

# Laravel 3

公式ドキュメント

日本語翻訳版



翻訳：川瀬 裕久

# Laravel 3 公式ドキュメント日本語版

Laravel 3 の公式ドキュメントを日本語に翻訳したものです。

Hirohisa Kawase

This book is for sale at <http://leanpub.com/laravel-3-japanese>

This version was published on 2013-04-26



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2012 - 2013 Hirohisa Kawase

# Contents

前書き	i
修正履歴	iv
1 概要	1
2 L a r a v e l 変更ログ	4
3 インストールと準備	8
4 ルーティング	12
5 コントローラー	21
6 モデルとライブラリー	27
7 ビューとレスポンス	31
8 アセットの管理	38
9 テンプレート	40
10 ペジネーション	46
11 HTMLの作成	49
12 フォームの作成	53
13 入力とクッキー	58
14 バンドル	63
15 クラスのオートロード	69
16 エラーとログ	71
17 プロファイラー	73
18 実行時の環境設定	75
19 リクエストの確認	76
20 URLの生成	78

## CONTENTS

21 イベント	81
22 バリデーション	84
23 ファイルの使用	97
24 文字列の使用	100
25 ローカリゼーション	102
26 暗号化	104
27 I o C コンテナ	105
28 ユニットテスト	107
29 データベース設定	109
30 生のクエリー	111
31 F l u e n t クエリービルダー	113
32 E l o q u e n t ORM	121
33 スキーマビルダー	136
34 マイグレーション	140
35 R e d i s	142
36 キャッシュ設定	144
37 キャッシュ使用法	147
38 セッション設定	149
39 セッション使用法	152
40 認証設定	154
41 認証使用法	155
42 A r t i s a n コマンド	158
43 タスク	161
44 G i t H u b の L a r a v e l	164
45 コマンドラインで L a r a v e l に貢献する	166
46 T o r t o i s e G i t で L a r a v e l に貢献する	170
おまけ	174

# 前書き

## この書籍について

この電子書籍は[Laravel 公式ドキュメント](http://laravel.kore1server.com)<sup>1</sup>を、個人的に翻訳したものです。公式ドキュメントはオリジナル英語版が Laravel の配布ファイルに含まれており、Laravel インストール後、/docs にアクセスすることで、閲覧することができます。

フレームワークを始め、新しいサービスを日本人が活用するには、日本語のドキュメントが必要です。Laravel の面白さと使いやすさに感動したため、急いで翻訳しました。翻訳したドキュメントは、以下のサイトで公開しております。Web で閲覧したい方は、どうぞ活用ください。もちろん、無料で閲覧していただけます。

- <http://laravel.kore1server.com>
- <http://laravel-ja.phpfogapp.com>
- <http://laravel-ja.pagodabox.com>

一番上のアドレスが、最新版となっております。以降のアドレスはコピーサイトです。他のサイトは最新版のサイトがダウンした場合のバックアップとして用意しました。

また、ローカルサーバーでこの日本語ドキュメントを利用したい方は、<http://github.com/HiroKws/Laravel-base-32/zipball/original-css>から、日本語ドキュメントを含んだ、配布 zip をダウンロードしていただけます。内容につきましては、<http://kore1server.com/laravel-tutorial/312-laravel-32-development-base-sample>をご覧ください。（メンテナンスの関係上、含めている日本語ドキュメントは常時最新版に保っていません。）

本来、Markdown により記述されており、HTML に変換し Web で表示するために書かれているドキュメントです。書籍としてそぐわない表現もあり、その部分に関しては多少書き換えました。

## サポート

この書籍のサポートは Laravel 3 のドキュメントのサポート期間内とさせていただきます。

現在リリースされている Laravel 3 のバージョンの開発期間中、リリースごとに原文のドキュメントが更新されます。原文の変更に合わせて、最新版を配布します。また、誤記や翻訳の修正、表示の改善は随時行います。

ちなみに、Laravel 4 のリリースは 2013 年 5 月です。

電子版を販売する Leanpub 社の更新通知システムを通じ、メールにて更新をお伝えします。その中のダウンロードリンクから無料で最新版を入手できます。

ダウンロードされる電子書籍のファイル名はいつも同じ名前になります。古いバージョンを保存しておきたい方は、上書きされないように管理してください。旧バージョンのダウンロードはできません。

---

<sup>1</sup><http://www.laravel.com/docs>

## ライセンス

電子書籍を含め、書籍として出版されたドキュメントは、私が著作権を保持します。ただし、同じ内容を Web 上で公表している日本語ドキュメント、配布パッケージに含まれている日本語ドキュメントに関しては、オリジナルのドキュメントと同様に MIT ライセンスでご利用いただけます。

## L a r a v e l とは

新しい軽量 PHP フレームワークの一つです。後発の利を生かし、様々なフレームワークから機能を取り入れています。コードが読みやすくなるように、設計されています。そのため、メンテナンスがしやすいフレームワークです。Laravel という名前は適当に創りだされた単語であり、特別の意味を持っていません。

Laravel が他のフレームワークとは毛色が異なる多くの特徴を持っているのは、開発者 Taylor Otwell 氏のバックボーンにあるのでしょう。

彼は、PHP による Web 開発の専門家ではありませんでした。そのため、多くの PHP フレームワークや SQL ライブラリーにありがちな、ソースの見づらさを当然のものと受け取りませんでした。彼の以前の経歴は Microsoft の .NET solutions に携わっていました。

当初、Laravel は Taylor 氏の遊びで作られたものです。しかし、これを利用して Web アプリを開発した会社が資金援助を行い、結果ここまで発展しました。実に一年での急成長です。

既に 2013 年の 2 月にカンファレンスがワシントン DC で開かれ、小中規模のスポンサーが 11 社も付きました。

もし読者の方が、大規模なアプリ開発に携わっており、上流からのきちんとした設計に基づく開発手法を取られ、人員も確保できるのでしたら、他の有名フレームワークをご利用されたほうがよろしいでしょう。Laravel はまだ未熟な部分もあります。

活用できるベストシナリオとしては「アジャイルスタイルの開発方法を取り、数人のグループで新しいアプリを約一ヶ月で開発する」ような場合でしょう。PHP フレームワークでの経験があれば、数日もあれば十分習得でき、コードの読みやすさは、顧客からの要求に対応しやすくなります。複雑なフレームワークの全体を把握するだけで一ヶ月かかることもありません。ちょっとした変更にも、自分の書いた難しいソースを読み解くこともありません。（もちろん、いくら Laravel を使用しても、複雑に書いてしまえば、元の木阿弥ですよ。）このベストシナリオは、ここに上げたような条件で Laravel を用い実際に開発を行った方が、彼の記事で Laravel を褒めていた内容を紹介したものです。

Laravel は学習コストが最低で済みます。たやすく習得できるフレームワークです。ですから、ある程度の人数で開発するのだが、共通のフレームワークの経験が見つからず、学習に手間取らないフレームワークを探している場合にピッタリです。

もしくは、多くの Web サイト開発見られるようなアジャイルスタイルの開発手法を取られているチームにも適しているでしょう。コードの読みやすさは抜群です。

更に、数多くの案件を個人で請け負ってられる方にも適しているでしょう。顧客からの修正依頼にも、今までより気軽に答えられるようになるでしょう。

また、新たにフレームワーク自身を勉強したい方にも適しています。最初に大きなフレームワークにとりかかるのでは、全体を把握するだけでも、時間がかかります。余りにも小さなフレームワークでは、使用するメリットを十分に感じられないでしょう。学習しやすく、それなりに機能を備えた Laravel は学習目的にも最適です。

開発するのが「楽しい」と感じさせる不思議なフレームワークです。趣味であれ、仕事であれ、開発の楽しみを感じさせてくれる Laravel を多くの方に使用していただきたいと思います。

## Leanpub

電子書籍版は Leanpub を利用し販売しています。

国内のサービスを介せず、Leanpub を利用するのは、メンテナンスの理由です。Leanpub で一度購入いただくと、書籍の内容を修正した場合、購入者にメールで通知を行い、新しいバージョンを無料でダウンロードしていただける仕組みになっています。

これにより、多くの時間を費やし、「完全な書物」を用意したため、読者にとって最適な時期に、書籍を手元に届けられない悲劇を回避できます。多くのテクニカルな書物で見られるような、「ちょっと機会を逃した」残念な出版を避ける事ができます。

現在の読者は Web 情報に新しさを、書物には正確さを期待しているとも言えます。Leanpub での出版物は最初は Web 情報と同じく、「新しいが、正確さに欠ける」かも知れません。しかし、アップデートが可能であるということは、出版物の内容の修正・追加が可能です。読者からのフィードバックを受け付け、直ぐに対応できるということです。その結果、内容を「常に新しく、正確」なものへと育ててすることができます。

出版ごとにサイトを用意し、正誤表を公開するという、著者と読者にとって手間がかかることを行わなくとも、常に最新版を読者の手元に置いていただけます。

このドキュメントは原文が存在するため、日本語版を勝手に変更はしません。ですが、誤字脱字、翻訳の間違いなどをご指摘いただければ、可能な限り迅速に対処させていただきます。

Leanpub は新しいサービスで、日本語の対応に関してまだ問題を抱えております。ですから、Leanpub の開発者と連絡を取り、少しずつクオリティーも上げていきたいと思います。

また無料開放ではなく、有料での配布にさせていただくのは、私が体調不良のため働けず、サイト維持のためいくらかのお金をご寄付いただかなければならないためです。元気でバリバリ働けるのであれば、私個人でも、サイト維持は難なく可能ですが、現状それが不可能です。（こうした状況であり、時間が取れるため、翻訳・出版も可能だったので、ネガティブな意味だけでありません。）

値段は、最低 5.55 ドル、希望価格 7.77 ドルにさせていただいております。当初は最低 1.11 ドルに設定していましたが、アップデートの頻度が結構高いため、作業量にそぐわなくなってしまう、値上げしました。この値段はドネーションの意味も込め、幅を持たせております。無理のない範囲で値段を決めていただけます。もし会社等で購入し、多少のドネーションを行なっても良いという場合は、表示される値段をクリックしていただければ、値段を直接ご指定いただけます。

支払いは Paypal およびカードです。今回の Laravel 3 公式ドキュメント日本語翻訳電子書籍版とは関係なく、日本語での情報提供に対しドネーションいただける方は、Paypal アカウント [hiro.soft@gmail.com](mailto:hiro.soft@gmail.com) へ寄付をお願いします。また、Laravel 開発者の Taylor Otwell 氏へ寄付されたい方は、[taylorotwell@gmail.com](mailto:taylorotwell@gmail.com) の Paypal アカウントへどうぞ。

川瀬 裕久

# 修正履歴

## 2012/07/16

- 出版（発行）

## 2012/07/27

- 誤記修正 (前書きでアジャイルを Ajax と誤記)

## 2012/08/16

- Laravel 3.2.4 に対応した。
- 書籍向きには未修正のまま。
- テーブル表示に対応した。
- 抜けていた認証の章を追加した。
- タイトルなどの読みやすさ調整した。

## 2012/08/17

- 3.2.4 翻訳完成版

## 2012/08/18

- 文書内クロスリンクに対応した。

## 2012/08/19

- 3.2.5 翻訳完成版
  - artisan のタスクをアプリケーションから呼び出すメソッドの使い方が追加された。
  - ネストした where の処理コードを 3.2.3 へ戻す緊急リリースらしく、ドキュメントに大きな変更は無い。

## 2012/09/03

- 3.2.7 翻訳完成版
  - 原文で Blade テンプレートの @parent の記述に多少の変更あり。
  - スキーマビルダーの外部キーに関して、定義時の注意が記述された。
  - Github での貢献に対し、プルリクエストへの指針が提示された。



## 2012/09/26

- 3.2.8 翻訳完成版
  - オリジナルの英語版のタイポ修正や整形作業が多いため、日本語訳に影響する部分は少ない。
  - ルーティングの章、コントローラルーティングに `Controller::detect()` 使用時の注意が追加された。また、`(:all)` の使用例も追加された。
  - Eloquent の章に、`touch()` と `timestamp()` メソッドの説明が追加された。

## 2012/11/01

- 3.2.11 翻訳完成版
  - ファイルアップロードの際のフォームの `open` に関する注意喚起が追加された。これはフォーラムに `HTML::open()` を使っているため、アップロードできないという質問が多いためと思われる。
  - マイグレーションのサブコマンドに `rebuild` が追加された。

## 2013/01/11

- 3.2.13 翻訳速報版
  - 機能は存在したが、マニュアルから抜けていた箇所がところどころ追加された。
  - プロファイルの使い方が新たなページとして追加された。
  - バリデーションで配列の要素数をバリディートできるようになった。
  - 多言語対応の URL を生成できるようになった。
  - Fluent クエリービルダーで `lists()` と `group_by()` の記述が追加された。

## 2013/04/26

- 3.2.14 翻訳速報版
  - Eloquent の章に追加した文章中のメソッド名間違いを修正した。(コードには正しい名前が指定されている)
  - Fluent の章で `where` をチェーンでつなぐと `AND WHERE` になることを明記した。
  - IoC の章に `unregister` メソッドを追加した。
  - Form の章で `label` の第4パラメーターを追加した。`false` でエスケープしない。
  - その他原文のタイポが修正されたが、日本語翻訳文には影響しない。

## 2013/04/27

- 3.2.14 翻訳正式版
  - おまけのコメント翻訳コアを最新版の 3.2.14 で作成
  - 日本翻訳文を読みやすさのため数カ所変更。内容の変更はなし

# 1 概要

## 1.1 初めに

Laravel のドキュメントへようこそ。このドキュメントはスタートガイドとして、さらに特徴の紹介としても役立つように書かれています。どこから読んでも学習できますが、以前に学んだ概念をもとに、その後に続くドキュメントは書かれていますので、初めから順番に読むことをお勧めします。

## 1.2 Laravel を楽しめるのは誰？

Laravel は柔軟性と読み書きしやすさを重視した、パワフルなフレームワークです。初めて Laravel に触れる方は、人気がある軽量な PHP フレームワークを使用して開発する時と同じ、安らぎを感じるでしょう。もうちょっと経験を積んだユーザーであれば、他のフレームワークではできない方法で、コードをモジュール化できることを評価するでしょう。Laravel の柔軟性は、要求に何度でも応じ、アプリケーションを修正しながら、形作ることを可能にし、表現性はあなたとあなたのチームが開発するコードをシンプルで読みやすくしてくれるでしょう。

## 1.3 Laravel はどこが違うの？

Laravel には他のフレームワークと違った特徴を数多く持っています。特に重要な点をいくつか紹介しましょう。

- バンドルは Laravel のモジュールパッキングシステムです。[Laravel バンドルリポジトリ](http://bundles.laravel.com/)<sup>1</sup>は、アプリケーションへ簡単に機能を付け加えられるように、予め用意されています。バンドルリポジトリから bundles ディレクトリにダウンロードしても良いですし、“Artisan” コマンドラインツールを使い、自動的にインストールすることもできます。
- **Eloquent ORM**は最も進化した PHP アクティブレコードを実装しています。リレーションシップとネストされた eager ローディングで簡単に制約を適用できる能力を使えば、自分のデーターを完全にコントロールでき、アクティブレコードの便利さを十分に体験できるでしょう。Eloquent は Laravel のクエリービルダーである Fluent のメソッドを完全にサポートしています。
- アプリケーションロジックを（多くの Web 開発者にはお馴染みの）コントローラーでアプリケーションに実装することもできますし、また Sinatra フレームワークと似たようなシンタックスを使い、ルートの定義に直接記述することもできます。Laravel は小さなサイトから、巨大なエンタープライズアプリケーションまで、必要に応じて全て作成できるだけの柔軟性を開発者に提供する哲学で、設計されています。
- リーバスルーティングで名前付きのルートヘリンクを作成できます。リンクを作成するときにルートの名前を使えば、Laravel は自動的に正しい URI を挿入します。これを使うことにより、後ほどルートを変更しても、Laravel がサイト中のリンク全部を適切に更新します。

---

<sup>1</sup><http://bundles.laravel.com/>

- **Rest** コントローラーは GET と POST のロジックを分ける一つの手法です。例えばログインにおいて、コントローラーの `get_login()` アクションでフォームを担当させ、コントローラーの `post_login()` アクションで、送信されたフォームを受け取り、バリデーションし、エラーメッセージと一緒にログインフォームにリダイレクトさせたり、各ユーザーのダッシュボードにリダイレクトさせたりできます。
- クラスのオートロードはオートロードの環境設定を保つ手間を省き、使用していない不必要なコンポーネントをロードしてしまうことを防げます。ライブラリーやモジュールを使いたいのですか？ローディングに悩むことはありません。どうぞ使ってください。後は **Laravel** が面倒を見ます。
- ビューコンポーサーはビューがロードされた時点で実行されるコードブロックです。良い例がブログのサイドナビに見られる、投稿をランダムにリスト表示するものです。コンポーサーは必要のあるブログポストを全てロードするロジックで構成されるでしょう。そうしてビューをロードすれば、表示する準備は全て予め済んでいるわけです。これにより、メソッドのページコンテンツに関連する、ビューのモジュールで使用するデータのロードを、全てのコントローラー側で確実に行わなくてはならない手間を省くことができます。
- **IoC** コンテナ (**Inversion of Control**) は新しいオブジェクトを生成するメソッドを提供し、随意にインスタンスを生成したり、シングルトンでの使用をできるようにするものです。IoC により、外部ライブラリーの使用準備を行う必要は減多になります。また、きっちりと決まった柔軟性のないファイル構造に係わる必要はなく、IoC を使用したオブジェクトにはコードのどこからでもアクセスできることも意味しています。
- マイグレーションはデータベーススキーマのバージョンコントロールで、**Laravel** に直接統合されています。生成も実行も **"Artisan"** コマンドラインユーティリティーを使用して行えます。他のメンバーがスキーマを変更したら、リポジトリからコピーをローカル環境に置き、マイグレーションを実行します。すると、あなたのデータベースもアップデートされます！
- ユニットテストは **Laravel** の大切な一部です。**Laravel** 自身も何百ものテストにより、新しい変更が予期せず他の部分を壊していないことを確認するために使っています。これは、**Laravel** が業界で最も安定してるフレームワークであると考えられている理由の一つです。さらに **Laravel** は皆さんが自分のコードにユニットテストを書くのを簡単にしてくれます。その後で、**"Artisan"** コマンドユーティリティーを使いテストを実行できます。
- 自動ペジネーションはアプリケーションロジックがペジネーションの設定のためにごちゃごちゃになることを防ぎます。現在のページを得て、DB のレコード数を取得し、`limit/offset` を使用してデーターを `SELECT` する代わりに、ただ **"paginate"** を呼び出し、ビューのどこにページリンクを出力するのか **Laravel** に教えて下さい。**Laravel** は自動的に残りの面倒を見ます。**Laravel** のペジネーションシステムは簡単に使用でき、簡単に変更できるように設計されています。強調しますが、**Laravel** がこれらを自動的に処理するからといっても、自分で呼び出したリ、システムを設定できないわけではありません。そうしたければ、手動で行えます。

これは他の PHP フレームワークとの違いを示す、わずかな例にすぎません。こうした特徴とその他すべて、このドキュメント全体を通して記述してあります。

## 1.4 アプリケーション構造

**Laravel** のディレクトリー構造は他の人気のある PHP フレームワークと似せて設計されています。他のフレームワークで採用されている方法と似ている構造を使うことで、どんなアプリケーションでも、どんなサイズのものでも簡単に作成できます。

Laravel のアーキテクチャがユニークだからといっても、アプリケーションに合わせて、開発者が独自の構造を構築することも可能です。これはコンテンツマネジメントシステムのような大きなプロジェクトに有効でしょう。こうした柔軟な構造は Laravel 独自のものです。

このドキュメントを通し、設置するのに最適なデフォルトの位置を指定していきたいと思います。

## 1.5 Laravel のコミュニティ

Laravel フォーラム<sup>2</sup>は手助けを得たり、手助けしたり、もしくは他の人が何を言っているかただ眺めたりできる素晴らしい場所です。

我々の多くは毎日 FreeNode の #laravel IRC チャンネルに接続しています。Laravel のフォーラム記事に接続方法が説明されています。<sup>3</sup> この IRC チャンネルにつながっぱなしにすることは、Laravel を使用する Web 開発について多くを学ぶ方法です。どうぞ質問をし、他の人の質問に答え、もしくはつないだままにして、他の人の質問と答から学んでください。私達は Laravel を愛していますし、Laravel について話すのも大好きです。ですからよそ者にはならないでください！

## 1.6 ライセンス情報

Laravel はMIT ライセンス<sup>4</sup>のもとにライセンスされているオープンソースのソフトウェアです。

---

<sup>2</sup><http://forums.laravel.com>

<sup>3</sup><http://forums.laravel.com/viewtopic.php?id=671>

<sup>4</sup><http://www.opensource.org/licenses/mit-license.php>

## 2 Laravel 変更ログ

### 2.1 Laravel 3.2.14

- IoC でデフォルトパラメーターを解決できるように追加
- Postgres で insert\_get\_id を FETCH\_ASSOC で使用した時のバグ修正

#### 3.2.13からのアップグレード

- `laravel` フォルダを置き換え

### 2.2 Laravel 3.2.13

- Symfony HttpFoundation を 2.1.6. へアップグレード
- フレームワークの数多くの不具合の修正

#### 3.2.12からのアップグレード

- `laravel` フォルダを置き換え

### 2.3 Laravel 3.2.12

- Clear sections on a complete render operation.

#### 3.2.11からのアップグレード

- `laravel` フォルダを置き換え

### 2.4 Laravel 3.2.11

- Eager ロードのマッチングのパフォーマンス改善
- 環境の決定時に `gethostname` をチェックするように変更

#### 3.2.10からのアップグレード

- `laravel` フォルダを置き換え

## 2.5 Laravel 3.2.10

- Eloquent モデルのバグフィックス

### 3.2.9からのアップグレード

- `laravel` フォルダーを置き換え

## 2.6 Laravel 3.2.9

- たとえ”logger” イベントリスナーのものであろうと、いつでも例外をログするように変更
- 見づらいビューでの例外メッセージを修正

### 3.2.8からのアップグレード

- `laravel` フォルダーを置き換え

## 2.7 Laravel 3.2.8

- “index.php” を付けずに言語を URL に指定した場合、スラッシュがつながるバグを修正
- 認証の”Remember me” クッキーでセキュリティ問題となり得る部分の修正

### 3.2.7からのアップグレード

- `laravel` フォルダーを置き換え

## 2.8 Laravel 3.2.7

- Eloquent の `to_array` メソッドのバグ修正
- 一般的なエラーページ表示のバグ修正

### 3.2.6からのアップグレード

- `laravel` フォルダーを置き換え

## 2.9 Laravel 3.2.6

- Blade クラスのコードを 3.2.3 ヘダウングレード

### 3. 2. 5からのアップグレード

- **laravel**フォルダーを置き換え

## 2.10 Laravel 3. 2. 5

- ネストした **where** に対するコードを 3.2.3 ヘダウングレード

### 3. 2. 4からのアップグレード

- **laravel**フォルダーを置き換え

## 2.11 Laravel 3. 2. 4

- 多対多 **eager** ローディングのスピードアップ
- **Eloquent::change()** メソッドの調節
- 多くのバグフィックスと機能向上

### 3. 2. 3からのアップグレード

- **laravel**フォルダーを置き換え

## 2.12 Laravel 3. 2. 3

- **Eloquent** の **eager** ローディングの修正
- 全ての **IoC** の解決に対し、“**laravel.resolving**” イベントを追加

### 3. 2. 2からのアップグレード

- **laravel**フォルダーを置き換え

## 2.13 Laravel 3. 2. 2

- **Postgres** サポートの全体的な改善
- **SQL** サーバースキーマ構文の問題を修正
- **eager** ローディングの“**first**”と“**find**”の問題を修正
- “**IoC::resolve**” に引数を渡されないことで発生するバグを修正
- 環境の設定で、**hostnames** の指定を行えるようにした
- “**DB::last\_query**” メソッドの追加
- **Auth** 設定で、“**password**” オプションを追加

### 3. 2. 1 からのアップグレード

- **laravel** フォルダを置き換え

## 2.14 L a r a v e l 3. 2. 1

- 同じリクエストで設定したクッキーを取得する場合のバグを修正
- プライマリーキーに対する SQL サーバーの構文に関するバグを修正
- PHP 5.4 のバリデーターに関するバグを修正
- リンクを生成する場合に HTTP/HTTPS を指定しない場合、現在のプロトコルを使用する
- Eloquent Auth ドライバーのバグ修正
- メッセージコンテナに "format" メソッドを追加

### 3. 2 からのアップグレード

- **laravel** フォルダを置き換え



## 3 インストールと準備

### 3.1 動作要件

- Apache か nginx、もしくは他の互換性のある Web サーバー
- Laravel は PHP 5.3 で導入されたパワフルで有益な機能を使用しています。最低でも PHP 5.3 が必要です。
- Laravel は[Fileinfo ライブラリー<sup>1</sup>](http://php.net/manual/ja/book.fileinfo.php)をファイルの mime タイプを判断するために使用しています。これはデフォルトで PHP5.3 に含まれています。しかしながら、Windows ユーザーは Fileinfo モジュールを有効にするために、php.ini に一行書き加える必要があります。もっと詳しく知るために、次のリンクを参照してください。[PHP.net : インストール／設定の詳細<sup>2</sup>](http://php.net/manual/ja/install.configuration.php)
- Laravel は[Mcrypt ライブラリー<sup>3</sup>](http://php.net/manual/ja/book.mcrypt.php)を暗号とハッシュの生成に使用しています。Mcrypt は通常ブリーインストールされています。もし phpinfo() の出力に Mcrypt が見つからない時には、あなたの LAMP インストールのベンダーサイトを参照するか、[PHP.net : インストール／設定の詳細<sup>4</sup>](http://php.net/manual/ja/install.configuration.php)を御覧ください。

### 3.2 インストール

1. [Laravel をダウンロード<sup>5</sup>](http://laravel.com/download)する
2. Laravel 圧縮ファイルを解凍し、コンテンツを Web サーバーにアップロードする
3. `config/application.php`ファイルの `key` オプションに、でたらめな 32 文字の値をセットする
4. “storage/views” ディレクトリーが書き込み可能になっていることを確認する
5. ブラウザでアプリケーションにアクセスする

全てを上手くやれば、Laravel の可愛いスプラッシュページが表示されるでしょう。準備をしてください。まだまだ多くのことを学ばなくてはなりません。

#### 追加の機能

Laravel の利便性を完全に利用するために、以下の機能もインストールすることができます。しかし、必要ではありません。

- SQLite、MySQL、PostgreSQL、もしくは SQL サーバーPDO ドライバー
- Memcached か APC

---

<sup>1</sup><http://php.net/manual/ja/book.fileinfo.php>

<sup>2</sup><http://php.net/manual/ja/fileinfo.installation.php>

<sup>3</sup><http://php.net/manual/ja/book.mcrypt.php>

<sup>4</sup><http://php.net/manual/ja/book.mcrypt.php>

<sup>5</sup><http://laravel.com/download>

## 問題ですか？

もし、何か問題があれば、以下を試してください。

- **public**ディレクトリーが Web サーバーのドキュメントルートになっているのを確認する（次のサーバー設定を参照してください）
- `mod_rewrite` を使用しているなら、`applicaton/config/application.php`にある `index`オプションに空文字列を指定する
- Web サーバー上の `storage` フォルダーとその中のサブフォルダーが書き込み可能になっていることを確認してください。

## 3.3 サーバー設定

多くの Web 開発フレームワークと同様に、Laravel は Web サーバーのドキュメントルートには公開する必要のあるファイルだけを設置することにより、あなたのアプリケーションコード、バンドル、ローカルストレージを保護するように設計されています。これはある種のサーバー設定のミスにより、Web を通じて（データベースパスワードやその他の設定情報を含んでいる）あなたのコードにアクセスされることを防ぎます。安全のためには良い方法です。

この例で Laravel を `/Users/JonSnow/Sites/MySite`ディレクトリーへインストールしたと考えてください。

MySite に対する最も基本的な Apache 仮想ホストの設定は、このようになるでしょう。

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

`/Users/JonSnow/Sites/MySite`へインストールされて、ドキュメントルートには `/Users/JonSnow/Sites/MySite/public`を指定していることに注目してください。

ドキュメントルートに公開 (`public`) フォルダーを指定するのは、よく行われるベストプラクティスです。これにより、ドキュメントルートはアップデートせず、Laravel を使用することもできるようになります。ドキュメントルートに関する別の手法が必要であれば、[Larave フォーラム<sup>6</sup>](http://forums.laravel.com/viewtopic.php?id=1258)でも見つけることができます。

## 3.4 基本設定

提供されている設定ファイルは全て `application` の下の `config` ディレクトリーに設置されています。どんなオプションが使用できるのか、基本的に理解するため、設定ファイルに目を通しておくことをお勧めします。`applicatoin/config/application.php` ファイルに基本的なアプリケーションに対するオプションが集まっているため、特に注目してください。

---

<sup>6</sup><http://forums.laravel.com/viewtopic.php?id=1258>

非常に重要なのは、サイトを公開する前に **application key** オプションを変更することです。このキーはフレームワーク全体を通じて暗号化やハッシュ、その他に利用されます。これは **config/application.php** ファイルの中にあり、でたらめな 32 文字の文字列を指定してください。標準準拠のアプリケーションキーは **Artisan** コマンドラインユーティリティーを使用し、自動的に生成できます。詳細は [Artisan コマンドインデックス](#) を御覧ください。

`mod_rewrite` を使用している場合は、`index` オプションに空文字列をセットしてください。

## 3.5 環境

ほとんどの場合、ローカル環境で指定するオプションと、実働サーバーで設定するオプションは異なっているでしょう。Laravel は URL ベースでデフォルトの環境を決めるメカニズムになっています。Laravel をインストールしたルートにある、`paths.php` を開いてください。次のような配列が存在しています。

```
$environments = array(

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

);
```

コードが示しているのは、Laravel は `localhost` で始まる URL、もしくは `.dev` で終わる URL であるならば、「ローカル」環境であると考えということです。

次に、`application/config/local` ディレクトリーを作成してください。このディレクトリーに置かれたファイルと、その中のオプションは、ベースとなる `application/config` ディレクトリーの中のオプションを置き換えます。例えば、`application.php` ファイルを新しい `local` 設定ディレクトリーに作成することができます。

```
return array(

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

);
```

この例では、ローカルの URL オプションは `application/config/application.php` 中の URL オプションをオーバーライドします。オーバーライドしたいオプションのみを指定すれば良いことに注意してください。

簡単ですね？もちろん、お望みの環境を自由に作ることができますよ。

## 3.6 クリーンURL

ほとんどの場合、アプリケーションの URL に”index.php” が含まれるのは、避けたいことでしょう。HTTP リライトルールを使用すれば、取り除けます。あなたが Apache をアプリケーションのサーバーとして利用しているのであれば、`mod_rewrite` を有効にし、`.htaccess`をあなたの **public**ディレクトリに確実に設置してください。

```
<IfModule mod_rewrite.c>
    RewriteEngine on

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

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

上の`.htaccess` ファイルが上手く動作しない？では、次を試してください。

```
Options +FollowSymLinks
RewriteEngine on

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

RewriteRule . index.php [L]
```

HTTP リライトの設定を済ましたら、`application/config/application.php`のなかにある、`index`設定オプションに空文字列をセットしてください。

注意：それぞれの Web サーバーは HTTP リライトに異なった方法を使っています。そのため、`.htaccess` ファイルに多少違った方法を取る必要があるでしょう。

## 4 ルーティング

### 4.1 基本

Laravel は PHP 5.3 の最新機能をルーティングをシンプルで記述的にするために使用しています。これは API から複雑な Web アプリケーションまで、全てをできる限り簡単に作成するために、重要なことです。通常ルートは **application/routes.php** で定義されます。

多くの他のフレームワークと異なり、Laravel は 2 つの方法でアプリケーションロジックを埋め込むことができます。とても一般的な方法ですがアプリケーションロジックをコントローラーに埋め込む事も、また **routes.php** に直接記述することもできます。これは特に数ページの小さなサイトにぴったりで、半ダースのメソッドを小さないくつかのコントローラーに書いたり、関連性の薄いメソッドをいくつかのコントローラーに詰め込んだりした後で、更にこうしたコントローラーヘルトを設定する必要はありません。

以降の例では、最初の引数は「登録する」ルートです。2 つ目の引数はそのルートで行うロジックを含む関数です。ルートには基本的に先頭の / は付けません。唯一の例外は、デフォルトルートの場合で、/ のみを指定します。

注意： ルートは登録された順番に評価されます。ですから、「全てに当てはまる」ルートは **routes.php** の最後に追加しましょう。

#### ” GET / ” に対応するルートを登録する

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

#### 全ての HTTP 変数 (GET、POST、PUT、DELETE) に対するルートを登録する

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

#### 他のリクエストメソッドに対するルートを登録する

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

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

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

一つの URI に複数の HTTP 変数を登録する

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

## 4.2 ワイルドカード

数字と一致する URI セグメント

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

英数字と一致する URI セグメント

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

残りの URI を制限なしに捉える

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

オプションの URI セグメント

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

## 4.3 404 イベント

アプリケーションにリクエストがあっても、どのルートにも一致しない場合は、404 イベントが発生します。デフォルトのイベントハンドラーは `application/routes.php` の中にあります。

### デフォルト 404 イベントハンドラー

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

あなたのアプリケーションにぴったりになるよう、自由に変更して下さい！

参照：

- [イベント](#)

## 4.4 フィルター

ルートフィルターは、あるルートが実行される前と後に実行されるものです。もし、“before” フィルターが値を返したら、その値はリクエストに対するレスポンスだと考え、そのルートは実行されません。これは認証フィルターなどを組み込み時に便利です。フィルターは基本的に `application/routes.php` で定義されます。

### フィルターを登録する

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

### ルートにフィルターを付ける

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

ルートに” a f t e r” フィルターを付ける

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

ルートに複数のフィルターを付ける

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

フィルターに引数を渡す

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

## 4.5 パターンフィルター

時々、特定の URI で始まるルート全部にフィルターを付けたいことがあるでしょう。例えば、”auth” フィルターを”admin” で始まる URI に対する全てのリクエストに適用したい場合です。どうやるのか、御覧ください。

フィルターを基にし、U R I パターンを定義する

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

配列を使用し、フィルター名とコールバックを URI と共に指定することで、直接フィルターを登録することも可能です。

フィルターとU R I パターンを同時に定義する



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

## 4.6 グローバルフィルター

Laravel には2つの「グローバル」フィルターが用意されており、**before**と**after**はアプリケーションに対する全てのリクエストで実行されます。両方共に **application/routes.php**の中で定義されています。これらのフィルターは共通のバンドルを開始したり、グローバルなアセットを追加するのに良い場所です。

注目：**after**フィルターは、現在のリクエストに対する **Response**オブジェクトを受け取ります。

## 4.7 ルートグループ

ルートグループは、コードをきれいにこぎつぱりと保ったまま、ルートのグループに対し、一連の属性を付け加えるために利用できます。

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

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

## 4.8 名前付きルート

いつも URL の生成やリダイレクトにルートの URI を使っていると、ルートを後で変更するときにトラブルが起き得ます。アプリケーション全体を通じて、ルートに付けた名前で参照するのは、便利な方法です。ルートの変更が発生しても、リンクは新しいルートを示しますので、それ以上の変更は必要ありません。

### 名前付きルートを登録する

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

### 名前付きルートに対する URL を生成する

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

### 名前付きルートへリダイレクトする

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

一度ルートに名前をつければ、現在のリクエストを処理しているルートが、名前を与えられたルートかどうか、簡単にチェックできます。

### リクエストを処理しているルートが、名前を与えられたものか判断する

```
if (Request::route()->is('home'))  
{  
    // "home" ルートは、リクエストを処理しています!  
}
```

## 4.9 HTTPS ルート

ルートを定義するときに、“https” アトリビュートを指定することで、そのルートに対する URL を生成したり、リダイレクトする時に HTTPS プロトコルを使用することができます。

### HTTPS ルートを定義する

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

### ”secure” ショートカットメソッドを使用する

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

## 4.10 バンドルルート

バンドルは Laravel のモジュールパッケージシステムです。アプリケーションに対するリクエストを簡単にバンドルに処理させるように設定することができます。[バンドルの詳細](#)については別のドキュメントを御覧ください。今のところは、このセクションを読み通して、バンドルの中で処理するようにルートを使えるだけでなく、バンドルの中でも登録できるという認識を持ていただければ結構です。

`application/bundles.php` ファイルを開き、なにか追加しましょう。

ルートを扱うバンドルを登録する

```
return array(

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

);
```

新しい **handles** オプションがバンドル設定配列にあるのに気づきましたか？これで Laravel に “admin” で始まる URI のリクエストは全て、Admin バンドルに行くように伝えています。

これで、バンドルにいくつかルートを登録する準備ができました。今度は、**routes.php** ファイルをあなたのバンドルのルートディレクトリーに作成し、以下のコードを付け加えてください。

バンドルのルート ( `root` ) ルートを登録する

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

この例を解説しましょう。( `:bundle` ) プレースホルダーに気が付きましたか？これはバンドルを登録するときに使用した、**handles** 節の値に置き換わります。これはコードを **D.R.Y.**<sup>1</sup> に保ち、あなたのバンドルを使用する人が、あなたの定義したルートを壊さずに、バンドルへのルート URI を変更できるようにしてくれます。ナイスでしょ？

もちろん、( `:bundle` ) プレースホルダーは、ルートルートだけでなく、ルート全てに使用できます。

バンドルのルートを登録する

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Don't\\_repeat\\_yourself](http://en.wikipedia.org/wiki/Don't_repeat_yourself)

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

## 4.11 コントローラールーティング

コントローラーはアプリケーションロジックを管理する別の方法です。もしコントローラーに不慣れでしたら、先に[コントローラーについて](#)読み、このセクションに戻ってきてください。

Laravel の全てのルートについて認識しておくべき重要なことは、コントローラーへのルートも含め、明確に定義されている必要があることです。これが意味するのは、ルート登録がされていないコントローラーメソッドに対してはアクセスできないということです。コントローラーの中のメソッドは、コントローラールート登録を使用すれば、自動的に全て定義することができます。コントローラールート登録は基本的に **application/routes.php** の中で定義します。

大抵の場合、アプリケーション中の”controllers” ディレクトリーに存在する、全てのコントローラーを一度に登録したいことでしょう。たった一文で出来ますよ。御覧ください。

### アプリケーションの全てのコントローラーに登録する

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

**Controller::detect**メソッドは、シンプルにアプリケーションで定義されているコントローラーを全て配列で返します。

もし、バンドルに含まれるコントローラーを自動的に突き止めたい時は、バンドル名をメソッドに渡すだけです。バンドル名が指定されない時は、**application** フォルダの **controllers** ディレクトリーが検索されます。

注意：このメソッドではコントローラーをロードする順番をコントロールできないことに注意してください。**Controller::detect()** はとても小さなサイトに対してのみ使用するべきでしょう。明確にアドバイスするなら、「手動」でコントローラーのルーティングを記述することは、よりルーティングをコントロールし、より読みやすくなるということです。

### ” a d m i n ” バンドルのコントローラーを全て登録する

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

### ルーターに” h o m e ” コントローラーに登録する

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

### ルーターに複数のコントローラーを登録する

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

一度コントローラーを登録すれば、メソッドにはシンプルな URI 規約でアクセスできます。

`http://localhost/コントローラー/メソッド/引数`

この規約は Codeigniter や他の人気のあるフレームワークで採用されているものと似ており、最初の引数がコントローラー、2 番目がメソッド、残りのセグメントはメソッドの引数として渡されます。もしメソッドセグメントがなければ、"index" メソッドがつかわれます。

このルーティング規約はすべての状況で好ましくは無いでしょう。そこでシンプルで直感的なシンタックスを使い、URI をコントローラーアクションに明確にルートすることもできます。

### コントローラーアクションを指定してルートを登録する

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

### コントローラーアクションを指定し、フィルターも使ったルートを登録する

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

### コントローラーアクションを示す名前付きルートを登録

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

## 4.12 CLI ルートテスト

Laravel の"Artisan"CLI を使い、ルートをテストすることができます。シンプルに使用したいリクエストメソッドと URI を指定してください。ルートのレスポンスが CLI で `var_dump` されます。

### Artisan CLI を通して、ルートを呼び出す

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

# 5 コントローラー

## 5.1 基本

コントローラーはユーザーの入力を受け取り、モデル、ライブラリー、ビュー間の相互関係を管理する責任を受け持つクラスのことです。典型的な動作は、モデルからデータを受け取り、それからユーザーに対しデータを表示するためにビューに返します。

現代的な Web 開発において、コントローラーはアプリケーションロジックを実現する最も一般的な方法として使用されています。しかしながら、Laravel は開発者にアプリケーションロジックをルーティングに含めて実装することも許しています。この詳細は[ルーティングのドキュメント](#)を御覧ください。新しいユーザーはコントローラーから始められるほうが良いでしょう。コントローラーでできないことは、ルートベースのアプリケーションロジックでもできません。

コントローラークラスは `application/controllers` に置かれ、`Base_Controller` クラスを拡張しなくてはなりません。コントローラークラスは Laravel により、インクルードされます。

### シンプルなコントローラーを作成する

```
class Admin_Controller extends Base_Controller
{

    public function action_index()
    {
        //
    }

}
```

アクションは Web からアクセスできるコントローラーメソッドの名前です。アクションは”action\_“で始まる名前を付けなくてはなりません。他のメソッドは、スコープにかかわらず、Web からアクセス出来ません。

**Base\_Controller** クラスは **Laravel** のメイン **Controller** クラスを拡張しており、多くのコントローラーに共通のメソッドを使いやすいように用意されています。

## 5.2 コントローラールーティング

コントローラーへのルートも含めて、Laravel の全てのルートは、明確に定義される必要があると認識するのは重要です。

これが意味するのは、ルート登録がされていないコントローラーメソッドに対してはアクセスできないということです。コントローラーの中のメソッドは、コントローラールート登録を使用すれば、自動的に全て定義することができます。コントローラールートの登録は基本的に **application/routes.php** の中で定義します。

コントローラーのルーティングについての詳細は[ルーティングページ](#)を参照してください。

## 5.3 バンドルコントローラー

バンドルは Laravel のモジュールパッケージシステムです。アプリケーションに対するリクエストを簡単にバンドルに処理させるように設定することができます。[バンドルの詳細](#)については別のドキュメントを御覧ください。

バンドルに属するコントローラーを作成するのは、アプリケーションのコントローラーを作成するのと同じです。コントローラーのクラス名の前にバンドル名をつけるだけです。もしバンドル名が”admin”であれば、コントローラーのクラスはこのようになります：

### バンドルコントローラークラスの作成

```
class Admin_Home_Controller extends Base_Controller
{

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

}
```

けど、ルーターにどうやってバンドルコントローラーを登録するのでしょうか？とても簡単です。ご覧のとおりです：

### ルーターにバンドルのコントローラーを登録する

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

素晴らしい！これで Web から”admin”バンドルの home コントローラーへアクセスできます！

注目：Laravel 全体を通して、連続するコロンの (::) はバンドルを意味します。バンドルに関するより多くの情報は[バンドルドキュメント](#)を御覧ください。

## 5.4 アクションフィルター

アクションフィルターはコントローラーアクションの前と後に実行できるメソッドです。Laravel では、フィルターをアクションに結び付けられるだけではありません。HTTP 変数 (post, get, put, delete) を選択して、フィルターを有効にさえできます。

“before” と “after” フィルターをコントローラーのコンストラクターの中で、コントローラーアクションと結びつけることも可能です。

### フィルターをすべてのアクションに結びつける

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

この例では、“auth” フィルターはこのコントローラーの中のすべてのアクションが行われる前に、実行されます。auth アクションは Laravel に備わっており、**application/routes.php**の中で見つかります。auth フィルターはそのユーザーがログインしていることを確認し、していない場合は“login”ヘリダイレクトします。

### いくつかのアクションにだけフィルターを結びつける

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

この例では、auth フィルターは `action_index()` と `action_list()` メソッドが実行される前に行われます。これらのページにアクセスする前に、ユーザーはログインしてなくてはなりません。しかしながら、このコントローラー中の他のアクションでは、認証されたセッションは要求されません。

### いくつかのアクションを除いた全てに、フィルターを結びつける

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

以前の例と同様、この宣言により、このコントローラーのアクションが実行される前に、フィルターが確実に実行されます。フィルターを適用するアクションを宣言する代わりに、認証セッションを要求しないアクションを宣言しています。場合によって“except”メソッドを使用したほうが安全なこともあります。新しいアクションをこのコントローラーに追加した時に、`only()` に指定し忘れる可能性があるからです。これにより認証されていないユーザーを意図せずコントローラーのアクションにアクセスさせてしまう可能性があります。

### POST に対してフィルターを結びつける

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

この例は、どうやって特定の HTTP 変数にだけフィルターを実行するかを示しています。この場合、CSRF フィルターをフォームがポストされた場合にのみ実行します。CSRF フィルターは他のシステムからのポスト（例えばボットなど）を防ぐように設計されており、Laravel にはデフォルトで用意されています。CSRF フィルターは **application/routes.php**の中で見つかります。

参照：



- ルートフィルター

## 5.5 コントローラーのネスト

コントローラーはメインの **application/controllers** フォルダーの下に、好きなだけのサブフォルダーを作成し、その中に置くこともできます。

コントローラークラスを作成し、**controllers/admin/panel.php** として設置します。

```
class Admin_Panel_Controller extends Base_Controller
{

    public function action_index()
    {
        //
    }

}
```

ネストしたコントローラーは「ピリオド」を使いルートに登録します。

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

ネストしたコントローラーを使う場合、優先順位を考慮し、いつもネストが深いものから浅いものの順に登録してください。

コントローラーの” `index` ”アクションにアクセスする

`http://localhost/admin/panel`

## 5.6 コントローラーレイアウト

コントローラーを使用するレイアウトについては、完全なドキュメントが、[テンプレートのページ](#)で見つけられます。

## 5.7 RESTコントローラー

コントローラーのアクションを”`action_`”で始める代わりに、対応させたい HTTP 変数名を付けることもできます。

## RESTフルプロパティをコントローラーに付け加える

```
class Home_Controller extends Base_Controller
{

    public $restful = true;

}
```

## RESTフルコントローラーアクションを作成する

```
class Home_Controller extends Base_Controller
{

    public $restful = true;

    public function get_index()
    {
        //
    }

    public function post_index()
    {
        //
    }

}
```

これは特に CRUD メソッドをフォームの生成と表示、バリデーションと結果の保存のロジックに分けて作成する場合に便利です。

## 5.8 依存の注入

もしあなたがテストしやすいコードを書くことに焦点を当てていたのでしたら、多分コントローラーのコンストラクターに依存性を注入したいことでしょう。問題ありません。コントローラーを**IoC コンテナ**に登録してください。コンテナにコントローラーに登録するときには、キーのプレフィックスに**controller**を付けてください。では、**application/start.php**ファイルの中で、コントローラーに登録してみましょう。こうなります：

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

アプリケーションでコントローラーに対するリクエストがあると、Laravel は自動的にそのコントローラーがコンテナに登録されているか調べ、登録されているならば、コントローラーのインスタンスを解決するためにコンテナを利用します。

コントローラーの依存性の注入に飛び込む前に、Laravel の美しい[IoC コンテナ](#)ドキュメントをお読みになりたいでしょう。

## 5.9 コントローラーファクトリー

例えばサードパーティの IoC コンテナなどを利用し、もっと自分でコントローラーのインスタンス化をコントロールしたければ、Laravel のコントローラーファクトリーを使う必要があります。

コントローラーのインスタンス化を処理するためにイベントを登録する

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

イベントは解決すべきコントローラのクラス名を受け取ります。必要なことはそのコントローラーのインスタンスを返してあげることが全てです。

## 6 モデルとライブラリー

### 6.1 モデル

モデルはあなたのアプリケーションの心臓です。アプリケーションロジック（コントローラー／ルート）とビュー（HTML）はユーザーとモデルとを関係付ける媒体にすぎません。モデルの中に組み込む、一番典型的なロジックは**ビジネスロジック**<sup>1</sup>でしょう。

モデルに入れ込む機能の例をご覧ください。

- データベース I/O
- ファイル I/O
- Web サービスとのやり取り

例えば、あなたがブログをプログラムしているとします。あなたは多分”Post” モデルを作りたいがるでしょう。ユーザーはポストにコメントしたがるでしょうから、”Comment” モデルも作ることでしょう。もしユーザーにコメントさせるのであれば、”User” モデルも必要になるでしょう。分かりましたか？

### 6.2 ライブラリー

ライブラリーとは、あなたのアプリケーションのためだけに仕事をするわけではないクラスのことです。例えば、HTML からコンバートしてくれる PDF 生成ライブラリーを考えてください。それは複雑でしょうが、あなたのアプリケーションのためだけに役立つわけでありません。ですから、「ライブラリー」にすることを考えてください。

ライブラリーを作成するのは、クラスをすることと同じように簡単で、`libraries` フォルダーの中に保存します。次の例として、渡されたテキストをエコーするメソッドを持つ、シンプルなライブラリーを作成してみましょう。`libraries` フォルダーの中の **printer.php** ファイルを作成し、次のコードを書いてください。

```
<?php
```

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

これであなたはアプリケーションのどんな所からも、`Printer::write('write メソッドからこのテキストはエコーされている！')` と呼び出せるようになりました。

---

<sup>1</sup><http://ja.wikipedia.org/wiki/%E3%83%93%E3%82%B8%E3%83%8D%E3%82%B9%E3%83%AD%E3%82%B8%E3%83%83%E3%82%AF>

## 6.3 オートロード

ライブラリーとモデルは、とても簡単に使用できるので、Laravel のオートローダーに感謝しています。オートローダーについてもっと知るためには[オートローダー](#)をチェックしてください。

## 6.4 ベストプラクティス

私達の頭の中には「コントローラーは簡単に!」というマントラが流れています。けど、どうやって実際の世界に適用すればいいのでしょうか？これは「モデル」という言葉が含む、問題の一部になります。一体何を意味しているのでしょうか？ただの便利な言葉なのでしょうか？「データベース」に関連付けられた多くの「モデル」が発生させるのは、データベースにアクセスするだけの軽いモデルと、でっぴり太ったコントローラーです。別の手法を取ってみましょう

“models” ディレクトリーが完全にスクラップになったからって、どうだって言うんですか？もっと便利な名前を付けましょう。実際、アプリケーションと同じ名前をつけましょう。多分衛星トラッキングサイトでしたら、名前は”Trackler” になるでしょう。それならば、application フォルダーの下に”trackler” ディレクトリーを作りましょう。

いいですね！次に、クラスを実体”entities”、サービス”services”、リポジトリ”repositories”に分割しましょう。では、”trackler” フォルダーの下に、これら3つのディレクトリーを作成することにしましょう。それぞれを見て行きましょう。

### Entities (実体)

entities はアプリケーションのデータコンテナだと考えてください。主としてプロパティのみで構成されています。ですから、アプリケーションに”Location”（位置）実体があるならば、”latitude”（経度）と”longitude”（緯度）のプロパティを持ちます。このようなコードになるでしょう：

```
<?php namespace Trackler\Entities;

class Location {

    public $latitude;
    public $longitude;

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

}
```

見た感じ、良いようです。これで、実体がひとつ出来ました。他の2つのフォルダーを見て行きましょう。

## S e r v i c e s (サービス)

services はアプリケーションの処理で構成されます。このまま、Trackler サンプルを使って説明していきます。このアプリケーションでは、ユーザーが GPS ロケーションを入力するフォームを持っているとします。ですから、座標が正しいフォーマットであるかバリデーションする必要があります。*location* 実体をバリデーションする必要があります。では、”services” ディレクトリーの中に、”validators” フォルダーを作成し、次のクラスを書いてください。

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

use Trackler\Entities\Location;

class Location_Validator {

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

}
```

いいですね！これでコントローラーとルートとは独立して、バリデーションをテストする素晴らしい方法が手に入りました！*location* のバリデーションは済みました。これで保存する準備ができました。では、どう行いましょうか？

## R e p o s i t o r i e s (リポジトリ)

repositories はアプリケーションのデータアクセス層です。あなたのアプリケーションの実体 (*entities*) を保存したり、入手することに責任を持ちます。引き続いて、この例では場所 (*location*) 実体を使って行きましょう。実体を保存する場所リポジトリが必要です。リレーショナルデータベースであろうと、Redis であろうと、次世代のホットなストレージだろうと、なんでも好きなメカニズムを使うことができます。例を見てみましょう。

```
<?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...
    }

}
```

これで、アプリケーションの実体、サービス、リポジトリ間の関係をきれいに分割できました。これが意味するのは、スタブリポジトリをサービスやコントローラーに注入することで、こうしたアプリケーションの一部分をデータベースとは独立してテストできるということです。しかも、サービスや実体、コントローラーには影響をあたえること無く、保存テクノロジーを変更できるようになります。これで、良い関係の分離を達成できました。

参照：

- [IoC コンテナ](#)

# 7 ビューとレスポンス

## 7.1 基本

ビューはアプリケーションを使用している人に送られる HTML で構成されています。アプリケーションのビジネスロジックから、ビューを分けることで、コードは綺麗になり、メンテナンスしやすくなります。

すべてのビューは **application/views** ディレクトリーの中に設置され、PHP ファイル拡張子を付けます。**View** クラスはビューを取得するシンプルな方法を提供します。それを使用者にリターンします。例を見てみましょう！

### ビューを作成する

```
<html>
    私はviews/home/index.phpに保存されている！
</html>
```

### ルートからビューを返す

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

### コントローラーからビューを返す

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

### ビューが存在するか確かめる

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

時々、ブラウザーに送るレスポンスをもうちょっとコントロールする必要があることでしょう。例えば、レスポンスにカスタムヘッダーをセットしたいとか、HTTP ステータスコードを変えたいとかです。こうやります：

### カスタムレスポンスを返す



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

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

データを渡し、ビューを含めたカスタムレスポンスを返す

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

J S O N レスポンスを返す

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

J S O N P レスポンスを返す

```
return Response::jsonp('myCallback', array('name' => 'Batman'));
```

E l o q u e n t モデルを J S O N で返す

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

## 7.2 ビューとデータの結合

典型的には、ルートかコントローラーは表示するビューのデータをモデルへリクエストします。ですから、データをビューに渡す方法が必要になります。やり方は色々ありますので、自分にベストな方法を選んでください！

ビューにデータを結びつける

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

結びつけたデータをビューの中でアクセスする

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

ビューにチェーンでデータを結びつける

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

データーを結びつけるために、配列を渡す

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

データーを結びつけるために、マジックメソッドを使う

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

データーを結びつけるために、配列アクセスインターフェイス方式を使用する。

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

## 7.3 ビューのネスト

時々、ビューの中からビューをネストしたい場合があるでしょう。ネストされたビューは時々「パーシャル」と呼ばれますが、ビューを小さく、モジュール分割するのに役に立ちます。

” `nest` ” メソッドを使い、ネストしたビューを結びつける

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

ネストしたビューにデーターを渡す

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

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

時々、ビューの中から直接他のビューを取り込みたいことがあるでしょう。**render**ヘルパー機能が使えます。

ビューを表示するのに” `render` ” ヘルパーを使用する

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

またこれもよくありますが、リスト中のデーターのインスタンスに応じて、パーシャルビューを表示することもあります。例えば、一つの注文に応じた詳細をパーシャルビューで表示する場合です。また別の例としては、注文の配列をループで処理し、それぞれのオーダーをパーシャルビューでレンダリングする場合もあるでしょう。このような場合は、シンプルに **render\_each** ヘルパーを使ってください。

配列の中のアイテムをそれぞれパーシャルビューでレンダリングする

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

最初の引数はパーシャルビューの名前で、2つ目はデータの配列です。3つ目は配列のアイテムそれぞれが、パーシャルビューに渡される時に参照される変数の名前です。

## 7.4 名前付きビュー

名前付きビューはコードを表現的で組織立てるのに、役立ちます。シンプルに使ってみましょう。

名前付きビューの登録

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

名前付きビューのインスタンスを得る

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

データを名前付きビューと結びつける

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

## 7.5 ビューコンポーサー

ビューが生成されるたびに”composer” イベントが発生します。このイベントを `listen` し、ビューが生成されるたびに資源や共通データをビューに結びつけることができます。一般的によくつかわれる機能としては、ブログのサイドナビにあるランダムポストのリストがあげられるでしょう。レイアウトビューの中でロードすることで、パーシャルビューをネストすることができます。それから、パーシャルに対し、コンポーサーを定義します。それから、コンポーサーはポストテーブルをクエリーし、ビューをレンダリングするのに必要なデータを全て集めます。ランダムロジックをあちこちにばらまかなくて済みます！一般的にコンポーサーは `application/routes.php` の中で定義されます。サンプルをどうぞ：

” `home` ” ビューにコンポーサーを登録する

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

これで”home”ビューが生成されるたびに、登録した無名関数に View のインスタンスが渡され、あなたがやりたいことがなれ、ビューに対し準備ができます。

### コンポーサーを複数のビューに対し登録する

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

注目：ビューは一つ以上のコンポーサーを持てます。使いまくってください！

## 7.6 リダイレクト

重要な注意点は、ルートでもコントローラーでも、”return” 文でレスポンスを返さなくてはならないことです。ユーザーをリダイレクトしたい場所で”Redirect::to()” を呼ぶ代わりに、”return Redirect::to()” を使いましょう。これは他の PHP フレームワークと一番異なっているという点でとても重要です。この実務的な重要点は、思いがけず簡単に見落とされてしまいます。

### 他のURLへリダイレクトする

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

### 特定のステータスでリダイレクトする

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

### セキュアなURLへリダイレクトする

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

### アプリケーションのルートへリダイレクトする

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

以前のアクションへリダイレクトする

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

名前付きルートへリダイレクトする

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

コントローラーアクションへリダイレクトする

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

時々、名前付きルートへリダイレクトする必要があるが、URI のワイルドカードの代わりに、特定の値を使いたい場合があることでしょう。ワイルドカードを特定の値へ簡単に置き換えられます。

ワイルドカード値を指定して、名前付きルートへリダイレクトする

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

ワイルドカード値を指定して、アクションへリダイレクトする

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

## 7.7 フラッシュデータと共にリダイレクト

アプリケーションにユーザー登録した後や、ログイン後に、ウェルカムもしくはステータスメッセージを表示するのは一般的です。しかし、どうやって次のリクエストのステータスメッセージをセットできるのでしょうか？リダイレクトのレスポンスに `with()` メソッドを使い、データをフラッシュ保存できます。

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

ビューからのメッセージは `Session` の `get` メソッドでアクセスできます。

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

参照：

- [セッション](#)

## 7.8 ダウンロード

ファイルダウンロードのレスポンスを送る

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

ファイル名を指定し、ダウンロードレスポンスを送る

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

## 7.9 エラー

特定のエラーレスポンスを生成するには、シンプルに返したいレスポンスコードを指定してください。  
**views/error**の中に保存されている、対応したビューが自動的にリターンされます。

4 0 4 エラーレスポンスを生成する

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

5 0 0 エラーレスポンスを生成する

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

## 8 アセットの管理

### 8.1 アセットを登録する

**Asset**クラスは CSS と Javascript をアプリケーションで使用する簡単な方法を提供します。アセットを登録するには、**Asset**クラスの **add**メソッドを呼び出すだけです。

アセットを登録する

```
Asset::add('jquery', 'js/jquery.js');
```

**add**メソッドは3つの引数を取ります。最初はアセットの名前で、2つ目はそのアセットの **public**ディレクトリーからの相対パスです。3つ目はアセットの依存リストです。（詳細は後ほど）登録するのが Javascript なのか css なのかをメソッドに伝えていないことに注目してください。**add**メソッドはファイルの拡張子から、登録するファイルのタイプを決定します。

### 8.2 アセットをダンプする

登録済みアセットのリンクをビューに表示する準備が済んだら、**styles**と **scripts**メソッドが使用できます。

ビューの中にアセットをダンプする

```
<head>
    <?php echo Asset::styles(); ?>
    <?php echo Asset::scripts(); ?>
</head>
```

### 8.3 アセットの依存

時々、アセットは他のアセットに依存していることを指定する必要があるでしょう。つまりビューの中で、あるアセットを宣言する前に、他のアセットを宣言しておく必要がある場合です。Laravel のアセット依存性管理は、これ以上簡単にできないほど簡単です。あなたがアセットに付けた「名前」を覚えていますか？**add**メソッドの第3引数に依存性の宣言を渡すことができます。

依存関係のあるアセットを登録する

```
Asset::add('jquery-ui', 'js/jquery-ui.js', 'jquery');
```

この例では、**jquery-ui**アセットを登録し、それと同時に **jquery**アセットに依存している指定を行っています。これで、ビューの中でアセットのリンクを置くときに、jQuery UI アセットの前にいつも、jQuery アセットが宣言されます。依存しているアセットは二つ以上ある？大丈夫です：

複数の依存関係があるアセットを登録する

```
Asset::add('jquery-ui', 'js/jquery-ui.js', array('first', 'second'));
```

## 8.4 アセットのコンテナ

レスポンスを早くするために、Javascript を HTML ドキュメントの最後に置くのは常識です。しかし、いくつかのアセットをドキュメントの head に置く必要がある時はどうしましょう？問題ありません。Asset クラスはアセットコンテナで簡単に管理できる方法を提供しています。Asset クラスの **container**メソッドを呼び出し、コンテナ名をつけてください。一度コンテナのインスタンスを作れば、今までと同じシンタックスを用いて、コンテナにアセットを自由に追加できます。

アセットコンテナを取得する

```
Asset::container('footer')->add('example', 'js/example.js');
```

コンテナに与えられたアセットをダンプする

```
echo Asset::container('footer')->scripts();
```

## 8.5 バンドルのアセット

バンドルアセットを便利に追加したり、ダンプする前に、**バンドルアセットを生成し、公開する**ドキュメントを読みたいかも知れませんね。

アセットを登録する時、通常パスは **public**ディレクトリーからの相対パスになります。しかしながら、これではバンドルアセットを取り扱うときに不便です。それらは **public/bundles**ディレクトリーの中にあるからです。でも、思い出してください。Laravel は人生をより簡単にするために存在していることを。ですから、シンプルに管理しているアセットコンテナにバンドルを指定してください。

管理しているアセットコンテナにバンドルを指定する

```
Asset::container('foo')->bundle('admin');
```

これでアセットを追加すれば、バンドルの **public** ディレクトリーへの相対パスで使用できます。Laravel は自動的に正しいフルパスを生成します。



## 9 テンプレート

### 9.1 基本

多分、あなたのアプリケーションでもほとんどのページに渡って、共通のレイアウトを使用していることでしょう。このレイアウトを手動で、全てのコントローラーアクションに生成するのは、辛いですね。コントローラーにレイアウトが指定出来れば、開発はもっと楽になります。では、これを行ってみましょう。

コントローラーに” `layout` ” プロパティを指定する

```
class Base_Controller extends Controller {  
  
    public $layout = 'layouts.common';  
  
}
```

コントローラーのアクションからレイアウトにアクセスする

```
public function action_profile()  
{  
    $this->layout->nest('content', 'user.profile');  
}
```

注目：レイアウトを使う場合、アクションは何もリターンしません。

### 9.2 セクション

ビューのセクションはネストしたビューからレイアウトにコンテンツを挿入するシンプルな方法を提供します。例えば、多分あなたはレイアウトのヘッダーの中にネストビューが必要としている Javascript を挿入したいとします。これを掘り下げてみましょう。

ビューの中にセクションを生成する

```
<?php Section::start('scripts'); ?>
    <script src="jquery.js"></script>
<?php Section::stop(); ?>
```

## セクションの内容をレンダリングする

```
<head>
    <?php echo Section::yield('scripts'); ?>
</head>
```

## B l a d e のショートカットを使いセクション操作する

```
@section('scripts')
    <script src="jquery.js"></script>
@endsection

<head>
    @yield('scripts')
</head>
```

## 9.3 B l a d e テンプレートエンジン

Blade はビューを書くことを至高の喜びにしてくれます。Blade ビューを作成するには、ファイルの拡張子を”.blade.php”にするだけです。Blade により、美しく控えめなシンタックスで、PHP コントロール構文やデーターのエコーを書くことができるようになります。例をご覧ください。

### B l a d e を使い、変数をエコーする

```
Hello, {{ $name }}.
```

### B l a d e を使い、関数の結果をエコーする

```
{{ Asset::styles() }}
```

### ビューをレンダーする

**@include**を使用し、他のビューの中にビューをレンダーすることができます。レンダーされるビューは自動的に、現在のビューの全てのデーターを継承します。

```
<h1>Profile</h1>
@include('user.profile')
```

同様に、**@include**と同じような働きをする **@render**も使用できます。違いはレンダー時に、現在のビューのデーターを継承しないことです。

```
@render('admin.list')
```

## Blade コメント

```
{{-- これがコメントです --}}
```

```
{{--
    これは
    複数行に渡る
    コメント例です。
--}}
```

注目 : Blade のコメントは、HTML コメントとは異なり、HTML ソースには出力されません。

## Blade コントロール構文

### For ループ :

```
@for ($i = 0; $i <= count($comments); $i++)
    コメントの内容は {{ $comments[$i] }}
@endfor
```

### Foreach ループ :

```
@foreach ($comments as $comment)
    コメントの内容は {{ $comment->body }}.
@endforeach
```

### While ループ :

```
@while ($something)
    まだループ中です !
@endwhile
```

### If 文 :

```
@if ( $message == true )
    メッセージを出力中 !
@endif
```

### If Else 文 :

```
@if (count($comments) > 0)
    コメントがあります！
@else
    コメントがありません！
@endif
```

### E l s e l f 文 :

```
@if ( $message == 'success' )
    成功した！
@elseif ( $message == 'error' )
    エラーが起きた。
@else
    ここに来るのかな？
@endif
```

### F o r E l s e 文 :

```
@forelse ($posts as $post)
    {{ $post->body }}
@empty
    配列中にはポストはありません！
@endforelse
```

### U n l e s s 文 :

```
@unless(Auth::check())
    Login
@endunless
```

// 同じ内容…

```
<?php if ( ! Auth::check()): ?>
    Login
<?php endif; ?>
```

## 9.4 B l a d e レイアウト

Blade はきれいでエレガントなシンタックスを PHP の一般的なコントロール構文に提供しているだけでなく、ビューのレイアウトに使用できる、美しい手法も用意しています。例えば、あなたのアプリケーションでは、共通のルック・アンド・フィールを提供するために、「マスター」ビューを使っているでしょう。それは多分、こんな感じだと思います :

```
<html>
    <ul class="navigation">
        @section('navigation')
            <li>Example Item 1</li>
            <li>Example Item 2</li>
        @endsection
    </ul>

    <div class="content">
        @yield('content')
    </div>
</html>
```

“content” セクションが生成されることに注目してください。このセクションに何かテキストを埋めるなくてはなりません。では、このレイアウトを使用する、別のビューを作成しましょう。

```
@layout('master')

@section('content')
    profileページへようこそ！
@endsection
```

素晴らしい！これで、ルートからシンプルに”profile” ビューをリターンできます。

```
return View::make('profile');
```

profile ビューはありがたいことに、**@layout**文により、Laravel は”master” テンプレートを自動的に使用してくれます。

重要： **@layout**はファイルの最初の一行で呼び出す必要があり、先頭にホワイトスペースをつけたり、途中で改行してはいけません。

## @ p a r e n t で追加する

場合により、セクションのレイアウトを置き換えてしまうよりは、追加したいこともあります。例えば、”master” レイアウトのナビゲーションリストを考えてください。ここに、新しいアイテムを追加してみましょう。こんな風になります：

```
@layout('master')

@section('navigation')
    @parent
    <li>Nav Item 3</li>
@endsection

@section('content')
    profileページへようこそ！
@endsection
```

**@parent**はレイアウトの *navigation*セクションの内容と置き換わります。これはレイアウトの拡張と継承を実現する美しくてパワフルな手法を提供しています。

# 10 ペジネーション

## 10.1 基本

Laravel のペジネーションは、導入するためにコードが取っ散らかるのを減らせるように、設計されています。

## 10.2 クエリービルダーを使用する

Fluent クエリービルダーを用い、完全な例を紹介しましょう。

クエリーの結果をページングする

```
$orders = DB::table('orders')->paginate($per_page);
```

クエリーのテーブルカラムを選択するためにオプションで配列を渡し、指定することもできます。

```
$orders = DB::table('orders')->paginate($per_page, array('id', 'name', 'created_at'));
```

ビューで結果を表示する

```
<?php foreach ($orders->results as $order): ?>
    <?php echo $order->id; ?>
<?php endforeach; ?>
```

ペジネーションリンクを生成する

```
<?php echo $orders->links(); ?>
```

links メソッドは、次のような賢いページのリンクを生成します。

前 1 2 ... 24 25 26 27 28 29 30 ... 78 79 次

ペジネーションはどのページを表示しているかを自動的に決め、結果とリンクをそれに合わせて変更してくれます。

また、「次」、「前」のリンクを生成することも可能です。

シンプルに「前」と「次」のリンクだけを生成する

```
<?php echo $orders->previous(). ' '. $orders->next(); ?>
```

参照：

- *Fluent* クエリービルダー

## 10.3 ペジネーションリンクに追加する

ペジネーションのリンクにソートしているカラム名などをクエリに含める必要があるかも知れません。

ペジネーションリンクにクエリを追加する

```
<?php echo $orders->appends(array('sort' => 'votes'))->links();
```

これで次のような URL が生成されます。

```
http://example.com/something?page=2&sort=votes
```

## 10.4 P a g i n a t o r s を手動で生成する

時にクエリビルダーを使用せず、Paginator インスタンスを手動で作成する必要があることもあるでしょう。こうしてください：

P a g i n a t o r インスタンスを手動で生成する

```
$orders = Paginator::make($orders, $total, $per_page);
```

## 10.5 ペジネーションスタイル

全てのペジネーションリンク要素は、CSS クラスが使用できます。links メソッドで生成される HTML 要素の一例をご覧ください：

```
<div class="pagination">
    <ul>
        <li class="previous_page"><a href="foo">前</a></li>

        <li><a href="foo">1</a></li>
        <li><a href="foo">2</a></li>

        <li class="dots disabled"><a href="#">...</a></li>
```



```
<li><a href="foo">11</a></li>
<li><a href="foo">12</a></li>

<li class="active"><a href="#">13</li>

<li><a href="foo">14</a></li>
<li><a href="foo">15</a></li>

<li class="dots disabled"><a href="#">...</a></li>

<li><a href="foo">25</a></li>
<li><a href="foo">26</a></li>

<li class="next_page"><a href="foo">次</a></li>
</ul>
</div>
```

結果の最初のページでは、「前」リンクは無効になります。同様に、「次」リンクは結果の最終ページでは、無効になります。生成される HTML はこのようになります。

```
<li class="disabled previous_page"><a href="#">前</a></li>
```

# 11 HTML の作成

## 11.1 HTML エンティティ

ユーザーのインプットをビューに表示するときには、HTML 中の重要なすべての文字を HTML エンティティ表現に変換することは重要です。

例えば、< シンボルは必ずエンティティ表現に変換します。HTML キャラクターをエンティティ表現にコンバートすることは、クロスサイトスクリプティングより、あなたのアプリケーションを守る手助けになります。

文字列を HTML エンティティ表現にコンバートする

```
echo HTML::entities('<script>alert(\'hi\');</script>');
```

グローバルヘルパーの ” e ” を使う

```
echo e('<script>alert(\'hi\');</script>');
```

## 11.2 スクリプトとスタイルシート

J a v a s c r i p t ファイルへの参照を生成する

```
echo HTML::script('js/scrollTo.js');
```

C S S ファイルへの参照を生成する

```
echo HTML::style('css/common.css');
```

メディアタイプを指定し、C S S ファイルへの参照を生成する

```
echo HTML::style('css/common.css', array('media' => 'print'));
```

参照：

- [アセットの管理](#)

## 11.3 リンク

U R L からリンクを生成する

```
echo HTML::link('user/profile', 'User Profile');
```

HTTPSを使用し、リンクを生成する

```
echo HTML::link_to_secure('user/profile', 'User Profile');
```

追加のHTML属性を指定し、リンクを生成する

```
echo HTML::link('user/profile', 'User Profile', array('id' => 'profile_link\'));
```

## 11.4 名前付きルートへのリンク

名前付きルートへリンクを生成する

```
echo HTML::link_to_route('profile');
```

ワイルドカード値と共に、名前付きルートへリンクを生成する

```
$url = HTML::link_to_route('profile', 'User Profile', array($username));
```

参照：

- [名前付きルート](#)

## 11.5 コントローラーアクションへのリンク

コントローラーアクションへリンクを生成する

```
echo HTML::link_to_action('home@index');
```

ワイルドカード値と共に、コントローラーアクションへリンクを生成する

```
echo HTML::link_to_action('user@profile', 'User Profile', array($username))\;
```

## 11.6 多言語へのリンク

同じページの他言語ページへのリンクを生成

```
echo HTML::link_to_language('fr');
```

ホームページを多言語で表示するリンクを生成

```
echo HTML::link_to_language('fr', true);
```

## 11.7 Mailtoリンク

ボットをあしらうために、HTML クラスの”mailto” メソッドは与えられたメールアドレスを分かりづらくします。

Mailtoリンクを生成する

```
echo HTML::mailto('example@gmail.com', 'E-Mail Me!');
```

メールアドレスをリンクテキストとして生成する

```
echo HTML::mailto('example@gmail.com');
```

## 11.8 画像

HTML イメージタグを生成する

```
echo HTML::image('img/smile.jpg', $alt_text);
```

追加のHTML 属性と共に、HTML イメージタグを生成する

```
echo HTML::image('img/smile.jpg', $alt_text, array('id' => 'smile'));
```

## 11.9 リスト

配列のアイテムからリストを生成する

```
echo HTML::ol(array('Get Peanut Butter', 'Get Chocolate', 'Feast'));
```

```
echo HTML::ul(array('Ubuntu', 'Snow Leopard', 'Windows'));
```

```
echo HTML::dl(array('Ubuntu' => 'An operating system by Canonical', 'Windows' => 'An operating system by Microsoft'));
```

## 11.10 カスタムマクロ

HTML クラスヘルパー、“macros”を使い、簡単に自分自身のカスタムマクロを定義できます。実例をどうぞ。最初に名前と無名関数でマクロを登録します。

### HTML マクロを登録する

```
HTML::macro('my_element', function()  
{  
    return '<article type="awesome">';  
});
```

次に、名前でマクロを呼び出します。

### カスタムHTMLマクロを呼び出す

```
echo HTML::my_element();
```

## 12 フォームの作成

注意：フォーム要素に表示されるすべての入力データは `HTML::entities` メソッドを通してフィルタリングされます。

### 12.1 フォームを開く

現在のURLへPOSTするフォームを開く

```
echo Form::open();
```

URIとリクエスト方法を指定し、フォームを開く

```
echo Form::open('user/profile', 'PUT');
```

HTTPSのURLへPOSTするフォームを開く

```
echo Form::open_secure('user/profile');
```

フォームタグに追加のHTML属性を指定する

```
echo Form::open('user/profile', 'POST', array('class' => 'awesome'));
```

ファイルアップロードを受け付けるフォームを開く

```
echo Form::open_for_files('users/profile');
```

HTTPSを使い、ファイルアップロードを受け付けるフォームを開く

```
echo Form::open_secure_for_files('users/profile');
```

フォームを閉じる

```
echo Form::close();
```

## 12.2 C S R F プロテクション

Laravel はクロスサイト・リクエスト・フォージェリからサイトを守る簡単な方法を提供しています。まず、ユーザーのセッションにランダムトークンを設置します。これは自動的に行われますので、何もする必要はありません。次に、フォームに隠し入力フィールドを生成し、ランダムトークンを埋め込みます。

セッションの C S R F トークンを埋め込む隠しフィールドを生成する

```
echo Form::token();
```

ルートに C S R F フィルターを追加する

```
Route::post('profile', array('before' => 'csrf', function()
{
    //
}));
```

C S R F トークン文字列を取得する

```
$token = Session::token();
```

Laravel の CSRF プロテクション機能を使用する前に、セッションドライバを指定する必要があります。

参照：

- ルートフィルター
- クロスサイト・リクエスト・フォージェリ<sup>1</sup>

## 12.3 ラベル

ラベル要素を生成する

```
echo Form::label('email', 'E-Mail Address');
```

ラベルに追加の H T M L 要素を指定する

---

<sup>1</sup><http://ja.wikipedia.org/wiki/%E3%82%AF%E3%83%AD%E3%82%B9%E3%82%B5%E3%82%A4%E3%83%88%E3%83%AA%E3%82%AF%E3%82%A8%E3%82%B9%E3%83%88%E3%83%95%E3%82%A9%E3%83%BC%E3%82%B8%E3%82%A7%E3%83%AA>

```
echo Form::label('email', 'E-Mail Address', array('class' => 'awesome'));
```

ラベルの表示内容のHTMLエスケープを行わない

```
echo Form::label('confirm', 'Are you <strong>sure</strong> you want to proceed?', null, false);
```

ラベルの表示内容の自動HTMLエスケープを行わないため、4番目の引数にオプションとして `false` を指定することもできます。

ラベルを生成後に、ラベルと一致する名前で作られるHTML要素は、その名前と同じIDも生成されます。

## 12.4 テキスト、テキストエリア、パスワード、隠しフィールド

テキスト入力要素の生成

```
echo Form::text('username');
```

テキスト入力要素にデフォルト値を指定する

```
echo Form::text('email', 'example@gmail.com');
```

注目：`hidden`と`textarea`メソッドは`text`メソッドと使い方は同じです。一つ覚えるだけで、3つまとめて学べます。

パスワード入力要素を生成する

```
echo Form::password('password');
```

## 12.5 チェックボックスとラジオボタン

チェックボックス要素を生成する



```
echo Form::checkbox('name', 'value');
```

チェック状態をデフォルトにして生成する

```
echo Form::checkbox('name', 'value', true);
```

注目： *radio*メソッドは *checkbox*と全く同じです。1つで2つ分ですね。

## 12.6 ファイル入力

ファイル入力要素を生成する

```
echo Form::file('image');
```

## 12.7 ドロップダウンリスト

配列の要素から、ドロップダウンリストを生成する

```
echo Form::select('size', array('L' => 'Large', 'S' => 'Small'));
```

一つのアイテムをデフォルトに指定し、ドロップダウンリストを生成する

```
echo Form::select('size', array('L' => 'Large', 'S' => 'Small'), 'S');
```

## 12.8 ボタン

S u b m i t ボタン要素を生成する

```
echo Form::submit('Click Me!');
```

注目： ボタン要素を生成する必要がある？ならば、*button*メソッドをお試しください。*submit*と使い方は同じです。

## 12.9 カスタムマクロ

カスタムフォームクラスヘルパー、通称「マクロ」を簡単に定義できます。実例を見て下さい。最初に、マクロを名前と無名関数を指定して、登録します。

### フォームマクロを登録する

```
Form::macro('my_field', function()  
{  
    return '<input type="awesome">';  
});
```

次に、名前でそのマクロを呼び出します。

カスタムマクロを呼び出す。

```
echo Form::my_field();
```

# 13 入力とクッキー

## 13.1 入力

`input`クラスはアプリケーションへ GET、POST、PUT、DELETE リクエストを通じて行われる入力を処理します。入力データへどうやってアクセスするのか、いくつか例を見てみましょう。

入力配列から値を得る

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

注目：“get” メソッドは、全てのリクエストタイプ (GET、POST、PUT、DELETE) の入力を扱います。GET だけではありません。

入力配列から、全部入力を得る

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

`$_FILES` 配列も含めて、全ての入力を得る

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

デフォルトでは、入力アイテムが存在しない場合、*null*が返されます。しかし、メソッドの2番目の引数として、他のデフォルト値を指定できます。

要求する入力項目が存在しない時の、デフォルト値を指定する

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

デフォルト値を返すために、無名関数を使用する

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

与えられた名前のアイテムが入力に存在するか確かめる

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

注目：“has”メソッドは、入力項目が空文字列の場合、*false*を返します。

## 13.2 JSON入力

例えば、Backbone のような Javascript MVC フレームワークを使用している時、アプリケーションでJSONのポストを受け取る必要があるでしょう。人生を楽しむために、“Input::json”メソッドを導入してあります。

アプリケーションで、JSON入力を受け取る

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

## 13.3 ファイル

全ての `$_FILES` 配列を受け取る

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

`$_FILES` 配列から、一つの項目を受け取る

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

`$_FILES` 配列から、特定のアイテムを受け取る

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

注目：ファイルアップロードを使用するには `Form::open_for_files()` を使用するか、自分で `multipart/form-data` を有効にしてください。

参照：

- [Opening Forms](#)

## 13.4 以前の入力

正しくないフォームが送信された後は、通常再表示する必要があります。Laravel の `Input` クラスは、この問題も心に留めて設計されています。ここに示すのは、いかに簡単に前のリクエストから入力を受け取れるかという例です。最初に、セッションに入力データを退避します。

セッションに入力を退避する

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

特定の入力をセッションに退避する

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

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

前回のリクエストで退避した入力を受け取る

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

注目：“old” メソッドを使用する前に、セッションドライバーを指定してください。

参照：

- [セッション](#)

## 13.5 以前の入力とリダイレクト

これで、どうやって入力をセッションに退避させるか理解できたでしょう。リダイレクトする場合、古い入力に多少手間をかけなくても良いように、ショートカットも使えます。

`Redirect` インスタンスで、入力を退避する

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

`Redirect` インスタンスで、特定の入力を退避する

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

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

## 13.6 クッキー

Laravel は `$_COOKIE` 配列を、ナイスにラップしています。しかしながら、使用する前にいくつかの点を認識しておく必要があります。最初に、全ての Laravel クッキーは「署名ハッシュ」で保存されます。これによりフレームワークが、そのクッキーはユーザーによって変更されていないことを確認できるようになります。第二に、クッキーを保存する場合、そのクッキーは直ぐにブラウザには送信されません。リクエストの最後まで保持し、一緒に送信されます。これが意味するのは、セットしたクッキーの値を同じリクエストの中で取得することはできないということを表します。

### クッキーの値を取得する

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

### 要求したクッキーが存在していない場合のデフォルト値を指定する

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

### 持続時間 60 分のクッキーを保存する

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

### 5 年間持続する、「永続」クッキーを作成する

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

### クッキーを削除する

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

## 13.7 マージと置換え

時々、現時点の入力とマージしたり、書き換えたりしたいことがあるでしょう。

### 現在の入力に新しいデータをマージする

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

### 入力全部を新しいデータに置き換える

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

## 13.8 入力のクリア

現在のリクエストの入力データーを全てクリアする場合は、“clear” メソッドを使用してください：

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

# 14 バンドル

## 14.1 基本

バンドルは Laravel3.0 の改善点の中心になるものです。シンプルな方法で、コードを便利な「バンドル（束）」にグルーピングできるようになりました。バンドルは、それぞれ独自のビュー、設定、ルート、マイグレーション、タスクなどが持てます。バンドルでデータベース ORM から強固な認証システムまで、何でも開発できます。このスコープのモジュール化は重要な一面で、Laravel における設計の決定全部を事実上行かせたものです。あなたは **application** フォルダを色々な捉え方をしているでしょうが、Laravel に置ける特別なデフォルトバンドルで一番最初にロードされ、使用されるようにプログラムされていると考えることができます。

## 14.2 バンドルを作成する

バンドル作成の最初のステップは、**bundles**ディレクトリーの中にフォルダーを作ることです。例えば、アプリケーションの管理者用バックエンドとして動作する、“admin” バンドルを作成してみましょう。**application/start.php**ファイルはアプリケーションをどのように実行するかを決めるための基本的な設定を提供しています。同じ目的で、新しいバンドルフォルダーの中に **start.php**ファイルを作成できます。これは、バンドルがロードされるといつも実行されます。これを作成しましょう。

### バンドル `start.php` ファイルの作成

<?php

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

この **start** ファイル中では、オートローダーに対し、名前空間“Admin”のクラスはバンドルの **models** ディレクトリーからロードするように、指示しています。**start** ファイルの中でなんでも好きな事ができますが、典型的な使用方法はクラスのオートロードを登録することです。しかし実際のところ、バンドルに **start** ファイルを作成しなければならない訳ではありません。

次に、アプリケーションにこのバンドルをどうやって登録するのかを見てみましょう。

## 14.3 バンドルを登録する

さあ、admin バンドルは出来ました。Laravle へ登録しなければなりません。**application/bundles.php**を開いてください。このファイルでアプリケーションに必要なバンドルを全て登録します。では、付け加えましょう。



## シンプルなバンドルを登録する

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

利便性のため、Laravel はルートレベルの `bundle` ディレクトリーにある `admin` バンドルを読み込もうとしますが、もしご希望なら他の場所を指定することもできます。

## 場所を指定し、バンドルを登録する

```
return array(  
  
    'admin' => array('location' => 'userscape/admin'),  
  
);
```

これで Laravel は `bundles/userscape/admin` でバンドルを探します。

# 14.4 バンドルとクラスのロード

通常、バンドルの `start.php` ファイルはオートローダーの登録のみ行います。ですから、`start.php` を飛ばして、バンドルのマッピングを登録配列でそのまま宣言できます。こうなります：

## バンドル登録で、オートローダーの定義を行う

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

それぞれのオプションが、Laravel のオートローダーの機能に対応していることに注目してください。実際、オプションは対応するオートローダーの機能に自動的に渡されています。

多分、`(:bundle)` プレースホルダーに気が付かれたことでしょう。使いやすように、これは自動的にバンドルへのパスへ置き換わります。簡単ですね。

## 14.5 バンドルを始める

さあ、これでバンドルを作成し、登録も済みました。しかしまだ使えません。最初にスタートする必要があります。

### バンドルを開始する

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

これで Laravel に、クラスのオートローダーを登録しているだろう、バンドルの **start.php** ファイルを実行するように、指示します。start メソッドは、**routes.php** がバンドルに存在するならば、それも読みこむように指示します。

バンドルは一度だけしか開始できません。続けて start メソッドを呼び出しても、無視されます。

もし、アプリケーション全体でバンドルを使用したいのであれば、リクエストのたびに開始したいでしょう。そんな場合は、**application/bundles.php** ファイルでバンドルを自動スタートするように設定しましょう。

### バンドルを自動スタートするように設定する

```
return array(  
  
    'admin' => array('auto' => true),  
  
);
```

いつもバンドルを明白に開始する必要はありません。実際、通常はバンドルを自動開始するように設定してしておけば、Laravel は残りを上手く処理します。例えば、バンドルのビュー、設定、言語、ルートやフィルターを使用しようとする、バンドルは自動的に開始されます。

バンドルが開始されると、毎回イベントが発生します。バンドルの開始をこんな風にして、リッスンできます：

### バンドルの開始イベントをリッスンする

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

更に、バンドルがスタートしないように、「不使用」にすることもできます。

バンドルを開始しないように、「不使用」にする

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

## 14.6 バンドルへのルーティング

ルーティングとバンドルの情報に関しては、[バンドルルーティング](#)と[バンドルコントローラー](#)のドキュメントを参照してください。

## 14.7 バンドルを使用する

前に述べた通り、バンドルはビュー、設定、言語ファイルを持てます。Laravel では、構文にダブルコロンの使い、それらのアイテムをロードします。いくつか例を見てみましょう。

バンドルのビューを読み込む

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

バンドルの設定アイテムを読み込む

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

バンドルの言語ファイルを読み込む

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

時々、存在しているかどうか、場所とか、設定配列に含まれているかどうか、バンドルの「メタ」情報を集めたい時もあることでしょう。こうなります：

バンドルが存在するか調べる

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

バンドルがインストールされている場所を取得する

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

バンドルの設定配列を取得する

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

インストールされているすべてのバンドル名を取得する

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

## 14.8 バンドルのアセット

もしバンドルにビューが含まれているのであれば、アプリケーションの **public**ディレクトリーに Javascript や画像などのアセットを用意する必要があります。問題ありませんただ、バンドルの中に **public**フォルダーを作成し、そこに全部のアセットをおいてください。

素晴らしい！でも、どうやってアプリケーションの **public**フォルダーへ移せばいいのでしょうか。Laravel の”artisan” コマンドラインは、全てのバンドルのアセットを **public** ディレクトリーへコピーする、シンプルなコマンドを提供しています。ご覧ください。

### publicディレクトリーでバンドルのアセットを公開する

```
php artisan bundle:publish
```

このコマンドは、バンドルのアセットのために **application** 下の **public/bundles**ディレクトリーの中にフォルダーを作成します。例えば、あなたのバンドルの名前が”admin” でしたら、**public/bundles/admin**フォルダーが作成され、あなたのバンドルの **public** フォルダー内にある全てのファイルがコピーされます。

**public** ディレクトリーに入れたバンドルアセットへのパスを便利に取得する情報は、[アセット管理](#)のドキュメントを参照してください。

## 14.9 バンドルのインストール

もちろんいつでもバンドルを手動でインストールできますが、”artisan” CLI はバンドルをインストール／アップグレードする素晴らしい方法を提供しています。フレームワークでは、インストールするバンドルをシンプルに Zip 解凍して使用します。実例をご覧ください：

### artisanでバンドルをインストールする

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

素晴らしい！これでバンドルがインストールされました。これを[登録](#)し、[アセットを公開](#)する準備ができました。

利用できるバンドルのリストが必要ですか？Laravel の[バンドルディレクトリー](#)<sup>1</sup>をチェックしてください。

## 14.10 バンドルのアップグレード

バンドルをアップグレードすれば、Laravel は自動的に古いバンドルを削除し、新しくコピーします。

---

<sup>1</sup><http://bundles.laravel.com>

## `artisan`でバンドルをアップグレードする

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

注目：バンドルをアップグレードしたあとは、[アセットを再公開](#)してください。

重要：バンドルはアップグレード時に完全に削除されますので、バンドルのコアに加えた変更を認識しておく必要があります。バンドルの設定オプションを変更する必要もあることでしょう。バンドルのコードを直接変更する代わりに、バンドルの `start` イベントで設定することができます。次のようなコードを `application/start.php` ファイルに書いてください。

## バンドルの `start` イベントをリッスンする

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

# 15 クラスのオートロード

## 15.1 基本

オートローディングであなたはクラスの読み込みで明確に *require* したり *include* したりする必要がなくなり、のんびりできます。アプリケーションにリクエストがあり、クラスが本当に必要になった時のみ読み込まれますから、あなたは関連するファイルを読み込むことなく、どんなクラスでも直ぐに使用できます。

デフォルトでは、**models** と **libraries** ディレクトリーが、**application/start.php** ファイルの中で、オートロードされるように登録されています。クラス名をすべて小文字にしたファイル名を使用する規約になっています。例えば、**"User"** クラスは、**models** ディレクトリーの中に、**"user.php"** という名前のファイルで設置しなくてはなりません。また、クラスをサブディレクトリーに入れ、ネストすることもできます。名前空間をディレクトリー構造に合わせるだけです。ですから、**"Entities\User"** クラスは、**models** ディレクトリーの中に、**"entities/user.php"** という名前で設置します。

## 15.2 ディレクトリーを登録する

上で示した通り、**models** と **libraries** ディレクトリーはデフォルトでオートロードされます。しかしながら、クラスと同じファイル名を付ける規約を使い、どのディレクトリーでも登録することができます。

オートロードするディレクトリーを登録する

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

## 15.3 マップを登録する

時には、手動でクラスに関連するファイルをマッピングしたい時もあるでしょう。これは一番効率が良い、クラスのローディング方法です。

オートローダーにクラスとファイルのマップを登録する

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

## 15.4 名前空間を登録する

多くのサードパーティライブラリーは PSR-0 規約に従った構造をしています。PSR-0 では、クラス名はファイル名と一致しており、ディレクトリー構造が名前空間になります。もし、PSR-0 ライブラリーを使用するならば、ルートの名前空間とディレクトリーをオートローダーに登録してください。

### 名前空間をオートローダーに登録する

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

PHP で名前空間が使えるようになる前は、多くのプロジェクトで下線 (   ) がディレクトリー構造を表していました。もし、あなたがこのような古風なライブラリーを使っているとしても、オートローダーに登録するのは簡単です。例えば、SwiftMailer を使っているのだしたら、全てのクラスは "Swift\_" で始まっていることに気がついているでしょう。では、下線を使っているプロジェクトをルートとして指定することで、"Swift" をオートローダーに登録してみましょう。

### 「下線」使用のライブラリーをオートローダーに登録する

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

# 16 エラーとログ

## 16.1 基本設定

エラーとログに関する設定オプションはすべて、`application/config/errors.php`にあります。早速、見てみましょう。

### エラーを無視する

`ignore`オプションは Laravel により、無視されるエラーレベルを配列で指定します。「無視する (ignore)」ことで、それらのエラーが発生しても、スクリプトの実行を止めません。しかし、ログが有効になっていれば、ログに残されます。

### エラー詳細

`detail`オプションで、エラーが発生した時にエラーメッセージとスタックトレースを表示するかをフレームワークに指示します。開発時は、`true`にしておきましょう。ですが、実機環境では `false`にセットします。`false` の時は、一般的なエラーメッセージの内容である `application/views/error/500.php`にあるビューが表示されます。

## 16.2 ログ

ログを有効にするためには、`error` 設定の中の `log` オプションを `"true"` にセットします。有効にすると、エラー発生時に、`logger` 設定アイテムで定義されている無名関数が実行されます。これにより、どのようにエラーをログするかをとても柔軟に取り扱えます。そのエラーを開発チームにメールで送信することもできます！

デフォルトでは、ログは `storage/logs` ディレクトリーに保存され、毎日新しいログファイルが作成されます。これでログファイルが、余りに込み入ってしまうことを防ぎます。

## 16.3 Logger クラス

時には、Laravel の `log` クラスをデバッグや、もしくはただ情報メッセージを取り扱うのに使いたい場合もあるでしょう。

### ログにメッセージを書く



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

ログメッセージタイプを指定するためにマジックメソッドを使う

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

# 17 プロファイラー

## 17.1 Enabling the プロファイラー

プロファイラーを有効にするには、`application/config/application.php`を編集しプロファイラーオプションに `true`を設定する必要があります。

```
'profiler' => true,
```

これにより、あなたがインストールした Laravel から戻ってくる全てのレスポンスにはプロファイラーのコードが付与されます。

注意：これを書いている時点の問題は、プロファイルを有効にしていると JSON を送り返すレスポンスにでさえ、プロファイルコードを含めてしまい、レスポンス中の JSON のシンタックスを壊してしまうというものです。

## 17.2 ログ

プロファイラーの一部にログを表示することも可能です。アプリケーションの全体を通しロガーを呼び出し、プロファイラーがレンダリングされる時に表示されます。

プロファイラーにログする

```
Profiler::log('info', ' 情報をプロファイラーにログ');
```

## 17.3 タイマーとベンチマーク

アプリケーションの速度測定とベンチマークはとてもシンプルで、プロファイラーの `tick()` 関数を使用します。また、アプリケーションに異なったタイマーを設定することも可能で、実行が終了時に結果を表示します。

それぞれのタイマーはタイムラインに指定された個別の名前を持ちます。同じ名前を持つタイマーは、最後に実行されたものがタイムラインに表示されます。それぞれのタイマーは実行される場合に、別の操作を行うためコールバックを実行できます。

無名のタイマータイムラインを使用

```
プロファイラー::tick();  
プロファイラー::tick();
```

分割されたタイムラインで複数の名前のタイマーを使用

```
プロファイラー::tick('myTimer');  
プロファイラー::tick('nextTimer');  
プロファイラー::tick('myTimer');  
プロファイラー::tick('nextTimer');
```

名前付きのタイマーをコールバックとともに使用

```
プロファイラー::tick('myTimer', function($timers) {  
    echo " 私はタイマーコールバックの内側にいます！";  
});
```

# 18 実行時の環境設定

## 18.1 基本

時々、実行時に設定オプションを取得したり、設定したりする必要があるでしょう。**Config**クラスが使用できます。Laravel の「ドット (.)」構文で、設定ファイルと項目にアクセスできます。

## 18.2 オプションを取得する

設定オプションを取得する

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

オプションが存在しない時、デフォルト値を返す

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

設定配列全体を取得する

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

## 18.3 オプションを設定する

設定オプションを設定する

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

# 19 リクエストの確認

## 19.1 U R I 操作

リクエストの現在のU R I を取得

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

U R I の特定のセグメントを取得

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

セグメントが存在しない場合、デフォルト値を返す

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

クエリ文字列も含んだ、完全なU R I を取得

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

時々、URI が与えられた文字列であるか、もしくは文字列で始まっているかを調べる必要が有ることでしょう。このために、`is()` メソッドをどうやって使用するかのサンプルです。

U R I が” h o m e ” であるか確かめる

```
if (URI::is('home'))
{
    // 現在の URI は"home" だ！
}
```

現在のU R I が” d o c s / ” で始まっているか確かめる

```
if (URI::is('docs/*'))
{
    // 現在の URI は"docs/" で始まっている！
}
```

## 19.2 他のリクエストヘルパー

現在のリクエストのメソッドを得る

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

**\$\_SERVER** グローバル配列にアクセスする

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

リクエスト場所の IP アドレスを取得する

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

現在のリクエストが HTTPS を使っているか確かめる

```
if (Request::secure())
{
    // このリクエストは HTTPS で送られてきた！
}
```

現在のリクエストが AJAX リクエストであるかチェックする

```
if (Request::ajax())
{
    // このリクエストは AJAX を使用している！
}
```

現在のリクエストが artisan CLI を通しているか確かめる

```
if (Request::cli())
{
    // このリクエストは、CLI からだ！
}
```

## 20 URLの生成

### 20.1 基本

アプリケーションのベースURLを取得

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

ベースURLからの相対アドレスより生成

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

HTTPSのURLを生成

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

現在のURLを取得

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

クエリー文字列も含めた現在のURLを取得

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

### 20.2 ルートへのURL

名前付きルートへのURLを生成

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

時々、名前付きルートへのURLを生成する場合に、URIのワイルドカードを指定した値で置き換えたことがあると思います。ワイルドカードを実際の値に置き換えるのは、簡単です。

ワイルドカード値と共に、名前付きルートを生成する

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

参照：

- [名前付きルート](#)

## 20.3 コントローラーアクションへのURL

コントローラーアクションへのURLを生成

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

ワイルドカード値と共に、アクションへのURLを生成

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

## 20.4 多言語へのURL

他の言語の同じページへのURLを生成

```
$url = URL::to_language('fr');
```

他の言語のホームページへのURLを生成

```
$url = URL::to_language('fr', true);
```

## 20.5 アセットへのURL

アセットへのURLには、“application.index”設定オプションの値は含まれません。

アセットへのURLを生成

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

## 20.6 URLヘルパー

あなたの人生を楽しみ、コードをクリーンにするように設計されたURL生成のためのグローバルfunctionがあります。

ベースURLからの相対アドレスより生成

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

アセットへのURLを生成



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

名前付きルートへのURLを生成

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

ワイルドカード値と共に、名前付きルートを生成する

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

コントローラーアクションへのURLを生成

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

ワイルドカード値と共に、アクションへのURLを生成

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

# 21 イベント

## 21.1 基本

イベントは複数のアプリケーションを独立させ、コードに手を加えることなく、アプリケーションのコアにプラグインを実現する素晴らしい手段を提供します。

## 21.2 イベントの発行

イベントを発生させるには、**Event**クラスに、発生させたいイベントの名前を指定します。

### イベント発生

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

**fire**メソッドの結果を変数に受け取っていることに注目してください。このメソッドは、すべてのイベントリスナーのレスポンスで構成された配列です。

場合により、最初のイベントを発生させ、最初のレスポンスを受け取りたい時があるでしょう。次のようにします。

### イベントを発生させ、最初のレスポンスを受け取る

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

注目：**first**メソッドは、リッスンしている全てのハンドラーにむけイベントを発行させますが、最初のレスポンスのみが返されます。

**Event::until**メソッドは、最初に **NULL** ではないレスポンスが返るまで、イベントハンドラを実行します。

### 最初の非NULLレスポンスが返ってくるまで、イベントを発生

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

## 21.3 イベントのリッスン

ところで、誰もイベントをリッスンしなければ、役に立ちませんよね？イベントが発生したときに、呼び出されるハンドラーを登録しましょう。

## イベントハンドラーを登録する

```
Event::listen('loaded', function()  
{  
    // 私は"loaded" イベントで実行される！  
});
```

メソッドに指定した無名関数は、“loaded” イベントが発生するたびに実行されます。

## 21.4 キューイングされるイベント

ある場合には、イベント発生を「キュー」にため、直ぐに発生させたくない場合もあることでしょう。これは、“queue” と “flush” メソッドで可能です。最初に、ユニークな識別子を指定し、キューにイベントを入れましょう。

### イベントをキューに登録する

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

このメソッドには3つの引数があります。最初はキューの名前です。2番目はキューに入れるアイテムのユニークな識別子です。3つはキューのフラッシャー(flusher) に渡すデーターの配列です。

次に、“foo” キューにフラッシャーを登録します。

### イベントフラッシャーを登録する

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

イベントフラッシャーは2つの引数を取ることに注意してください。最初は、キューイベントのユニークな識別子です。この場合はユーザーID です。2つ目（残りの）引数は、そのキューイベントに対する実行アイテムです。

最後に、フラッシャーを実行し、全てのキューに溜まっているイベントを実行させるために“flush” メソッドを使用してください。

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

## 21.5 Laravel イベント

Laravel のコアにより、発生させられるイベントもいくつか存在します。確認してください。

### バンドル開始時に発生するイベント

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

データベースクエリが実行された時に発生するイベント

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

ブラウザにレスポンスが送信される直前に発生するイベント

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

Logクラスを用い、メッセージをログした時に発生するイベント

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

## 22 バリデーション

### 22.1 基本

ほとんどのインタラクティブな Web アプリケーションは、データのバリデーションが必要です。例えば、登録フォームでは、パスワードの再確認が必要でしょう。多分、メールアドレスは重複してはいけません。データのバリデーションは堅苦しいプロセスです。ありがたいことに、Laravel では、そうではありません。Validator クラスはデータのバリデーションを簡単にしてくれる素晴らしいヘルパーを用意してくれています。一例を見てみましょう。

バリデーションしたいデータを配列で獲得

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

データに対するバリデーションルールを定義

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

Validator インスタンスを作成し、実行する

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

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

`errors` プロパティは、エラーメッセージの取り扱いを簡単にしてくれる、シンプルな message collector クラスです。もちろん、デフォルトのエラーメッセージは全てのバリデーションルールに用意してあります。デフォルトのメッセージは `language/en/validation.php` にあります。

これで、基本的な Validator クラスの使い方に慣れました。データをバリデーションするのに使用するルールについて、掘り下げて学ぶ用意ができました。

### 22.2 バリデーションルール

必須項目

存在し、空文字列ではないことをバリデートする属性です。

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

あるフィールドが入力済みの場合、同時に入力されていることをバリデートする属性です。

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

## 文字種指定

英文字だけで構成されていることをバリデートする属性です。

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

英文字と数字だけで構成されていることをバリデートする属性です。

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

英数字とダッシュ、下線で構成されていることをバリデートする属性です。

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

## サイズ

与えられた文字数であること、もしくは数字項目の場合はその値であることをバリデートする属性です。

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

サイズが与えられた範囲内であることをバリデートする属性です。

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

注目：最低値と最高値も含まれます。

与えられたサイズ以上であることをバリデートする属性です。

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

与えられたサイズ以下であることをバリデートする属性です。

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

## 数字項目

数字であることをバリデートする属性です。

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

整数であることをバリデートする属性です。

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

## 内包と除外

リストの値の中にあることをバリデートする属性です。

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

リストの値の中に無いことをバリデートする属性です。

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

## 確認項目

*confirmed*ルールは *attribute\_confirmation*項目が存在し、その値と一致していることをバリデートする属性です。

### 確認項目と一致していることをバリデート

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

この例で Validator は、*password*項目が、配列の中の *password\_confirmation*項目と一致していることを、確認します。

## 受け入れの確認

*accepted*ルールは項目が *yes*か *1*であることをバリデートします。このルールは「サービスの規約」のようなフォームのチェックボックスのバリデーションに役立ちます。

### その項目が受け入れられたかバリデートする

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

## 22.3 他項目との比較

項目値が、他のフィールドの値と同じ事をバリデートする

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

2つの項目の値が異なることをバリデートする

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

### 正規表現

*match*ルールは与えられた正規表現と一致することをバリデートします。

正規表現と一致することをバリデートする

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

### 一意と存在

値が与えられたデータベーステーブルで一意であることをバリデートする

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

上記の例では、*email*項目は *users*テーブルで、ユニークであるかチェックされます。その項目名とカラム名が異なっている時にもユニークであることを確かめたいのですか？問題ありません。

### uniqueルールでカスタムカラム名を指定する

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

レコードを更新する場合、通常は *unique* ルールを使用しても、更新するそのレコードに対しては適用を除外したいことはよくあります。例えば、ユーザープロフィールの更新では、メールアドレスの変更は許可されていることでしょう。しかし、*unique*ルールが効いていると、そのユーザーがメールアドレスを変更しなかった場合、*unique*ルールは失敗してしまいます。そのため、更新するユーザーに対しては、このルール適用を飛ばす必要があります。

IDを指定し、*unique*ルールを無視するよう強制する



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

データベーステーブルに項目の値が存在していることをバリデートする

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

`exists` ルールにカスタムカラム名を指定する

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

日付

指定日付以前であることをバリデートする

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

指定日付以降であることをバリデートする

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

注目：`before`と `after`バリデーションルールは日付の解析に、PHP の関数である `strtotime`を利用しています。

日付が与えられたフォーマットであることをバリデートする

```
'start_date' => 'date_format:H\\\:i'),
```

注意：パラメーターのセパレーター `t` として扱われないように、コロンをバックスラッシュでエスケープすること

日付に対するフォーマットのオプションについては[PHP ドキュメント<sup>1</sup>](http://php.net/manual/ja/datetime.createfromformat.php#refsect1-datetime.createfromformat-parameters)に記述されています。

メールアドレス

メールアドレスとして正しいかバリデートする

---

<sup>1</sup><http://php.net/manual/ja/datetime.createfromformat.php#refsect1-datetime.createfromformat-parameters>

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

注目：このルールは PHP 組み込み関数の *filter\_var* メソッドを使用しています。

## URL

有効なURLであるかバリデートする

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

アクティブなURLであるかバリデートする

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

注目： *active\_url* ルールは URL がアクティブであるか判断するために *checkdnsr* を使用しています。

## アップロードファイル

*mimes* ルールはアップロードファイルが指定された MIME タイプであるかバリデートします。このルールは、そのファイルの内容を読み、実際の MIME タイプを決めるために、PHP Fileinfo 拡張を使用しています。*config/mimes.php* の中で定義されている拡張子で、引数で指定されたものは、このルールを通されます

指定されたタイプの一つであることをバリデートする

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

注目：ファイルをバリデートする時は、*Input::file()* か *Input::all()* を入力項目収集に使用してください。

ファイルが画像であることをバリデートする

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

ファイルが指定キロバイトより小さいことをバリデートする

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

## 配列

配列をバリデートする

```
'categories' => 'array'
```

ちょうど3要素を持つ配列をバリデートする

```
'categories' => 'array|count:3'
```

1 から3要素を持つ配列をバリデートする

```
'categories' => 'array|countbetween:1,3'
```

2つ以上の要素を持つ配列をバリデートする

```
'categories' => 'array|countmin:2'
```

多くて2つの要素を持つ配列をバリデートする

```
'categories' => 'array|countmax:2'
```

## 22.4 エラーメッセージの取得

Laravel では、シンプルなエラー収集クラスを使用し、手軽にエラーメッセージを取り扱えるようになっています。Validator のインスタンスで *passes*か *fails*メソッドを呼び出した後に、*errors*プロパティを利用してアクセスできます。メッセージを取得するためにいくつかの関数が用意されています。

一項目にエラーメッセージがあるか確かめる

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

その項目の最初のエラーメッセージを取得する

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

時には、HTML 要素でラップしたエラーメッセージが必要なこともあるでしょう。大丈夫です。2 番目の引数に、:message プレースホルダーを使い、フォーマットを指定してください。

エラーメッセージをフォーマットする

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

指定された項目の、すべてのエラーメッセージを取得

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

指定された項目の、すべてのエラーメッセージをフォーマット

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

全ての項目の、全てのエラーメッセージを取得

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

全ての項目の、全てのエラーメッセージをフォーマット

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

## 22.5 バリデーション実例

一度バリデーションを実行すれば、簡単にビューにそれを表示できます。Laravel では、驚異的なシンプルさで行えます。典型的なシナリオに沿って、行なってみましょう。2つのルートを定義します。

```
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);
    }
});
```

素晴らしいですね！2つのシンプルな登録のためのルートができました。一つはフォームを表示処理し、もうひとつはフォームの投稿を処理します。POST ルートでは、入力に対してバリデーションを行っています。バリデーションが失敗した場合、表示に使えるようにバリデーションエラーをセッションに退避 (flash) させ、登録フォームへリダイレクトします。

しかし、GET ルートで **errors** とビューを明確に結びつけていないことに注目してください。それでも、エラー変数 (\$errors) はビューで使用できます。賢明なことに Laravel は、errors がセッションにあれば、あなたのため、ビューに渡してくれます。errors がセッションに存在していなければ、からのメッセージコンテナがビューに渡されます。あなたはビューの中で、errors 編集を通して、いつもメッセージコンテナが存在すると思われています。私たちは、あなたの人生を楽しむことが大好きです。

例えば、メールアドレスのバリデーションに失敗すれば、セッション変数の \$errors の中に 'email' を見つけることができます。

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

Blade を使い、ビューにエラーメッセージを条件付きで付け加えることもできます。

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

これは例えば Twitter Bootstrap のようなものを使用しているときに、条件付きでクラスを付け加えたい時に便利に使えます。例えば、メールアドレスのバリデーションに失敗したら、Bootstrap の "error" クラスを `div class="control-group"` 文に付け加えたいことでしょう。

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

バリデーションが失敗したら、ビューには `error` クラスが付け加え表示されるでしょう。

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

## 22.6 カスタムエラーメッセージ

エラーメッセージをデフォルトから変更したいのですか？たぶん、項目名とルールを指定して、カスタムエラーメッセージを使いたい場合さえあるでしょう。どちらにしても、`Validator` クラスが簡単に実現してくれます。

`Validator` に渡すカスタムメッセージの配列を作成

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

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

素晴らしいですね！これで、バリデーションのチェック時に失敗した時、いつでもカスタムメッセージが使用できます。けど、`:attribute` なんてらは、メッセージの中でどうなるんでしょう？あなたが楽になるように、`Validator` クラスは、`attribute` プレースホルダーを実際の項目の名前に置き換えてくれます！項目名の下線も取り除いてくれます。

エラーメッセージを作成するときには、他にも `:other`、`:size`、`:min`、`:max`、`:values` プレースホルダーも使用できます。

他のバリデーションプレースホルダー

```
$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',
);
```

でも、カスタムメッセージが使えると言っても、`email` 項目に対してしか指定できないのでしょうか？大丈夫です。項目\_ルールのネーミングルールを使い、メッセージを指定して下さい。

与えられた項目のカスタムメッセージを指定する

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

上記の例のように、要求されたカスタムメッセージは `email` 項目に使用されますが、他のすべての項目にはデフォルトのメッセージが使用されます。

しかし、たくさんのカスタムエラーメッセージを使用するために、コードの中で指定すれば、扱いにくくめちゃくちゃになるでしょう。ですから、バリデーション言語ファイルの中の **custom** 配列で、カスタムメッセージを指定して下さい。

### バリデーション言語ファイルにカスタムエラーメッセージを追加する

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

## 22.7 カスタムバリデーションルール

Laravel は多くのパワフルなバリデーションルールを提供しています。しかし、結局自分用に作成する必要が起きるのは、よくあるでしょう。バリデーションルールを作成するには2つのシンプルな方法が用意されています。両方共素晴らしいので、プロジェクトにあった方をお使いください。

### カスタムバリデーションルールを登録

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

この例は、Validator に新しいバリデーションルールを登録しています。ルールは3つの引数を取ります。最初はバリデーションを行う項目名です。2つ目はバリデーションを行う値で、3つ目はルールに指定されるパラメーターです。

あなたのカスタムバリデーションルールを使うには次のように呼び出します。

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

もちろん、新しいルールのエラーメッセージを定義する必要があります。これは、その場で直ぐに定義する方法と：

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

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

もしくは、**language/en/validation.php**の中にあなたのルールに対するエントリーを付け加える方法があります。

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

前に述べたように、カスタムルールに引数のリストを指定し、受け取ることができます：

```
// 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];
});
```

この場合、バリデーションルールの引数は、要素が”yes”だけの配列を受け取ります。

バリデーションルールを作成し保存する、もうひとつの方法は **Validator** クラス自身を拡張することです。拡張して新しいバージョンの **Validator** を作成すれば、既に存在する機能を全部使用しつつ、あなたのカスタム機能を追加できます。もし望むのであれば、デフォルトのメソッドを置き換えることもできます。例を見ていきましょう。

最初に、**Laravel\Validator** を拡張し、**application/libraries** に設置します。

## カスタム Validator クラスを定義

```
<?php

class Validator extends Laravel\Validator {}
```

次に、**config/application.php** から **Validator** の別名 (alias) を削除します。これは必要です。そうしないと2つの”**Validator**” という名前がコンフリクトを起こしてしまいます。

次に、”**awesome**” ルールを新しいクラスに付け加えます。

## カスタムバリデーションルールを付け加える



```
<?php
```

```
class Validator extends Laravel\Validator {  
  
    public function validate_awesome($attribute, $value, $parameters)  
    {  
        return $value == 'awesome';  
    }  
  
}
```

メソッドの名前に、**validate\_**ルール命名規則を使っていることに注目してください。”awesome”という名前のルールのメソッドは、”validate\_awesome”にしなくてはなりません。これがカスタムルールを登録する時と、Validator クラスを拡張する時の、違いの一つです。Validator クラスはシンプルに true か false をリターンします。これでおしまいです！

自分で作ったバリデーションルールのカスタムメッセージを作成する必要があることも、心に留めておいてください。そうしてもらえらるのでしたら、どんなルールを定義してもらってもかまいません！

## 23 ファイルの使用

### 23.1 ファイルの読み込み

ファイルの内容を読み込む

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

### 23.2 ファイルの書き込み

ファイルに書き込む

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

ファイルに追加する

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

### 23.3 ファイルの削除

ファイルを削除する

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

### 23.4 ファイルのアップロード

**\$\_FILE** を指定場所に移動

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

アップロードしたファイルは、[Validator クラス](#)を使用して、簡単にバリデーションできます。

## 23.5 ファイル拡張子

ファイル名から拡張子を得る

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

## 23.6 ファイルタイプを調べる

ファイルが与えられたタイプであるか調べる

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

**is**メソッドは、ただ拡張子をチェックするだけではありません。Fileinfo PHP 拡張を使用し、そのファイルの本当の MIME タイプを調べるために使用されます。

注目： **application/config/mimes.php**の中に定義されている拡張子を指定できます。注目： **Fileinfo** PHP 拡張がこの機能に必要です。情報は、[PHP Fileinfo ページ<sup>a</sup>](http://php.net/manual/en/book.fileinfo.php)をご覧ください。

<sup>a</sup><http://php.net/manual/en/book.fileinfo.php>

## 23.7 M I M E タイプを取得

拡張子と関連付けられたM I M Eタイプを取得

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

注目：このメソッドは **application/config/mimes.php**で定義されている、拡張子の MIME タイプを返すだけです。

## 23.8 ディレクトリーのコピー

指定された場所のディレクトリーを再帰的にコピー

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

## 23.9 ディレクトリーの削除

再帰的なディレクトリーの削除

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

## 24 文字列の使用

### 24.1 大文字小文字変換など

**Str**クラスは文字変換を行う3つの便利なメソッド、**upper**、**lower**、**title**を提供しています。これら3つはPHPの**strtoupper**<sup>1</sup>、**strtolower**<sup>2</sup>、**ucwords**<sup>3</sup>メソッドをより賢くしたバージョンです。どこが賢いかといえば、**マルチバイト文字列**<sup>4</sup>PHP拡張があなたのWebサーバーにインストールされていれば、UTF-8を処理できるようになっています。使用方法は、単に文字列をメソッドに渡すだけです。

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

*// i am a string.*

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

*// I AM A STRING.*

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

*// I Am A String.*

### 24.2 文字数と語数

文字列の文字数を制限

```
echo Str::limit("Lorem ipsum dolor sit amet", 10);
```

*// Lorem ipsu...*

```
echo Str::limit_exact("Lorem ipsum dolor sit amet", 10);
```

*// Lorem i...*

文字列の語数を制限

```
echo Str::words("Lorem ipsum dolor sit amet", 3);
```

*// Lorem ipsum dolor...*

### 24.3 ランダム文字列の生成

英数字のランダム文字列を生成

---

<sup>1</sup><http://php.net/manual/ja/function.strtoupper.php>

<sup>2</sup><http://php.net/manual/ja/function strtolower.php>

<sup>3</sup><http://php.net/manual/ja/function.ucwords.php>

<sup>4</sup><http://php.net/manual/ja/book.mbstring.php>

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

英文字のランダム文字列を生成

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

## 24.4 単数形と複数形

複数形を取得

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

単数形を取得

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

指定された数字が2以上であれば、複数形を返す

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

## 24.5 スラグ

URLフレンドリーなスラグを生成

```
return Str::slug('My First Blog Post!');  
// my-first-blog-post
```

与えられたセパレーターを使用し、URLフレンドリーなスラグを生成

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

# 25 ローカライゼーション

## 25.1 基本

ローカライズはアプリケーションを他の言語に翻訳する処理のことです。**Lang**クラスは多言語対応アプリケーションを構成し、テキストを入手するための、シンプルなメカニズムを提供しています。

全ての言語ファイルは、**application/language**ディレクトリ下に置かれます。あなたのアプリケーションで扱うそれぞれの言語のディレクトリを **application/language**ディレクトリ下に作成してください。ですから例えば、英語とスペイン語をアプリケーションで取り扱うのであれば、**language**ディレクトリの下に、**en**と**es**ディレクトリを作成すれば良いのです。

それぞれの言語ディレクトリは異なった言語ファイルで構成されています。それぞれの言語ファイルは、シンプルにその言語の文字列の配列です。事実、言語ファイルの構造は、設定ファイルと同じです。例えば、**application/language/en**ディレクトリの中で、**marketing.php**ファイルを作成するならば、こんなふうになります。

言語ファイルを作成

```
return array(  
  
    'welcome' => 'Welcome to our website!',  
  
);
```

次に、対応する **marketing.php** ファイルを **application/language/es**ディレクトリに作成します。このファイルは次のような形式になります。

```
return array(  
  
    'welcome' => 'Bienvenido a nuestro sitio web!',  
  
);
```

ナイスですね！これで、どうやって言語ファイルとディレクトリを用意すれば良いか、理解できたでしょう。続けて、ローカライズしましょう！

## 25.2 ローカライズ文字列の取得

ローカライズした文字列を取得

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

”\_\_” ヘルパーを使用し、ローカライズした文字列を取得

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

“marketing”と”welcome”を分けるピリオドをどう使用しているかに注目してください。ピリオドの前の文字列は言語ファイルを表し、ピリオドの後に続く文字列は、そのファイルの中で指定されているキーを表します。

デフォルトの言語以外の文字列を取得する必要があるのですか？問題ありません。**get**メソッドで言語を指定するだけです。

指定された言語のローカライズ文字列を取得

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

## 25.3 プレースホルダーと置換

では、もうちょっとウェルカムメッセージを続けましょう。”Welcome to our website!”は余りにも一般的すぎるメッセージです。誰を歓迎しているのか、名前を指定できたら良くなります。しかし、それぞれのユーザーごとに言語ファイル中に指定するのは、時間を浪費し、馬鹿馬鹿しいですね。ありがたいことに、する必要はありません。言語ファイル中の翻訳テキスト中に「プレースホルダー」が使用できます。プレースホルダーはコロン(:)で始まります。

プレースホルダーを使い、翻訳テキストを作成

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

置換されたローカライズ文字列を取得

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

”\_\_”を使用し、置換されたローカライズ文字列を取得

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



# 26 暗号化

## 26.1 基本

Laravel の **Crypter** クラスは、セキュアな復元可能暗号化を扱うシンプルなインターフェイスを提供します。デフォルトで **Crypter** クラスは、強固な AES-256 暗号化と復元を **Mcrypt PHP** 拡張を通して提供しています。

注目：サーバーに **Mcrypt PHP** 拡張をインストールするのを忘れないでください。

## 26.2 文字列の暗号化

与えら得た文字列を暗号化する

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

## 26.3 文字列の復元

文字列を復元する

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

とても重要なのは、暗号化メソッドは、文字列の暗号化と復元に、あなたのアプリケーションキーを使用しているということです。

# 27 IoC コンテナ

## 27.1 定義

IoC コンテナはオブジェクトの生成を管理するシンプルな方法です。複雑なオブジェクトの生成を定義しておき、プロジェクト全体のどこからでも、一行のコードで生成することができます。更に、依存性の「注入」をクラスやコントローラーに導入することもできます。

IoC コンテナはあなたのアプリケーションを柔軟でテストしやすくするのに役立ちます。コンテナでインターフェイスの実装の別バージョンを登録できますから、[スタブとモック](#)<sup>1</sup>を使い、外部の依存からコードのテストを独立させることができます。

## 27.2 オブジェクトの登録

IoC コンテナで、リゾルバを登録する

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

素晴らしい！これでコンテナに SwiftMailer のリゾルバを登録できました。しかし、必要になるたびに毎回メイラーのインスタンスをコンテナに生成させたくないですよね？多分、最初に生成したインスタンスと同じインスタンスがコンテナから返される方が良いでしょう。その場合、ただコンテナに、そのオブジェクトはシングルトンであると伝えてください。

コンテナにシングルトンを登録する

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

既に存在しているインスタンスをコンテナにシングルトンとして登録することもできます。

存在しているインスタンスをコンテナに登録する

---

<sup>1</sup><http://martinfowler.com/articles/mocksArentStubs.html>

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

## 27.3 オブジェクトの解決

これで SwiftMailer はコンテナに登録されました。IoCクラスの **resolve**メソッドを使うことで、解決することができます。

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

注目：コンテナにコントローラーを登録することもできます。

## 27.4 インスタンスの登録を取り消す

テスト目的でいくつかのコンテナの登録を取り消したいこともあるでしょう。

mailerクラスを取り消す例：

```
IoC::unregister('mailer');
```

## 28 ユニットテスト

### 28.1 基本

ユニットテストはあなたのコードをテストし、正しく動作していることを確認します。実際、多くの推奨者達はコードを書く前に、テストを書くことさえ行なっています。Laravel は人気のあるPHPUnit<sup>1</sup>テストライブラリーを美しく統合し、テストを書き始めることを簡単にしてくれています。事実、Laravel フレームワーク自身も何百ものユニットテストが行われています。

### 28.2 テストクラスを作成する

すべてのアプリケーションのテストは、**application/tests**ディレクトリーに置かれます。このディレクトリーに、基本的な **example.test.php** ファイルを見つけられることでしょう。開いて、クラスの構成を見て下さい。

```
<?php

class TestExample extends PHPUnit_Framework_TestCase {

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

ファイルのサフィックス、**.test.php**に特に気をつけてください。これはLaravelにテスト実行時に、このファイルはテストケースのクラスであると告げています。**test** ディレクトリーにある、このサフィックスが付いていないファイルは全て、テストケースとして扱われません。

もし、バンドルにテストを書くのであれば、バンドルの中の **tests**ディレクトリーに置いてください。Laravel が残りの面倒を見ます！

テストケース作成に関係する、残りの情報は[PHPUnit ドキュメント](http://www.phpunit.de/manual/current/ja/)<sup>2</sup>をご覧ください。

---

<sup>1</sup><http://www.phpunit.de/manual/current/ja/>

<sup>2</sup><http://www.phpunit.de/manual/current/ja/>

## 28.3 テストを実行する

テストの実行には、Laravel の `artisan` コマンドラインユーティリティが使用できます。

`artisan` で、アプリケーションテストを実行する

```
php artisan test
```

バンドルのユニットテストを実行

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

## 28.4 テストからコントローラーを呼び出す

テストからコントローラーをどうやって呼び出すかのサンプルです。

テストからコントローラーを呼び出す

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

テストで、コントローラーのインスタンスを生成する

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

注目 : `Controller::call` を使用し、コントローラーアクションを実行する時、コントローラーのアクションフィルターは実行されます。

## 29 データベース設定

Laravel は以下のデータベースをサポートしています。

全てのデータベース設定オプションは、`application/config/database.php`の中にあります。

### 29.1 SQLite クイックスタート

SQLite<sup>1</sup>は素晴らしいですね。設定が必要ないデータベースシステムです。デフォルトで、Laravel は SQLite データベースを使用するように設定されています。本当に、何も変える必要はありません。ただ SQLite データベース名の `application.sqlite` を `application/storage/database` ディレクトリーに投下するだけです。これでおしまいです。

もちろん、“application” 以外の名前を付けたいのであれば、`application/config/database.php` ファイルにある、データベースオプション SQLite セクションを変更できます。

```
'sqlite' => array(
    'driver'    => 'sqlite',
    'database' => 'あなたのデータベース名',
)
```

もし、あなたのアプリケーションが一日 10 万ヒット以下のアクセスを受けるのであれば、SQLite がアプリケーションの実稼働には最適でしょう。それ以上でしたら、MySQL か PostgreSQL の使用を考えてください。

良い SQLite の管理ソフトが必要ですか？この[Firefox 拡張<sup>a</sup>](https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/)をチェックしてみてください。

<sup>a</sup><https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/>

### 29.2 他のデータベースの設定

もし、MySQL、SQL Server、PostgreSQL を使用するのであれば、`application/config/database.php` の中の設定オプションを変数する必要があります。設定ファイルの中を閲覧になれば、それぞれのシステム毎に、サンプル設定が用意されているのが分かるでしょう。サーバーに合うように必要なオプションを変更し、デフォルトの接続名を設定してください。

<sup>1</sup><http://sqlite.org>

## 29.3 デフォルト接続名の設定

多分、お気づきになられたでしょうが、それぞれのデータベースの接続オプションは `application/config/database.php` ファイル中で名前が定義されています。デフォルトでは、3 接続形態が定義されています。`sqlite`、`mysql`、`sqlsrv`、`pgsql`です。（訳注：原文のまま3接続と訳していますが、4接続が正しいかと思います。）ご自由に接続名を変更してください。デフォルトの接続は `default` で設定します。

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

デフォルト接続は、常に **Fluent クエリービルダー** で使用されます。もし、リクエスト中にデフォルト接続を変更する必要がある場合は、`Congig::set()` メソッドを使用してください。

## デフォルトのPDOオプションをオーバーライドする

PDO connector クラス (`laravel/database/connectors/connector.php`) はデフォルトのPDO属性が定義しており、それぞれのシステム毎にオプションの配列でオーバーライドできます。例えばテーブルに大文字やキャメルケースでフィールドを定義してあっても、デフォルト属性としてカラム名を小文字へと強要 (`PDO::CASE_LOWER`) しています。このため、デフォルト属性では、クエリー結果のオブジェクトの変数は小文字でのみアクセスできます。デフォルトのPDO属性にMySQLシステムの設定を追加する例：

```
'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,
),
```

より多くのPDO接続属性については [PHP マニュアル<sup>2</sup>](http://php.net/manual/ja/pdo.setattribute.php) をご覧ください。

---

<sup>2</sup><http://php.net/manual/ja/pdo.setattribute.php>

## 30 生のクエリー

### 30.1 基本

`query`メソッドは、データベース接続に対して、そのままの SQL を任意に実行できるように用意されています。

データベースからレコードをセレクト

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

バインディングを使いデータベースからレコードをセレクト

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

データベースにレコードを挿入する

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

レコードを更新し、影響を受けたレコード数を取得

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

テーブルから削除し、影響を受けたレコード数を取得

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

### 30.2 その他のクエリメソッド

Laravel はデータベースへのクエリーをシンプルに出来るように、他のメソッドも用意しています。おおまかな使い方です。

SELECTクエリーを実行し、最初の結果を取得

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

SELECTクエリーを実行し、ひとつのカラムの値だけを取得



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

## 30.3 P D O 接続

時には、Laravel 接続オブジェクトの後ろで使用されている PDO 接続に直接アクセスしたい場合もあるでしょう。

データベースへの P D O 接続を取得

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

注目：コネクション名が指定されない場合は、**default**の接続がリターンされます。

# 31 Fluent クエリービルダー

## 31.1 基本

Fluent クエリビルダーは、データベースを取り扱うための、Laravel のパワフルで流暢 (fluent) な SQL クエリビルダーです。全てのクエリーはプリペAREDステートメントを使用し、SQL インジェクションから保護されます。

DB クラスの **table** メソッドを使用し、流暢にクエリーを開始しましょう。クエリーする対象テーブルを指示だけしましょう。

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

これで、”users” テーブルに対する Fluent クエリービルダーのインスタンスを手に入れました。このクエリービルダーを使い、テーブルに対しレコードを取得、挿入、更新もしくは削除することができます。

## 31.2 レコード取得

データベースから、レコードを配列で取得

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

注目： **get** メソッドはテーブルのカラムに対応するプロパティを持つオブジェクトの配列をリターンします。

データベースから一件のみ取得する

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

プライマリーキーを指定し、一件のみ取得する

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

注目：模試結果が見つからない場合は、**first**メソッドは NULL をリターンします。**get**メソッドは、空配列をリターンします。

データベースから、1 カラムの値だけを取得

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

データベースから、特定のカラムのみ取得

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

指定されたカラムの値を配列で取得

```
$users = DB::table('users')->take(10)->lists('email', 'id');
```

注目：第2引数はオプション

**distinct** を指定し、データベースからセレクト

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

## 31.3 Where 節の生成

**where** と **or\_where**

様々なメソッドが **where** 節を生成する手助けを行います。最も基本的なメソッドは、**where** と **or\_where** です。使い方をご覧ください：

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

AND WHERE と同じ働きのコードを書くには、シンプルに他の **where** でクエリーをつなげてください。

```
return DB::table('users')
    ->where('id', '=', 1)
    ->where('activated', '=', 1)
    ->first();
```

もちろん、ただ値の同じレコードを指定するだけに制限されているわけではありません。以上、以下、等しくない、**like**も使用できます。

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

あなたの考えている通り、**where**メソッドは AND 条件でクエリーします。OR 条件の場合は、**or\_where**を使います。

**where\_in**、**wherenotin**、**orwherein**、**orwherenotin**

**where\_in**メソッドは、配列の中の値で検索するクエリーを簡単に作るために使用できます。

```
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();
```

**wherenull**、**wherenotnull**、**orwherenull**、**orwherenotnull**

**where\_null**メソッドは、NULL 値を簡単にチェックするのに最適です

```
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();
```

**where\_between**、**where\_not\_between**、**or\_where\_between**、**or\_where\_not\_between**

**where\_between**とその類似メソッドは最小値と最大値の間の値をととても簡単にチェックできます。

```
return DB::table('users')->where_between($column, $min, $max)->get();
```

```
return DB::table('users')->where_between('updated_at', '2000-10-10', '2012-10-10')->get();
```

```
return DB::table('users')->where_not_between('updated_at', '2000-10-10', '2012-01-01')->get();
```

```
return DB::table('users')
    ->where('email', '=', 'example@gmail.com')
    ->or_where_between('updated_at', '2000-10-10', '2012-01-01')
    ->get();
```

```
return DB::table('users')
    ->where('email', '=', 'example@gmail.com')
    ->or_where_not_between('updated_at', '2000-10-10', '2012-01-01')
    ->get();
```

## 31.4 Where節のネスト

WHERE 節でカッコを使用し、グループにする必要があることもあります。そんな場合は、**whrere**や**where\_or**メソッドのパラメーターに無名関数を渡してください。

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

上の例は、以下の様なクエリーを生成します。

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

## 31.5 動的Where節

動的 where メソッドはコードの読みやすさを上げる、素晴らしい手法です。いくつかサンプルを挙げましょう。

```
$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');
```

## 31.6 テーブル接合

他のテーブルを join する必要がありますか？**join**と **left\_join**を試してください。

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

最初の引数に join したいテーブルを指定します。残りの3引数は、join の **ON**節を構成します。

join メソッドの使い方を覚えれば、**left\_join**も使用できるようになります。メソッドの引数は同じです。

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

join の第2パラメーターに無名関数を使用することで、**ON**節の複数条件を指定できます。

```
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'));
```

## 31.7 結果の順序

クエリー結果の並び順は **order\_by** メソッドを使用し、簡単に指定できます。メソッドに並べ替えるカラムと方向 (desc か asc) を指定してください。

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

もちろん、好きなだけカラムを指定できます。

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

## 31.8 グループ集計

クエリー結果を **group\_by** メソッドを使用し、簡単にグルーピングできます。

```
return DB::table(...)->group_by('email')->get();
```

## 31.9 レコードスキップと取得数制限

クエリーで受け取る結果の数を制限 (**limit**) したい時は、**take** メソッドが使用できます。

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

オフセットをクエリーに指定するには、**skip** メソッドを使用します。

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

## 31.10 集計

最大、最小、平均、合計、件数を求めたいのですか？でしたら、クエリーにカラム名を渡してください。

```
$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();
```

もちろん、初めに WHERE 節を使い、クエリーに条件をつけることもできます。

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

## 31.11 式

場合により、**NOW()**のような SQL 関数をカラムの値にセットする場合があります。通常、**now()** は自動的にエスケープ処理でクオートされてしまいます。これを防ぐには、**DB**クラスの **raw**メソッドを使います。このようになります：

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

**row**メソッドは、クエリーに式の内容をパラメーターではなく、文字列として挿入するように、指示するものです。例えば、項目の値をインクリメントする式に使えます。

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

もちろん、便利な **increment**、**decrement**メソッドも提供しています。

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

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

## 31.12 レコード挿入

**insert** メソッドは挿入する値の配列を受け取ります。**insert** メソッドは **true** か **false** をクエリーが成功したか示すために、リターンします。

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

挿入したレコードは自動的に ID がインクリメントされるはずですが、**insert\_get\_id**メソッドで、レコードを挿入し、その ID を取得できます。



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

注目： **insert\_get\_id**メソッドは、自動的に増分されるカラムの名前が”id”であることを前提として動作します。

## 31.13 レコード更新

レコードを更新するには、**update**メソッドに配列で値を渡します。

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

もちろん、いくつかのレコードだけを更新したい時には、**update**メソッドを呼び出す前に、**WHERE**節を付け加えてください。

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

## 31.14 レコード削除

データベースからレコードを削除したい時には、シンプルに **delete**メソッドを呼び出してください。

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

レコードを ID で手早く削除したいのですか？大丈夫です。**delete**メソッドに、その ID を渡してください。

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

## 32 Eloquent ORM

### 32.1 基本

ORM はオブジェクトリレーショナルマッパー<sup>1</sup>で、Laravel では本当に使いやすくなっています。名前は”Eloquent（雄弁な）”です。なぜなら、データベースオブジェクトと関連性を扱うのに、雄弁で表現的な構文を使用するからです。一般的には、データベースのテーブルそれぞれを Eloquent モデルとして定義します。最初に、シンプルなモデルを定義しましょう。

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

ナイスですね！Eloquent クラスを拡張したモデルであることに注目してください。このクラスはデータベースを表情豊かに取り扱うために、必要な機能を全て提供しています。

注目：通常、Eloquent モデルは、**application/models**ディレクトリーに設置されます。

### 32.2 規約

Eloquent では、データベース構造に関して、いくつかの基本的な規約があります。

- それぞれのテーブルは、**id**という名前のプライマリーキーを持つ。
- それぞれのテーブル名は、対応するモデルの複数形の名前で無くてはならない。

時には、モデルの複数形ではない名前をテーブル名に使用したり、異なった主キーを使用したりする必要がありますよね。大丈夫です。モデルに、**static** な **table** プロパティを追加してください。

```
class User extends Eloquent {  
  
    public static $table = 'my_users';  
  
    public static $key = 'my_primary_key';  
  
}
```

<sup>1</sup><http://ja.wikipedia.org/wiki/%E3%82%AA%E3%83%96%E3%82%B8%E3%82%A7%E3%82%AF%E3%83%88%E9%96%A2%E4%BF%82%E3%83%9E%E3%83%83%E3%83%94%E3%83%B3%E3%82%B0>

## 32.3 モデルの取得

Eloquent を使用して、モデルを取得するのは、心地よいほど簡単です。最も基本的な Eloquent モデルを取得する方法は static の **find** メソッドです。これはプライマリーキーを指定し、テーブルの各カラムに対応したプロパティを持つモデルをリターンするメソッドです。

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

```
echo $user->email;
```

find メソッドは、次のようなクエリーを実行します。

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

テーブル全体を取得する必要がありますか？static の **all** メソッドを使ってください。

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

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

もちろん、テーブル全部を取得できてそれほど便利ではありません。ありがたいことに、**Fluent** クエリービルダーの全てのメソッドは、**Eloquent** でも使用可能です。モデルを最初は static な **クエリービルダー** から始め、**get** か **first** メソッドで query を実行してください。get メソッドはモデルの配列を返し、一方の first メソッドは、モデルをひとつだけ返します。

```
$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();
```

注目：結果が見つからない場合、**first** モデルは NULL をリターンします。**all** メソッドと **get** メソッドは空の配列を返します。

## 32.4 集計

最小、最大、平均、合計、件数が必要ですか？適したメソッドにカラム名を渡すだけです。

```
$min = User::min('id');  
  
$max = User::max('id');  
  
$avg = User::avg('id');  
  
$sum = User::sum('id');  
  
$count = User::count();
```

もちろん、初めに WHERE 節を使い、クエリーの数を経ることもできます。

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

## 32.5 モデルの挿入と更新

Eloquent モデルを挿入するのは、多少手間がかかります。最初は新しいモデルのインスタンスを作成します。次に、プロパティをセットします。最後に **save** メソッドを実行します。

```
$user = new User;  
  
$user->email = 'example@gmail.com';  
$user->password = 'secret';  
  
$user->save();
```

他の手法として、**create** メソッドは、新しいレコードをデータベースに挿入し、その新しいレコードのインスタンスを返します。挿入に失敗すると、**false** を返します。

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

モデルの更新はとてもシンプルです。新しいモデルのインスタンスを作成する代わりに、データベースから 1 つデータを獲得します。それから、プロパティをセットし、**save** します。

```
$user = User::find(1);  
  
$user->email = 'new_email@gmail.com';  
$user->password = 'new_secret';  
  
$user->save();
```

データベースレコードの作成日時と更新日時のタイムスタンプをメンテする必要がある？ Eloquent を使うのでしたら、心配する必要はありません。モデルに **timestamps** プロパティを付け加えてください。

```
class User extends Eloquent {  
  
    public static $timestamps = true;  
  
}
```

次にテーブルへ **create\_at**と **update\_at**を **date** 型で付け加えてください。これで、いつでもモデルを保存すれば、自動的に作成／更新タイプスタンプがセットされます。なにに、どういたしまして。

場合により、モデルのデータを実際には変更しないのだけれど、**updated\_at**データカラムを更新できると便利です。こんな時は **touch**メソッドを使用してください。即時に変更し、自動的に更新します。

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

また、**update\_at**データカラムをアップデートしたいが、すぐにモデルを保存しない場合は **timestamp**関数を使用してください。もし、実際にモデルのデータを変更するのでしたら、この作業は舞台裏で密に行われることに注意してください。

```
$comment = Comment::find(1);  
$comment->timestamp();  
// $comment モデルのデータを変更しない、他のコードがここに入る  
$comment->save();
```

注目：アプリケーションのデフォルトタイムゾーンは **application/config/application.php**ファイルで変更できます。

## 32.6 関係付け

下手なやり方をしない限り、データベーステーブルは他のテーブルと関連を持つでしょう。例えば、注文はユーザーに所属しているものです。また、ポストは多くのコメントを持ちます。Eloquent は関連性を定義でき、関連するモデルをシンプルかつ直感的に取得できるようにします。Laravel は3タイプの関係をサポートしています。

- 1 対 1
- 1 対多
- 多対多

Eloquent モデルに関係を定義するためには、ただ **has\_one**、**has\_many**、**belongs\_to**、**has\_many\_and\_belongs\_to**メソッドを結果としてリターンするメソッドを作成してください。詳細を1つずつ、確かめて行きましょう。

## 1 対 1

1 対 1 の関係は、関係の中でも一番基本的な形です。例えば、ユーザーが一つの電話を持っていることを表してみましょう。Eloquent では、この関係をシンプルに記述できます。

```
class User extends Eloquent {  
  
    public function phone()  
    {  
        return $this->has_one('Phone');  
    }  
  
}
```

関係するモデル名を **has\_one** メソッドに渡していることに注目してください。これで、ユーザーの電話を **phone** メソッドを通して、取得できるようになりました。

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

この文で、どんな SQL が実行されるのか確認しましょう。2 つのクエリーが実行されます。一つはユーザーを取得し、もうひとつはユーザーの電話を取得します。

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

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

関連付けの外部キーとして、Eloquent は **user\_id** を使用していることに気をつけてください。ほとんどの外部キーが、このモデル **\_id** 規約に従っていることでしょう。しかしながら、他のカラム名を外部キーとして使っているのでしたら、メソッドの 2 番目の引数として渡してください。

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

ユーザーの電話を **first** メソッドを使用せず取得したいのですか？大丈夫です。動的 **phone** プロパティを使ってください。Eloquent は自動的に関係を読み取り、頭の良いことに **get**（1 対多関係の場合）メソッドか **first**（1 対 1 関係の場合）メソッドを呼び出します。

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

電話のユーザーを取得する必要がある？ **phones** テーブルに外部キー(**user\_id**) があるのですから、この関係を **belongs\_to**（所属する）メソッドで表現する必要があります。理屈に合っているでしょう？電話はユーザーに所属しています。**belongs\_to** メソッドを使う場合、リレーションシップメソッドの名前は対応する外部キー(**\_id**無し) にする必要があります。外部キーの名前が **user\_id** ですから、リレーションシップメソッドの名前は **user** になります。

```
class Phone extends Eloquent {

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

}
```

素晴らしい！リレーションシップメソッドか動的プロパティを使い、Phone モデルを通して User モデルにアクセスできるようになりました。

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

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

## 1 対多

ブログポストは多くのコメントを持っていると仮定できます。この関係は **has\_many** メソッドを使用して、簡単に定義できます。

```
class Post extends Eloquent {

    public function comments()
    {
        return $this->has_many('Comment');
    }

}
```

では、リレーションシップメソッドか動的プロパティを利用し、ポストコメントにアクセスしてみましょう。

```
$comments = Post::find(1)->comments()->get();
```

```
$comments = Post::find(1)->comments;
```

この両方の命令は、次のような SQL を実行するでしょう。

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

```
SELECT * FROM "comments" WHERE "post_id" = 1
```

他の外部キーを結び付けたい？大丈夫です。メソッドの第2引数として、渡してください。

```
return $this->has_many('Comment', 'my_foreign_key');
```

多分、あなたは不思議に思っていることでしょう。もし動的プロパティが関係を返し、キーストロークを節約してくれるのなら、どうしてリレーションシップメソッドを使う必要があるのだろう？実は、リレーションシップメソッドは、とてもパワフルなんです。関連を取得する前に、クエリーメソッドをチェーンして続けることができるのです。確認してみましょう。

```
echo Post::find(1)->comments()->order_by('votes', 'desc')->take(10)->get();
```

## 多対多

多対多関係は、3つの中で一番込み入っています。けれど心配しないでください。これを理解できますよ。例えば、ユーザー(User)は多くの役割(Roles)を持ち、役割(Role)も多くのユーザー(Users)に所属されます。この関係を実現するために3つのデータベーステーブルを作成する必要があります。**users**テーブル、**roles**テーブル、**role\_user**テーブルです。それぞれのテーブルの構造は、次のようになるでしょう。

### users:

```
id      - INTEGER
email   - VARCHAR
```

### roles:

```
id      - INTEGER
name    - VARCHAR
```

### role\_user:

```
id        - INTEGER
user_id   - INTEGER
role_id   - INTEGER
```

テーブルは多くのレコードにより構成されています。ですから複数形で名づけます。**has\_many\_and\_belongs\_to**関係で使用されるピボットテーブルは、関連する2モデルの単数形の名前をアルファベット順に、アンダースコアでつないだ名前にします。

これで、**has\_many\_and\_belongs\_to**メソッドを用い、モデルの関係を定義できるようになりました。



```
class User extends Eloquent {

    public function roles()
    {
        return $this->has_many_and_belongs_to('Role');
    }

}
```

素晴らしい！それでは、ユーザーの役割を取得してみましょう。

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

または、通常動的プロパティを通して、関係を取得するでしょうね。

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

もし、テーブルの命名規則に従わない場合は、**has\_many\_and\_belongs\_to**メソッドの第2引数として、テーブル名を渡してください。

```
class User extends Eloquent {

    public function roles()
    {
        return $this->has_many_and_belongs_to('Role', 'user_roles');
    }

}
```

デフォルトでは、ピボットテーブルに確実に存在するフィールドがリターンされます。(2つの **id** フィールドとタイムスタンプ) もしあなたのピボットテーブルに他のカラムを追加しているのでしたら、それらを **with()** メソッドを用い取得することもできます。

```
class User extends Eloquent {

    public function roles()
    {
        return $this->has_many_and_belongs_to('Role', 'user_roles')->with\
('column');
    }

}
```

## 32.7 関係したモデルを挿入

**Post**モデルは多くのコメントを持つと想定してください。与えられたポストに対して、新しいコメントをしばしば挿入する必要があります。モデルの **post\_id** 外部キーを手動でセットする代わりに、新しいコメントを所有されている **Post** モデルから挿入することが可能です。次のようなコードになります。

```
$comment = new Comment(array('message' => 'A new comment.'));

$post = Post::find(1);

$comment = $post->comments()->insert($comment);
```

親のモデルを通して関連するモデルを挿入する場合、外部キーは自動的に設定されます。ですからこの場合、新しく挿入されたコメントでしたら、“**post\_id**” に“1” が自動的にセットされます。

“**has\_many**” 関係を取り扱っている場合、“**save**” メソッドを関連するモデルの挿入／更新に使用できます。

```
$comments = array(
    array('message' => 'A new comment.'),
    array('message' => 'A second comment.'),
);

$post = Post::find(1);

$post->comments()->save($comments);
```

### 関係したモデルを挿入（多対多）

こうした機能は、多対多関係で、更に便利になります。例えば、**User**モデルは、多くの役割 (**roles**) を持っていると考えてください。同様に、**Role**モデルは、多くのユーザーに所属しています。ですから、この関係の中間テーブルは、“**user\_id**” と “**role\_id**” カラムを持っています。では、ユーザーへ新しい役割を挿入してみましょう。

```
$role = new Role(array('title' => 'Admin'));

$user = User::find(1);

$role = $user->roles()->insert($role);
```

役割が挿入された時、“**roles**” テーブルに **Role** が挿入されただけでなく、中間テーブルにもレコードが挿入されます。面倒ですからね！

しかしながら、しばしば新しいレコードを中間テーブルに挿入したくなるでしょう。多分、既に存在するユーザーに、役割を追加したい時などです。**attach** メソッドを使用してください。

```
$user->roles()->attach($role_id);
```

また、中間テーブル（ピボットテーブル）のフィールドにデーターを追加することも可能です。追加したいデータを含んだ追加コマンドを2つ目の変数配列として付け加えます。

```
$user->roles()->attach($role_id, array('expires' => $expires));
```

別の方法として、“sync” メソッドも使用できます。中間テーブルで同期 (sync) させたい ID の配列を渡します。この操作が完了すると、配列の中の ID だけが中間テーブルに存在することになります。

```
$user->roles()->sync(array(1, 2, 3));
```

## 32.8 中間テーブルの操作

多分ご存知でしょうが、多対多関係は中間テーブルを必要としています。Eloquent はこのテーブルの管理を簡単にしてくれています。例えば、**User**モデルが、多くの役割 (roles) を持っていると考えてください。そして同様に、**Role**モデルはたくさんのユーザーを持っています。ですから、中間テーブルは“user\_id” と “role\_id” カラムを持ちます。この関係のピボットテーブルにアクセスできます。

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

```
$pivot = $user->roles()->pivot();
```

一度ピボットテーブルのインスタンスを作成してしまえば、他の Eloquent モデルと同様に使用することができます。

```
foreach ($user->roles()->pivot()->get() as $row)
{
    //
}
```

さらに、レコードを指定することで、特定の中間テーブルのレコードにアクセスすることもできます。例えば：

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

```
foreach ($user->roles as $role)
{
    echo $role->pivot->created_at;
}
```

私達が取得した、個々の関係した **Role**モデルは、自動的に **pivot**属性を結びつけます。この属性は、関係するモデルに結びついた、中間テーブルのレコードを表すモデルで構成されています。

時々、指定したリレーションシップモデルの中間テーブルから全てのレコードを削除したい場合もあるでしょう。例えば、あるユーザーに結びつけた役割を全て削除したい場合です。どうやるか見てみましょう。

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

$user->roles()->delete();
```

これは役割を”roles” テーブルから削除しないことに注意してください。ただ、指定したユーザーに結びついている役割のレコードを中間テーブルから削除しているだけです。

## 32.9 Eager ロード

Eager ローディングは N+1 クエリー問題を和らげるために存在しています。具体的にはどんな問題でしょうか？えー、それぞれの本は著者に所属するとしましょう。この関係を表すと、次のようになります。

```
class Book extends Eloquent {

    public function author()
    {
        return $this->belongs_to('Author');
    }

}
```

では、次のコードを試してみましょう。

```
foreach (Book::all() as $book)
{
    echo $book->author->name;
}
```

一体いくつのクエリーが実行されるのでしょうか？ええと、まず一つはテーブルのすべての本を取得するために実行されます。それから、著者を取得するため、それぞれの本についてクエリーが実行されます。25冊の著者の名前を表示するために、26回のクエリーが必要になります。合計でどのくらいの速度になると思いますか？

ありがたいことに、**with**メソッドを使えば、author モデルを Eager ロードできます。Eager ロードしたい関係の関数名を使ってください。

```
foreach (Book::with('author')->get() as $book)
{
    echo $book->author->name;
}
```

この例でしたら、たったの2クエリーだけが実行されます

```
SELECT * FROM "books"
```

```
SELECT * FROM "authors" WHERE "id" IN (1, 2, 3, 4, 5, ...)
```

明らかに、Eager ロードを使えば、アプリケーションのパフォーマンスをドラマティックに改善してくれます。上の例でしたら、Eager ロードで実行時間は半分になります。

2つ以上の関連で Eager ロードを使用する必要がありますか？簡単です。

```
$books = Book::with(array('author', 'publisher'))->get();
```

注目：Eager ロードを使用する時は、いつも static の **with** メソッドでクエリーを開始してください。

ネストした関係の Eager ロードをしたい場合もあるでしょう。例えば、**Author** モデルが "contacts" 関係を持っているとしましょう。Book モデルから、両方の関係の Eager ロードが可能です。

```
$books = Book::with(array('author', 'author.contacts'))->get();
```

もし、同じモデルに対し頻繁に eager ローディングを使っているのに気づいたなら、**\$includes** をモデル中で使いたくなるでしょう。

```
class Book extends Eloquent {  
  
    public $includes = array('author');  
  
    public function author()  
    {  
        return $this->belongs_to('Author');  
    }  
  
}
```

**\$includes** は **with** と同じ引数を取ります。これで、以下のコードで eager ロードされるようになります。

```
foreach (Book::all() as $book)  
{  
    echo $book->author->name;  
}
```

注意: **with**はモデルの **\$includes**をオーバーライドします。

## 32.10 Eager ロードの構成

Eager ロードだけではなく、Eager ロードに条件を付けたい時もあるでしょう。簡単です。次のようなコードになります。

```
$users = User::with(array('posts' => function($query)
{
    $query->where('title', 'like', '%first%');
}))->get();
```

この例ではユーザーのポストを Eager ロードしていますが、ポストのタイトルに”first” がつかわれているものだけを選択しています。

## 32.11 ゲッターとセッターメソッド

セッターはカスタムメソッドを使用し、属性を結びつけられるようにしてくれます。セッターは、属性の名前に”set\_“を付けたもので定義してください。

```
public function set_password($password)
{
    $this->set_attribute('hashed_password', Hash::make($password));
}
```

セッターは変数のように括弧無しで呼び出し、名前は”set\_“プリフィックスを除いたメソッド名になります。

```
$this->password = "my new password";
```

ゲッターも似ています。属性がリターンされる前に更新するためにつかわれます。属性の名前の前に”get\_“を付け、定義してください。

```
public function get_published_date()
{
    return date('M j, Y', $this->get_attribute('published_at'));
}
```

ゲッターも変数のように括弧無しで呼び出し、”get\_”無しのメソッド名になります。

```
echo $this->published_date;
```

## 32.12 複数代入

複数代入は連想配列を渡し、モデルの属性にその配列の値を埋める方法です。複数代入はモデルのコンストラクターに配列を渡すことで行うことができます。

```
$user = new User(array(
    'username' => 'first last',
    'password' => 'disgaea'
));

$user->save();
```

もしくは、**fill**メソッドを使うことでも複数代入が行えます。

```
$user = new User;

$user->fill(array(
    'username' => 'first last',
    'password' => 'disgaea'
));

$user->save();
```

複数代入のデフォルトでは、全ての属性のキー／値のペアが保存されます。しかし、値をセットできる属性のホワイトリストを作成することも可能です。アクセスできる属性のホワイトリストがセットされると、指定されている属性だけが複数代入されます。

アクセスできる属性を指定するには、**static** の **accessible** 配列に設定してください。この配列は、複数代入可能な属性で構成されています。

```
public static $accessible = array('email', 'password', 'name');
```

もしくは **accessible** メソッドをモデルに対し使用してください。

```
User::accessible(array('email', 'password', 'name'));
```

注目：複数代入をユーザーの入力に対して使用する時は、最大限の注意を行なってください。技術の過信は、重大なセキュリティ脆弱性を生みます。

## 32.13 モデルを配列に変換

JSON API を作成している場合、頻繁にモデルを配列にコンバートするでしょう。そのために簡単にシリアライズできるようになっています。本当にシンプルです。

### モデルを配列に変換

```
return json_encode($user->to_array());
```

“to\_array” メソッドはモデルの属性全てへ、自動的に用意されます。

場合により、例えばパスワードのようなモデルの配列に含みたくない属性もあるでしょう。これを行うには、モデルに”hidden” 属性を追加してください。

### 配列で指定した属性を含めないようにする

```
class User extends Eloquent {  
  
    public static $hidden = array('password');  
  
}
```

## 32.14 モデル削除

Eloquent は Fluent クエリービルダーの全ての機能とメソッドを継承しているため、モデルもさっと削除できます。

```
$author->delete();
```

しかしながら、注意してもらいたいのは、[外部キー](#)と連鎖削除 (cascade delete) を指定していない限り、関連しているモデルは削除されないことです。(例えば、この著者の全ての Book モデルはまだ存在しています。)



## 33 スキーマビルダー

### 33.1 基本

スキーマビルダーはデータベーステーブルの作成と変更のメソッドを提供します。スラスラ書ける構文で、ベンダー限定の何かにとらわれず、テーブルを操作できます。

参照：

- [マイグレーション](#)

### 33.2 テーブルの作成と削除

**Schema**クラスはテーブルを作成／修正するために使います。さっそく、例を見てみましょう。

簡単なデータベーステーブルを作成

```
Schema::create('users', function($table)
{
    $table->increments('id');
});
```

このサンプルを確認して行きましょう。スキーマビルダーに **create**メソッドでこれは新しいテーブルで、作成する必要があると伝えます。2つ目の引数で、無名関数を渡し、**Table** インスタンスを受けます。この **Table** オブジェクトを利用し、カラムを足したり引いたり、テーブルに索引を付けたり、すらすら書けます。

データベースからテーブルを削除

```
Schema::drop('users');
```

指定したデータベース接続のテーブルを削除

```
Schema::drop('users', 'connection_name');
```

時々、スキーマ操作を行うデータベース接続を指定する必要があるかも知れません。

操作を行う接続を指定

```
Schema::create('users', function($table)
{
    $table->on('connection');
});
```

## 33.3 カラム追加

Fluent テーブルビルダーのメソッドは、特定のベンダーの SQL を使用せず、カラムを追加できます。まずはメソッドです。見て行きましょう。

| コマンド  | 説明  |
|---|---|
| <code>\$table-&gt;increments('id');</code>        | 自動増分される ID をテーブルへ                         |
| <code>\$table-&gt;string('email');</code>         | VARCHAR のカラム                              |
| <code>\$table-&gt;string('name', 100);</code>     | 長さ指定の VARCHAR                             |
| <code>\$table-&gt;integer('votes');</code>        | INTEGER をテーブルへ                            |
| <code>\$table-&gt;float('amount');</code>         | FLOAT をテーブルへ                              |
| <code>\$table-&gt;decimal('amount', 5, 2);</code> | 最大桁数と少数桁を指定し DECIMAL を追加                  |
| <code>\$table-&gt;boolean('confirmed');</code>    | BOOLEAN をテーブルへ                            |
| <code>\$table-&gt;date('created_at');</code>      | 日付をテーブルへ                                  |
| <code>\$table-&gt;timestamp('added_on');</code>   | TIMESTAMP をテーブルへ                          |
| <code>\$table-&gt;timestamps();</code>            | <b>created_at</b> と <b>updated_at</b> を追加 |
| <code>\$table-&gt;text('description');</code>     | TEXT をテーブルへ                               |
| <code>\$table-&gt;blob('data');</code>            | BLOB をテーブルへ                               |
| <code>-&gt;nullable();</code>                     | NULL 値可能を指定                               |
| <code>-&gt;default(\$value)</code>                | そのカラムのデフォルト値を宣言                           |
| <code>-&gt;unsigned();</code>                     | 整数を符号なしに設定                                |

追記：Laravel の”boolean” タイプはすべてのデータベースシステムで small integer カラムにマップされます。

### テーブルの作成とカラム追加例

```
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();
});
```

## 33.4 カラム削除

データベーステーブルからカラムを削除

```
$table->drop_column('name');
```

データベーステーブルから複数のカラムを削除

```
$table->drop_column(array('name', 'email'));
```

## 33.5 インデックス追加

スキーマビルダーは多くのタイプのインデックスをサポートしています。インデックスを付け加えるためには2つの方法があります。それぞれのインデックスタイプごとにメソッドがあります。しかしながら、カラムを追加時に索引を定義することもできます。見てみましょう。

インデックス付きで `string` カラムを作成

```
$table->string('email')->unique();
```

もし別の行でインデックスを定義するなら、もっと様々な指定ができます。インデックスメソッドの例をご覧ください。

| コマンド   | 説明              |
|--|-----------------|
| <code>\$table-&gt;primary('id');</code>                    | プライマリキーを追加      |
| <code>\$table-&gt;primary(array('fname', 'lname'));</code> | 複合キーの追加         |
| <code>\$table-&gt;unique('email');</code>                  | ユニークキーの追加       |
| <code>\$table-&gt;fulltext('description');</code>          | フルテキストインデックスの追加 |
| <code>\$table-&gt;index('state');</code>                   | 基本インデックスの追加     |

## 33.6 インデックス削除

インデックスを削除するには、名前を指定しなくてはなりません。Laravel はすべてのインデックスに適した名前をつけます。シンプルにテーブル名に続け、インデックスしているカラムの名前、それからインデックスのタイプです。例をご覧ください。

| コマンド  | 説明                               |
|---|----------------------------------|
| <code>\$table-&gt;drop_primary('users_id_primary');</code>              | “users” テーブルのプライマリーキーを削除         |
| <code>\$table-&gt;drop_unique('users_email_unique');</code>             | “users” テーブルのユニークインデックスを削除       |
| <code>\$table-&gt;drop_fulltext('profile_description_fulltext');</code> | “profile” テーブルから、フルテキストインデックスを削除 |
| <code>\$table-&gt;drop_index('geo_state_index');</code>                 | “geo” テーブルから、基本インデックスを削除         |

## 33.7 外部キー

Schema クラスの記述的なインターフェイスを使用し、テーブルに外部キー束縛を簡単に追加できます。例えば、**posts**テーブルに、**user\_id**があり、**users**テーブルの **id**カラムを参照しているとしましょう。カラムに外部キー束縛を付け加える方法です。

```
$table->foreign('user_id')->references('id')->on('users');
```

更に、「削除 (on delete)」と「更新 (on update)」アクションを外部キーに指定できます。

```
$table->foreign('user_id')->references('id')->on('users')->on_delete('restrict');
```

```
$table->foreign('user_id')->references('id')->on('users')->on_update('cascade');
```

また、簡単に外部キーを削除することもできます。スキームビルダーにおけるデフォルトの外部キーの名前は、他のインデックスを作成する場合と同じ規則に従っています。サンプルをどうぞ。

```
$table->drop_foreign('posts_user_id_foreign');
```

注意：外部キーで参照されるフィールドは自動増分項目であり、そのため自動的に **unsigned integer** になります。ですから、外部キーのフィールドは **unsigned()** で作成し、両方共に同じタイプであることを確認してください。さらに、両方のテーブルはエンジンに **InnoDB** をセットしていること、参照されるテーブルは、外部キーのテーブルの前に作成することも確実に行ってください。

```
$table->engine = 'InnoDB';
```

```
$table->integer('user_id')->unsigned();
```

# 34 マイグレーション

## 34.1 基本

データベースのバージョンコントロールを行うマイグレーションを考えてみましょう。あなたは開発チームにあなたも関わっていること、そしてローカル開発環境に全てを整えたことを伝えるましょう。エリックのやつがデータベースに変更を加え、新しく追加したカラムを使用するコードをチェックインしました。あなたはそのコードをプルし、アプリケーションは動かなくなります。だって、あなたは新しいカラムを作っていませんからね。どう対処しましょう？マイグレーションが答えです。もっと深く掘り下げ、どうやって使用するか見てみましょう！

## 34.2 データベースの準備

マイグレーションを始める前に、データベースに幾らかの準備を行う必要があります。Laravel は実行されたマイグレーションの記録を保持するために特別のテーブルを使用しています。このテーブルを作成するには、Artisan コマンドラインを使うだけです。

**Laravel** マイグレーションテーブルを作成

```
php artisan migrate:install
```

## 34.3 マイグレーションの作成

Laravel の”Artisan”CLI を使い簡単にマイグレーションを作成できます。次のように行います：

マイグレーションの作成

```
php artisan migrate:make create_users_table
```

では、**application/migrations** フォルダを調べてみましょう。新しいマイグレーションが見つかるはずです！ファイル名にはタイプスタンプも含まれていることに注目してください。これで Laravel は正しい順序でマイグレーションを実行できます。

You may also create migrations for a bundle.

バンドルのマイグレーションを作成

```
php artisan migrate:make bundle::create_users_table
```

参照：

- [スキーマビルダー](#)

## 34.4 マイグレーションの実行

実行されていないアプリケーションとバンドルのマイグレーションを全て実行

```
php artisan migrate
```

アプリケーションの未実行なマイグレーションを全部実行

```
php artisan migrate application
```

バンドルの未実行なマイグレーションを全部実行

```
php artisan migrate bundle
```

## 34.5 ロールバック

マイグレーションをロールバックすると、Laravel はそのマイグレーション「操作」全体をロールバックします。ですから、もし最後のマイグレーションコマンドで 122 個のマイグレーションを実行していたならば、その 122 個全部がロールバックされます。

最後のマイグレーション操作をロールバック

```
php artisan migrate:rollback
```

まだ実行していないすべてのマイグレーションを全部ロールバック

```
php artisan migrate:reset
```

全てをロールバックし、続けて全部マイグレーションをやり直す

```
php artisan migrate:rebuild
```

# 35 Redis

## 35.1 基本

Redis<sup>1</sup> is an open source, advanced key-value store. Redis はキーに文字列<sup>2</sup>、ハッシュ<sup>3</sup>、リスト<sup>4</sup>、セット<sup>5</sup>、ソート済みセット<sup>6</sup>で構成できるため、データ構造サーバーとして多く参照されます。

## 35.2 設定

アプリケーションの Redis 設定は `application/config/database.php` ファイルの中です。このファイルの `redis` 配列は Redis サーバーの設定で構成されています。

```
'redis' => array(

    'default' => array('host' => '127.0.0.1', 'port' => 6379),

),
```

デフォルトの設定で開発には充分でしょう。しかし、環境に合わせてこの配列を自由に変更してください。シンプルに、Redis サーバーの名前と、ホスト、ポート番号を指定します。

## 35.3 使用法

Redis クラスの `db` メソッドを呼び出し、Redis インスタンスを取得します。

```
$redis = Redis::db();
```

これでデフォルトの Redis サーバーのインスタンスが得られます。Redis 設定で定義したサーバーを指定するために `db` メソッドに名前を渡すこともできます。

```
$redis = Redis::db('redis_2');
```

素晴らしい！これで Redis インスタンスが入手できました。このインスタンスでどんな Redis コマンド<sup>7</sup>も発行できます。Laravel では、Redis サーバーにコマンドを渡すために、マジックメソッドを使用しています。

---

<sup>1</sup><http://redis.io>

<sup>2</sup><http://redis.shibu.jp/commandreference/strings.html>

<sup>3</sup><http://redis.shibu.jp/commandreference/hashtes.html>

<sup>4</sup><http://redis.shibu.jp/commandreference/lists.html>

<sup>5</sup><http://redis.shibu.jp/commandreference/sets.html>

<sup>6</sup><http://redis.shibu.jp/commandreference/sortedsets.html>

<sup>7</sup><http://redis.io/commands>

```
$redis->set('name', 'Taylor');  
  
$name = $redis->get('name');  
  
$values = $redis->lrange('names', 5, 10);
```

引数をコマンドにマジックメソッドで渡していることに注目してください。もちろん、マジックメソッドを使用しなければならないわけではなく、サーバーにコマンドを渡すのには **run** メソッドも使用できます。

```
$values = $redis->run('lrange', array(5, 10));
```

デフォルト Redis サーバーでコマンドを実行したいだけですか？Redis クラスの静的マジックメソッドが使用できます。

```
Redis::set('name', 'Taylor');  
  
$name = Redis::get('name');  
  
$values = Redis::lrange('names', 5, 10);
```

注目：Redis を使用する **キャッシュ** と **セッション** のドライバーは **Laravel** に含まれています。



# 36 キャッシュ設定

## 36.1 基本

あなたのアプリケーションでユーザーによって投票された、人気ソングトップ10を表示していると想像してください。誰かがあなたのサイトを訪れるたびに10曲を探すのは、ほんとうに必要ですか？10分ごと、もしくは1時間ごとに保存し、それを表示するとしたら、ドラマティックなスピードアップが望めると思いませんか？Laravelのキャッシュはシンプルに実現します。

Laravelは5つのキャッシュドライバを提供しています。

- ファイルシステム
- データベース
- Memcached
- APC
- Redis
- メモリー(配列)

Laravelはデフォルト設定でファイルシステムキャッシュドライバを使用します。これは最初から、設定なしで利用できます。ファイルシステムドライバは **cache** ディレクトリの中にファイルとしてアイテムを保存します。もし、このドライバに満足したならば、他の設定は必要ないでしょう。もう、使用を開始する準備が済んでいます。

ファイルシステムキャッシュドライバを使用する前に、**storage/cache**ディレクトリーが書き込めるようにしてください。

## 36.2 データベース

データベースキャッシュドライバはデータベーステーブルをシンプルなキー／値の保存場所として使用します。使い始めるには、最初に **application/config/cache.php**の中にデータベーステーブルの名前を設定してください。

```
'database' => array('table' => 'laravel_cache'),
```

次にデータベーステーブルを作成します。テーブルに次の3カラムを作成してください。

- key (varchar)

- value (text)
- expiration (integer)

これだけです！一度設定を行い、テーブルを用意したら、キャッシュを使う準備ができました。

## 36.3 Memcached

[Memcached](#)<sup>1</sup>はウィキペディアやフェイスブックのようなサイトで使用されている究極な速さの、メモリーオブジェクトキャッシュシステムであり、オープンソースで配布されています。Laravel の Memcached ドライバーを使用し始める前に、Memcached をインストール、設定します。それと PHP Memcache 拡張をサーバーに入れておく必要があります。

一度、Memcached がインストールされたら、`application/config/cache.php`ファイルの **driver**を設定してください。

```
'driver' => 'memcached'
```

それから、**servers**に Memcached サーバーを付け加えます。

```
'servers' => array(
    array('host' => '127.0.0.1', 'port' => 11211, 'weight' => 100),
)
```

## 36.4 Redis

[Redis](#)<sup>2</sup>はオープンソースの、進化したキー／値のデータストアです。Redis はキーに文字列<sup>3</sup>、ハッシュ<sup>4</sup>、リスト<sup>5</sup>、セット<sup>6</sup>、ソート済みセット<sup>7</sup>で構成できるため、データ構造サーバーとして多く参照されます。

Redis キャッシュドライバを使用する前に、[Redis サーバーの設定](#)を行なってください。そうしたら、`application/config/cache.php`の中の **driver**を設定します。

```
'driver' => 'redis'
```

### キャッシュキー

APC や Redis、Memcached サーバーを使用している他のアプリケーションとの間に名前の衝突が起きるのを防ぐため、Laravel はこれらのドライバーを使っているキャッシュの中で、各アイテムに **key**を付加しています。自由にこの値を変更してください。

---

<sup>1</sup><http://memcached.org>

<sup>2</sup><http://redis.io>

<sup>3</sup><http://redis.shibu.jp/commandreference/strings.html>

<sup>4</sup><http://redis.shibu.jp/commandreference/hashtes.html>

<sup>5</sup><http://redis.shibu.jp/commandreference/lists.html>

<sup>6</sup><http://redis.shibu.jp/commandreference/sets.html>

<sup>7</sup><http://redis.shibu.jp/commandreference/sortedsets.html>

```
'key' => 'laravel'
```

## オンメモリキャッシュ

「メモリー」キャッシュドライバーはディスクに何もキャッシュしません。現在のリクエストに対して、キャッシュデータをただの内部配列として管理します。これは、他のストレージメカニズムから独立して、アプリケーションをテストするためにピッタリです。決して、「本当」のキャッシュドライバーとしては使用しないでください。

## 37 キャッシュ使用法

### 37.1 アイテムの保存

キャッシュにアイテムを保存するのは、実に簡単です。Cache クラスの **put** メソッドをただ呼び出してください。

```
Cache::put('name', 'Taylor', 10);
```

最初の引数はキャッシュされるアイテムのキーです。このキーを使い、キャッシュからアイテムを取得します。2つ目の引数は値です。3つ目の引数は、アイテムがキャッシュに保存される分数です。

キャッシュに有効期限を付けない場合は、”forever” を使ってください。

```
Cache::forever('name', 'Taylor');
```

キャッシュにアイテムを保存するときにシリアライズの必要はありません。

### 37.2 アイテムの取得

キャッシュからアイテムを取得するのは、保存するより簡単です。**get** メソッドを使います。ただ、取得したいアイテムのキーを指定してください。

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

アイテムの有効期限が過ぎているか、存在していない場合、デフォルトとして **NULL** 値が返されます。ですが、他の値を第2引数に指定することで、デフォルト値を指定できます。

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

これで、”name” キャッシュアイテムの期限が切れていたり、存在していない場合、”Fred” がリターンされます。

キャッシュアイテムが存在しない場合、データベースから値を取りたい時はどうしましょう？解決策はシンプルです。**get** メソッドのデフォルト値に無名関数を渡してください。無名関数は、キャッシュアイテムが存在していない場合のみ、実行されます。

```
$users = Cache::get('count', function() {return DB::table('users')->count()\n;});
```

この例を、もうちょっと進めてみましょう。アプリケーションの登録済みユーザー数を取得することをイメージしてください。もし、値がキャッシュされていなければ、デフォルト値をキャッシュに保存したい場合は、**remember**メソッドを使います。

```
$users = Cache::remember('count', function() {return DB::table('users')->count();}, 5);
```

この例を解説しましょう。**count**アイテムがキャッシュに存在するならば、その値が返されます。もし存在しなければ、無名関数の結果がキャッシュに5分間保存され、かつメソッドの戻り値になります。絶妙でしょう？

Laravel は **has**メソッドでキャッシュアイテムが存在しているかを確認するシンプルな方法も提供しています。

```
if (Cache::has('name'))\n{\n    $name = Cache::get('name');\n}
```

## 37.3 アイテムの削除

キャッシュアイテムを削除したい？問題ありません。**forget**メソッドにアイテムの名前を指定してください。

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

# 38 セッション設定

## 38.1 基本

Web はステートレスな環境です。つまり、アプリケーションに対する毎回のリクエストは、前回のリクエストと関係が無いことを意味しています。しかし、セッションは任意のデータをあなたのアプリケーションの各訪問者毎に任意のデータを保持してくれます。それぞれの訪問者に対するセッションデータは Web サーバーに保存され、同時にセッション ID が訪問者のコンピューターのクッキーに保存されます。クッキーはそのユーザーのセッションをアプリケーションに「思い出させ」、その後に続くリクエストでセッションを取得させてくれる役目をします。

注目：セッションを使い始める前に、`application/config/application.php` ファイルのアプリケーションキーを指定してください。

6 つのセッションドライバが用意されています。

- クッキー
- ファイルシステム
- データベース
- Memcached
- Redis
- メモリー(配列)

## 38.2 クッキーセッション

クッキーベースのセッションはライトウェイトで早いセッション情報の保持メカニズムです。しかも、セキュアです。それぞれのクッキーは、強力な AES-256 を使用して暗号化されます。しかしながら、クッキーは 4 Kb の容量しか無いため、より多くの情報をセッションに保存したい場合、他のドライバを使ってください。

クッキーセッションを使い始める場合は、`application/config/session.php` ファイルの `driver` オプションをセットするだけです。

```
'driver' => 'cookie'
```

## 38.3 ファイルシステムセッション

ほとんどの場合、ファイルシステムセッションはあなたのアプリケーションで上手く機能するでしょう。しかしながら、非常に多くのトラフィックを扱うか、サーバーの形態で運用する場合は、データベースか、Memchach セッションを使ってください。

ファイルシステムセッションを使い始める場合は、`application/config/session.php`ファイルの `driver` オプションをセットするだけで済みます。

```
'driver' => 'file'
```

これだけです。準備完了です！

注目：ファイルシステムは `storage/sessions`ディレクトリーに保存されます。ですから、書き込み可能に設定してください。

## 38.4 データベースセッション

データベースセッションを使い始める前、最初に[データベース接続の設定](#)が必要です。

次にセッションテーブルを作成する必要があります。以降に、参考になるように SQL 文を紹介します。しかしながら、Laravel の”Artisan” コマンドラインでテーブルの生成もできます！

A r t i s a n

```
php artisan session:table
```

S Q L i t e

```
CREATE TABLE "sessions" (  
    "id" VARCHAR PRIMARY KEY NOT NULL UNIQUE,  
    "last_activity" INTEGER NOT NULL,  
    "data" TEXT NOT NULL  
);
```

M y S Q L

```
CREATE TABLE `sessions` (  
    `id` VARCHAR(40) NOT NULL,  
    `last_activity` INT(10) NOT NULL,  
    `data` TEXT NOT NULL,  
    PRIMARY KEY (`id`)  
);
```

もし別のテーブル名を使用したいのであれば、`application/config/session.php`ファイルの `table` オプションを変更してください。

```
'table' => 'sessions'
```

後、行うべきなのは `application/config/session.php` でドライバーの設定です。

```
'driver' => 'database'
```

## 38.5 Memcache セッション

Memcache セッションを使い始める前に、[Memcache サーバーの設定](#)を行なってください。

後は、`applicaton/config/session.php` ファイルでドライバーを設定します。

```
'driver' => 'memcached'
```

## 38.6 Redis セッション

Redis セッションを使い始める前に、[Redis サーバーの設定](#)を行なってください。

後は、`applicaton/config/session.php` ファイルでドライバーを設定します。

```
'driver' => 'redis'
```

## 38.7 オンメモリセッション

“memory” セッションドライバーは現在のリクエストのため、ただ配列に保存するだけです。このドライバーはディスクに何も書き込みませんから、アプリケーションのユニットテストにピッタリです。これは「本当」のセッションドライバーではありません。



## 39 セッション使用法

### 39.1 アイテムの保存

セッションにアイテムを保存するには、Session クラスの `put` メソッドを使います。

```
Session::put('name', 'Taylor');
```

最初の引数は、セッションアイテムのキーです。このキーを指定してセッションからアイテムを取得します。2つ目の引数は、そのアイテムの値です。

### 39.2 アイテムの取得

フラッシュデータも含め、セッションからアイテムを取得するには、Session クラスの `get` メソッドを使用します。取得したいアイテムのキーを渡してください。

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

セッションアイテムが存在しない場合、デフォルトでは `NULL` 値が返されます。ですが、`get` メソッドの第2引数に、デフォルト値を指定することもできます。

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

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

これで”name”アイテムがセッションに存在しない場合、”Fred” がリターンされます。

Laravel はセッションアイテムが存在しているか確認するために `has` メソッドも提供しています。

```
if (Session::has('name'))
{
    $name = Session::get('name');
}
```

### 39.3 アイテムの削除

セッションからアイテムを削除するには、Session クラスの `forget` メソッドを使います。

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

さらに、セッションからすべてのアイテムを削除したい場合は、**flush**メソッドを使ってください。

```
Session::flush();
```

## 39.4 フラッシュアイテム

**flash**メソッドは、次のセッションの後に消去されるアイテムを保存します。これは、ステータスメッセージやエラーメッセージのような一時的なデータを保存するのに便利です。

```
Session::flash('status', 'Welcome Back!');
```

フラッシュアイテムは次のリクエストで消えますが、**reflash**か **keep**メソッドで、その先のリクエストまで保持することも可能です。

全てのアイテムを再度保持する：

```
Session::reflash();
```

特定のアイテムを保持する：

```
Session::keep('status');
```

複数のアイテムを保持する：

```
Session::keep(array('status', 'other_item'));
```

## 39.5 再生成

時々、セッション ID を「再生成」したい場合もあるでしょう。これはランダムな新しいセッション ID が、そのセッションに与えられることをシンプルに意味します。実例をどうぞ。

```
Session::regenerate();
```

## 40 認証設定

### 40.1 基本

ほとんどの対話型アプリケーションはユーザーをログイン／ログアウトさせます。Laravel はユーザー認証し、現在のユーザーに関する情報を取得できるシンプルなクラスを提供しています。

初めましょう。**application/config/auto.php**をご覧ください。認証設定は認証を使い始めるのに役に立つ基本的なオプションで構成されています。

### 40.2 認証ドライバー

Laravel の認証はドライバーベースです。これが意味するのは、認証するユーザーの取得に関する責任は、各種の「ドライバー」に任されているということです。Eloquent と Fluent は最初から用意されていますが、必要であれば自分でドライバーを自由に書いてください！

**Eloquent**ドライバーは Eloquent ORM をユーザー情報の取得に使用し、デフォルトになっています。**Fluent**ドライバーは Fluent クエリビルダーをユーザー情報の取得に使用します。

### 40.3 デフォルト「ユーザー名」

設定ファイル中、2つ目のオプションはユーザーのデフォルト「ユーザー名」を元に決定するかです。典型的にはデータベースの”users” テーブル中のカラムで、通常は”email” か”username” でしょう。

### 40.4 認証モデル

**Eloquent**認証ドライバーを使用する場合、このオプションはユーザーを読み込む際に使用する Eloquent モデルを決めます。

### 40.5 認証テーブル

**Fluent**認証ドライバーを使用する場合、このオプションは、アプリケーションのユーザーを構成するデータベーステーブルを決めます。

# 41 認証使用法

注目：Auth クラスを使用する前に、セッションドライバを設定する必要があります。

## 41.1 ソルトとハッシュ

Auth クラスを使用する場合、すべてのパスワードにハッシュとソルトを使用するように、強く推奨します。Web の開発は、責任をもって行われるべきです。ソルトとハッシュはユーザーのパスワードに対するレインボーテーブル攻撃の効力を無くします。

パスワードのソルトとハッシュは **Hash** クラスを使用します。Hash クラスは暗号化 (**bcrypt**) ハッシュアルゴリズムを使用します。例を確認してください。

```
$password = Hash::make('secret');
```

Hash クラスの **make** メソッドは、60 文字のハッシュ文字列を返します。

Hash クラスの **check** メソッドを使えば、ハッシュされていない値とハッシュされた値を比較することができます。

```
if (Hash::check('secret', $hashed_value))
{
    return 'The password is valid!';
}
```

## 41.2 ログイン

アプリケーションのログイン処理は、Auth クラスの **attempt** メソッドを使ってください。ただ、ユーザー名とパスワードをメソッドに渡すだけです。ログインに必要な内容は配列で渡します。柔軟性を高めるため、ドライバ毎に必要な引数の数は異なります。ログイン内容が有効であれば、**true** が返されます。そうでなければ、**false** です。

```
$credentials = array('username' => 'example@gmail.com', 'password' => 'secret');

if (Auth::attempt($credentials))
{
    return Redirect::to('user/profile');
}
```

ユーザーのログイン内容が有効であった場合、ID はセッションに補完され、その後続くアプリケーションへのリクエストで、「ログイン」状態として取り扱われます。

アプリケーション中でユーザーがログインしているかを判断するには、**check**メソッドを使用します。

```
if (Auth::check())
{
    return "You're logged in!";
}
```

ユーザーが登録した後などに、チェックを行わずログインさせる場合は、**login**メソッドを使ってください。Just pass the user's ID:

```
Auth::login($user->id);
```

```
Auth::login(15);
```

## 41.3 ルートの保護

ログイン済みのユーザーだけに特定のルートへアクセスさせるのは、一般的です。Laravel では、**auth フィルター**でこれを実現しています。ユーザーがログインしていれば、通常通りリクエストは処理されます。しかし、ユーザーがログインしていなければ、“login” という**名前付きルート**へリダイレクトされます。

ルートを保護するためには、**auth** フィルターを付け加えるだけです。

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

注目：お好きなように、**auth** フィルターは編集できます。**application/routes.php**にデフォルトの実装があります。

## 41.4 ログインしたユーザーの取得

一度ユーザーがアプリケーションにログインしたら、Auth クラスの **user** メソッドを使用し、ユーザーモデルにアクセスできます。

```
return Auth::user()->email;
```

注目：もし、ユーザーがログインしてなければ、**user**メソッドは NULL 値を返します。

## 41.5 ログアウト

ユーザーをアプリケーションからログアウトさせる用意ができましたか？

```
Auth::logout();
```

このメソッドはセッションからユーザーID を取り除き、以降のアプリケーションに対するリクエストで、ユーザーはログアウトしているとして扱われます。

## 42 Artisan コマンド

### 42.1 Help

| 説明   | コマンド                                   |
|--|--|
| View a list of available artisan commands. | <code>php artisan help:commands</code> |

### 42.2 アプリケーション設定 (追加情報)

| 説明   | コマンド                                  |
|--|---------------------------------------|
| セキュアなアプリケーションキーを生成<br><code>config/application.php</code> の application key が空文字列の場合、キーを生成する | <code>php artisan key:generate</code> |

### 42.3 データベースセッション (追加情報)

| 説明           | コマンド                                   |
|--------------|--|
| セッションテーブルの生成 | <code>php artisan session:table</code> |

### 42.4 マイグレーション (追加情報)

| 説明                        | コマンド   |
|---------------------------|--|
| Laravel マイグレーションテーブルを生成   | <code>php artisan migrate:install</code>                 |
| マイグレーションを生成               | <code>php artisan migrate:make create_users_table</code> |
| バンドルのマイグレーションを生成          | <code>php artisan migrate:make bundle::tablename</code>  |
| 未実行のマイグレーションを実施           | <code>php artisan migrate</code>                         |
| アプリケーションの未実行なマイグレーションを実施  | <code>php artisan migrate application</code>             |
| バンドルの未実行なマイグレーションを実施      | <code>php artisan migrate bundle</code>                  |
| 最後のマイグレーション操作をロールバック      | <code>php artisan migrate:rollback</code>                |
| 今まで実行した全てのマイグレーションをロールバック | <code>php artisan migrate:reset</code>                   |

### 42.5 バンドル (追加情報)

| 説明              | コマンド  |
|-----------------|---|
| バンドルをインストール     | <code>php artisan bundle:install eloquent</code>    |
| バンドルをアップデート     | <code>php artisan bundle:upgrade eloquent</code>    |
| 全てのバンドルをアップデート  | <code>php artisan bundle:upgrade</code>             |
| バンドルのアセットを公開    | <code>php artisan bundle:publish bundle_name</code> |
| 全てのバンドルのアセットを公開 | <code>php artisan bundle:publish</code>             |

インストール後には、[バンドルの登録](#)<sup>a</sup>が必要です。

<sup>a</sup>[../bundles/#registering-bundles](#)

## 42.6 タスク (追加情報)

| 説明             | コマンド  |
|----------------|---|
| タスクの呼び出し       | <code>php artisan notify</code>               |
| 引数を渡し、タスクの実行   | <code>php artisan notify taylor</code>        |
| タスクの特定なメソッドを実行 | <code>php artisan notify:urgent</code>        |
| バンドルのタスクを実行    | <code>php artisan admin::generate</code>      |
| バンドルの特定メソッドを実行 | <code>php artisan admin::generate:list</code> |

## 42.7 ユニットテスト (追加情報)

| 説明              | コマンド                                      |
|-----------------|---|
| アプリケーションのテストを実行 | <code>php artisan test</code>             |
| バンドルのテストを実行     | <code>php artisan test bundle-name</code> |

## 42.8 ルーティング (追加情報)

| 説明       | コマンド   |
|----------|--|
| ルートを呼び出す | <code>php artisan route:call get api/user/1</code> |

注目 : get を post、put、delete、その他なんでも置き換えられます。

## 42.9 アプリケーションキー

| 説明            | コマンド                                  |
|---------------|---------------------------------------|
| アプリケーションキーを生成 | <code>php artisan key:generate</code> |



違った長さをコマンドの追加引数で渡すこともできます。

## 42.10 C L I オプション

| 説明                   | コマンド   |
|----------------------|--|
| Laravel の環境をセットする    | <code>php artisan foo --env=local</code>           |
| デフォルトのデータベース接続をセットする | <code>php artisan foo --database=sqlitename</code> |

## 43 タスク

### 43.1 基本

Laravel のコマンドラインツールは Artisan です。Artisan でマイグレーション、クエリジョブ、ユニットテストなどの「タスク」をなんでも実行できます。

### 43.2 タスクの作成と実行

タスクを作成するには、`application/tasks`ディレクトリーに新しいクラスを書きます。クラスの名前はサフィックスとして”\_Task”を付け、”run”メソッドを含まなくてはなりません。このように：

タスククラスを作成

```
class Notify_Task {  
  
    public function run($arguments)  
    {  
        // 素晴らしいことを知らせる…  
    }  
  
}
```

それから、コマンドラインでタスクのメソッドを”run”するために呼び出します。引数も渡せます

コマンドラインからタスクを呼び出す

```
php artisan notify
```

引数を渡し、タスクを呼び出す

```
php artisan notify taylor
```

アプリケーションからタスクを呼び出す

```
Command::run(array('notify'));
```

引数を渡しアプリケーションからタスクを呼び出す

```
Command::run(array('notify', 'taylor'));
```

タスクに存在する特定のメソッドを呼び出せることを覚えておいてください。Notify タスクに urgent メソッドを追加しましょう。

### タスクにメソッドを追加

```
class Notify_Task {

    public function run($arguments)
    {
        // 素晴らしいことを知らせる…
    }

    public function urgent($arguments)
    {
        // これは緊急だ！
    }

}
```

これで”urgent” メソッドを呼び出せます。

### タスクの特定メソッドを呼び出す

```
php artisan notify:urgent
```

## 43.3 バンドルタスク

バンドルにタスクを作成するには、クラス名の先頭にタスク名を付けます。ですから、バンドル名が”admin”でしたら、タスクはこんなふうになるでしょう。

### バンドルに所属するタスクを作成

```
class Admin_Generate_Task {

    public function run($arguments)
    {
        // admin を作成！
    }

}
```

タスクを実行するには、いつもどおり二重のコロン構文でバンドルを指定します。

バンドル所属のタスクを実行

```
php artisan admin::generate
```

バンドルに所属するタスクの中の、特定メソッドを実行

```
php artisan admin::generate:list
```

## 43.4 C L I オプション

L a r a v e l 環境を設定

```
php artisan foo --env=local
```

デフォルトデータベース接続を設定

```
php artisan foo --database=sqlite
```

# 44 G i t H u b の L a r a v e l

## 44.1 基本

Laravel の開発とソース管理は GitHub で行われているので、誰でも貢献することが可能です。誰でもバグを修正し、機能を追加し、ドキュメントを改善できます。

プロジェクトに修正が提出されると、Laravel チームはその変更をレビューし、Laravel のコアに取り入れるか決定します。

## 44.2 リポジトリ

Laravel の GitHub のホームは[github.com/laravel](https://github.com/laravel)<sup>1</sup>です。Laravel has several repositories. 基本的には、**laravel**リポジトリ、だけに注意を払っておけば十分です。

## 44.3 ブランチ

Laravelリポジトリは複数のブランチを持っており、それぞれ別の目的を持っています。

- **master** - これは Laravel のリリースブランチです。このブランチでは開発中のものを含みません。このブランチには最新で、安定版の Laravel コアコードだけです。Laravel を[laravel.com](https://laravel.com)<sup>2</sup>からダウンロードする時、このマスターブランチから直接ダウンロードしているのです。このブランチに対しては、プルリクエストをしてはいけません。
- **develop** - これは開発作業用のブランチです。コードの変更やコミュニティによる貢献など全ての提案は、このブランチへ **pull** します。Laravel プロジェクトへプルリクエストをするのでしたら、このブランチへ希望するプルリクエストを行なってください。

一度、明確なマイルストーンに到達する、そして／もしくは Taylor Otwell と Laravel チームが現在の開発ブランチの安定性と追加機能にハッピーと感じたら、**develop**ブランチを、**master**ブランチへプルし、最新の安定版 Laravel を世界中で使ってもらうために、リリースします。

## 44.4 プルリクエスト

GitHub プルリクエスト<sup>3</sup>は Laravel のコミュニティの全員が Laravel のコードベースに貢献できる素晴らしい方法です。バグを発見しましたか？あなたのフォークで修正し、プルリクエストを送ってください。これはレビューされ、良ければメインリポジトリにマージされます。

---

<sup>1</sup><https://github.com/laravel>

<sup>2</sup><http://laravel.com/>

<sup>3</sup><https://help.github.com/articles/using-pull-requests>

多くの人々の貢献により、コードベースをきれいで安定したまま高品質に保つために、高品質なプルリクエストのガイドラインが必要とされます。

- ブランチ：旧バージョンに関連した、急ぎのドキュメント修正でない限り、プルリクエストは `develop` ブランチだけに送ってください。プルリクエストを送るときは、このブランチがターゲットとして選ばれているか確認してください。（GitHub は自動的に選びません。）
- ドキュメント：新しい機能を追加する、または API を適切に変更する場合は、ドキュメントも修正してください。ドキュメントのファイルはコアリポジトリに含まれています。
- ユニットテスト：古いバグが再現したり、全体的な品質を高く保つため、Laravel のコアはユニットテストを通しています。ですから、プルリクエストを作成するときには、あなたの追加した新しいコードについてユニットテストが期待されています。どんなバグ修正であっても、そのバグが再度発生しないことを確実にするために、リグレッションテストを追加してください。どうテストを書いたら良いのかよくわからないときは、コアチームもしくは他のコントリビューターの皆さんが快くお手伝いします。

#### 参照

- [コマンドラインで Laravel に貢献する](#)
- [TortoiseGit で Laravel に貢献する](#)

## 45 コマンドラインでLaravelに貢献する

### 45.1 始めよう

このチュートリアルはコマンドラインを使用し、[GitHub<sup>1</sup>](#)のプロジェクトに貢献する基礎を説明しています。ワークフローはGitHub上のほとんどのプロジェクトに対して適用できますが、今回は特に[Laravel<sup>2</sup>](#)プロジェクトに焦点をあわせていきます。このチュートリアルはOSX、Linux、Windowsに適用しています。

このチュートリアルの前提は[Git<sup>3</sup>](#)がインストール済みで、[GitHub<sup>4</sup>](#)にアカウントを作成していることです。もし、まだご覧になっていなければ、[GitHubのLaravelドキュメント](#)をご覧ください。Laravelのリポジトリとブランチを理解しておいてください。

### 45.2 LaravelをForkする

GitHubにログインし、[Laravelリポジトリ<sup>5</sup>](#)を訪れてください。**Fork**ボタンをクリックします。これにより、あなた自身のGitHubアカウントにLaravelのフォーク（プロジェクトの分岐）が作成されます。あなたのLaravelフォークは<https://github.com/username/laravel>に置かれます。（*username*はあなたのGitHubユーザー名に置き換えてください。）

### 45.3 Laravelをクローンする

コマンドラインか端末を開き、Laravelの開発修正を行う新しいディレクトリを作成します。

```
# mkdir laravel-develop
# cd laravel-develop
```

次に、Laravelリポジトリをクローンします。（あなたの作成したフォークではありません。）

```
# git clone https://github.com/laravel/laravel.git .
```

注目：（あなたの作成したフォークではなく）Laravelのオリジナルリポジトリをクローンした理由は、あなたのローカルリポジトリにLaravelのリポジトリから最新の変更をいつでも取り込める

---

<sup>1</sup><https://github.com/>

<sup>2</sup><https://github.com/laravel/laravel>

<sup>3</sup><http://git-scm.com/>

<sup>4</sup><https://github.com/signup/free>

<sup>5</sup><https://github.com/laravel/laravel>

(pull) ようにするためです。

## 45.4 Forkを追加する

次に、あなたの作成したフォークをリモートリポジトリとして追加します。

```
# git remote add fork git@github.com:username/laravel.git
```

**username**をあなたの GitHub ユーザー名に置き換えるのを忘れないでください。大文字・小文字も間違えないように。追加したあなたのフォークを確認するには：

```
# git remote
```

これで、Laravel リポジトリの真新しいクローンしたのに加え、あなたのフォークをリモートリポジトリにできました。新しい機能を付け加えたり、バグを修正する用意ができました。

## 45.5 ブランチを作成する

最初に、**develop** ブランチで作業していることを確認してください。もし **master** ブランチに変更を送信してしまうと、残念なことに、近い将来取り除かれてしまいます。この理由については[GitHub の Laravel](#)をお読みください。**develop** ブランチに切り替えるには：

```
# git checkout develop
```

次に、最新の Laravel リポジトリにアップデートしてください。もし新しい機能やバグの修正をあなたのクローンした Laravel プロジェクトに付け加えるのであれば、それらの変更を確実にローカルリポジトリに全て含めてください。これが重要なステップなのは、あなたのフォークの代わりに、オリジナルの Laravel リポジトリからクローンしているからです。

```
# git pull origin develop
```

では、新しい機能がバグ修正のために、新しいブランチを作成し、準備をしましょう。新しいブランチを作成するときは、便利ですからわかりやすい名前をつけてください。例えば、Eloquent のバグを修正するのでしたら、*bug/eloquent*と名づけましょう。



```
# git branch bug/eloquent
# git checkout bug/eloquent
Switched to branch 'bug/eloquent'
```

もしくは、例えばローカライズのためのドキュメントのように、作成するものに対する新しいドキュメントを付け加えたり、変更をしたいのであれば：

```
# git branch feature/localization-docs
# git checkout feature/localization-docs
Switched to branch 'feature/localization-docs'
```

注意：新しい機能やバグフィックス毎に、新しいブランチを作成してください。これは組織に協力することです。機能／修正が限定的で独立していれば、Laravel チームはコアに変更を取り込みやすくなります。

これで、自分のブランチを作成し、それに切り替えました。では、コアの変更にとりかかりましょう。新しい変更を付け加えるか、バグを修正してください。

## 45.6 コミットする

コーディングと変更に対するテストが終了したら、次にローカルリポジトリにコミットしましょう。最初に、追加／修正したファイルを `add` します。

```
# git add laravel/documentation/localization.md
```

次に、リポジトリに変更をコミットします。

```
# git commit -s -m "I added some more stuff to the Localization documentati\
on."
```

- `-s` はあなたの名前でコミットを承認することを意味します。これは Laravel チームに Laravel コアにあなたのコードを付け加えることに、あなたが個人的に同意していると、伝えることになります。
- `** -m` はコミットする内容のメッセージです。何を付け加えたのか、修正したのかを簡単に説明します。

## 45.7 フォークにプッシュする

これであなたのローカルリポジトリに変更がコミットされました。次に、GitHub のあなたのフォークに対し、新しいブランチを `push` (もしくは `sync`) します。

```
# git push fork feature/localization-docs
```

これで GitHub のあなたのフォークにブランチが push されました。

## 45.8 プルリクエストを送る

最後のステップは Laravel リポジトリにプルリクエストを送ることです。これは Laravel チームに対し、あなたの Laravel コアの修正を pull、merge するようにリクエストすることを意味します。ブラウザであなたのフォーク、<https://github.com/username/laravel><sup>6</sup>を訪れてください。Pull Requestをクリックしてください。次は、慎重にリポジトリのベースとヘッド、ブランチを選択します。

- **base repo:** laravel/laravel
- **base branch:** develop
- **head repo:** username/laravel
- **head branch:** feature/localization-docs

なぜあなたがこの変更を作成したのか、更に細かい説明をフォームに書き入れてください。最後に、Send pull requestをクリックします。これでおしまいです！変更は Laravel チームに送られました。

## 45.9 次は何？

付け加えたい新しい機能が修正したいバグが、他にもありますか？最初に、いつもあなたの新しいブランチの base は develop ブランチから確実に初めます。

```
# git checkout develop
```

それから、Laravel のリポジトリから最新の変更を pull しましょう。

```
# git pull origin develop
```

これで、新しいブランチを作成し、再度コーディングをスタートする用意ができました！

Jason Lewis<sup>a</sup>のブログポスト、GitHub プロジェクトに貢献する<sup>b</sup>がこの記事の主要なヒントとなりました。

<sup>a</sup><http://jasonlewis.me/>

<sup>b</sup><http://jasonlewis.me/blog/2012/06/how-to-contributing-to-a-github-project>

<sup>6</sup><https://github.com/username/laravel>

# 46 T o r t o i s e G i t で L a r a v e l に 貢献する

## 46.1 始めよう

このチュートリアルは[GitHub](https://github.com/)<sup>1</sup>上のプロジェクトに Windows の[TortoiseGit](https://code.google.com/p/tortoisegit/)<sup>2</sup>を用い貢献する方法を説明しています。ワークフローは GitHub 上のほとんどのプロジェクトに対して適用できますが、今回は特に[Laravel](https://github.com/laravel/laravel)<sup>3</sup>プロジェクトに焦点をあわせていきます。

このチュートリアル的前提は TortoiseGit for Windows がインストール済みで、[GitHub account](https://github.com/)<sup>4</sup>にアカウントを作成していることです。（翻訳注：以降の TortoiseGit の説明は公式サイトで配布されている日本語言語ファイルインストーラーを適用し、言語表示を日本語に設定している前提で翻訳しています。）もし、まだご覧になっていなければ、[GitHub の Laravel](https://github.com/laravel/laravel)ドキュメントをご覧になり、Laravel のリポジトリとブランチを理解しておいてください。

## 46.2 L a r a v e l を F o r k する

GitHub にログインし、[Laravel リポジトリ](https://github.com/laravel/laravel)<sup>5</sup>を訪れてください。**Fork**ボタンをクリックします。これにより、あなた自身の GitHub アカウントに Laravel のフォーク（プロジェクトの分岐）が作成されます。あなたの Laravel フォークは <https://github.com/username/laravel>に置かれます。（*username*はあなたの GitHub ユーザー名に置き換えてください。）

## 46.3 L a r a v e l をクローンする

Windows のエクスプローラーを開き、Laravel への変更を開発するための、新しいディレクトリを作成してください。

- Laravel のディレクトリを右クリックし、コンテキストメニューを開きます。**Git クローン...**をクリックします。
- Git clone
- **Url:** <https://github.com/laravel/laravel.git>
- ディレクトリ:直前のステップで作成したディレクトリ
- **OK**をクリック

---

<sup>1</sup><https://github.com/>

<sup>2</sup><http://code.google.com/p/tortoisegit/>

<sup>3</sup><https://github.com/laravel/laravel>

<sup>4</sup><https://github.com/signup/free>

<sup>5</sup><https://github.com/laravel/laravel>

注目：（あなたの作成したフォークではなく）Laravel のオリジナルリポジトリをクローンした理由は、あなたのローカルリポジトリに Laravel のリポジトリから最新の変更をいつでも取り込める (pull) ようにするためです。

## 46.4 Fork を追加する

クローンが終了したら、あなたのフォーク（プロジェクトの分岐）をリモートリポジトリとして追加しましょう。

- Laravel ディレクトリを右クリックし、**TortoiseGit** → 設定を選んでください。
- **Git**/リモートセクションを選びます。Add a new remote:
  - リモート: fork
  - URL: <https://github.com/username/laravel.git>
  - 新規に追加／保存をクリック
  - **OK**をクリック

**username**を自分の GitHub ユーザー名に置き換えるのを忘れなく大文字小文字は区別されます。

## 46.5 ブランチを作成する

では、新しい機能かバグ修正のために、新しいブランチを作成し、準備をしましょう。新しいブランチを作成するときは、便利ですからわかりやすい名前をつけてください。例えば、Eloquent のバグを修正するのでしたら、*bug/eloquent*と名づけましょう。もしくは、例えばローカライズのためのドキュメントを変更をしたいのであれば、*feature/localization-docs*と名付けます。良いネーミング規則は組織と強調し、あなたのブランチの目的を他の人が理解するのを助けます。

- Laravel ディレクトリを右クリックし、TortoiseGit → ブランチを作成を選んでください。
  - ブランチ: *feature/localization-docs*
  - 基点 → ブランチ: *remotes/origin/develop*
  - チェック 追跡
  - チェック 新しいブランチに切り替える
  - **OK**をクリック

これで新しい *feature/localization-docs* ブランチが作成され、これに切り替えられます。

注意：新しい機能やバグフィックス毎に、新しいブランチを作成してください。これは組織に協力することです。機能／修正が限定的で独立していれば、Laravel チームはコアに変更を取り込みやすくなります。

これで、自分のブランチを作成し、それに切り替えました。では、コアの変更にとりかかりましょう。新しい変更を付け加えるか、バグを修正してください。

## 46.6 コミット

変更部分のコーディングとテストを終えたら、次はローカルリポジトリにコミットします。：

- Laravel ディレクトリーを右クリックし、**Git** コミット -> “feature/localization-docs”...を選びます。
- コミット
- メッセージ：付け加えたもの、変更したものの簡単な説明
- **Click Sign** - これは Laravel チームに Laravel コアにあなたのコードを付け加えることに、あなたが個人的に同意していると、伝えることになります。
- 変更した項目：すべての追加／変更ファイルをチェックする
- **OK**をクリック

## 46.7 フォークにプッシュする

これであなたのローカルリポジトリに変更がコミットされました。次に、GitHub のあなたのフォークに対し、新しいブランチを **push**（もしくは **sync**）します。

- Laravel ディレクトリーを右クリックし、**Git** 同期…を選びます。
- **Git** 同期
- ローカルブランチ feature/localization-docs
- リモートブランチ 空白にする
- リモート fork
- プッシュをクリック
- “username:” を尋ねられたら、Github のユーザー名を大文字小文字そのままを入力します。
- “password:” を尋ねられたら、Github のパスワードを大文字小文字そのままを入力します。

これで GitHub のあなたのフォークにブランチが **push** されました。

## 46.8 プルリクエストを送る

最後のステップは Laravel リポジトリにプルリクエストを送ることです。これは Laravel チームに対し、あなたの Laravel コアの修正を pull、merge するようにリクエストすることを意味します。ブラウザであなたのフォーク、<https://github.com/username/laravel><sup>6</sup>を訪れてください。Pull Requestをクリックしてください。次は、慎重にリポジトリのベースとヘッド、ブランチを選択します。

- **base repo:** laravel/laravel
- **base branch:** develop
- **head repo:** username/laravel
- **head branch:** feature/localization-docs

なぜあなたがこの変更を作成したのか、更に細かい説明をフォームに書き入れてください。最後に、Send pull requestをクリックします。これでおしまいです！変更は Laravel チームに送られました。

## 46.9 次は何？

付け加えたい新しい機能が修正したいバグが、他にもありますか？前の**ブランチを作成する**セクションの指示と同じことを行なってください。全ての機能追加や修正ごとに新しいブランチを作成すること、そしていつでも新しいブランチは *remotes/origin/develop* ブランチから作成することを覚えておいてください。

---

<sup>6</sup><https://github.com/username/laravel>

# おまけ

Laravel 公式ドキュメント電子書籍版を購入された方に、おまけとして PHPDoc のコメント形式部分を翻訳した、コアのソースファイルを添付します。

翻訳の品質は、一度通しで翻訳し終わった状態です。コースの内容を理解しながら翻訳したわけではなく、ただコメントを訳しただけです。そのため、翻訳品質は高くはありません。ご了承ください。

ダウンロード先は以下の通りです \* <https://dl.dropboxusercontent.com/u/52897388/14liners.zip>(Laravel3.2.14 版) \* <https://dl.dropboxusercontent.com/u/52897388/12eyes.zip>(Laravel3.2.12 版)

ご自身で API を生成される場合や、コード補完時に PHPDoc 形式のコメントを表示してくれる IDE などをご利用ください。

しばらくは、この電子書籍の特典としますので、翻訳したコメント部分の著作権は保持いたします。

内容：

- **laravel** フォルダ：翻訳済みのコアファイル。バージョン 3.2.12 および 14。コアの **laravel** フォルダにそのまま書き添付してご利用いただけます。
- **public/api-ja** : **apigen** で生成した Laravel コアの API ドキュメント HTML。 [laravel.kore1server.com/api](http://laravel.kore1server.com/api) で公開しているものです。
- **apigen.neon** : 自分で **apigen** を使用して API ドキュメントを作成したい方のためのオプションファイルです。 **apigen** をインストールし、Laravel のルートディレクトリーで、 **apigen -c apigen.neon** で起動すると、 **public/api** 下にドキュメントが生成されます。詳しくは **apigen.neon** の内容をご覧ください。コメントを入れてあります。
- **html2ja** : **apigen** で生成した **public/api** は、 **apigen** 自体の生成テンプレートが英語なため、完全に日本語になりません。テンプレートに全部手を入れるのは作業が膨大になります。そこで **sed** を利用し、英語の部分日本語に書き換えております。 **apigen** で生成後、 **sh html2ja** を実行しますと、 **public/api-ja** フォルダの中に変換結果が書き込まれます。
- **html2ja.sed** : 上記で利用している SED スクリプトです。
- **japanize.php** : Laravel が出力する例外を日本語で表示するために、コアを書き換える Laravel のタスクです。