Configuration

Introduction

All of the configuration files for the Laravel framework are stored in the `config` directory. Each option is documented, so feel free to look through the files and get familiar with the options available to you.

Environment Configuration

It is often helpful to have different configuration values based on the environment where the application is running. For example, you may wish to use a different cache driver locally than you do on your production server.

To make this a cinch, Laravel utilizes the DotEnv PHP library by Vance Lucas. In a fresh Laravel installation, the root directory of your application will contain a `.env.example` file. If you install Laravel via Composer, this file will automatically be renamed to `.env`. Otherwise, you should rename the file manually.

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would get exposed.

If you are developing with a team, you may wish to continue including a `.env.example` file with your application. By putting placeholder values in the example configuration file, other developers on your team can clearly see which environment variables are needed to run your application. You may also create a `.env.testing` file. This file will override the `.env` file when running PHPUnit tests or executing Artisan commands with the `--env=testing` option.

Any variable in your `.env` file can be overridden by external environment variables such as server-level or system-level environment variables.

Environment Variable Types

All variables in your `.env` files are parsed as strings, so some reserved values have been created to allow you to return a wider range of types from the `env()` function:

`.env` Value	`env()` Value
true	(bool) true
(true)	(bool) true
false	(bool) false
(false)	(bool) false
empty	(string) ''
(empty)	(string) ''
null	(null) null
(null)	(null) null

If you need to define an environment variable with a value that contains spaces, you may do so by enclosing the value in double quotes.

```
APP_NAME="My Application"
```

Retrieving Environment Configuration

All of the variables listed in this file will be loaded into the `$_ENV` PHP super-global when your application receives a request. However, you may use the `env` helper to retrieve values from these variables in your configuration files. In fact, if you review the Laravel configuration files, you will notice several of the options already using this helper:

```
'debug' => env('APP_DEBUG', false),
```

The second value passed to the `env` function is the "default value". This value will be used if no environment variable exists for the given key.

Determining The Current Environment

The current application environment is determined via the `APP_ENV` variable from your `.env` file. You may access this value via the `environment` method on the `App` facade:

```
$environment = App::environment();
```

You may also pass arguments to the `environment` method to check if the environment matches a given value. The method will return `true` if the environment matches any of the given values:

```
if (App::environment('local')) {
    // The environment is local
}

if (App::environment(['local', 'staging'])) {
    // The environment is either local OR staging...
}
```

The current application environment detection can be overridden by a server-level `APP_ENV` environment variable. This can be useful when you need to share the same application for different environment configurations, so you can set up a given host to match a given environment in your server's configurations.

Hiding Environment Variables From Debug Pages

When an exception is uncaught and the `APP_DEBUG` environment variable is `true`, the debug page will show all environment variables and their contents. In some cases you may want to obscure certain variables. You may do this by updating the `debug_blacklist` option in your `config/app.php` configuration file.

Some variables are available in both the environment variables and the server / request data. Therefore, you may need to blacklist them for both `$_ENV` and `$_SERVER`:

```
return [

    // ...

    'debug_blacklist' => [
        '_ENV' => [
            'APP_KEY',
            'DB_PASSWORD',
        ],

        '_SERVER' => [
            'APP_KEY',
            'DB_PASSWORD',
        ],

        '_POST' => [
            'password',
        ],
    ],
];
```

Accessing Configuration Values

You may easily access your configuration values using the global `config` helper function from anywhere in your application. The configuration values may be accessed using "dot" syntax, which includes the name of the file and option you wish to access. A default value may also be specified and will be returned if the

configuration option does not exist:

```
$value = config('app.timezone');
```

To set configuration values at runtime, pass an array to the `config` helper:

```
config(['app.timezone' => 'America/Chicago']);
```

Configuration Caching

To give your application a speed boost, you should cache all of your configuration files into a single file using the `config:cache` Artisan command. This will combine all of the configuration options for your application into a single file which will be loaded quickly by the framework.

You should typically run the `php artisan config:cache` command as part of your production deployment routine. The command should not be run during local development as configuration options will frequently need to be changed during the course of your application's development.

If you execute the `config:cache` command during your deployment process, you should be sure that you are only calling the `env` function from within your configuration files. Once the configuration has been cached, the `.env` file will not be loaded and all calls to the `env` function will return `null`.

Maintenance Mode

When your application is in maintenance mode, a custom view will be displayed for all requests into your application. This makes it easy to "disable" your application while it is updating or when you are performing maintenance. A maintenance mode check is included in the default middleware stack for your application. If the application is in maintenance mode, a `MaintenanceModeException` will be thrown with a status code of 503.

To enable maintenance mode, execute the `down` Artisan command:

```
php artisan down
```

You may also provide `message` and `retry` options to the `down` command. The `message` value may be used to display or log a custom message, while the `retry` value will be set as the `Retry-After` HTTP header's value:

```
php artisan down --message="Upgrading Database" --retry=60
```

Even while in maintenance mode, specific IP addresses or networks may be allowed to access the application using the command's `allow` option:

```
php artisan down --allow=127.0.0.1 --allow=192.168.0.0/16
```

To disable maintenance mode, use the `up` command:

```
php artisan up
```

You may customize the default maintenance mode template by defining your own template at `resources/views/errors/503.blade.php`.

Maintenance Mode & Queues

While your application is in maintenance mode, no queued jobs will be handled. The jobs will continue to be handled as normal once the application is out of maintenance mode.

Alternatives To Maintenance Mode

Since maintenance mode requires your application to have several seconds of

downtime, consider alternatives like Envoyer to accomplish zero-downtime deployment with Laravel.

Routing
Blade Templates
Authentication
Authorization
Artisan Console
Database
Eloquent ORM
Testing

Laracon
Laracon EU
Laracon AU
Jobs
Certification
Forums

Kirschbaum
Byte 5
64Robots
Cubet
DevSquad
Ideil
Cyber-Duck
ABOUT YOU
Become A Partner

Envoyer
Horizon
Lumen
Nova
Echo
Valet
Mix
Spark
Cashier
Homestead
Dusk
Passport
Scout
Socialite