Colibri-Handbuch

Release 0.1.0

Pro Sales GmbH

20.12.2017

Erste Schritte

Mit diesem Handbuch hast Du alles an der Hand, was Du für den erfolgreichen Einsatz des Colibri Basissystem benötigst: Von den ersten Schritten der richtigen Installation und Konfiguration bis hin zu den konkreten Einsatzmöglichkeiten in der Praxis. Wir sind uns sicher, dass Du mit diesem Handbuch die ideale Einführung in Colibri findest; unabhängig davon, ob Du bereits früher mit dem Laravel PHP Framework gearbeitet hast oder nicht.

Vieles innerhalb dieses Handbuches richtet sich an einen Firmen internen Standard der Pro Sales GmbH aus. Da die meisten Entwickler der Pro Sales GmbH bspw. einen Mac verwenden, findest du immer wieder Kommandos oder Anweisungen für das Mac-Betriebssystem. Wir erwarten, dass Du selbständig solche Kommandos oder Anweisungen an Dein eigenes Betriebsystem anpassen kannst.

Die meisten Methoden und Konzepte welche in diesem Handbuch vorgestellt werden benötigen einen gewissen Grad an Vorkenntnisse um dem Inhalt Gut folgen zu können. So ist es bspw. von Vorteil wenn Du die nachfolgenden Begriffe bereits kennst und weisst wie diese Eingesetzt werden:

- Composer, Yarn
- Docker
- MariaDB, MySQL, Redis
- PHP 7
- WebPack
- Sass, ES6

KAPITEL 1

Einleitung

Die Grundanforderungen an internetbasierten Systemen ist oft identisch: Es müssen (teil-) dynamische HTML-Seiten dargestellt sowie Verbindungen zu einer Datenbank hergestellt werden. Diese grundlegenden Funktionalitäten werden daher in einem Basissystem zusammengefasst, auf das Du als Programmierer zugreifen kannst, um schnell und flexibel Inhalt und Logik zu erstellen.

1.1 Asset-Management

Damit Du mit allem ausgestattet bist was man zum entwickeln einer modernen Webanwendung benötigt unterstützt Dich Colibri ebenfalls beim Verwalten Deiner Assetsdateien. Hierbei greifen wir auf Laravel-Mix zurück, welches mit verschiedenen Kompilierungs- und Verarbeitungsmethoden ausgestattet ist. Basierend auf WebPack können so bspw. individuell Stylesheets oder JavaScript Komponenten zusammengestellt werden. Mehr Informationen über Laravel-Mix findest Du unter: https://laravel.com/docs/5.5/mix#introduction.

1.2 PHP-Framework

Es gibt viele Gründe ein PHP-Framework als Fundament einzusetzen, denn so können wir kostbare Entwicklungszeit einsparen und in der Regel auf hochwertige Komponenten zurückgreifen, die wir nicht ständig neu entwicklen müssen. Das Laravel PHP-Framework spielt dabei eine besondere Rolle, denn es hat sich bei vielen Entwicklern einen guten Namen gemacht und wird auf breiter Basis eingesetzt. Du hast vollen Zugriff auf alle Möglichkeiten, welche Dir das Laravel PHP-Framework (v.5.5) zu bieten hat.

1.3 Unterschiede zu Laravel

...

1.4 Entwicklungsumgebung

Genau wie bei einer Bergwanderung gern das falsche Schuhwerk genutzt wird, wird bei der Programmierung auch häufig nicht auf die richtige Entwicklungsumgebung geachtet. Dabei spielt es eine entscheidende Rolle wie Du deine Entwicklungsumgebung einrichtest um Tests und Veröffentlichung eines Projektes zu erleichtern.

Die Pro Sales GmbH verwendet bei fast allen Webprojekten Docker um Entwicklungs-, Test- oder Produktionsumgebungen zu schaffen und zu automatisieren.

1.5 Pakete / Erweiterungen

Erweiterungen können wie bei jedem normalen PHP-Projekt mithilfe von Yarn oder Composer hinzugefügt werden. Dabei stehen Dir alle Erweiterungen zur Verfügung die Du mit jedem anderen Laravel Projekt (v.5.5) ebenfalls hättest. Zusätzlich kannst du aber auch unsere Colibri spezifischen Erweiterungen verwenden, welche teilweise öffentlich zur Verfügung stehen.

- Pro Sales Erweiterungen
- Laravel Erweiterungen
- Spatie Erewiterungen

KAPITEL 2

Installationsanleitung

Wir beginnen mit der Auflistung der Systemanforderungen des Colibri Basissystems:

- PHP >= 7.0.0
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

2.1 Schritt-01: Composer Projekt

Composer ist ein unabhängiger Installer für PHP-Pakete, welcher Abhängigkeiten auflösen und installieren kann (https://getcomposer.org/). Er wird mittlerweile in vielen PHP-Frameworks sowie Open-Source-Projekten verwendet und auch Colibri stellt Composer-Pakete zur Installation bereit. Ein Composer-Paket kann auf der Webseite https://packagist.org/ bekannt gemacht werden.

Sobald Du Composer auf deinem Rechner installiert hast (https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx), kannst du mit der Installation von Colibri beginnen:

```
composer create-project pro-sales/colibri:* blog/
```

2.2 Schritt-02: Yarn Abhängigkeiten (optional)

Optional kannst Du auch gleich die Yarn (https://yarnpkg.com/lang/en/) Abhängigkeiten hinzu installieren. Diese Abhängigkeiten werden für das *Asset-Management* benötigt und sollten nur installiert werden falls Du Änderungen am Design oder den JavaScript Komponenten tätigen möchtest.

Sobald Du Yarn auf deinem Rechner installiert hast (https://yarnpkg.com/en/docs/install/), kannst du mit der Installation der Yarn-Abhängigkeiten beginnen:

```
cd blog/
yarn install
```

KAPITEL 3

Basissystem Einrichten

3.1 Konfiguration

Alle Konfigurationsdaten des Colibri Basissystems befinden sich im Konfigurationsverzeichnis (*config/*). Jeder dieser Konfigurationsdateien ist mithilfe von DocBlock Kommentaren dokumentiert und sollte als Deine erste Anflaufstelle zum Gesammtüberblick des Basissystems dienen.

- *Cache*
- *Datenbanken*
- *Lokalisierung*
- *Sessions*

3.2 Verzeichnisschutz

Eventuell brauchen gewisse Verzeichnisse besondere Rechte. Nachfolgend findest Du die dazugehörigen Kommandos um die etwaigen Rechte korrekt zu vergeben:

```
sudo chgrp -R www-data storage/ bootstrap/cache/
sudo chmod -R ug+rwx storage/ bootstrap/cache/
```

3.3 Webserver

3.3.1 Lokaler PHP-Server

Falls Du bereits PHP-7 auf Deinem Rechner installiert hast, kannst Du mithilfe des `serve` Artisan Kommandos, einen lokalen Entwicklungsserver starten. Dabei versucht PHP das Basissystem unter der URL: http://localhost:8000 abzubilden. Eine Datenbank wird Dir dabei jedoch nicht zur Verfügung gestellt, lediglich Dein PHP steht zur Verfügung:

```
php artisan serve
```

Eine weit aus fortgeschrittenere Methode um eine Entwicklungsumgebung einzurichten, findest Du unter: *Entwicklungsumgebung*. Dieses Artisan Kommando sollte wirklich nur gebraucht werden um bspw. kurz einen Einblick auf die Startseite zu erhaschen, nicht jedoch um professionelle Webanwendungen zu kreiieren.

3.3.2 Pretty URL's

Apache

Mithilfe der *public/.htaccess* Dateien werden alle Anfragen an den Front-Controller weitergeleitet, ohne dass Du die *index.php* Datei angeben musst. Damit diese Weiterleitung korrekt funktioniert musst Du jedoch gewährleisten dass das *mod_rewrite* Modules Deines Apache Webserver aktiviert wurde, ohne dieses Modul werden keine *.htaccess* Dateien von Deinem Webserver akzeptiert.

Falls die reguläre Lösung in *public/.htaccess* bei Dir nicht funktioniert kannst Du es mit nachfolgender Alternative versuchen:

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

Nginx

Wenn Du hingegegen einen Nginx Webserver verwendest, kannst du folgende Konfiguration setzen, damit alle Anfragen korrekt an den *Front-Controller* weitergeleitet werden können:

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

3.4 Anwendungsschlüssel

Der Anwendungsschlüssel oder auch Application-Key genannt, ist eine willküllriche Zeichenkette, welcher zum verschlüsseln von Sessiondaten vewendet wird. Normalerweise handelt es sich um eine Zeichenkette (*string*) bestehend aus 32-Zeichen.

Bei der Installtion von Colibri sollte automatisch ein Anwendungsschlüssel generiert worden sein. Diesen Schlüssel findest Du in Deiner *.env* Datei (`APP_KEY`). Solltest Du einmal keinen Schlüssel vorfinden oder falls Du einen neuen Schlüssel generieren möchtest, kannst Du mithilfe des `key:generate` Artisan Kommando's tun:

```
php artisan key:generate
```

> **Warnung:** Sollte der Anwendungsschlüssel aus irgendwelchen Gründen fehlen, werden die Sessiondaten Deiner Benutzer nicht verschlüsselt und sind somit nicht sicher!

3.5 DotEnv-Konfigurationen

Oftmals möchte man Konfigurationen an der aktuellen Umgebung des Basissystems anpassen. So ist es bspw. möglich, dass Du ein komplett anderes Cache-System innerhalb der Entwickung verwenden möchtest als Du bspw. für die Produktion einsetzt.

Colibri greift dabei auf die DotEnv PHP Bibliothek von Lance Lucas zu, um solche umgebungsspezifische Variablen in einer *.env* Datei auszulagern. Bei einer frischen Colibri Installation wird die *.env.dist* Datei automatisch kopiert, umbenannt in *.env* und mit einem Schlüssel versehen; all diese Schritte sind dabei innerhalb der *composer.json* Datei beschrieben.

Da Du innerhalb der *.env* Datei Deine Zugangsdaten und geheimen Konfigurationen tätigst, solltest Du niemals die *.env* Datei im Versionskontrollsystem aufnehmen, um Deine Daten zu schützen. Colibri übernimmt diese Aufgabe bereits für Dich, indem der entsprechende Eintrag in der *.gitignore* Datei hinterlegt wurde.

Falls Du zusammen mit einem Team arbeitest, wird im Regelfall die *.env.dist* Datei als Orientierungspunkt für Teammitglieder verwendet. Dabei können bspw. Platzhalterwerte eingesetzt werden, um anderen einen Hinweis zu geben wie sie die Konfiguration zu definieren haben.

Eine weitere DotEnv Datei welche von Colibri verwendet wird ist bspw. die *.env.testing*. Diese Datei überschreibt die *.env* Datei beim Einsatz der Testumgebung (PHPUnit) oder beim Ausführen von Artisan Kommandos mit der `--env="testing"` Option.

Hinweis: Jede Umgebungsvariable innerhalb der *.env* Datei kann von externen Umgebungsvariablen, wie bspw. Server-Level oder System-Level abhängigen Umgebungsvariablen überschrieben werden.

3.6 Wartungsmodus

Sollte sich die Anwendung im Wartungsmodus befinden, werden alle Anfragen an eine bestimmte Seite weitergeleitet. Dies erleichtert Dir ein temporäres „abschalten“ des Basissystems während Du bspw. Updates ausführen oder Wartungsarbeiten vornimmst.

Colibri verfügt über eine eigene *Middleware*, welche überprüft ob sich das Basissystem im Wartungsmodus befindet. Sollte dieser Fall eintreffen wird eine *MaintenanceModeException* Exception ausgeworfen. Dabei werden alle Anfragen mit dem Statuscode *503* beantwortet und an die entsprechende Seite weitergeleitet.

Während des Wartungsmodus werden keine *Aufgaben* aus den *Warteschlangen (Queues)* abgearbeitet. Diese Aufgaben werden erst dann ausgeführt wenn sich das Basissystem nicht mehr im Wartungsmodus befindet.

3.6.1 Wartungsmodus Aktivieren

Das `down` Artisan Kommando erlaubt uns dabei das aktivieren des Wartungsmodus für das Basissystem:

```
php artisan down
```

Optional kannst Du auch eine Nachricht (`message`) und Wiederholungsversuch (`retry`), in Sekunden, innerhalb des `down` Artisan Kommandos beifügen. Der Nachrichtenwert (`message`) wird dabei als Protokoll- oder Ausgabenachricht verwendet und sollte den Klienten über die Wartungsarbeiten informieren. Die `retry` Option hingegegen erlaubt es eine Zeitspanne, zu definieren, wann die Wartungsarbeiten möglicherweise abgeschlossen sind. Die Antwort wird dabei als *Retry-After* HTTP Header definiert:

```
php artisan down --message="Datenbank wird aktualisiert." --retry=60
```

3.6.2 Die 503-Seite

Das Blade-Template für die 503-Seite findest Du unter: *resources/views/errors/503.blade.php*. Du kannst die Seite beliebig anpassen wie jede andere View des Basissystems. Der Webseite steht dabei eine eigenes Stylesheet zur Verfügung. Die Resourcen dieses Stylesheets findest du unter: *resources/assets/sass/pages/503.sass*. Vergiss nicht die Datei zu kompilieren, falls Du Änderungen innerhalb dieses Stylesheets tätigst.

3.6.3 Wartungsmodus Deaktivieren

Um den Wartungsmodus zu beenden kannst Du das `up` Arisan Kommando verwenden:

```
php artisan up
```

3.6.4 Alternativen zum Wartungsmodus

Da die Umschaltung des Basissystems in den Wartungsmodus eine gewisse Anzahl an Sekunden eine sogenannte Downtime verursacht, lohnt es sich nach Alternativen umzusehen, welche diese Probleme lösen. Eine Alternative ist hierbei Envoyer welche ohne Downtime solche veröffentlichungen vornehmen kann.

KAPITEL 4

Struktur & Aufbau

Colibri ist ideal geeignet für mittelere bis grössere Webprojekte und ist bereits mit allen notwendigen Verzeichnissen und Klassen ausgestattet. Natürlich steht es Dir frei die Struktur von Colibri jederzeit zu ändern. Hierbei gibt es Grundsätzlich keine Einschränkungen, solgange Du mithilfe des Composer-Autoloading diese Klassen selbst einbinden kannst.

Hinweis: Trotz der Freiheiten welche sich Dir hier bieten, ein Hinweis: Zu viele grundlegende Änderungen solltest Du in Deinem Projekt nicht einführen. Ansonsten erschwert dies Neulingen in Deinem Team den Einstieg.

4.1 Wo befinden sich die Models/Entitäten?

Viele Entwickler scheinen darüber verwirrt zu sein, dass beim Basissystem kein spezifisches Verzeichnis für Models oder Entinitäten vorgesehen ist. Wir vertreten hierbei die Meinung von Laravel und behaupten dass das Wort Model bei vielen Entwicklern unterschiedliche Bedeutung hat. Zum Beispiel finden die einen Entwickler - ein Model enthält alle Business-Logiken die gebraucht werden. Wiederum andere Entwickler verstehen unter einem Model die Beziehung zu einer Datenbank.

Aus diesen Gründen wurde das Standardverzeichnis für Eloquent Models im *app/* Verzeichnis vorgesehen, dies erlaubt es den Entwicklern selbständig ein Verzeichnis zu wählen, in welchem die Models aufbewahrt werden sollen.

4.2 Hauptverzeichnis (./)

Anwendungsverzeichnis Im Anwendungsverzeichnis (*app/*) werden alle Kernklassen Deiner Anwendung aufbewahrt. Wir werden dieses Verzeichnis *weiter unten* detailierter behandeln; für den Anfang solltest Du lediglich wissen, dass sich alle Klassen Deiner Anwendung in diesem Verzeichnis befinden.

Bootstrap-Verzeichnis Innerhalb des Bootstrap-Verzeichnis (*bootstrap/*) befindet sich die *app.php* Datei welche für das Starten (Bootstraping) des Basissystems zuständig ist. Ebenfalls werden in diesem Verzeichnis auch generierte Cache Dateien, welche der Performance des Basissystems dienen, aufbewahrt.

Konfigurationsverzeichnis Alle Konfigurationsdaten des Colibri Basissystems befinden sich im Konfigurationsverzeichnis (*config/*). Jeder dieser Konfigurationsdateien ist mithilfe von DocBlock Kommentaren dokumentiert und sollte als Deine erste Anflaufstelle zum Gesammtüberblick des Basissystems dienen.

Datenbank-Verzeichnis Im Datenbank-Verzeichnis (*database/*) befinden sich Datenbank spezifieschen Dateien wie bspw. Migration oder Seedings. Unter Umständen wird dieses Verzeichnis auch zur Bereitstellung von SQLite Datenbanken genutzt.

Public-Verzeichnis Das Public-Verzeichnis (*public/*) enthält alle öffentlich abrufbaren Dateien. Neben der Front-Controller Datei (*index.php*) sind dies Grafiken, CSS und JavaScript Dateien.

Assets-Verzeichnis Im Assets-Verzeichnis (*public/assets/*) befinden sich öffentliche Grafiken, CSS und JavaScript Dateien des Basissystems.

Resourcen-Verzeichnis Das Resourcen-Verzeichnis (*resources/*) beinhaltet die Views, sowie Rohdateien welche bspw. mithilfe des Asset-Manager's kompliert werden. Hierzu gehören Dateien wie SASS für die Generierung von Stylesheets oder ES6 JavaScript Komponenten.

Route-Verzeichnis Im Route-Verzeichnis (*routes/*) wird das sogenannte Mapping aller bekannten URL's für die Controller oder zur direkten Ausgabe definiert. Im Regelfall werden diese Aufgaben in verschidene Dateien ausgelagert. Mehr zum Thema Routing findest Du jedoch unter: *Routing*.

Sammel-Verzeichnis Das Sammel-Verzeichnis (*storage/*) beinhaltet Deine kompilierten Blade-Templates, dateibasierte Sessiondaten, Caching Dateien sowie andere Dateien welche durch das Basissystem generiert werden.

Anwendungsspeicher-Verzeichnis Im Anwendungspeicher-Verzeichnis (*storage/app/*) werden alle generierten Dateien des Basissystems aufbewahrt. Eine Idee dieses Verzeichnis richtig zu Nutzen ist bspw. das *storage/app/public/* Unterverzeichnis anzulegen. Darin können Sie öffentliche Dateien hinterlegen welche mithilfe des Basissystems generiert wurden, bspw. Avatare oder ähnliches. Damit Colibri weiss dass Du diese Dateien öffentlich zur Verfügung stellen möchtest, solltest Du einen Symobllink unter *public/storage* anlegen, welcher auf dieses Verzeichnis verweist. Einen solchen Link kannst Du mithilfe des *php artisan storage:link* Kommandos zuweisen.

Frameworkspeicher-Verzeichnis Im Frameworkspeicher-Verzeichnis (*storage/framework/*) befinden sich alle Dateien welche durch das Laravel PHP Framework generiert wurden. Hierzu gehören auch die generierten Cache Dateien des Laravel PHP Framework's.

Protokoll-Verzeichnis Im Protokoll-Verzeichnis (*storage/logs/*) werden alle generieten Protokolle des Basissystems, des Frameworks oder des Servers hinterlegt, sofern alle Konfigurationen korrekt gesetzt wurden.

Testverzeichnis Wie der Name bereits Vermuten lässt, handelt es sich beim Testverzeichnis (*tests/*) um den Aufbewahrungsort für Funktions, HTTP, Browser Tests des Basissystems. Colibri stellt Dir dabei einen grundlegenden Test zur Verfügung, welcher zugleich als Orientierungspunkt verwendet werden kann.

Vendor-Verzeichnis Das Vendor-Verzeichnis (*vendor/*) ist für alle Fremdmodule gedacht, die Du in Deiner Anwendung einsetzt. Das wichtigste Fremdmodul ist das Laravel PHP-Framework, welches das Fundament von Colibri setzt.

Node-Verzeichnis Ein weiteres Verzeichnis welches Abhängigkeiten innerhalb des Basissystems aufbewahrt ist das Modul-Verzeichnis (*node_modules/*). In diesem Verzeichnis werden alle Abhängigkeiten aus der *package.json* Datei installiert.

4.3 Anwendungsverzeichnis (app/)

Im Anwendungsverzeichnis befinden sich die wichtigsten Dateien des Basissystems. Der Standard Namespace welcher innerhalb dieses Verzeichnisses verwendet wird ist `App` und wird mithilfe des Composer und dem PSR4-Standard Autoloading automatisch ins Basissystem geladen.

Tipp: Vieler dieser Klassen innerhalb des *app/* Verzeichnisses, können mithilfe von Artisan Kommandos generiert werden. Um eine Liste mit möglichen Befehlen zum generieren von Dateien im Anwendungsverzeichnis, zu erhalten nutze den `php artisan list make` Artisan Befehl.

Konsolenverzeichnis Im Konsolenverzeichnis (*app/Console/*) werden all Deine Artisan Kommandos definiert. Ein neues Artisan Kommando kannst Du bspw. mithilfe des `make:command` Artisan Befehls generieren. In diesem Verzeichnis befindet sich der Konsolen-Kernel welcher zum registrieren der Artisan Kommandos und cron-jobs verwendet wird.

Eventsverzeichnis Das Eventsverzeichnis (*app/Events/*) existiert nicht gleich zu beginn. Es sollte mithilfe des `event:generate` oder des `make:event` Artisan Kommandos generiert werden. Dieses Verzeichnis beinhaltet die Klassen für *Aufgaben & Events*. Events werden zum informieren von weiteren Bestandteilen des Basissystems verwendet falls ein bestimmter Event eintrifft, dies ermöglicht ein flexibles coden mit der Möglichkeit Klassen untereinander zu entkoppeln.

Exception-Verzeichnis Das Exception-Verzeichnis (*app/Exceptions/*) beinhaltet die Exception-Handler Klasse, welche zum behandeln von Fehlern eingesetzt wird, sowie verschiedene Exception-Klassen die Du selber generieren kannst. Falls Du die Fehlerbehandlung anpassen möchtest, kannst Du dies in der *Handler* Klasse in diesem Verzeichnis tun.

HTTP-Verzeichnis Im HTTP-Verzeichnis (*app/Http/*) befinden sich Deine Controller, Middlewares und Form-Requests. Eigentlich alles was zum bearbeiten von Anfragen benötigt wird.

Aufgabenverzeichnis Das Aufgabenverzeichnis (*app/Jobs/*) existiert nicht gleich zu Beginn. Es sollte mithilfe des `make:job` Artisan Kommandos generiert werden. Dieses Verzeichnis beinhaltet alle Aufgaben innerhalb einer *Wartenschlange*. Aufgaben können dabei einzeln abgearbeitet werden oder synchron mit der Anfrage vearbeitet werden. Aufgaben welche synchron mit der Anfrage verarbeitet werden, können auch als Kommandos deklariert werden da diese dem Kommando-Entwurfsmuster folgen.

Listener-Verzeichnis Das Listener-Verzeichnis (*app/Listeners/*) existiert nicht gleich zu Beginn. Es sollte mithilfe des `event:generate` oder des `make:listener` Artisan Kommandos generiert werden. Dieses Verzeichnis beinhaltet die Klassen welche zum berarbeiten der *Aufgaben & Events* verwendet werden. Ein Listener wartet bis ein Event innerhalb der Anwendung ausgelöst wird und bestimmt dabei das weitere Verfahren mit diesem Event. Als Beispiel könnte somit eine E-Mail an einen Benutzer zugestellt werden ohne dass die E-Mail selbst im Benutzer-Model definiert werden muss.

Mail-Verzeichnis Das Mail-Verzeichnis (*app/Mail/*) existiert nicht gleich zu Beginn. Es sollte mithilfe des `make:mail` Artisan Kommandos generiert werden. Dieses Verzeichnis beinhaltet alle Deine Mail-Klassen welche Du innerhalb der Anwendung versenden möchtest. Mail-Objekte erlauben es Dir Logiken und Inhalte von E-Mails in eine eigene Klasse zu extrahieren, welche Du dann mithilfe der `Mail::send` Methode versenden kannst.

Nachrichten-Verzeichnis (Notifications) Das Nachrichten-Verzeichnis (*app/Notifications/*) existiert nicht gleich zu Beginn. Es sollte mithilfe des `make:notification` Artisan Kommandos generiert werden. In diesem Verzeichnis befinden sich Klassen welche für verschiedene Nachrichtendienste eingesetzt werden können um die Benutzer über einen Event zu informieren. Dabei stehen Dir eine Vielzahl an Treibern zur Verfügung, um Deine Nachricht zu verbreiten: E-Mail, Slack, SMS oder speichern in der Datenbank.

Zugriff-Verzeichnis (Policies) Das Zugriff-Verzeichnis (*app/Policies/*) existiert nicht gleich zu Beginn. Es sollte mithilfe des `make:policy` Artisan Kommandos generiert werden. In diesem Verzeichnis befinden sich die Zugriff (Policy) Klassen der Anwendung. Mithilfe von Policies kannst Du den Zugriff auf eine bestimmte Resource abhängig von den Benutzern steuern, mehr zu diesem Thema findest Du unter: *Autorisierung*.

Provider-Verzeichnis Im Provider-Verzeichnis (*app/Providers/*) befinden sich alle *Service Provider Klassen* des Basissystems. Mitihilfe von Service-Providern werden alle Abhängigkeiten innerhalb des Basissystems aufgelöst, alle Events werden registiert und alles was für den Betrieb des Basissystems notwendig ist wird darin vorbereitet.

Bei einer Neuinstallation von Colibri findest Du bereits einige solcher Service-Provider Klassen welche für den Betrieb des Systems benötigt werden. Fühl Dich wie zuhause und versuche Deine eigene Services in Dein Basissystem zu integrieren damit Du Zugriff auf all Deine Objekte erhälst.

Validierungsverzeichnis Das Validierungsverzeichnis (*app/Rules/*) existiert nicht gleich zu Beginn. Es sollte mithilfe des `make:rule` Artisan Kommandos generiert werden. In diesem Verzeichnis befinden sich Deine eigenen Validierungsregeln. Damit kannst Du die Logiken welche zur Validierung benötigt werden in ein eigenes Objekt auslagern, mehr Informationen zu diesem Thema findest Du unter: *Validierung*.

KAPITEL 5

HTTP-Verarbeitung

5.1 Front-Controller

Abb. 5.1: Der Front-Controller lädt Composer's Autoloading, sowie das Illuminate (Laravel) PHP-Framework und ist für das Bootstraping des Basissystems, sowie der Verarbeitung von HTTP-Anfragen verantwortlich.

Eine MVC-Applikation besitzt immer einen zentralen Einstiegspunkt, den Front-Controller, welcher auch gern Bootstrap-File genannt wird. Der Front-Controller, ist im Endeffekt nur dafür zuständig die Anfragen entgegenzunehmen und diese mithilfe des HTTP-Kernels zu beantworten. Im Hintergrund wird hierzu zuerst das Composer Autoloading, sowie das Illuminate (Laravel) PHP-Framework geladen. Anschliessend wird mithilfe des Bootstraping das Basissystem hochgefahren. Nun kann die Anfrage mithilfe des HTTP-Kernels verarbeitet und beantwortet werden.

Das erweckt eventuell den Anschein das vieles einfach so, oder mithilfe von Magie entsteht. Naja das ist so nicht Ganz richtig, denoch ist es Momentan noch etwas zu früh um Dir diesen Schritt detailiert zu erklären. Aber wir veratten schon einmal, dass bspw. der *Service-Provider* eingesetzt wird um alle Klassen sowie das Basissystem selbst in die Anwendung zu laden.

5.2 HTTP / Console Kernels

Nachdem der Front-Controller geprüft hat um welche Art von Anfrage es sich handelt, wird die Anfrage an den HTTP- oder Console-Kernel weitergeleitet, welcher die Anfrage schlussendlich verarbeitet. Alle Anfragen passieren den einen oder anderen Kernel und sind fest im Anfrage-Prozess verankert. Für den Moment konzentrieren wir uns aber auf den HTTP-Kernel (*app/Http/Kernel.php*).

Der HTTP-Kernel erweitert die `Illuminate\Foundation\Http\Kernel` Klasse und definiert ein Array mit `bootstrappers` die ausgeführt werden, bevor eine Anfrage verarbeitet wird. Dabei wird bspw. die Fehlerbehandlung eingerichtet, das Protokollieren wird eingeschaltet, die *Umgebungsvariablen* werden eingspeisst und andere Aufgaben, welche für eine sichere Verarbeitung der Anfrage benötigt werden implementiert der HTTP-Kernel für Dich.

Abb. 5.2: Der HTTP-Kernel funktioniert ähnlich wie eine Play-Doh Maschiene ...

Des weiteren definiert der HTTP-Kernel eine Liste mit aktiven *Middleware* welche die Anfrage durchlaufen musss, bevor die Anwendung selbst um die Anfrage kümmert. Diese Middleware sind Hauptsächlich zum lesen und schreiben von *HTTP-Sessions*, zum einschalten des *Wartungsmodus*, *zum verifizieren des CSRF-Token* und mehr zuständig.

Die Verarbeitung (`handle` Methode) einer Anfrage durch den HTTP-Kernel ist simpel: Empfange eine Anfrage (`Request`) und liefere eine Antwort (`Response`). Stell Dir unter dem HTTP-Kernel eine grosse *Play-Doh* Maschiene vor, bei welcher die Anfragen als Knettmasse eingehen und als wundeschöne Formen (Antworten) wieder zurückgeliefert werden.

5.2.1 Service-Provider Anfordern

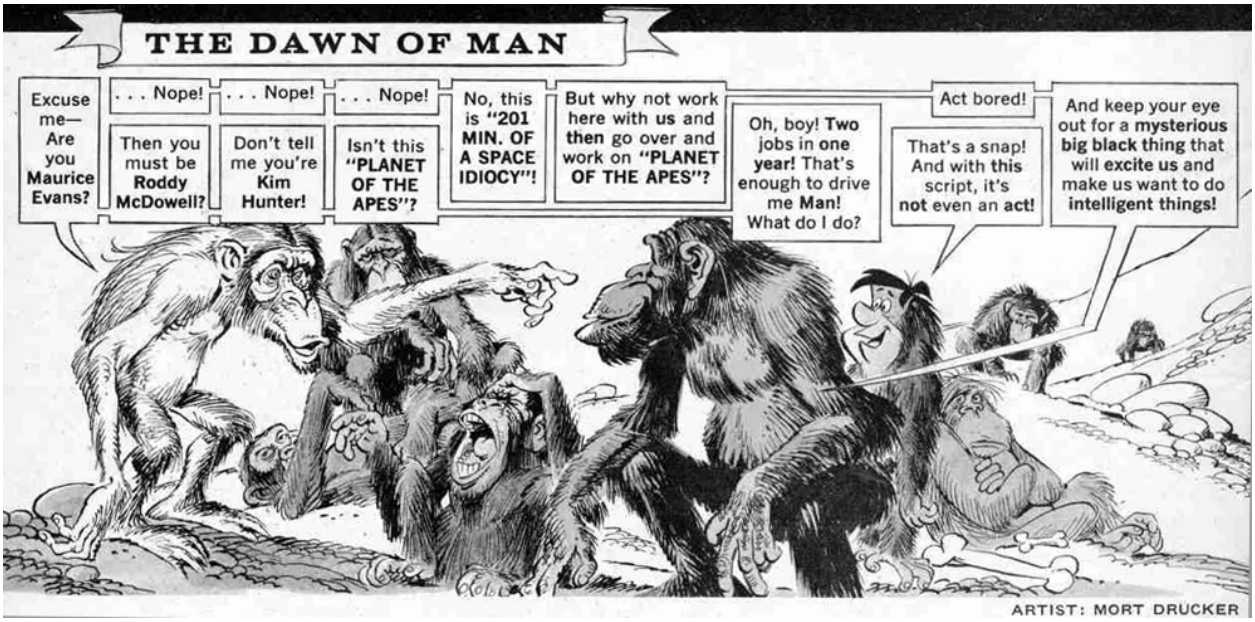
Einer der wohl wichtigsten Aufgaben des Kernels ist, dass anfordern der *Service-Provider* für Deine Anwendung, dazu gehören auch die Service-Provider des Basissystems. Alle Service-Provider des Basissystems werden in der *config/app.php* innerhalb des `providers` Schlüssel des Konfigurationsarrays aufgeführt. Bei allen Service-Provider wird immer zuerst die `register` Methode ausgeführt, sobald sich alle Service-Provider „registriert“ haben, werden die `boot` Methoden aufgerufen.

Beim Service-Provider handelt es sich um ein Entwurfsmuster für eine zentrale Registrierung von Objekten. Da das Basissystem selbst aus einzelnen Komponenten besteht, fällt auch das Basissystem selbst in diese Kategorie. Der Service-Provider übernimmt dabei die wohl entscheindenste Rolle, wenn es um das Verwalten von Objekten innerhalb des Basissystems geht.

5.2.2 Anfrage Verarbeiten

Sobald das Basissystem zur Verfügung steht (bootstraping) und alle Service-Provider registriert wurden wird ein `Request` Objekt generiert und an den Router übermittelt. Der Router wiederum versucht das `Request` Objekt einer Route oder einer Controller Methode zuzuweisen und durchläuft dabei die Route spezifischen Middlewares.

5.3 Die Rolle der Service-Provider



Beim *Service-Provider* handelt es sich um ein Entwurfsmuster für eine zentrale Registrierung von Objekten. Wir verwenden Service-Provider um Kontrolle über die Instanziierungen von neuen oder bestehenden Klassen innerhalb des Basissystem zu erhalten. Eine Instanz des Basissystems wird erstellt, alle Service-Provider werden registriert und gestartet, anschliessend wird die Anfrage an das Basisystem übermittelt. So einfach ist dieses Prinzip!

5.4 Routing & Middleware

Um die HTTP-Anfragen zu bearbeiten stellt das Basissystem die nachfolgenden Konzepte bereit. Mithilfe von Routing kannst Du das sogenannte URL-Mapping vornehmen. Die Anfrage wird dabei einer bestimmten Controller Methode oder Closure Funktion zugewiesen und durchläuft dabei die Middlewares:

5.4.1 Routing

Die Aufgabe des Routing ist es, eine Anfrage auf den entsprechenden Controller oder Closure-Funktion abzubilden. Dabei wird die URL der Anfrage mithilife von mehrerer Routing-Regeln analysiert. Wird eine Route als passend identifiziert, werden die ermittelten Angaben zu Controller, Action und weitere Parameter in einem Objekt gespeichert und weiter verarbeitet.

Die Routes befinden sich dabei in einem eigenen Verzeichnis (*routes/*) und wird dabei auf die folgenden Dateien aufgeteilt: *routes/api.php*, *routes/channel.php*, *routes/console.php* und *routes/web.php*.

Grundlagen

Die simpelste Variante einer Route benötigt eine URI, sowie eine Closure Funktion um eine Anfrage verarbeiten zu können:

```
Route::get('foo', function () {
    return 'Hello World';
});
```

Standard-Route-Dateien

API-Routes

Routes welche für die Programmierungsschnittstelle (API) benötigt werden können in der *routes/api.php* definiert werden. Diese Routes sind dabei Zustandslos und werden mithilfe der `api` middleware Gruppe verarbeitet.

```
<?php

use Illuminate\Http\Request;

Route::middleware('auth:api')->get('/user', function (Request $request) {
    return $request->user();
});
```

Broadcasting-Kanäle

Innerhalb der *routes/channels.php* Datei werden alle Event basierten Kanäle (Broadcasting Channels) des Basissystems definiert. Um nun bspw. eine Aktion auszuführen, muss die Callback Funktion aufgerufen werden. Sobald diese Callback Funktion den Wert Wahr (`true`) zurückliefert, darf die entsprechende Aktion ausgeführt werden.

```
<?php

Broadcast::channel('App.User.{id}', function ($user, $id) {
    return (int) $user->id === (int) $id;
});
```

Falls Du jetzt schon in das Broadcasting eintauchen möchtest (was ich nicht empfehle), solltest Du Dir ein Video, von Taylor Otwell, dem Gründer des Laravel PHP Framework: https://laracasts.com/lessons/broadcasting-events-in-laravel-5-1.

Konsole-Routes

In der *routes/console.php* Datei kannst Du mithilfe einer Closure Funktion Deine eigenen Artisan Kommandos definieren. Jeder dieser Closure Funktionen ist dabei an eine *command* Instanz gebunden welche Dir die Funktionen bereitstellt um ein eigenes Artisan Kommando zu generieren:

```
<?php

Artisan::command('inspire', function () {
    $this->comment(
        'Das Leben ist wie eine Schachtel Pralinen, man weiss nie, was man bekommt.'
    );
})->describe('Display an inspiring quote');
```

Web-Routes

Bei den meisten Projekten beginnst Du das definiereren von Routes wahrscheinlich mit der *routes/web.php* Datei. Diese Datei beinhaltet alle öffentlichen Routes welche Du bspw. mithilfe Deines Browser aufrufen kannst.

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

Router-Methoden

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

5.4.2 Middleware

KAPITEL 6

MVC-Struktur

Das Model-View-Controller Entwurfsmuster, kurz auch als MVC bezeichnet, ist eines der objektorientierenten Entwurfsmuster. Die Idee des MVC ist, eine Anwendung in verschiedene Teile auzugliedern, nämlich Model, View und den Controller. Hintergrund des Ansatzes ist, dass mnan die Anwendung besser und deutlicher strukturieren möchte, so dass man genau weiss, welchen Teil des Basissystems welche Funktion hat.

6.1 Model

Ein Modell (Model) enthält grundsätzlich die jeweils für den aktuellen Anwendungskontext relevanten Daten. Die Implementierung kann in einigen Fällen auch Geschäftslogik enthalten. Oft existiert für jedes Objekt der Realwelt eine Modellklasse und auf der Datenbankseite eine Tabelle.

Weiter Lesen ...

6.2 View

Die Darstellung (View) ist für die Anzeige und für die Entgegennahme von Benutzerinteraktionen zuständig. Dazu benötigt die View-Klasse die Daten aus dem Modell. Hierin werden im Falle von Webanwendungen alle Formulare und HTML-Elemente implementiert.

Weiter Lesen ...

6.3 Controller

Die Programmlogik (Controller) ist für die Verwaltung der Darstellung (gegebenfalls auch mehrerer gleichzeitig) und das Holen, sowie Aktualisieren der benötigten Daten zuständig. Aktionen, welche von einem Benutzer in einem View ausgelöst werden, werden hier ebenfalls verarbeitet. Das Manipulieren von Daten allerdings ist nicht Sache der Programmlogik.

Weiter Lesen ...

KAPITEL 7

Services

7.1 Service-Container

7.2 Service-Provider

KAPITEL 8

Facades & Contracts

8.1 Facade

8.2 Contract

KAPITEL 9

Kernfunktionen

KAPITEL 10

Daten & Dateien

KAPITEL 11

HTTP

KAPITEL 12

Sicherheit

KAPITEL 13

Aufgaben & Events

13.1 Warteschlangen (Queues)

13.2 Aufgaben-Verwaltung (Cron-Jobs)

KAPITEL 14

Ausgabe

KAPITEL 15

Datenbanken

KAPITEL 16

Eloquent ORM

KAPITEL 17

Testen

17.1 HTTP-Tests

17.2 Browser-Tests

17.3 Datenbank-Tests

17.4 Fake-Tests

KAPITEL 18

Suchen

search

KAPITEL 19

Glossar

Controller Die Programmlogik (Controller) ist für die Verwaltung der Darstellung (gegebenfalls auch mehrerer gleichzeitig) und das Holen, sowie Aktualisieren der benötigten Daten zuständig. Aktionen, welche von einem Benutzer in einem View ausgelöst werden, werden hier ebenfalls verarbeitet. Das Manipulieren von Daten allerdings ist nicht Sache der Programmlogik.

Front-Controller Eine MVC-Applikation besitzt immer einen zentralen Einstiegspunkt, den Front-Controller, welcher auch gern Bootstrap-File genannt wird. Der Front-Controller, ist im Endeffekt nur dafür zuständig die Anfragen entgegenzunehmen und diese mithilfe des HTTP-Kernels zu beantworten. Im Hintergrund wird hierzu zuerst das Composer Autoloading, sowie das Illuminate (Laravel) PHP-Framework geladen. Anschliessend wird mithilfe des Bootstraping das Basissystem hochgefahren. Nun kann die Anfrage mithilfe des HTTP-Kernels verarbeitet und beantwortet werden.

HTTP-Kernel Der HTTP-Kernel erweitert die `Illuminate\Foundation\Http\Kernel` Klasse und definiert ein Array mit `bootstrappers` die ausgeführt werden, bevor eine Anfrage verarbeitet wird. Dabei wird bspw. die Fehlerbehandlung eingerichtet, das Protokollieren wird eingeschaltet, die *Umgebungsvariablen* werden eingspeisst und andere Aufgaben, welche für eine sichere Verarbeitung der Anfrage benötigt werden implementiert der HTTP-Kernel für Dich.

Routing Die Aufgabe des Routing ist es, eine Anfrage auf den entsprechenden Controller oder Closure-Funktion abzubilden. Dabei wird die URL der Anfrage mithilife von mehrerer Routing-Regeln analysiert. Wird eine Route als passend identifiziert, werden die ermittelten Angaben zu Controller, Action und weitere Parameter in einem Objekt gespeichert und weiter verarbeitet.

Service-Provider Beim *Service-Provider* handelt es sich um ein Entwurfsmuster für eine zentrale Registrierung von Objekten. Wir verwenden Service-Provider um Kontrolle über die Instanziierungen von neuen oder bestehenden Klassen innerhalb des Basissystem zu erhalten. Eine Instanz des Basissystems wird erstellt, alle Service-Provider werden registriert und gestartet, anschliessend wird die Anfrage an das Basisystem übermittelt. So einfach ist dieses Prinzip!

Model Ein Modell (Model) enthält grundsätzlich die jeweils für den aktuellen Anwendungskontext relevanten Daten. Die Implementierung kann in einigen Fällen auch Geschäftslogik enthalten. Oft existiert für jedes Objekt der Realwelt eine Modellklasse und auf der Datenbankseite eine Tabelle.

View Die Darstellung (View) ist für die Anzeige und für die Entgegennahme von Benutzerinteraktionen zuständig. Dazu benötigt die View-Klasse die Daten aus dem Modell. Hierin werden im Falle von Webanwendungen alle

Formulare und HTML-Elemente implementiert.

KAPITEL 20

Stichwortverzeichnis

Stichwortverzeichnis