'
```

Validate that an attribute only contains letters, numbers, dashes, or underscores:

```
'username' => 'alpha_dash'
```

Size

Validate that an attribute is a given length, or, if an attribute is numeric, is a given value:

```
'name' => 'size:10'
```

Validate that an attribute size is within a given range:

```
'payment' => 'between:10,50'
```

> **Note:** All minimum and maximum checks are inclusive.

Validate that an attribute is at least a given size:

```
'payment' => 'min:10'
```

Validate that an attribute is no greater than a given size:

```
'payment' => 'max:50'
```

Numeric

Validate that an attribute is numeric:

```
'payment' => 'numeric'
```

Validate that an attribute is an integer:

```
'payment' => 'integer'
```

Inclusion & Exclusion

Validate that an attribute is contained in a list of values:

```
'size' => 'in:small,medium,large'
```

Validate that an attribute is not contained in a list of values:

```
'language' => 'not_in:cobol,assembler'
```

Confirmation

The *confirmed* rule validates that, for a given attribute, a
matching *attribute_confirmation* attribute exists.

Validate that an attribute is confirmed:

```
'password' => 'confirmed'
```

Given this example, the Validator will make sure that the *password* attribute matches the *password_confirmation*attribute in the array being validated.

Acceptance

The *accepted* rule validates that an attribute is equal to *yes* or *1*. This rule is helpful for validating checkbox form fields such as "terms of service".

Validate that an attribute is accepted:

```
'terms' => 'accepted'
```

Same & Different

Validate that an attribute matches another attribute:

```
'token1' => 'same:token2'
```

Validate that two attributes have different values:

```
'password' => 'different:old_password',
```

Regular Expression Match

The *match* rule validates that an attribute matches a given regular expression.

Validate that an attribute matches a regular expression:

```
'username' => 'match:/[a-z]+/';
```

Uniqueness & Existence

Validate that an attribute is unique on a given database table:

```
'email' => 'unique:users'
```

In the example above, the *email* attribute will be checked for uniqueness on the *users* table. Need to verify uniqueness on a column name other than the attribute name? No problem:

Specify a custom column name for the unique rule:

```
'email' => 'unique:users,email_address'
```

Many times, when updating a record, you want to use the unique rule, but exclude the row being updated. For example, when updating a user's profile, you may allow them to change their e-mail address. But, when the*unique* rule runs, you want it to skip the given user since they may not have changed their address, thus causing the *unique* rule to fail. It's easy:

Forcing the unique rule to ignore a given ID:

```
'email' => 'unique:users,email_address,10'
```

Validate that an attribute exists on a given database table:

```
'state' => 'exists:states'
```

Specify a custom column name for the exists rule:

```
'state' => 'exists:states,abbreviation'
```

Dates

Validate that a date attribute is before a given date:

```
'birthdate' => 'before:1986-28-05';
```

Validate that a date attribute is after a given date:

```
'birthdate' => 'after:1986-28-05';
```

> **Note:** The **before** and **after** validation rules use the **strtotime** PHP function to convert your date to something the rule can understand.

E-Mail Addresses

Validate that an attribute is an e-mail address:

```
'address' => 'email'
```

> **Note:** This rule uses the PHP built-in *filter_var* method.

URLs

Validate that an attribute is a URL:

```
'link' => 'url'
```

Validate that an attribute is an active URL:

```
'link' => 'active_url'
```

> **Note:** The *active_url* rule uses *checkdnsr* to verify the URL is active.

Uploads

The *mimes* rule validates that an uploaded file has a given MIME type. This rule uses the PHP Fileinfo extension to read the contents of the file and determine the actual MIME type. Any extension defined in the *config/mimes.php*file may be passed to this rule as a parameter:

Validate that a file is one of the given types:

```
'picture' => 'mimes:jpg,gif'
```

> **Note:** When validating files, be sure to use Input::file() or Input::all() to gather the input.

Validate that a file is an image:

```
'picture' => 'image'
```

Validate that a file is no more than a given size in kilobytes:

```
'picture' => 'image|max:100'
```

Retrieving Error Messages

Laravel makes working with your error messages a cinch using a simple error collector class. After calling the*passes* or *fails* method on a Validator instance, you may access the errors via the *errors* property. The error collector has several simple functions for retrieving your messages:

Determine if an attribute has an error message:

```
if ($validation->errors->has('email'))
{
    // The e-mail attribute has errors...
}
```

Retrieve the first error message for an attribute:

```
echo $validation->errors->first('email');
```

Sometimes you may need to format the error message by wrapping it in HTML. No problem. Along with the :message place-holder, pass the format as the second parameter to the method.

Format an error message:

```
echo $validation->errors->first('email', '<p>:message</p>');
```

Get all of the error messages for a given attribute:

```
$messages = $validation->errors->get('email');
```

Format all of the error messages for an attribute:

```
$messages = $validation->errors->get('email', '<p>:message</p>');
```

Get all of the error messages for all attributes:

```
$messages = $validation->errors->all();
```

Format all of the error messages for all attributes:

```
$messages = $validation->errors->all('<p>:message</p>');
```

Validation Walkthrough

Once you have performed your validation, you need an easy way to get the errors back to the view. Laravel makes it amazingly simple. Let's walk through a typical scenario. We'll define two routes:

```
Route::get('register', function()
{
    return View::make('user.register');
});

Route::post('register', function()
{
    $rules = array(...);

    $validation = Validator::make(Input::all(), $rules);

    if ($validation->fails())
    {
        return Redirect::to('register')->with_errors($validation);
    }
});
```

Great! So, we have two simple registration routes. One to handle displaying the form, and one to handle the posting of the form. In the POST route, we run some validation over the input. If the validation fails, we redirect back to the registration form and flash the validation errors to the session so they will be available for us to display.

But, notice we are not explicitly binding the errors to the view in our GET route. However, an errors variable ($errors) will still be available in the view. Laravel intelligently determines if errors exist in the session, and if they do, binds them to the view for you. If no errors exist in the session, an empty message container will still be bound to the view. In your views, this allows you to always assume you have a message container available via the errors variable. We love making your life easier.

For example, if email address validation failed, we can look for 'email' within the $errors session var.

```
$errors->has('email')
```

Using Blade, we can then conditionally add error messages to our view.

```
{{ $errors->has('email') ? 'Invalid Email Address' : 'Condition is false. Can be
left blank' }}
```

This will also work great when we need to conditionally add classes when using something like Twitter Bootstrap.
For example, if the email address failed validation, we may want to add the "error" class from Bootstrap to our*div class="control-group"* statement.

```
<div class="control-group {{ $errors->has('email') ? 'error' : '' }}">
```

When the validation fails, our rendered view will have the appended *error* class.

```
<div class="control-group error">
```

Custom Error Messages

Want to use an error message other than the default? Maybe you even want to use a custom error message for a given attribute and rule. Either way, the Validator class makes it easy.

Create an array of custom messages for the Validator:

```
$messages = array(
    'required' => 'The :attribute field is required.',
);

$validation = Validator::make(Input::get(), $rules, $messages);
```

Great! Now our custom message will be used anytime a required validation check fails. But, what is this**:attribute** stuff in our message? To make your life easier, the Validator class will replace the **:attribute** place-holder with the actual name of the attribute! It will even remove underscores from the attribute name.

You may also use the **:other**, **:size**, **:min**, **:max**, and **:values** place-holders when constructing your error messages:

Other validation message place-holders:

```
$messages = array(
    'same'    => 'The :attribute and :other must match.',
    'size'    => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute must be between :min - :max.',
    'in'      => 'The :attribute must be one of the following types: :values',
);
```

So, what if you need to specify a custom required message, but only for the email attribute? No problem. Just specify the message using an **attribute_rule** naming convention:

Specifying a custom error message for a given attribute:

```
$messages = array(
    'email_required' => 'We need to know your e-mail address!',
);
```

In the example above, the custom required message will be used for the email attribute, while the default message will be used for all other attributes.

However, if you are using many custom error messages, specifying inline may become cumbersome and messy. For that reason, you can specify your custom messages in the **custom** array within the validation language file:

Adding custom error messages to the validation language file:

```
'custom' => array(
    'email_required' => 'We need to know your e-mail address!',
)
```

Custom Validation Rules

Laravel provides a number of powerful validation rules. However, it's very likely that you'll need to eventually create some of your own. There are two simple methods for creating validation rules. Both are solid so use whichever you think best fits your project.

Registering a custom validation rule:

```
Validator::register('awesome', function($attribute, $value, $parameters)
{
    return $value == 'awesome';
});
```

In this example we're registering a new validation rule with the validator. The rule receives three arguments. The first is the name of the attribute being validated, the second is the value of the attribute being validated, and the third is an array of parameters that were specified for the rule.

Here is how your custom validation rule looks when called:

```
$rules = array(
    'username' => 'required|awesome',
);
```

Of course, you will need to define an error message for your new rule. You can do this either in an ad-hoc messages array:

```
$messages = array(
    'awesome' => 'The attribute value must be awesome!',
);

$validator = Validator::make(Input::get(), $rules, $messages);
```

Or by adding an entry for your rule in the **language/en/validation.php** file:

```
'awesome' => 'The attribute value must be awesome!',
```

As mentioned above, you may even specify and receive a list of parameters in your custom rule:

```
// When building your rules array...

$rules = array(
    'username' => 'required|awesome:yes',
);

// In your custom rule...

Validator::register('awesome', function($attribute, $value, $parameters)
{
    return $value == $parameters[0];
});
```

In this case, the parameters argument of your validation rule would receive an array containing one element: "yes".

Another method for creating and storing custom validation rules is to extend the Validator class itself. By extending the class you create a new version of the validator that has all of the pre-existing functionality combined with your own custom additions. You can even choose to replace some of the default methods if you'd like. Let's look at an example:

First, create a class that extends **Laravel\Validator** and place it in your **application/libraries** directory:

Defining a custom validator class:

```
<?php

class Validator extends Laravel\Validator {}
```

Next, remove the Validator alias from **config/application.php**. This is necessary so that you don't end up with 2 classes named "Validator" which will certainly conflict with one another.

Next, let's take our "awesome" rule and define it in our new class:

Adding a custom validation rule:

```
<?php

class Validator extends Laravel\Validator {

    public function validate_awesome($attribute, $value, $parameters)
    {
        return $value == 'awesome';
    }

}
```

Notice that the method is named using the **validate_rule** naming convention. The rule is named "awesome" so the method must be named "validate_awesome". This is one way in which registering your custom rules and extending the Validator class are different. Validator classes simply need to return true or false. That's it!

Keep in mind that you'll still need to create a custom message for any validation rules that you create. The method for doing so is the same no matter how you define your rule!

WORKING WITH FILES

Reading Files

Getting the contents of a file:

```
$contents = File::get('path/to/file');
```

Writing Files

Writing to a file:

```
File::put('path/to/file', 'file contents');
```

Appending to a file:

```
File::append('path/to/file', 'appended file content');
```

Removing Files

Deleting a single file:

```
File::delete('path/to/file');
```

File Uploads

Moving a $_FILE to a permanent location:

```
Input::upload('picture', 'path/to/pictures', 'filename.ext');
```

Note: You can easily validate file uploads using the Validator class.

File Extensions

Getting the extension from a filename:

```
File::extension('picture.png');
```

Checking File Types

Determining if a file is given type:

```
if (File::is('jpg', 'path/to/file.jpg'))
{
    //
}
```

The **is** method does not simply check the file extension. The Fileinfo PHP extension will be used to read the content of the file and determine the actual MIME type.

Note: You may pass any of the extensions defined in the **application/config/mimes.php** file to the **is**method.
Note: The Fileinfo PHP extension is required for this functionality. More information can be found on thePHP Fileinfo page.

Getting MIME Types

Getting the MIME type associated with an extension:

```
echo File::mime('gif'); // outputs 'image/gif'
```

Note: This method simply returns the MIME type defined for the extension in the**application/config/mimes.php** file.

Copying Directories

Recursively copy a directory to a given location:

```
File::cpdir($directory, $destination);
```

Removing Directories

Recursively delete a directory:

```
File::rmdir($directory);
```

WORKING WITH STRINGS

Capitalization, Etc.

The **Str** class also provides three convenient methods for manipulating string capitalization: **upper**, **lower**, and**title**. These are more intelligent versions of the PHP strtoupper, strtolower, and ucwords methods. More intelligent because they can handle UTF-8 input if the multi-byte string PHP extension is installed on your web server. To use them, just pass a string to the method:

```
echo Str::lower('I am a string.');

echo Str::upper('I am a string.');

echo Str::title('I am a string.');
```

Word & Character Limiting

Limiting the number of characters in a string:

```
echo Str::limit($string, 10);
echo Str::limit_exact($string, 10);
```

Limiting the number of words in a string:

```
echo Str::words($string, 10);
```

Generating Random Strings

Generating a random string of alpha-numeric characters:

```
echo Str::random(32);
```

Generating a random string of alphabetic characters:

```
echo Str::random(32, 'alpha');
```

Singular & Plural

The String class is capable of transforming your strings from singular to plural, and vice versa.

Getting the plural form of a word:

```
echo Str::plural('user');
```

Getting the singular form of a word:

```
echo Str::singular('users');
```

Getting the plural form if given value is greater than one:

```
echo Str::plural('comment', count($comments));
```

Slugs

Generating a URL friendly slug:

```
return Str::slug('My First Blog Post!');
```

Generating a URL friendly slug using a given separator:

```
return Str::slug('My First Blog Post!', '_');
```

LOCALIZATION

The Basics

Localization is the process of translating your application into different languages.
The **Lang** class provides a simple mechanism to help you organize and retrieve the text of your multilingual application.

All of the language files for your application live under the **application/language** directory. Within the**application/language** directory, you should create a directory for each language your application speaks. So, for example, if your application speaks English and Spanish, you might create **en** and **sp** directories under the**language** directory.

Each language directory may contain many different language files. Each language file is simply an array of string values in that language. In fact, language files are structured identically to configuration files. For example, within the **application/language/en** directory, you could create a **marketing.php** file that looks like this:

Creating a language file:

```
return array(

     'welcome' => 'Welcome to our website!',

);
```

Next, you should create a corresponding **marketing.php** file within the **application/language/sp** directory. The file would look something like this:

```
return array(

     'welcome' => 'Bienvenido a nuestro sitio web!',

);
```

Nice! Now you know how to get started setting up your language files and directories. Let's keep localizing!

Retrieving A Language Line

Retrieving a language line:

```
echo Lang::line('marketing.welcome')->get();
```

Retrieving a language line using the "__" helper:

```
echo __('marketing.welcome');
```

Notice how a dot was used to separate "marketing" and "welcome"? The text before the dot corresponds to the language file, while the text after the dot corresponds to a specific string within that file.

Need to retrieve the line in a language other than your default? Not a problem. Just mention the language to the**get** method:

Getting a language line in a given language:

```
echo Lang::line('marketing.welcome')->get('sp');
```

Place Holders & Replacements

Now, let's work on our welcome message. "Welcome to our website!" is a pretty generic message. It would be helpful to be able to specify the name of the person we are welcoming. But, creating a language line for each user of our application would be time-consuming and ridiculous. Thankfully, you don't have to. You can specify "place-holders" within your language lines. Place-holders are preceded by a colon:

Creating a language line with place-holders:

```
'welcome' => 'Welcome to our website, :name!'
```

Retrieving a language line with replacements:

```
echo Lang::line('marketing.welcome', array('name' => 'Taylor'))->get();
```

Retrieving a language line with replacements using "__":

```
echo __('marketing.welcome', array('name' => 'Taylor'));
```

ENCRYPTION

The Basics

Laravel's **Crypter** class provides a simple interface for handling secure, two-way encryption. By default, the Crypter class provides strong AES-256 encryption and decryption out of the box via the Mcrypt PHP extension.

> **Note:** Don't forget to install the Mcrypt PHP extension on your server.

Encrypting A String

Encrypting a given string:

```
$encrypted = Crypter::encrypt($value);
```

Decrypting A String

Decrypting a string:

```
$decrypted = Crypter::decrypt($encrypted);
```

> **Note:** It's incredibly important to point out that the decrypt method will only decrypt strings that were encrypted using **your** application key.

IOC CONTAINER

Definition

An IoC container is simply a way of managing the creation of objects. You can use it to define the creation of complex objects, allowing you to resolve them throughout your application using a single line of code. You may also use it to "inject" dependencies into your classes and controllers.

IoC containers help make your application more flexible and testable. Since you may register alternate implementations of an interface with the container, you may isolate the code you are testing from external dependencies using stubs and mocks.

Registering Objects

Registering a resolver in the IoC container:

```
IoC::register('mailer', function()
{
    $transport = Swift_MailTransport::newInstance();

    return Swift_Mailer::newInstance($transport);
});
```

Great! Now we have registered a resolver for SwiftMailer in our container. But, what if we don't want the container to create a new mailer instance every time we need one? Maybe we just want the container to return the same instance after the initial instance is created. Just tell the container the object should be a singleton:

Registering a singleton in the container:

```
IoC::singleton('mailer', function()
{
    //
});
```

You may also register an existing object instance as a singleton in the container.

Registering an existing instance in the container:

```
IoC::instance('mailer', $instance);
```

Resolving Objects

Now that we have SwiftMailer registered in the container, we can resolve it using the **resolve** method on the **IoC**class:

```
$mailer = IoC::resolve('mailer');
```

Note: You may also register controllers in the container.

UNIT TESTING

The Basics

Unit Testing allows you to test your code and verify that it is working correctly. In fact, many advocate that you should even write your tests before you write your code! Laravel provides beautiful integration with the popular PHPUnit testing library, making it easy to get started writing your tests. In fact, the Laravel framework itself has hundreds of unit tests!

Creating Test Classes

All of your application's tests live in the **application/tests** directory. In this directory, you will find a basic **example.test.php** file. Pop it open and look at the class it contains:

```
<?php

class TestExample extends PHPUnit_Framework_TestCase {

    /**
     * Test that a given condition is met.
     *
     * @return void
     */
    public function testSomethingIsTrue()
    {
        $this->assertTrue(true);
    }

}
```

Take special note of the **.test.php** file suffix. This tells Laravel that it should include this class as a test case when running your test. Any files in the test directory that are not named with this suffix will not be considered a test case.

If you are writing tests for a bundle, just place them in a **tests** directory within the bundle. Laravel will take care of the rest!

For more information regarding creating test cases, check out the PHPUnit documentation.

Running Tests

To run your tests, you can use Laravel's Artisan command-line utility:

Running the application's tests via the Artisan CLI:

```
php artisan test
```

Running the unit tests for a bundle:

```
php artisan test bundle-name
```

Calling Controllers From Tests

Here's an example of how you can call your controllers from your tests:

Calling a controller from a test:

```
$response = Controller::call('home@index', $parameters);
```

Resolving an instance of a controller from a test:

```
$controller = Controller::resolve('application', 'home@index');
```

> **Note:** The controller's action filters will still run when using Controller::call to execute controller actions.

DATABASE

DATABASE CONFIGURATION

Laravel supports the following databases out of the box:

- MySQL
- PostgreSQL
- SQLite
- SQL Server

All of the database configuration options live in the **application/config/database.php** file.

Quick Start Using SQLite

SQLite is an awesome, zero-configuration database system. By default, Laravel is configured to use a SQLite database. Really, you don't have to change anything. Just drop a SQLite database named **application.sqlite** into the **application/storage/database** directory. You're done.

Of course, if you want to name your database something besides "application", you can modify the database option in the SQLite section of the **application/config/database.php** file:

```
'sqlite' => array(
     'driver'   => 'sqlite',
     'database' => 'your_database_name',
)
```

If your application receives less than 100,000 hits per day, SQLite should be suitable for production use in your application. Otherwise, consider using MySQL or PostgreSQL.

> **Note:** Need a good SQLite manager? Check out this Firefox extension.

Configuring Other Databases

If you are using MySQL, SQL Server, or PostgreSQL, you will need to edit the configuration options in**application/config/database.php**. In the configuration file you can find sample configurations for each of these systems. Just change the options as necessary for your server and set the default connection name.

Setting The Default Connection Name

As you have probably noticed, each database connection defined in the **application/config/database.php** file has a name. By default, there are three connections defined: **sqlite**, **mysql**, **sqlsrv**, and **pgsql**. You are free to change these connection names. The default connection can be specified via the **default** option:

```
'default' => 'sqlite';
```

The default connection will always be used by the fluent query builder. If you need to change the default connection during a request, use the **Config::set** method.

Overwriting The Default PDO Options

The PDO connector class (**laravel/database/connectors/connector.php**) has a set of default PDO attributes defined which can be overwritten in the options array for each system. For example, one of the default attributes is to force column names to lowercase (**PDO::CASE_LOWER**) even if they are defined in UPPERCASE or CamelCase in the table. Therefore, under the default attributes, query result object variables would only be accessible in lowercase.
An example of the MySQL system settings with added default PDO attributes:

```
'mysql' => array(
    'driver'   => 'mysql',
    'host'     => 'localhost',
    'database' => 'database',
    'username' => 'root',
    'password' => '',
    'charset'  => 'utf8',
    'prefix'   => '',
    PDO::ATTR_CASE              => PDO::CASE_LOWER,
    PDO::ATTR_ERRMODE           => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_ORACLE_NULLS      => PDO::NULL_NATURAL,
    PDO::ATTR_STRINGIFY_FETCHES => false,
    PDO::ATTR_EMULATE_PREPARES  => false,
),
```

More about the PDO connection attributes can be found in the PHP manual.

RAW QUERIES

The Basics

The **query** method is used to execute arbitrary, raw SQL against your database connection.

Selecting records from the database:

```
$users = DB::query('select * from users');
```

Selecting records from the database using bindings:

```
$users = DB::query('select * from users where name = ?', array('test'));
```

Inserting a record into the database

```
$success = DB::query('insert into users values (?, ?)', $bindings);
```

Updating table records and getting the number of affected rows:

```
$affected = DB::query('update users set name = ?', $bindings);
```

Deleting from a table and getting the number of affected rows:

```
$affected = DB::query('delete from users where id = ?', array(1));
```

Other Query Methods

Laravel provides a few other methods to make querying your database simple. Here's an overview:

Running a SELECT query and returning the first result:

```
$user = DB::first('select * from users where id = 1');
```

Running a SELECT query and getting the value of a single column:

```
$email = DB::only('select email from users where id = 1');
```

PDO Connections

Sometimes you may wish to access the raw PDO connection behind the Laravel Connection object.

Get the raw PDO connection for a database:

```
$pdo = DB::connection('sqlite')->pdo;
```

> **Note:** If no connection name is specified, the **default** connection will be returned.

FLUENT QUERY BUILDER

The Basics

The Fluent Query Builder is Laravel's powerful fluent interface for building SQL queries and working with your database. All queries use prepared statements and are protected against SQL injection.

You can begin a fluent query using the **table** method on the DB class. Just mention the table you wish to query:

```
$query = DB::table('users');
```

You now have a fluent query builder for the "users" table. Using this query builder, you can retrieve, insert, update, or delete records from the table.

Retrieving Records

Retrieving an array of records from the database:

```
$users = DB::table('users')->get();
```

> **Note:** The **get** method returns an array of objects with properties corresponding to the column on the table.

Retrieving a single record from the database:

```
$user = DB::table('users')->first();
```

Retrieving a single record by its primary key:

```
$user = DB::table('users')->find($id);
```

> **Note:** If no results are found, the **first** method will return NULL. The **get** method will return an empty array.

Retrieving the value of a single column from the database:

```
$email = DB::table('users')->where('id', '=', 1)->only('email');
```

Only selecting certain columns from the database:

```
$user = DB::table('users')->get(array('id', 'email as user_email'));
```

Selecting distinct results from the database:

```
$user = DB::table('users')->distinct()->get();
```

Building Where Clauses

where and or_where

There are a variety of methods to assist you in building where clauses. The most basic of these methods are the**where** and **or_where** methods. Here is how to use them:

```
return DB::table('users')
    ->where('id', '=', 1)
    ->or_where('email', '=', 'example@gmail.com')
    ->first();
```

Of course, you are not limited to simply checking equality. You may also use **greater-than**, **less-than**, **not-equal**, and **like**:

```
return DB::table('users')
    ->where('id', '>', 1)
    ->or_where('name', 'LIKE', '%Taylor%')
    ->first();
```

As you may have assumed, the **where** method will add to the query using an AND condition, while the**or_where** method will use an OR condition.

where_in, where_not_in, or_where_in, and or_where_not_in

The suite of **where_in** methods allows you to easily construct queries that search an array of values:

```
DB::table('users')->where_in('id', array(1, 2, 3))->get();

DB::table('users')->where_not_in('id', array(1, 2, 3))->get();

DB::table('users')
    ->where('email', '=', 'example@gmail.com')
    ->or_where_in('id', array(1, 2, 3))
    ->get();

DB::table('users')
    ->where('email', '=', 'example@gmail.com')
    ->or_where_not_in('id', array(1, 2, 3))
    ->get();
```

where_null, where_not_null, or_where_null, and or_where_not_null

The suite of **where_null** methods makes checking for NULL values a piece of cake:

```
return DB::table('users')->where_null('updated_at')->get();

return DB::table('users')->where_not_null('updated_at')->get();

return DB::table('users')
    ->where('email', '=', 'example@gmail.com')
    ->or_where_null('updated_at')
    ->get();

return DB::table('users')
    ->where('email', '=', 'example@gmail.com')
    ->or_where_not_null('updated_at')
    ->get();
```

Nested Where Clauses

You may discover the need to group portions of a WHERE clause within parentheses. Just pass a Closure as parameter to the **where** or **or_where** methods:

```
$users = DB::table('users')
    ->where('id', '=', 1)
    ->or_where(function($query)
    {
        $query->where('age', '>', 25);
        $query->where('votes', '>', 100);
    })
    ->get();
```

The example above would generate a query that looks like:

```
SELECT * FROM "users" WHERE "id" = ? OR ("age" > ? AND "votes" > ?)
```

Dynamic Where Clauses

Dynamic where methods are great way to increase the readability of your code. Here are some examples:

```
$user = DB::table('users')->where_email('example@gmail.com')->first();

$user = DB::table('users')->where_email_and_password('example@gmail.com',
'secret');

$user = DB::table('users')->where_id_or_name(1, 'Fred');
```

Table Joins

Need to join to another table? Try the **join** and **left_join** methods:

```
DB::table('users')
    ->join('phone', 'users.id', '=', 'phone.user_id')
    ->get(array('users.email', 'phone.number'));
```

The **table** you wish to join is passed as the first parameter. The remaining three parameters are used to construct the **ON** clause of the join.

Once you know how to use the join method, you know how to **left_join**. The method signatures are the same:

```
DB::table('users')
    ->left_join('phone', 'users.id', '=', 'phone.user_id')
    ->get(array('users.email', 'phone.number'));
```

You may also specify multiple conditions for an **ON** clause by passing a Closure as the second parameter of the join:

```
DB::table('users')
    ->join('phone', function($join)
    {
        $join->on('users.id', '=', 'phone.user_id');
        $join->or_on('users.id', '=', 'phone.contact_id');
    })
    ->get(array('users.email', 'phone.number'));
```

Ordering Results

You can easily order the results of your query using the **order_by** method. Simply mention the column and direction (desc or asc) of the sort:

```
return DB::table('users')->order_by('email', 'desc')->get();
```

Of course, you may sort on as many columns as you wish:

```
return DB::table('users')
    ->order_by('email', 'desc')
    ->order_by('name', 'asc')
    ->get();
```

Skip & Take

If you would like to **LIMIT** the number of results returned by your query, you can use the **take** method:

```
return DB::table('users')->take(10)->get();
```

To set the **OFFSET** of your query, use the **skip** method:

```
return DB::table('users')->skip(10)->get();
```

Aggregates

Need to get a **MIN**, **MAX**, **AVG**, **SUM**, or **COUNT** value? Just pass the column to the query:

```
$min = DB::table('users')->min('age');

$max = DB::table('users')->max('weight');

$avg = DB::table('users')->avg('salary');

$sum = DB::table('users')->sum('votes');

$count = DB::table('users')->count();
```

Of course, you may wish to limit the query using a WHERE clause first:

```
$count = DB::table('users')->where('id', '>', 10)->count();
```

Expressions

Sometimes you may need to set the value of a column to a SQL function such as **NOW()**. Usually a reference to now() would automatically be quoted and escaped. To prevent this use the **raw** method on the **DB** class. Here's what it looks like:

```
DB::table('users')->update(array('updated_at' => DB::raw('NOW()')));
```

The **raw** method tells the query to inject the contents of the expression into the query as a string rather than a bound parameter. For example, you can also use expressions to increment column values:

```
DB::table('users')->update(array('votes' => DB::raw('votes + 1')));
```

Of course, convenient methods are provided for **increment** and **decrement**:

```
DB::table('users')->increment('votes');

DB::table('users')->decrement('votes');
```

Inserting Records

The insert method expects an array of values to insert. The insert method will return true or false, indicating whether the query was successful:

```
DB::table('users')->insert(array('email' => 'example@gmail.com'));
```

Inserting a record that has an auto-incrementing ID? You can use the **insert_get_id** method to insert a record and retrieve the ID:

```
$id = DB::table('users')->insert_get_id(array('email' => 'example@gmail.com'));
```

Note: The **insert_get_id** method expects the name of the auto-incrementing column to be "id".

Updating Records

To update records simply pass an array of values to the **update** method:

```
$affected = DB::table('users')->update(array('email' => 'new_email@gmail.com'));
```

Of course, when you only want to update a few records, you should add a WHERE clause before calling the update method:

```
$affected = DB::table('users')
    ->where('id', '=', 1)
    ->update(array('email' => 'new_email@gmail.com'));
```

Deleting Records

When you want to delete records from your database, simply call the **delete** method:

```
$affected = DB::table('users')->where('id', '=', 1)->delete();
```

Want to quickly delete a record by its ID? No problem. Just pass the ID into the delete method:

```
$affected = DB::table('users')->delete(1);
```

ELOQUENT ORM

The Basics

An ORM is an object-relational mapper, and Laravel has one that you will absolutely love to use. It is named "Eloquent" because it allows you to work with your database objects and relationships using an eloquent and expressive syntax. In general, you will define one Eloquent model for each table in your database. To get started, let's define a simple model:

```
class User extends Eloquent {}
```

Nice! Notice that our model extends the **Eloquent** class. This class will provide all of the functionality you need to start working eloquently with your database.

Note: Typically, Eloquent models live in the **application/models** directory.

Conventions

Eloquent makes a few basic assumptions about your database structure:

- Each table should have a primary key named **id**.
- Each table name should be the plural form of its corresponding model name.

Sometimes you may wish to use a table name other than the plural form of your model. No problem. Just add a static **table** property your model:

```
class User extends Eloquent {

     public static $table = 'my_users';

}
```

Retrieving Models

Retrieving models using Eloquent is refreshingly simple. The most basic way to retrieve an Eloquent model is the static **find** method. This method will return a single model by primary key with properties corresponding to each column on the table:

```
$user = User::find(1);

echo $user->email;
```

The find method will execute a query that looks something like this:

```
SELECT * FROM "users" WHERE "id" = 1
```

Need to retrieve an entire table? Just use the static **all** method:

```
$users = User::all();

foreach ($users as $user)
{
     echo $user->email;
}
```

Of course, retrieving an entire table isn't very helpful. Thankfully, **every method that is available through the fluent query builder is available in Eloquent**. Just begin querying your model with a static call to one of thequery builder methods, and execute the query using the **get** or **first** method. The get method will return an array of models, while the first method will return a single model:

```
$user = User::where('email', '=', $email)->first();
```

```
$user = User::where_email($email)->first();

$users = User::where_in('id', array(1, 2, 3))->or_where('email', '=', $email)-
>get();

$users = User::order_by('votes', 'desc')->take(10)->get();
```

Note: If no results are found, the **first** method will return NULL. The **all** and **get** methods return an empty array.

Aggregates

Need to get a **MIN**, **MAX**, **AVG**, **SUM**, or **COUNT** value? Just pass the column to the appropriate method:

```
$min = User::min('id');

$max = User::max('id');

$avg = User::avg('id');

$sum = User::sum('id');

$count = User::count();
```

Of course, you may wish to limit the query using a WHERE clause first:

```
$count = User::where('id', '>', 10)->count();
```

Inserting & Updating Models

Inserting Eloquent models into your tables couldn't be easier. First, instantiate a new model. Second, set its properties. Third, call the **save** method:

```
$user = new User;

$user->email = 'example@gmail.com';
$user->password = 'secret';

$user->save();
```

Alternatively, you may use the **create** method, which will insert a new record into the database and return the model instance for the newly inserted record, or **false** if the insert failed.

```
$user = User::create(array('email' => 'example@gmail.com'));
```

Updating models is just as simple. Instead of instantiating a new model, retrieve one from your database. Then, set its properties and save:

```
$user = User::find(1);

$user->email = 'new_email@gmail.com';
$user->password = 'new_secret';

$user->save();
```

Need to maintain creation and update timestamps on your database records? With Eloquent, you don't have to worry about it. Just add a static **timestamps** property to your model:

```
class User extends Eloquent {

     public static $timestamps = true;

}
```

Next, add **created_at** and **updated_at** date columns to your table. Now, whenever you save the model, the creation and update timestamps will be set automatically. You're welcome.

In some cases it may be useful to update the **updated_at** date column without actually modifying any data within the model. Simply use the **touch** method, which will also automatically save the changes immediately:

```
$comment = Comment::find(1);
$comment->touch();
```

You can also use the **timestamp** function to update the **updated_at** date column without saving the model immediately. Note that if you are actually modifying the model's data this is handled behind the scenes:

```
$comment = Comment::find(1);
$comment->timestamp();
//do something else here, but not modifying the $comment model data
$comment->save();
```

Note: You can change the default timezone of your application in the **application/config/application.php**file.

Relationships

Unless you're doing it wrong, your database tables are probably related to one another. For instance, an order may belong to a user. Or, a post may have many comments. Eloquent makes defining relationships and retrieving related models simple and intuitive. Laravel supports three types of relationships:

- One-To-One
- One-To-Many
- Many-To-Many

To define a relationship on an Eloquent model, you simply create a method that returns the result of either the**has_one**, **has_many**, **belongs_to**, or **has_many_and_belongs_to** method. Let's examine each one in detail.

One-To-One

A one-to-one relationship is the most basic form of relationship. For example, let's pretend a user has one phone. Simply describe this relationship to Eloquent:

```
class User extends Eloquent {

     public function phone()
     {
          return $this->has_one('Phone');
     }

}
```

Notice that the name of the related model is passed to the **has_one** method. You can now retrieve the phone of a user through the **phone** method:

```
$phone = User::find(1)->phone()->first();
```

Let's examine the SQL performed by this statement. Two queries will be performed: one to retrieve the user and one to retrieve the user's phone:

```
SELECT * FROM "users" WHERE "id" = 1

SELECT * FROM "phones" WHERE "user_id" = 1
```

Note that Eloquent assumes the foreign key of the relationship will be **user_id**. Most foreign keys will follow this**model_id** convention; however, if you want to use a different column name as the foreign key, just pass it in the second parameter to the method:

```
return $this->has_one('Phone', 'my_foreign_key');
```

Want to just retrieve the user's phone without calling the first method? No problem. Just use the **dynamic phone property**. Eloquent will automatically load the relationship for you, and is even smart enough to know whether to call the get (for one-to-many relationships) or first (for one-to-one relationships) method:

```
$phone = User::find(1)->phone;
```

What if you need to retrieve a phone's user? Since the foreign key (**user_id**) is on the phones table, we should describe this relationship using the **belongs_to** method. It makes sense, right? Phones belong to users. When using the **belongs_to** method, the name of the relationship method should correspond to the foreign key (sans the **_id**). Since the foreign key is **user_id**, your relationship method should be named **user**:

```
class Phone extends Eloquent {

     public function user()
     {
          return $this->belongs_to('User');
     }

}
```

Great! You can now access a User model through a Phone model using either your relationship method or dynamic property:

```
echo Phone::find(1)->user()->first()->email;

echo Phone::find(1)->user->email;
```

One-To-Many

Assume a blog post has many comments. It's easy to define this relationship using the **has_many** method:

```
class Post extends Eloquent {

     public function comments()
     {
          return $this->has_many('Comment');
     }

}
```

Now, simply access the post comments through the relationship method or dynamic property:

```
$comments = Post::find(1)->comments()->get();

$comments = Post::find(1)->comments;
```

```
Both of these statements will execute the following SQL:
SELECT * FROM "posts" WHERE "id" = 1

SELECT * FROM "comments" WHERE "post_id" = 1
```

Want to join on a different foreign key? No problem. Just pass it in the second parameter to the method:

```
return $this->has_many('Comment', 'my_foreign_key');
```

You may be wondering: *If the dynamic properties return the relationship and require less keystrokes, why would I ever use the relationship methods?* Actually, relationship methods are very powerful. They allow you to continue to chain query methods before retrieving the relationship. Check this out:

```
echo Post::find(1)->comments()->order_by('votes', 'desc')->take(10)->get();
```

Many-To-Many

Many-to-many relationships are the most complicated of the three relationships. But don't worry, you can do this. For example, assume a User has many Roles, but a Role can also belong to many Users. Three database tables must be created to accomplish this relationship: a **users** table, a **roles** table, and a **role_user** table. The structure for each table looks like this:

Users:

```
id    - INTEGER
email - VARCHAR
```

Roles:

```
id   - INTEGER
name - VARCHAR
```

Role_User:

```
id      - INTEGER
user_id - INTEGER
role_id - INTEGER
```

Now you're ready to define the relationship on your models using the **has_many_and_belongs_to** method:

```
class User extends Eloquent {

     public function roles()
```

```
	{
		return $this->has_many_and_belongs_to('Role');
	}

}
```

Great! Now it's time to retrieve a user's roles:

```
$roles = User::find(1)->roles()->get();
```

Or, as usual, you may retrieve the relationship through the dynamic roles property:

```
$roles = User::find(1)->roles;
```

As you may have noticed, the default name of the intermediate table is the singular names of the two related models arranged alphabetically and concatenated by an underscore. However, you are free to specify your own table name. Simply pass the table name in the second parameter to the **has_and_belongs_to_many** method:

```
class User extends Eloquent {

	public function roles()
	{
		return $this->has_many_and_belongs_to('Role', 'user_roles');
	}

}
```

By default only certain fields from the pivot table will be returned (the two **id** fields, and the timestamps). If your pivot table contains additional columns, you can fetch them too by using the **with()** method :

```
class User extends Eloquent {

	public function roles()
	{
		return $this->has_many_and_belongs_to('Role', 'user_roles')-
>with('column');
	}

}
```

Inserting Related Models

Let's assume you have a **Post** model that has many comments. Often you may want to insert a new comment for a given post. Instead of manually setting the **post_id** foreign key on your model, you may insert the new comment from it's owning Post model. Here's what it looks like:

```
$comment = new Comment(array('message' => 'A new comment.'));

$post = Post::find(1);

$comment = $post->comments()->insert($comment);
```

When inserting related models through their parent model, the foreign key will automatically be set. So, in this case, the "post_id" was automatically set to "1" on the newly inserted comment.

When working with `has_many` relationships, you may use the `save` method to insert / update related models:

```
$comments = array(
    array('message' => 'A new comment.'),
    array('message' => 'A second comment.'),
);

$post = Post::find(1);

$post->comments()->save($comments);
```

Inserting Related Models (Many-To-Many)

This is even more helpful when working with many-to-many relationships. For example, consider a **User** model that has many roles. Likewise, the **Role** model may have many users. So, the intermediate table for this relationship has "user_id" and "role_id" columns. Now, let's insert a new Role for a User:

```
$role = new Role(array('title' => 'Admin'));

$user = User::find(1);

$role = $user->roles()->insert($role);
```

Now, when the Role is inserted, not only is the Role inserted into the "roles" table, but a record in the intermediate table is also inserted for you. It couldn't be easier!

However, you may often only want to insert a new record into the intermediate table. For example, perhaps the role you wish to attach to the user already exists. Just use the attach method:

```
$user->roles()->attach($role_id);
```

It's also possible to attach data for fields in the intermediate table (pivot table), to do this add a second array variable to the attach command containing the data you want to attach:

```
$user->roles()->attach($role_id, array('expires' => $expires));
```

Alternatively, you can use the `sync` method, which accepts an array of IDs to "sync" with the intermediate table. After this operation is complete, only the IDs in the array will be on the intermediate table.

```
$user->roles()->sync(array(1, 2, 3));
```

Working With Intermediate Tables

As your probably know, many-to-many relationships require the presence of an intermediate table. Eloquent makes it a breeze to maintain this table. For example, let's assume we have a **User** model that has many roles. And, likewise, a **Role** model that has many users. So the intermediate table has "user_id" and "role_id" columns. We can access the pivot table for the relationship like so:

```
$user = User::find(1);

$pivot = $user->roles()->pivot();
```

Once we have an instance of the pivot table, we can use it just like any other Eloquent model:

```
foreach ($user->roles()->pivot()->get() as $row)
{
    //
}
```

You may also access the specific intermediate table row associated with a given record. For example:

```
$user = User::find(1);

foreach ($user->roles as $role)
{
    echo $role->pivot->created_at;
}
```

Notice that each related **Role** model we retrieved is automatically assigned a **pivot** attribute. This attribute contains a model representing the intermediate table record associated with that related model.

Sometimes you may wish to remove all of the record from the intermediate table for a given model relationship. For instance, perhaps you want to remove all of the assigned roles from a user. Here's how to do it:

```
$user = User::find(1);

$user->roles()->delete();
```

Note that this does not delete the roles from the "roles" table, but only removes the records from the intermediate table which associated the roles with the given user.

Eager Loading

Eager loading exists to alleviate the N + 1 query problem. Exactly what is this problem? Well, pretend each Book belongs to an Author. We would describe this relationship like so:

```
class Book extends Eloquent {

     public function author()
     {
          return $this->belongs_to('Author');
     }

}
```

Now, examine the following code:

```
foreach (Book::all() as $book)
{
     echo $book->author->name;
}
```

How many queries will be executed? Well, one query will be executed to retrieve all of the books from the table. However, another query will be required for each book to retrieve the author. To display the author name for 25 books would require **26 queries**. See how the queries can add up fast?

Thankfully, you can eager load the author models using the **with** method. Simply mention the **function name** of the relationship you wish to eager load:

```
foreach (Book::with('author')->get() as $book)
{
     echo $book->author->name;
}
```

In this example, **only two queries will be executed**!

```
SELECT * FROM "books"

SELECT * FROM "authors" WHERE "id" IN (1, 2, 3, 4, 5, ...)
```

Obviously, wise use of eager loading can dramatically increase the performance of your application. In the example above, eager loading cut the execution time in half.

Need to eager load more than one relationship? It's easy:

```
$books = Book::with(array('author', 'publisher'))->get();
```

> **Note:** When eager loading, the call to the static **with** method must always be at the beginning of the query.

You may even eager load nested relationships. For example, let's assume our **Author** model has a "contacts" relationship. We can eager load both of the relationships from our Book model like so:

```
$books = Book::with(array('author', 'author.contacts'))->get();
```

If you find yourself eager loading the same models often, you may want to use **$includes** in the model.

```
class Book extends Eloquent {

     public $includes = array('author');

     public function author()
     {
          return $this->belongs_to('Author');
     }

}
```

$includes takes the same arguments that **with** takes. The following is now eagerly loaded.

```
foreach (Book::all() as $book)
{
     echo $book->author->name;
}
```

> **Note:** Using **with** will override a models **$includes**.

Constraining Eager Loads

Sometimes you may wish to eager load a relationship, but also specify a condition for the eager load. It's simple. Here's what it looks like:

```
$users = User::with(array('posts' => function($query)
{
    $query->where('title', 'like', '%first%');

}))->get();
```

In this example, we're eager loading the posts for the users, but only if the post's "title" column contains the word "first".

Getter & Setter Methods

Setters allow you to handle attribute assignment with custom methods. Define a setter by appending "set_" to the intended attribute's name.

```
public function set_password($password)
{
    $this->set_attribute('hashed_password', Hash::make($password));
}
```

Call a setter method as a variable (without parenthesis) using the name of the method without the "set_" prefix.

```
$this->password = "my new password";
```

Getters are very similar. They can be used to modify attributes before they're returned. Define a getter by appending "get_" to the intended attribute's name.

```
public function get_published_date()
{
    return date('M j, Y', $this->get_attribute('published_at'));
}
```

Call the getter method as a variable (without parenthesis) using the name of the method without the "get_" prefix.

```
echo $this->published_date;
```

Mass-Assignment

Mass-assignment is the practice of passing an associative array to a model method which then fills the model's attributes with the values from the array. Mass-assignment can be done by passing an array to the model's constructor:

```
$user = new User(array(
    'username' => 'first last',
    'password' => 'disgaea'
));

$user->save();
```

Or, mass-assignment may be accomplished using the **fill** method.

```
$user = new User;

$user->fill(array(
    'username' => 'first last',
    'password' => 'disgaea'
));

$user->save();
```

By default, all attribute key/value pairs will be store during mass-assignment. However, it is possible to create a white-list of attributes that will be set. If the accessible attribute white-list is set then no attributes other than those specified will be set during mass-assignment.

You can specify accessible attributes by assigning the **$accessible** static array. Each element contains the name of a white-listed attribute.

```
public static $accessible = array('email', 'password', 'name');
```

Alternatively, you may use the **accessible** method from your model:

```
User::accessible(array('email', 'password', 'name'));
```

Note: Utmost caution should be taken when mass-assigning using user-input. Technical oversights could cause serious security vulnerabilities.

Converting Models To Arrays

When building JSON APIs, you will often need to convert your models to array so they can be easily serialized. It's really simple.

Convert a model to an array:

```
return json_encode($user->to_array());
```

The `to_array` method will automatically grab all of the attributes on your model, as well as any loaded relationships.

Sometimes you may wish to limit the attributes that are included in your model's array, such as passwords. To do this, add a `hidden` attribute definition to your model:

Excluding attributes from the array:

```
class User extends Eloquent {

    public static $hidden = array('password');

}
```

Deleting Models

Because Eloquent inherits all the features and methods of Fluent queries, deleting models is a snap:

```
$author->delete();
```

Note, however, than this won't delete any related models (e.g. all the author's Book models will still exist), unless you have set up foreign keys and cascading deletes.

SCHEMA BUILDER

The Basics

The Schema Builder provides methods for creating and modifying your database tables. Using a fluent syntax, you can work with your tables without using any vendor specific SQL.

Further Reading:

- Migrations

Creating & Dropping Tables

The **Schema** class is used to create and modify tables. Let's jump right into an example:

Creating a simple database table:

```
Schema::create('users', function($table)
{
    $table->increments('id');
});
```

Let's go over this example. The **create** method tells the Schema builder that this is a new table, so it should be created. In the second argument, we passed a Closure which receives a Table instance. Using this Table object, we can fluently add and drop columns and indexes on the table.

Dropping a table from the database:

```
Schema::drop('users');
```

Dropping a table from a given database connection:

```
Schema::drop('users', 'connection_name');
```

Sometimes you may need to specify the database connection on which the schema operation should be performed.

Specifying the connection to run the operation on:

```
Schema::create('users', function($table)
{
    $table->on('connection');
});
```

Adding Columns

The fluent table builder's methods allow you to add columns without using vendor specific SQL. Let's go over it's methods:

Command	**Description**
`$table->increments('id');`	Incrementing ID to the table
`$table->string('email');`	VARCHAR equivalent column
`$table->string('name', 100);`	VARCHAR equivalent with a length
`$table->integer('votes');`	INTEGER equivalent to the table
`$table->float('amount');`	FLOAT equivalent to the table

Command	**Description**
`$table->decimal('amount', 5, 2);`	DECIMAL equivalent with a precision and scale
`$table->boolean('confirmed');`	BOOLEAN equivalent to the table
`$table->date('created_at');`	DATE equivalent to the table
`$table->timestamp('added_on');`	TIMESTAMP equivalent to the table
`$table->timestamps();`	Adds **created_at** and **updated_at** columns
`$table->text('description');`	TEXT equivalent to the table
`$table->blob('data');`	BLOB equivalent to the table
`->nullable()`	Designate that the column allows NULL values
`->default($value)`	Declare a default value for a column
`->unsigned()`	Set INTEGER to UNSIGNED

Note: Laravel's "boolean" type maps to a small integer column on all database systems.

Example of creating a table and adding columns

```
Schema::table('users', function($table)
{
    $table->create();
    $table->increments('id');
    $table->string('username');
    $table->string('email');
    $table->string('phone')->nullable();
    $table->text('about');
    $table->timestamps();
});
```

Dropping Columns

Dropping a column from a database table:

```
$table->drop_column('name');
```

Dropping several columns from a database table:

```
$table->drop_column(array('name', 'email'));
```

Adding Indexes

The Schema builder supports several types of indexes. There are two ways to add the indexes. Each type of index has its method; however, you can also fluently define an index on the same line as a column addition. Let's take a look:

Fluently creating a string column with an index:

```
$table->string('email')->unique();
```

If defining the indexes on a separate line is more your style, here are example of using each of the index methods:

Command	**Description**
`$table->primary('id');`	Adding a primary key
`$table->primary(array('fname', 'lname'));`	Adding composite keys
`$table->unique('email');`	Adding a unique index
`$table->fulltext('description');`	Adding a full-text index
`$table->index('state');`	Adding a basic index

Dropping Indexes

To drop indexes you must specify the index's name. Laravel assigns a reasonable name to all indexes. Simply concatenate the table name and the names of the columns in the index, then append the type of the index. Let's take a look at some examples:

Command	**Description**
`$table->drop_primary('users_id_primary');`	Dropping a primary key from the "users" table
`$table->drop_unique('users_email_unique');`	Dropping a unique index from the

Command	Description
	"users" table
`$table->drop_fulltext('profile_description_fulltext');`	Dropping a full-text index from the "profile" table
`$table->drop_index('geo_state_index');`	Dropping a basic index from the "geo" table

Foreign Keys

You may easily add foreign key constraints to your table using Schema's fluent interface. For example, let's assume you have a **user_id** on a **posts** table, which references the **id** column of the **users** table. Here's how to add a foreign key constraint for the column:

```
$table->foreign('user_id')->references('id')->on('users');
```

You may also specify options for the "on delete" and "on update" actions of the foreign key:

```
$table->foreign('user_id')->references('id')->on('users')->on_delete('restrict');

$table->foreign('user_id')->references('id')->on('users')->on_update('cascade');
```

You may also easily drop a foreign key constraint. The default foreign key names follow the same convention as the other indexes created by the Schema builder. Here's an example:

```
$table->drop_foreign('posts_user_id_foreign');
```

Note: The field referenced in the foreign key is very likely an auto increment and therefore automatically an unsigned integer. Please make sure to create the foreign key field with **unsigned()** as both fields have to be the exact same type, the engine on both tables has to be set to **InnoDB**, and the referenced table must be created **before** the table with the foreign key.

```
$table->engine = 'InnoDB';

$table->integer('user_id')->unsigned();
```

MIGRATIONS

The Basics

Think of migrations as a type of version control for your database. Let's say your working on a team, and you all have local databases for development. Good ole' Eric makes a change to the database and checks in his code that uses the new column. You pull in the code, and your application breaks because you don't have the new column. What do you do? Migrations are the answer. Let's dig in deeper to find out how to use them!

Prepping Your Database

Before you can run migrations, we need to do some work on your database. Laravel uses a special table to keep track of which migrations have already run. To create this table, just use the Artisan command-line:

Creating the Laravel migrations table:

```
php artisan migrate:install
```

Creating Migrations

You can easily create migrations through Laravel's "Artisan" CLI. It looks like this:

Creating a migration

```
php artisan migrate:make create_users_table
```

Now, check your **application/migrations** folder. You should see your brand new migration! Notice that it also contains a timestamp. This allows Laravel to run your migrations in the correct order.

You may also create migrations for a bundle.

Creating a migration for a bundle:

```
php artisan migrate:make bundle::create_users_table
```

Further Reading:

- Schema Builder

Running Migrations

Running all outstanding migrations in application and bundles:

```
php artisan migrate
```

Running all outstanding migrations in the application:

```
php artisan migrate application
```

Running all outstanding migrations in a bundle:

```
php artisan migrate bundle
```

Rolling Back

When you roll back a migration, Laravel rolls back the entire migration "operation". So, if the last migration command ran 122 migrations, all 122 migrations would be rolled back.

Rolling back the last migration operation:

```
php artisan migrate:rollback
```

Roll back all migrations that have ever run:

```
php artisan migrate:reset
```

REDIS

The Basics

Redis is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and sorted sets.

Configuration

The Redis configuration for your application lives in the **application/config/database.php** file. Within this file, you will see a **redis** array containing the Redis servers used by your application:

```
'redis' => array(

    'default' => array('host' => '127.0.0.1', 'port' => 6379),

),
```

The default server configuration should suffice for development. However, you are free to modify this array based on your environment. Simply give each Redis server a name, and specify the host and port used by the server.

Usage

You may get a Redis instance by calling the **db** method on the **Redis** class:

```
$redis = Redis::db();
```

This will give you an instance of the **default** Redis server. You may pass the server name to the **db** method to get a specific server as defined in your Redis configuration:

```
$redis = Redis::db('redis_2');
```

Great! Now that we have an instance of the Redis client, we may issue any of the Redis commands to the instance. Laravel uses magic methods to pass the commands to the Redis server:

```
$redis->set('name', 'Taylor');

$name = $redis->get('name');

$values = $redis->lrange('names', 5, 10);
```

Notice the arguments to the comment are simply passed into the magic method. Of course, you are not required to use the magic methods, you may also pass commands to the server using the **run** method:

```
$values = $redis->run('lrange', array(5, 10));
```

Just want to execute commands on the default Redis server? You can just use static magic methods on the Redis class:

```
Redis::set('name', 'Taylor');

$name = Redis::get('name');

$values = Redis::lrange('names', 5, 10);
```

Note: Redis cache and session drivers are included with Laravel.

CACHING

CACHE CONFIGURATION

The Basics

Imagine your application displays the ten most popular songs as voted on by your users. Do you really need to look up these ten songs every time someone visits your site? What if you could store them for 10 minutes, or even an hour, allowing you to dramatically speed up your application? Laravel's caching makes it simple.

Laravel provides five cache drivers out of the box:

- File System
- Database
- Memcached
- APC
- Redis
- Memory (Arrays)

By default, Laravel is configured to use the **file** system cache driver. It's ready to go out of the box with no configuration. The file system driver stores cached items as files in the **cache** directory. If you're satisfied with this driver, no other configuration is required. You're ready to start using it.

> **Note:** Before using the file system cache driver, make sure your **storage/cache** directory is writeable.

Database

The database cache driver uses a given database table as a simple key-value store. To get started, first set the name of the database table in **application/config/cache.php**:

```
'database' => array('table' => 'laravel_cache'),
```

Next, create the table on your database. The table should have three columns:

- key (varchar)
- value (text)
- expiration (integer)

That's it. Once your configuration and table is setup, you're ready to start caching!

Memcached

Memcached is an ultra-fast, open-source distributed memory object caching system used by sites such as Wikipedia and Facebook. Before using Laravel's Memcached driver, you will need to install and configure Memcached and the PHP Memcache extension on your server.

Once Memcached is installed on your server you must set the **driver** in the **application/config/cache.php** file:

```
'driver' => 'memcached'
```

Then, add your Memcached servers to the **servers** array:

```
'servers' => array(
     array('host' => '127.0.0.1', 'port' => 11211, 'weight' => 100),
)
```

Redis

Redis is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and sorted sets.

Before using the Redis cache driver, you must configure your Redis servers. Now you can just set the **driver** in the**application/config/cache.php** file:

```
'driver' => 'redis'
```

Cache Keys

To avoid naming collisions with other applications using APC, Redis, or a Memcached server, Laravel prepends a**key** to each item stored in the cache using these drivers. Feel free to change this value:

```
'key' => 'laravel'
```

In-Memory Cache

The "memory" cache driver does not actually cache anything to disk. It simply maintains an internal array of the cache data for the current request. This makes it perfect for unit testing your application in isolation from any storage mechanism. It should never be used as a "real" cache driver.

CACHE USAGE

Storing Items

Storing items in the cache is simple. Simply call the **put** method on the Cache class:

```
Cache::put('name', 'Taylor', 10);
```

The first parameter is the **key** to the cache item. You will use this key to retrieve the item from the cache. The second parameter is the **value** of the item. The third parameter is the number of **minutes** you want the item to be cached.

You may also cache something "forever" if you do not want the cache to expire:

```
Cache::forever('name', 'Taylor');
```

> **Note:** It is not necessary to serialize objects when storing them in the cache.

Retrieving Items

Retrieving items from the cache is even more simple than storing them. It is done using the **get** method. Just mention the key of the item you wish to retrieve:

```
$name = Cache::get('name');
```

By default, NULL will be returned if the cached item has expired or does not exist. However, you may pass a different default value as a second parameter to the method:

```
$name = Cache::get('name', 'Fred');
```

Now, "Fred" will be returned if the "name" cache item has expired or does not exist.

What if you need a value from your database if a cache item doesn't exist? The solution is simple. You can pass a closure into the **get** method as a default value. The closure will only be executed if the cached item doesn't exist:

```
$users = Cache::get('count', function() {return DB::table('users')->count();});
```

Let's take this example a step further. Imagine you want to retrieve the number of registered users for your application; however, if the value is not cached, you want to store the default value in the cache using the**remember** method:

```
$users = Cache::remember('count', function() {return DB::table('users')->count();},
5);
```

Let's talk through that example. If the **count** item exists in the cache, it will be returned. If it doesn't exist, the result of the closure will be stored in the cache for five minutes **and** be returned by the method. Slick, huh?

Laravel even gives you a simple way to determine if a cached item exists using the **has** method:

```
if (Cache::has('name'))
{
     $name = Cache::get('name');
}
```

Removing Items

Need to get rid of a cached item? No problem. Just mention the name of the item to the **forget** method:

```
Cache::forget('name');
```

AUTHENTICATION

AUTH CONFIGURATION

The Basics

Most interactive applications have the ability for users to login and logout. Laravel provides a simple class to help you validate user credentials and retrieve information about the current user of your application.

To get started, let's look over the **application/config/auth.php** file. The authentication configuration contains some basic options to help you get started with authentication.

The Authentication Driver

Laravel's authentication is driver based, meaning the responsibility for retrieving users during authentication is delegated to various "drivers". Two are included out of the box: Eloquent and Fluent, but you are free to write your own drivers if needed!

The **Eloquent** driver uses the Eloquent ORM to load the users of your application, and is the default authentication driver. The **Fluent** driver uses the fluent query builder to load your users.

The Default "Username"

The second option in the configuration file determines the default "username" of your users. This will typically correspond to a database column in your "users" table, and will usually be "email" or "username".

Authentication Model

When using the **Eloquent** authentication driver, this option determines the Eloquent model that should be used when loading users.

Authentication Table

When using the **Fluent** authentication drivers, this option determines the database table containing the users of your application.

AUTHENTICATION USAGE

> **Note:** Before using the Auth class, you must specify a session driver.

Salting & Hashing

If you are using the Auth class, you are strongly encouraged to hash and salt all passwords. Web development must be done responsibly. Salted, hashed passwords make a rainbow table attack against your user's passwords impractical.

Salting and hashing passwords is done using the **Hash** class. The Hash class is uses the **bcrypt** hashing algorithm. Check out this example:

```
$password = Hash::make('secret');
```

The **make** method of the Hash class will return a 60 character hashed string.

You can compare an unhashed value against a hashed one using the **check** method on the **Hash** class:

```
if (Hash::check('secret', $hashed_value))
{
    return 'The password is valid!';
}
```

Logging In

Logging a user into your application is simple using the **attempt** method on the Auth class. Simply pass the username and password of the user to the method. The credentials should be contained in an array, which allows for maximum flexibility across drivers, as some drivers may require a different number of arguments. The login method will return **true** if the credentials are valid. Otherwise, **false** will be returned:

```
$credentials = array('username' => 'example@gmail.com', 'password' => 'secret');

if (Auth::attempt($credentials))
{
     return Redirect::to('user/profile');
}
```

If the user's credentials are valid, the user ID will be stored in the session and the user will be considered "logged in" on subsequent requests to your application.

To determine if the user of your application is logged in, call the **check** method:

```
if (Auth::check())
{
     return "You're logged in!";
}
```

Use the **login** method to login a user without checking their credentials, such as after a user first registers to use your application. Just pass the user's ID:

```
Auth::login($user->id);

Auth::login(15);
```

Protecting Routes

It is common to limit access to certain routes only to logged in users. In Laravel this is accomplished using the auth filter. If the user is logged in, the request will proceed as normal; however, if the user is not logged in, they will be redirected to the "login" named route.

To protect a route, simply attach the **auth** filter:

```
Route::get('admin', array('before' => 'auth', function() {}));
```

Note: You are free to edit the **auth** filter however you like. A default implementation is located in**application/routes.php**.

Retrieving The Logged In User

Once a user has logged in to your application, you can access the user model via the **user** method on the Auth class:

```
return Auth::user()->email;
```

Note: If the user is not logged in, the **user** method will return NULL.

Logging Out

Ready to log the user out of your application?

```
Auth::logout();
```

This method will remove the user ID from the session, and the user will no longer be considered logged in on subsequent requests to your application.

ARTISAN CLI

TASKS

The Basics

Laravel's command-line tool is called Artisan. Artisan can be used to run "tasks" such as migrations, cronjobs, unit-tests, or anything that want.

Creating & Running Tasks

To create a task create a new class in your **application/tasks** directory. The class name should be suffixed with "_Task", and should at least have a "run" method, like this:

Creating a task class:

```
class Notify_Task {

    public function run($arguments)
    {
        // Do awesome notifying...
    }

}
```

Now you can call the "run" method of your task via the command-line. You can even pass arguments:

Calling a task from the command line:

```
php artisan notify
```

Calling a task and passing arguments:

```
php artisan notify taylor
```

Calling a task from your application:

```
Command::run(array('notify'));
```

Calling a task from your application with arguements:

```
Command::run(array('notify', 'taylor'));
```

Remember, you can call specific methods on your task, so, let's add an "urgent" method to the notify task:

Adding a method to the task:

```
class Notify_Task {

    public function run($arguments)
    {
        // Do awesome notifying...
    }

    public function urgent($arguments)
    {
        // This is urgent!
    }

}
```

Now we can call our "urgent" method:

Calling a specific method on a task:

```
php artisan notify:urgent
```

Bundle Tasks

To create a task for your bundle just prefix the bundle name to the class name of your task. So, if your bundle was named "admin", a task might look like this:

Creating a task class that belongs to a bundle:

```
class Admin_Generate_Task {

    public function run($arguments)
    {
        // Generate the admin!
    }

}
```

To run your task just use the usual Laravel double-colon syntax to indicate the bundle:

Running a task belonging to a bundle:

```
php artisan admin::generate
```

Running a specific method on a task belonging to a bundle:

```
php artisan admin::generate:list
```

CLI Options

Setting the Laravel environment:

```
php artisan foo --env=local
```

Setting the default database connection:

```
php artisan foo --database=sqlite
```

ARTISAN COMMANDS

Help

Description	**Command**
View a list of available artisan commands.	`php artisan help:commands`

Application Configuration (More Information)

Description	**Command**

Description	Command
Generate a secure application key. An application key will not be generated unless the field in **config/application.php** is empty.	`php artisan key:generate`

Database Sessions (More Information)

Description	Command
Create a session table	`php artisan session:table`

Migrations (More Information)

Description	Command
Create the Laravel migration table	`php artisan migrate:install`
Creating a migration	`php artisan migrate:make create_users_table`
Creating a migration for a bundle	`php artisan migrate:make bundle::tablename`
Running outstanding migrations	`php artisan migrate`
Running outstanding migrations in the application	`php artisan migrate application`
Running all outstanding migrations in a bundle	`php artisan migrate bundle`
Rolling back the last migration operation	`php artisan migrate:rollback`
Roll back all migrations that have ever run	`php artisan migrate:reset`

Bundles (More Information)

Description	Command
Install a bundle	`php artisan bundle:install eloquent`
Upgrade a bundle	`php artisan bundle:upgrade eloquent`
Upgrade all bundles	`php artisan bundle:upgrade`
Publish a bundle assets	`php artisan bundle:publish bundle_name`
Publish all bundles assets	`php artisan bundle:publish`

Note: After installing you need to register the bundle

Tasks (More Information)

Description	Command
Calling a task	`php artisan notify`
Calling a task and passing arguments	`php artisan notify taylor`
Calling a specific method on a task	`php artisan notify:urgent`
Running a task on a bundle	`php artisan admin::generate`
Running a specific method on a bundle	`php artisan admin::generate:list`

Unit Tests (More Information)

Description	Command
Running the application tests	`php artisan test`
Running the bundle tests	`php artisan test bundle-name`

Routing (More Information)

Description	Command
Calling a route	`php artisan route:call get api/user/1`

Note: You can replace get with post, put, delete, etc.

Application Keys

Description	Command
Generate an application key	`php artisan key:generate`

Note: You can specify an alternate key length by adding an extra argument to the command.

CLI Options

Description	Command
Setting the Laravel environment	`php artisan foo --env=local`
Setting the default database connection	`php artisan foo --database=sqlitename`

LARAVEL ON GITHUB

The Basics

Because Laravel's development and source control is done through GitHub, anyone is able to make contributions to it. Anyone can fix bugs, add features or improve the documentation.

After submitting proposed changes to the project, the Laravel team will review the changes and make the decision to commit them to Laravel's core.

Repositories

Laravel's home on GitHub is at github.com/laravel. Laravel has several repositories. For basic contributions, the only repository you need to pay attention to is the **laravel** repository, located at github.com/laravel/laravel.

Branches

The **laravel** repository has multiple branches, each serving a specific purpose:

- **master** - This is the Laravel release branch. Active development does not happen on this branch. This branch is only for the most recent, stable Laravel core code. When you download Laravel from laravel.com, you are downloading directly from this master branch. *Do not make pull requests to this branch.*

- **develop** - This is the working development branch. All proposed code changes and contributions by the community are pulled into this branch. *When you make a pull request to the Laravel project, this is the branch you want to pull-request into.*

Once certain milestones have been reached and/or Taylor Otwell and the Laravel team is happy with the stability and additional features of the current development branch, the changes in the **develop** branch are pulled into the **master** branch, thus creating and releasing the newest stable version of Laravel for the world to use.

Pull Requests

GitHub pull requests are a great way for everyone in the community to contribute to the Laravel codebase. Found a bug? Just fix it in your fork and submit a pull request. This will then be reviewed, and, if found as good, merged into the main repository.

In order to keep the codebase clean, stable and at high quality, even with so many people contributing, some guidelines are necessary for high-quality pull requests:

- **Branch:** Unless they are immediate documentation fixes relevant for old versions, pull requests should be sent to the `develop` branch only. Make sure to select that branch as target when creating the pull request (GitHub will not automatically select it.)
- **Documentation:** If you are adding a new feature or changing the API in any relevant way, this should be documented. The documentation files can be found directly in the core repository.
- **Unit tests:** To keep old bugs from re-appearing and generally hold quality at a high level, the Laravel core is thoroughly unit-tested. Thus, when you create a pull request, it is expected that you unit test any new code you add. For any bug you fix, you should also add regression tests to make sure the bug will never appear again. If you are unsure about how to write tests, the core team or other contributors will gladly help.

Further Reading

- Contributing to Laravel via Command-Line
- Contributing to Laravel using TortoiseGit

CONTRIBUTING TO LARAVEL VIA COMMAND-LINE

Getting Started

This tutorial explains the basics of contributing to a project on GitHub via the command-line. The workflow can apply to most projects on GitHub, but in this case, we will be focused on the Laravel project. This tutorial is applicable to OSX, Linux and Windows.

This tutorial assumes you have installed Git and you have created a GitHub account. If you haven't already, look at the Laravel on GitHub documentation in order to familiarize yourself with Laravel's repositories and branches.

Forking Laravel

Login to GitHub and visit the Laravel Repository. Click on the **Fork** button. This will create your own fork of Laravel in your own GitHub account. Your Laravel fork will be located at **https://github.com/username/laravel**(your GitHub username will be used in place of *username*).

Cloning Laravel

Open up the command-line or terminal and make a new directory where you can make development changes to Laravel:

```
# mkdir laravel-develop
# cd laravel-develop
```

Next, clone the Laravel repository (not your fork you made):

```
# git clone https://github.com/laravel/laravel.git .
```

Note: The reason you are cloning the original Laravel repository (and not the fork you made) is so you can always pull down the most recent changes from the Laravel repository to your local repository.

Adding your Fork

Next, it's time to add the fork you made as a **remote repository**:

```
# git remote add fork git@github.com:username/laravel.git
```

Remember to replace *username** with your GitHub username. *This is case-sensitive*. You can verify that your fork was added by typing:

```
# git remote
```

Now you have a pristine clone of the Laravel repository along with your fork as a remote repository. You are ready to begin branching for new features or fixing bugs.

Creating Branches

First, make sure you are working in the **develop** branch. If you submit changes to the **master** branch, it is unlikely they will be pulled in anytime in the near future. For more information on this, read the documentation for Laravel on GitHub. To switch to the develop branch:

```
# git checkout develop
```

Next, you want to make sure you are up-to-date with the latest Laravel repository. If any new features or bug fixes have been added to the Laravel project since you cloned it, this will ensure that your local repository has all of those changes. This important step is the reason we originally cloned the Laravel repository instead of your own fork.

```
# git pull origin develop
```

Now you are ready to create a new branch for your new feature or bug-fix. When you create a new branch, use a self-descriptive naming convention. For example, if you are going to fix a bug in Eloquent, name your branch*bug/eloquent*:

```
# git branch bug/eloquent
# git checkout bug/eloquent
Switched to branch 'bug/eloquent'
```

Or if there is a new feature to add or change to the documentation that you want to make, for example, the localization documentation:

```
# git branch feature/localization-docs
# git checkout feature/localization-docs
Switched to branch 'feature/localization-docs'
```

Note: Create one new branch for every new feature or bug-fix. This will encourage organization, limit interdependency between new features/fixes and will make it easy for the Laravel team to merge your changes into the Laravel core.

Now that you have created your own branch and have switched to it, it's time to make your changes to the code. Add your new feature or fix that bug.

Committing

Now that you have finished coding and testing your changes, it's time to commit them to your local repository. First, add the files that you changed/added:

```
# git add laravel/documentation/localization.md
```

Next, commit the changes to the repository:

```
# git commit -s -m "I added some more stuff to the Localization documentation."
```

"- **-s** means that you are signing-off on your commit with your name. This tells the Laravel team know that you personally agree to your code being added to the Laravel core.
"- **-m** is the message that goes with your commit. Provide a brief explanation of what you added or changed.

Pushing to your Fork

Now that your local repository has your committed changes, it's time to push (or sync) your new branch to your fork that is hosted in GitHub:

```
# git push fork feature/localization-docs
```

Your branch has been successfully pushed to your fork on GitHub.

Submitting a Pull Request

The final step is to submit a pull request to the Laravel repository. This means that you are requesting that the Laravel team pull and merge your changes to the Laravel core. In your browser, visit your Laravel fork athttps://github.com/username/laravel. Click on **Pull Request**. Next, make sure you choose the proper base and head repositories and branches:

- **base repo:** laravel/laravel
- **base branch:** develop
- **head repo:** username/laravel
- **head branch:** feature/localization-docs

Use the form to write a more detailed description of the changes you made and why you made them. Finally, click **Send pull request**. That's it! The changes you made have been submitted to the Laravel team.

What's Next?

Do you have another feature you want to add or another bug you need to fix? First, make sure you always base your new branch off of the develop branch:

```
# git checkout develop
```

Then, pull down the latest changes from Laravel's repository:

```
# git pull origin develop
```

Now you are ready to create a new branch and start coding again!

> Jason Lewis's blog post Contributing to a GitHub Project was the primary inspiration for this tutorial.

CONTRIBUTING TO LARAVEL USING TORTOISEGIT

Getting Started

This tutorial explains the basics of contributing to a project on GitHub using TortoiseGit for Windows. The workflow can apply to most projects on GitHub, but in this case, we will be focused on the Laravel project.

This tutorial assumes you have installed TortoiseGit for Windows and you have created a GitHub account. If you haven't already, look at the Laravel on GitHub documentation in order to familiarize yourself with Laravel's repositories and branches.

Forking Laravel

Login to GitHub and visit the Laravel Repository. Click on the **Fork** button. This will create your own fork of Laravel in your own GitHub account. Your Laravel fork will be located at **https://github.com/username/laravel**(your GitHub username will be used in place of *username*).

Cloning Laravel

Open up Windows Explorer and create a new directory where you can make development changes to Laravel.

- Right-click the Laravel directory to bring up the context menu. Click on **Git Clone...**
- Git clone
    - **Url:** https://github.com/laravel/laravel.git
    - **Directory:** the directory that you just created in the previous step
    - Click **OK**

> **Note**: The reason you are cloning the original Laravel repository (and not the fork you made) is so you can always pull down the most recent changes from the Laravel repository to your local repository.

Adding your Fork

After the cloning process is complete, it's time to add the fork you made as a **remote repository**.

- Right-click the Laravel directory and goto **TortoiseGit > Settings**
- Goto the **Git/Remote** section. Add a new remote:
    - **Remote**: fork
    - **URL**: https://github.com/username/laravel.git
    - Click **Add New/Save**
    - Click **OK**

Remember to replace *username* with your GitHub username. *This is case-sensitive*.

Creating Branches

Now you are ready to create a new branch for your new feature or bug-fix. When you create a new branch, use a self-descriptive naming convention. For example, if you are going to fix a bug in Eloquent, name your branch*bug/eloquent*. Or if you were going to make changes to the localization documentation, name your branch*feature/localization-docs*. A good naming convention will encourage organization and help others understand the purpose of your branch.

- Right-click the Laravel directory and goto **TortoiseGit > Create Branch**
    - **Branch:** feature/localization-docs
    - **Base On Branch:** remotes/origin/develop
    - **Check** *Track*
    - **Check** *Switch to new branch*
    - Click **OK**

This will create your new *feature/localization-docs* branch and switch you to it.

> **Note:** Create one new branch for every new feature or bug-fix. This will encourage organization, limit interdependency between new features/fixes and will make it easy for the Laravel team to merge your changes into the Laravel core.

Now that you have created your own branch and have switched to it, it's time to make your changes to the code. Add your new feature or fix that bug.

Committing

Now that you have finished coding and testing your changes, it's time to commit them to your local repository:

- Right-click the Laravel directory and goto **Git Commit -> "feature/localization-docs"...**
- Commit
  - **Message:** Provide a brief explaination of what you added or changed
  - Click **Sign** - This tells the Laravel team know that you personally agree to your code being added to the Laravel core
  - **Changes made:** Check all changed/added files
  - Click **OK**

Pushing to your Fork

Now that your local repository has your committed changes, it's time to push (or sync) your new branch to your fork that is hosted in GitHub:

- Right-click the Laravel directory and goto **Git Sync...**
- Git Syncronization
  - **Local Branch:** feature/localization-docs
  - **Remote Branch:** leave this blank
  - **Remote URL:** fork
  - Click **Push**
  - When asked for "username:" enter your GitHub *case-sensitive* username
  - When asked for "password:" enter your GitHub *case-sensitive* account

Your branch has been successfully pushed to your fork on GitHub.

Submitting a Pull Request

The final step is to submit a pull request to the Laravel repository. This means that you are requesting that the Laravel team pull and merge your changes to the Laravel core. In your browser, visit your Laravel fork athttps://github.com/username/laravel. Click on **Pull Request**. Next, make sure you choose the proper base and head repositories and branches:

- **base repo:** laravel/laravel
- **base branch:** develop

- **head repo:** username/laravel
- **head branch:** feature/localization-docs

Use the form to write a more detailed description of the changes you made and why you made them. Finally, click **Send pull request**. That's it! The changes you made have been submitted to the Laravel team.

What's Next?

Do you have another feature you want to add or another bug you need to fix? Just follow the same instructions as before in the Creating Branches section. Just remember to always create a new branch for every new feature/fix and don't forget to always base your new branches off of the *remotes/origin/develop* branch.