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.

These configuration files allow you to configure things like your database connection information, your mail server information, as well as various other core configuration values such as your application URL and encryption key.

The `about` Command

Laravel can display an overview of your application's configuration, drivers, and environment via the `about` Artisan command.

```
php artisan about
```

If you're only interested in a particular section of the application overview output, you may filter for that section using the `--only` option:

```
php artisan about --only=environment
```

Or, to explore a specific configuration file's values in detail, you may use the `config:show` Artisan command:

```
php artisan config:show database
```

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. In a fresh Laravel installation, the root directory of your application will contain a `.env.example` file that defines many common environment variables. During the Laravel installation process, this file will automatically be copied to `.env`.

Laravel's default `.env` file contains some common configuration values that may differ based on whether your application is running locally or on a production web server. These values are then read by the configuration files within the `config` directory using Laravel's `env` function.

If you are developing with a team, you may wish to continue including and updating the `.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.

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

Environment File Security

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.

However, it is possible to encrypt your environment file using Laravel's built-in environment encryption. Encrypted environment files may be placed in source control safely.

Additional Environment Files

Before loading your application's environment variables, Laravel determines if an `APP_ENV` environment variable has been externally provided or if the `--env` CLI argument has been specified. If so, Laravel will attempt to load an `.env.[APP_ENV]` file if it exists. If it does not exist, the default `.env` file will be loaded.

Environment Variable Types

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

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

The second value passed to the `env` function is the "default value". This value will be returned 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:

```
use Illuminate\Support\Facades\App;
$environment = App::environment();
```

You may also pass arguments to the `environment` method to determine 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...
}
```

> [!NOTE] The current application environment detection can be overridden by defining a server-level `APP_ENV` environment variable.

Encrypting Environment Files

Unencrypted environment files should never be stored in source control. However, Laravel allows you to encrypt your environment files so that they may safely be added to source control with the rest of your application.

Encryption

To encrypt an environment file, you may use the `env:encrypt` command:

```
php artisan env:encrypt
```

Running the `env:encrypt` command will encrypt your `.env` file and place the encrypted contents in an `.env.encrypted` file. The decryption key is presented in the output of the command and should be stored in a secure password manager. If you would like to provide your own encryption key you may use the `--key` option when invoking the command:

```
php artisan env:encrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```

> [!NOTE] The length of the key provided should match the key length required by the encryption cipher being used. By default, Laravel will use the `AES-256-CBC` cipher which requires a 32 character key. You are free to use any cipher supported by Laravel's encrypter by passing the `--cipher` option when invoking the command.

If your application has multiple environment files, such as `.env` and `.env.staging`, you may specify the environment file that should be encrypted by providing the environment name via the `--env` option:

```
php artisan env:encrypt --env=staging
```

Decryption

To decrypt an environment file, you may use the `env:decrypt` command. This command requires a decryption key, which Laravel will retrieve from the `LARAVEL_ENV_ENCRYPTION_KEY` environment variable:

```
php artisan env:decrypt
```

Or, the key may be provided directly to the command via the `--key` option:

```
php artisan env:decrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```

When the `env:decrypt` command is invoked, Laravel will decrypt the contents of the `.env.encrypted` file and place the decrypted contents in the `.env` file.

The `--cipher` option may be provided to the `env:decrypt` command in order to use a custom encryption cipher:

```
php artisan env:decrypt --key=qUWuNRdfuImXcKxZ --cipher=AES-128-CBC
```

If your application has multiple environment files, such as `.env` and `.env.staging`, you may specify the environment file that should be decrypted by providing the environment name via the `--env` option:

```
php artisan env:decrypt --env=staging
```

In order to overwrite an existing environment file, you may provide the `--force` option to the `env:decrypt` command:

```
php artisan env:decrypt --force
```

Accessing Configuration Values

You may easily access your configuration values using the `Config` facade or global `config` 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:

```
use Illuminate\Support\Facades\Config;

$value = Config::get('app.timezone');

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

// Retrieve a default value if the configuration value does not exist...
$value = config('app.timezone', 'Asia/Seoul');
```

To set configuration values at runtime, you may invoke the `Config` facade's `set` method or pass an array to the `config` function:

```
Config::set('app.timezone', 'America/Chicago');

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

To assist with static analysis, the `Config` facade also provides typed configuration retrieval methods. If the retrieved configuration value does not match the expected type, an exception will be thrown:

```
Config::string('config-key');
Config::integer('config-key');
Config::float('config-key');
Config::boolean('config-key');
Config::array('config-key');
```

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 can be quickly loaded by the framework.

You should typically run the `php artisan config:cache` command as part of your production deployment process. 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.

Once the configuration has been cached, your application's `.env` file will not be loaded by the framework during requests or Artisan commands; therefore, the `env` function will only return external, system level environment variables.

For this reason, you should ensure you are only calling the `env` function from within your application's configuration (`config`) files. You can see many examples of this by examining Laravel's default configuration files. Configuration values may be accessed from anywhere in your application using the `config` function described above.

The `config:clear` command may be used to purge the cached configuration:

```
php artisan config:clear
```

> [!WARNING] 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; therefore, the `env` function will only return external, system level environment variables.

Configuration Publishing

Most of Laravel's configuration files are already published in your application's `config` directory; however, certain configuration files like `cors.php` and `view.php` are not published by default, as most applications will never need to modify them.

However, you may use the `config:publish` Artisan command to publish any configuration files that are not published by default:

```
php artisan config:publish

php artisan config:publish --all
```

Debug Mode

The `debug` option in your `config/app.php` configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the `APP_DEBUG` environment variable, which is stored in your `.env` file.

> [!WARNING] For local development, you should set the `APP_DEBUG` environment variable to `true`. **In your production environment, this value should always be `false`. If the variable is set to `true` in production, you risk exposing sensitive configuration values to your application's end users.**

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 `Symfony\Component\HttpKernel\Exception\HttpException` instance will be thrown with a status code of 503.

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

```
php artisan down
```

If you would like the `Refresh` HTTP header to be sent with all maintenance mode responses, you may provide the `refresh` option when invoking the `down` command. The `Refresh` header will instruct the browser to automatically refresh the page after the specified number of seconds:

```
php artisan down --refresh=15
```

You may also provide a `retry` option to the `down` command, which will be set as the `Retry-After` HTTP header's value, although browsers generally ignore this header:

```
php artisan down --retry=60
```

Bypassing Maintenance Mode

To allow maintenance mode to be bypassed using a secret token, you may use the `secret` option to specify a maintenance mode bypass token:

```
php artisan down --secret="1630542a-246b-4b66-afa1-dd72a4c43515"
```

After placing the application in maintenance mode, you may navigate to the application URL matching this token and Laravel will issue a maintenance mode bypass cookie to your browser:

```
https://example.com/1630542a-246b-4b66-afa1-dd72a4c43515
```

If you would like Laravel to generate the secret token for you, you may use the `with-secret` option. The secret will be displayed to you once the application is in maintenance mode:

```
php artisan down --with-secret
```

When accessing this hidden route, you will then be redirected to the `/` route of the application. Once the cookie has been issued to your browser, you will be able to browse the application normally as if it was not in maintenance mode.

> [!NOTE] Your maintenance mode secret should typically consist of alpha-numeric characters and, optionally, dashes. You should avoid using characters that have special meaning in URLs such as `?` or `&`.

Maintenance Mode on Multiple Servers

By default, Laravel determines if your application is in maintenance mode using a file-based system. This means to activate maintenance mode, the `php artisan down` command has to be executed on each server hosting your application.

Alternatively, Laravel offers a cache-based method for handling maintenance mode. This method requires running the `php artisan down` command on just one server. To use this approach, modify the maintenance mode variables in your application's `.env` file. You should select a cache `store` that is accessible by all of your servers. This ensures the maintenance mode status is consistently maintained across every server:

```
APP_MAINTENANCE_DRIVER=cache
APP_MAINTENANCE_STORE=database
```

Pre-Rendering the Maintenance Mode View

If you utilize the `php artisan down` command during deployment, your users may still occasionally encounter errors if they access the application while your Composer dependencies or other infrastructure components are updating. This occurs because a significant part of the Laravel framework must boot in order to determine your application is in maintenance mode and render the maintenance mode view using the templating engine.

For this reason, Laravel allows you to pre-render a maintenance mode view that will be returned at the very beginning of the request cycle. This view is rendered before any of your application's dependencies have loaded. You may pre-render a template of your choice using the `down` command's `render` option:

```
php artisan down --render="errors::503"
```

Redirecting Maintenance Mode Requests

While in maintenance mode, Laravel will display the maintenance mode view for all application URLs the user attempts to access. If you wish, you may instruct Laravel to redirect all requests to a specific URL. This may be accomplished using the `redirect` option. For example, you may wish to redirect all requests to the `/` URI:

```
php artisan down --redirect=/
```

Disabling Maintenance Mode

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

```
php artisan up
```

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

Maintenance Mode and 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 running your applications on a fully-managed platform like Laravel Cloud to accomplish zero-downtime deployment with Laravel.