Niclas Kahlmeier

Laravel 7 für Fortgeschrittene

Ein Webmasters Press Lernbuch

Version 1.0.5 vom 06.07.2020

Autorisiertes Curriculum für das Webmasters Europe Ausbildungs- und Zertifizierungsprogramm

Inhaltsverzeichnis

Authentifizieren – Login

4

Du lernst in dieser Lektion

- wie du das Auth-System des Laravel Frameworks verwendest.
- wie du das AuthSystem für deine Bedürfnisse anpassen und konfigurieren kannst.
- wie du eine Zwei-Faktor-Authentifizierung umsetzen kannst.
- wie du OAuth verwendest.

4.1 Laravel Auth - in wenigen Schritten zum Login-System

Du kennst dies sicherlich von den meisten Internetseiten: Auf einer öffentlich-zugänglichen Seite wird dir eine Möglichkeit angezeigt, dich anzumelden. Wenn du dies tust, gelangst du in einen geschlossenen Bereich. Genau so ein Login-System wollen wir nun in unsere Applikation integrieren. Glücklicherweise stattet uns Laravel mit den notwendigen Werkzeugen aus, um sehr schnell und einfach eine Nutzer-Authentifizierung einbauen. Was meinst du, wie lange wir dafür brauchen? Ich sage, dass wir uns in fünf Minuten in unserem Projekt anmelden können. Topp, die Wette gilt!

Für ein Login-System benötigen wir ein zusätzliches Composer Package. Genau so etwas bietet uns Laravel, allerdings ist das vor einigen Versionen in ein eigenes dediziertes Composer Package gewandert. Das hat vor allem mit dem Frontend zu tun: Mit diesem Package erhalten wir eine vollständig implementierte Auth-Lösung. Neben dieser ist aber auch der Aufbau für die Javascript Frontend Frameworks **Vue** und **React** im Package enthalten. Alles ist vorkonfiguriert und funktioniert ohne zusätzlichen Aufwand. Wir können aber – typisch Laravel – selbst Anpassungen vornehmen. Dazu aber später mehr. Wir wollen uns ja in nun nur noch vier Minuten anmelden können. Ich erkläre es im Anschluss ausführlich, aber jetzt gehen wir den Prozess Schritt für Schritt durch:

Schritt für Schritt 1:

1. Gehe via SSH in deine Vagrant Box.
2. Navigiere in das Verzeichnis, in dem deine Laravel-Applikation liegt.
3. Füge das gerade angesprochene Composer Package hinzu. `composer require laravel/ui`.
4. `php artisan ui vue --auth`: Führe den Artisan Command aus, um das Auth-System in deine Applikation zu holen.
5. Migriere die Datenbank mit `php artisan migrate`.
6. Rufe die */* Route auf. Dort müsstest du schon »Login« und »Register« in der oberen rechten Ecke sehen. Wenn du aber auf »Login« klickst, wird eine ungestylte Seite angezeigt. Deshalb müssen wir noch die Javascript- und CSS-Dateien erhalten.
7. Führe in deinem Applikationsverzeichnis (immer noch in der Vagrant Box) den Befehl `npm install` aus. Npm ist wie Composer nur für Javascript.
8. Anschließend muss der Befehl `npm run dev` ausgeführt werden. Dadurch wird ein sogenannte Bundler ausgeführt, der jeweils eine Javascript- und CSS-Datei im Verzeichnis *public/css* und *public/js* erstellt.
   Wenn du jetzt die */* Route öffnest und auf »Register« klickst, kannst du dich in deiner Applikation registrieren und anschließend mit diesen Daten anmelden. Habe ich zuviel versprochen? Je nachdem wie deine Internetleitung ist, haben wir ein Login-System in unter fünf Minuten hinzugefügt.

Du siehst, dass es funktioniert, aber du weißt wahrscheinlich noch nicht wie und warum. Ich habe dafür ein Video aufgenommen, in dem ich dir die Prozesse zeige, die hinter Registrieren und Anmelden stehen.

https://s.webmasters.de/2O

Laravel hat bereits mehrere Auth Controller integriert. Diese sind in Ordner *app/http/controller/auth*. Die Aufgaben der Controller sind selbsterklärend. Das von uns hinzugefügte Composer Package *laravel/ui* erstellt aus sogenannten Stubs die Views, die wir fürs Auth benötigen. ***Stubs*** sind vorgefertigte Dateien, die dann in unsere Applikation eingesetzt werden. Außerdem werden noch einige weitere Dinge wie der Home Controller und die Routes hinzugefügt. Einzelne Routes schließt du aus, indem du im Routing bei `Auth::routes` den Route-Namen auf `false` setzt:

```
Auth::route(["login'' => false])
```

Du kannst bei einer frischen Laravel-Installation das Auth-System gleich inkludieren. Hänge dazu das Flag `--auth` an:

```
laravel new advanced-laravel --auth
```

An dieser stelle muss ich anmerken, dass ich persönlich Bootstrap nicht sonderlich mag. Wenn wir das *laravel/ui*-Package nutzen, wird automatisch im View Layout ein Base Layout erstellt, welches Bootstrap verwendet. Natürlich darfst du Bootstrap gerne einsetzen, aber mein Favorit ist zur Zeit ***Tailwind CSS***, das in der Laravel-Community gerade sehr hoch im Kurs steht. Änderungen nimmst du entweder von Hand vor oder durch ein entsprechendes Frontend Preset, von denen für Laravel eine ganze Menge bereitstehen. Diese kannst du anstatt des *laravel/ui*-Package verwenden. Folge einfach der Anleitung des jeweiligen Presets. Presets: https://github.com/laravel-frontend-presets.

Hier ist meine persönliche Top 5 der Frontend Presets:

- 1. Tailwind (CSS) https://github.com/laravel-frontend-presets/tailwindcss
- 2. Vue Preset (nutzt standardmäßig Bootstrap, was sich aber manuell ändern lässt) *laravel/ui*
- 3. Inertia.js https://github.com/laravel-frontend-presets/inertiajs
- 4. Bulma (CSS) https://github.com/laravel-frontend-presets/bulma
- 5. React Preset (nutzt standardmäßig Bootstrap, was sich aber manuell ändern lässt) *laravel/ui*

In diesem kleinen Video zeige ich dir, wie wir von Hand Bootstrap durch Tailwind ersetzen.

https://s.webmasters.de/2P

Die Laravel-Authentifizierung basiert auf ***Guards*** und ***Providern***. Die Guards definieren, wie ein Nutzer bei jeder Anfrage authentifiziert wird. Die Provider kümmern sich darum, wie der Nutzer aus einem persistenten Speicher erhalten wird. Standardmäßig schaut der Provider mit dem Eloquent Driver in der Datenbank. Die Konfiguration der Provider und Guards kannst du in der Auth Config *config/auth* ansehen. Dort sehen wir direkt, dass der Guard für unsere Route Group *web* den Driver *session* und den Provider *users* nutzt. Für die API Routes nutzten wir den *token*-Driver. Das ist sinnvoll, da eine JSON API keine Session nutzt. Beide nutzen aber den *users*-Provider. Dieser wiederum nutzt den *eloquent*-Driver bei unserem User-Model. Wenn du in deiner Applikation kein Eloquent verwendest, greifst du eben auf

den *database*-Driver zurück. Dieser nutzt den Query Builder. Sicher fällt dir der Abschnitt auf, bei dem es um das Zurücksetzen der Passwörter geht. Darauf komme ich aber an geeigneter Stelle zurück.

Lass uns zunächst die Migration der Nutzer genauer anschauen. Wir benötigen eine einzigartige E-Mail-Adresse, eine Passwort-Spalte, die mindestens 60 Zeichen speichern kann, sowie den *rememberToken*. Diese Spalten sollten bei dir ebenso vorhanden sein. Dir dürfte außerdem aufgefallen sein, dass wir nach einer erfolgreichen Anmeldung immer zur Route */home* weitergeleitet werden. Dieses Verhalten kannst du natürlich auch ändern. Im *RouteServieProvider* konfigurierst du den Weiterleitungspfad unter der Konstanten `HOME`.

Generell sind, typisch für Laravel, eine ganze Reihe von Anpassungen möglich. Normalerweise wird die E-Mail zur Authentifizierung genutzt, aber du kannst natürlich auch den Nutzernamen eines Nutzers verwenden. Stelle dabei sicher, dass eine gleichnamige Spalte in der Datenbank verfügbar ist. Der Nutzername sollte ebenfalls einzigartig sein. Stell dir vor, mehrere Nutzer könnten den gleichen Nutzernamen auswählen. Das würde zu einem riesigen Chaos führen. Grundsätzlich kannst du jede Spalte anstatt der E-Mail nutzen. Dafür muss die Methode `username` im *LoginController* überschrieben werden. Die Methode stammt aus dem Trait *AuthenticatesUsers*. Als Wiedergabewert muss der zu verwendende Spaltenname angegeben werden.

Im *RegisterController* legst du die für die Registrierung notwendigen Formularattribute fest. Der Validator validiert die einzelnen Attribute der Request so wie du es kennst. Die `create`-Methode fungiert wie ein Mutator und speichert die Daten in der Datenbank. Wichtig ist, dass das Passwort als Hash gespeichert wird, also nicht in reiner Textform in der Datenbank vorliegt. Nach dem Hashing ist es nicht mehr möglich, das ursprünglich eingegebene Passwort herausfinden. Wir können nur noch einen Wert, meist eine Eingabe, hashen und mit dem vorhandenen Hash vergleichen. Wenn dir Hashing noch nicht ganz klar ist, schaue dir noch einmal das Kapitel »Sicherheit« der vorherigen Class an. Wie gesagt, habe ich mir für diese Class mit dir noch einiges vorgenommen. Natürlich wiederhole ich einige Sachen, dennoch gehe ich davon aus, dass dir die Konzepte der vorherigen Class klar sind.

Zugriff auf den angemeldeten Nutzer

Natürlich benötigen wir Zugriff auf den derzeit angemeldeten Nutzer. Anhand dieses Zugriffs können wir auch entscheiden, ob der derzeitige Nutzer in den geschützen Bereich darf oder eben nicht. Ich wette, dass du die Lösung mittlerweile selbst weißt: Wir können die *Auth*-Facade oder die Helper-Funktion `auth()` verwenden und auf beiden die verschiedenen Methoden aufrufen. Mit `Auth::user()` oder `auth()->user()` erhalten wir die Nutzerinstanz des derzeit angemeldeten Nutzers. Anschließend kannst du natürlich auf alle Attribute der Modelinstanz zugreifen. Sobald ein Nutzer angemeldet ist, erhältst du diesen auch über die Request-Instanz `$request->user()`. Die Request gibst du an eine Methode weiter, dann wird die Abhängigkeit automatisch vom Service Container »injected«. Das kennst du ja bereits.

Du kannst über die *Auth*-Facade oder die Helper-Funktion auch einen booleschen Wert dafür erhalten, ob der Nutzer angemeldet ist. Nutze dafür die Methode `check()`. Meist nutzten wir aber Middlewares anstatt der `check`-Methode, um herauszufinden, ob der Nutzer angemeldet ist. Dies hat den Vorteil, dass der Nutzer die Route bzw. den Controller noch nicht erreicht. Laravel hat die Middleware *auth* inkludiert (*Illuminate\Auth\Middleware\Authenticate*). Die Middleware wird auch für dich registriert. Du brauchst sie nur noch einzusetzen. Wenn der Nutzer angemeldet ist, `Auth::check()` also *true* ist, wird der Nutzer zur nächsten Zwiebelschicht weitergeleitet. Ist der Rückgabewert *false*, so wird der Nutzer zu der Route weitergeleitet, die in der `redirectTo`-Methode der *Authenticate* Middleware (*app/Http/Middleware/Authenticate*) angegeben ist. An diesem Ort kannst du die Weiterleitung auch anpassen, wenn der Nutzer nicht zur Standardanmeldungs-Route weitergeleitet werden soll. Du kannst bei der Middleware *auth* auch den zu nutzenden Guard angeben. Gib den Guard-Namen als Parameter mit.

Ein anderer Punkt sind natürlich die Views. Bei unserer Route »/« wird in der View geprüft, ob ein Nutzer angemeldet ist oder nicht. Anhand dessen werden dem Nutzer dynamisch die entsprechenden Schaltflächen im Header angezeigt. Hilfreich sind dabei die Blade Directives `@auth @endauth` und `@guest`

`@endguest`. Beiden Methoden kannst du einen Guard-Namen mitgeben, um einen speziellen *Auth* Guard zu nutzen.

Übung 5: Erste Schritte mit Laravels Login System

Hier ist wieder eine kleine Aufgabe, mit der ich dich testen möchte: Implementiere in deiner Laravel-Applikation, falls noch nicht geschehen, das Login-System. Wenn du dir bei einigen Punkten unsicher bist, kannst du natürlich in das Walkthrough schauen. Deine Aufgabe ist es, das Login-System zu nutzen. Auf dem Homescreen, also der URI */home*, soll der Name des Nutzers angezeigt werden, wenn dieser angemeldet ist. Außerdem soll eine URI */secret*, die den Text »gesicherter Bereich« wiedergibt, nur für angemeldete Nutzer aufrufbar sein.

Jetzt, wo wir mit dem Login-System die Basis geschaffen haben, denken wir einmal an den typischen Ablauf und die typischen Funktionen, wenn du dich auf anderen Websites wie z. B. Github bewegst.

- Normalerweise sollte ein böswilliger Nutzer nicht unendlich oft Passwörter bei E-Mails bzw. Nutzernamen ausprobieren dürfen.
- Der Nutzer sollte die Möglichkeit haben, angemeldet zu bleiben.
- Der Nutzer muss sich abmelden können.
- Wir müssen natürlich überprüfen ob der Nutzer auch wirklich eine E-Mail angegeben hat, auf die er Zugriff hat.
- Wenn der Nutzer sensible Daten wie die E-Mail-Adresse oder Zahlungsdaten ändern möchte, muss er diese Aktion mit seinem Passwort bestätigen.
- Was kann ein Nutzer machen, wenn er sein Passwort vergessen hat? Wir wollen ja nicht von Hand die Passwörter ändern.
- Was mir auch immer sehr gut gefällt, ist die Anmeldung mit Google oder anderen vorhandenen Accounts.

Puh, das ist eine ganze Menge Arbeit. Lass uns das nun Schritt für Schritt durchgehen:

Falsche Logins begrenzen

Stell dir mal vor, ein Kollege oder Freund möchte sich Zugang zu deinem E-Mail-Konto verschaffen. Er kennt deine E-Mail, aber nicht dein Passwort. Er nutzt eine sogenannte ***Brute-Force-Attacke***. Das bedeutet, dass er dein Passwort durch das Ausprobieren unzähliger Passwörter erraten möchte. Je nachdem, wie komplex dein Passwort ist, dauert das mal länger, mal kürzer. Angreifer benutzen neben dem simplen Probieren von Kombination meist sogenannte Wordlists. Auf diesen sind die gängigsten Passwörter gespeichert. *12345678*, *asdfghjk* oder *admin* zählen zu den unsichersten Passwörtern, da diese auf den Wordlists auftauchen und trotz der offensichtlichen Unsicherheit von vielen Nutzern verwendet werden. Die Zeit, die ein Angreifer zum Erraten eines Passworts benötigt, skaliert sehr schnell. Gehen wir von 62 möglichen Zeichen, also Groß- und Kleinbuchstaben sowie Zahlen aus, benötigt ein durchschnittlicher Rechner zum Erraten eines Passworts mit sechs Stellen 26 Sekunden, um 56800235584 Kombinationen auszuprobieren. Dabei gehen wir nicht von der Verwendung einer Wordlist aus, sondern von reinem Raten. Diese Daten habe ich von der Seite *1pw.de*, wo du sehr spannende Tabellen zur Passwortsicherheit im Bezug zu Brute-Force-Atttacken finden wirst: https://www.1pw.de/brute-force.php.

Passwortsicherheit

Auch wenn es nicht wirklich zum Thema dieser Class gehört, ist es mir ein Exkurs zum Thema Passwortsicherheit ein persönliches Anliegen. Weshalb mache ich das? Es gibt leider immer noch viel zu viele Menschen, die nicht nur einfache Passwörter, sondern oft auch immer wieder die gleichen Passwörter nutzen. Mir rollen sich regelmäßig die Fuß- und Fingernägel auf, wenn ich mitbekomme, dass das Passwort *12345678* verwendet wird, besonders bei produktiven Anwendungen und nicht nur in der Entwicklung. Ich müsste lügen, wenn behauptete, noch nie *admin* oder *12345678* in der Entwicklungsumgebung als Passwort verwendet zu haben. Das Problem mit den doppelten Passwörtern vieler Nutzer wird besonders bei kleineren Websites und Foren zur Gefahr, denn dort reicht schon ein kleines Sicherheitsleck. Das Passwort wird auf diesen Sites oft aufgrund mangelnden Wissens nicht gehasht, sondern in Klartextform in der Datenbank mit der E-Mail gespeichert. Der Angreifer könnte nach dem Angriff z. B. versuchen, sich mit diesem Passwort bei Paypal anzumelden. Benutzt du überall das gleiche Passwort, hast du jetzt ein Problem. Gleiches gilt für die Komplexität. Das Passwort sollte kein gängiges und auch nicht zu erratendes Passwort sein. Also nicht deinen Geburtstag, Wohnort oder die Namen deiner Kinder. Dies wären Daten, an die Dritte kommen könnte.

Schau doch mal auf die von mir bereits erwähnten Seite https://www.1pw.de/brute-force.php: Dort wird dir beispielhaft vorgerechnet, dass ein Computer für ein Passwort mit 12 Stellen und 62 verschiedenen Zeichen (Groß-, Kleinbuchstaben + Zahlen) immerhin 47000 Jahre rechnen muss, um es zu erraten. Wenn noch Sonderzeichen hinzukommen, wird es fast unmöglich. Daher empfehle ich zwei grundlegende Dinge für sichere Passwörter:

Tipp 1: Nutze einen Passwortmanager und generiere Passwörter aus Zahlen, Buchstaben und Sonderzeichen. Diese Kombinationen ergeben keinen Sinn und erschließen sich somit anderen Personen nicht. Nachteile: Leider sind sie schwer zu merken und ein guter Passwortmanager kostet etwas. Ich nutze einen Passwortmanager, neige aber mehr und mehr dazu, Tipp zwei zu verwenden.

Tipp 2: Bilde mittels Sätzen Eselsbrücken für Passwörter. Ich bilde Sätze, die sich leicht merken lassen, aber für einen Außenstehenden keinen Sinn ergeben. Hier ein Beispiel für ein Github-Passwort: *ichbrauchegithubwegenderlaravelclassabermeinpinguinhathunger*. Ich hänge immer *A!1* hinten an, damit ich die Konditionen für Sonderzeichen, Großbuchstaben und Zahlen erfülle. Außerdem lässt sich das leicht merken. Also: *ichbrauchegithubwegenderlaravelclassabermeinpinguinhathungerA!1*.

Das Passwort wird niemand erraten, weil es schlicht keinerlei Sinn ergibt. Ich bin mir aber sicher, dass es sich einfacher merken lässt als: *Nk}d/\?56!k,*RZ@*. Obendrein ist es sogar noch um einiges sicherer, da es deutlich länger ist. Wir nehmen also einen Satz (»Ich brauche Github wegen der Laravel Class, aber mein Pinguin hat Hunger«), schreiben diesen klein und ohne Leer- oder Satzzeichen und hängen wegen der Passwortanforderungen die Endung an. Probier diese Technik bei deinem nächsten Passwort aus :)

Kommen wir zurück zum eigentlichen Thema. Auch wenn es schon bei kurzen Passwörtern extrem viele Kombinationen gibt, kann ein moderner Rechner kurze Passwörter ohne Sonderzeichen relativ zügig knacken. Daher müssen wir dieses Ausprobieren auf unserer Seite verhindern. Außerdem können wir die Konditionen für ein Passwort festlegen. Laravel nutzt eingebaut das ***Login Throttling***. Den Trait findest du unter *Illuminate\Foundation\Auth\ThrottlesLogins*. Wenn du den Trait öffnest und die Methoden `maxAttempts` und `decayMinutes` anschaust, sieht du, wie viele falsche Versuche ein Nutzer pro eingestellter Minuten hat. Dies kannst du natürlich anpassen. Normalerweise hat der Nutzer fünf falsche Versuche pro Minute. Dieses Throttling fungiert über die angegebene E-Mail in Kombination mit der IP-Adresse, von der die Anfrage kommt. Wenn du die Anzahl der falschen Versuche oder die Zeit, bis neue Versuche wieder möglich sind, ändern möchtest, kannst du wie in den Methoden des Traits ersichtlich, eine Eigenschaft überschreiben. Diese setzten wir im Constructor des Login Controllers. Ich habe mir erlaubt, die Anzahl der Fehlversuche auf eins und den Cooldown auf 30 Minuten zu ändern:

```
<?php

public function __construct()
{
    $this->middleware('guest')->except('logout');
    $this->maxAttempts = '1';
    $this->decayMinutes = '30';
}
```

Codebeispiel 9

Passwortkonditionen

Ich ändere außerdem sehr gerne die Passwortkonditionen. Am besten eignet sich zur Validierung des Passworts ***Regex***. Regex wie? Regex was? Regex steht für *regular expression*, also regulärer Ausdruck. Damit lassen sich Zeichenketten syntaktisch beschreiben. Wir beschreiben unsere Passwortkonditionen syntaktisch und gleichen das Passwort des Nutzers mit unserer vorgegebenen Beschreibung ab. Warum müssen wir so etwas Kompliziertes machen? Wenn du in die verfügbaren Validierungsregeln schaust, ist keine im Framework enthaltene Validierung wirklich passend. Die Rule *password* prüft, ob der Wert mit dem aktuellen Passwort des Nutzer übereinstimmt, also nicht passend, falls du daran gedacht hast. ;) Mit der Validierung *regex* können wir, wie gesagt, unsere eigene Zeichenvaliderung nutzen. Auf https://ihateregex.io/ siehst du, wie der Ausdruck für das Passwort im Detail funktioniert. Wenn dich das nicht interessiert, kopiere den angegebenen String. Dieser prüft, ob mindestens acht Zeichen, ein Groß- und ein Kleinbuchstabe, eine Zahl und ein Sonderzeichen im Passwort enthalten sind. Du solltest dem Nutzer aber bei der Registrierung die Passwortbedingungen angeben, da der Nutzer sonst im Dunkeln tappt und nicht versteht, warum das Passwort nicht funktioniert. Die Validierung kannst du im *RegisterController* in der `validator`-Methode für das Passwort ändern. Schreibe den regulären Ausdruck zwischen zwei Forward Slashes. Diese begrenzen den regulären Ausdruck.

```
<?
//regulärer Ausdruck für das Passwort:
^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[#?!@$ %^&amp;*-]).{8,}$

//im Validator
'password' => ['required', 'string', 'min:8', 'confirmed',
'regex:/^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[#?!@$ %^&amp;*-]).{8,}$/'],
```

Codebeispiel 10

https://s.webmasters.de/2Q

Bestimmte Aktionen mit dem Passwort schützen

Sicher kennst du das: Obwohl du bereits angemeldet bist, musst du bei Änderungen sensibler Daten wie der E-Mail-Adresse die Änderung mit dem Passwort bestätigen. Das ist auch gut so. Bei Laravel wurde mitgedacht, denn auch diese Funktion gibt es out of the box. Nutze bei der entsprechenden Route bzw. Action die Middleware *password.confirm* im Routing oder im Constructor des Controllers. Der Nutzer wird zur View *resources/views/auth/passwords/confirm* weitergeleitet. Diese kannst du nach Belieben bearbeiten, so wie auch die anderen Auth Views. Hat der Nutzer das Passwort korrekt eingegeben, gelangt er eine Zwiebelschicht weiter. Ebenfalls wird die Middleware für einen definierten Zeitraum außer Kraft gesetzt, sodass der Nutzer nicht dauerhaft das Passwort eingeben muss. Die Zeit kannst du in der Auth Konfig mit der Eigenschaft *password_timeout* einstellen. Der definierte Wert von 10800 ist in Sekunden angegeben und repräsentiert drei Stunden. (10800 / 60 / 60)

Angemeldet bleiben

Du kennst bestimmt auch solche Seiten, die es sich einfach nicht merken können, dass ich mich bereits vor fünf Minuten zuvor angemeldet habe, aber mich zur neuen Anmeldung zwingen, wenn ich den Tab neu lade. Um dem Nutzer die Möglichkeit zu geben, angemeldet zu bleiben, bis er sich manuell ausloggt, muss die Spalte *remember_token* in der Nutzertabelle vorhanden sein. Die Schaltfläche *remember me* ist in der View bereits vorhanden. Ansonsten klappt die Funktion von Haus aus ohne weiteres Zutun. Der *remember_token* wird in einem Cookie gespeichert, und bei Aufruf der Seite wird der im Cookie gespeicherte Token mit dem Token in der Datenbank verglichen. Ohne das Speichern im Cookie wird die erfolgreiche Anmeldung nur in der Session gespeichert. Du kannst dieses Verhalten gut prüfen. Verwende die Methode `viaRemember` der *Auth*-Facade oder die Helper-Funktion *auth*. Die Methode gibt einen booleschen Wert wieder, ob die Anmeldung mittels des Cookies *remember me* erfolgte. Die Session kannst du, wie bereits behandelt, mit `request()->session()->flush()` löschen.

Ein paar Worte zum Abmelden noch, denn dafür brauchen wir keinen eigenen Abschnitt: Nutze die `logout`-Methode auf der *Auth*-Facade oder der `auth()`-Helper-Funktion. Dabei werden Session und Cookie gelöscht. Beim Abmelden gibt es noch ein weiteres praktisches Feature. Versetze dich in folgende Situation: Du bist auf Facebook unterwegs und erinnerst dich an meinen kleinen Exkurs zur Passwortsicherheit. Dein Passwort war zwar immer halbwegs sicher, aber auf jeder Plattform das Gleiche. Du entscheidest dich zu Änderung deines Passworts. Du änderst dein Passwort zwar am PC, bist aber gleichzeitig mit dem Mobilgerät auf Facebook angemeldet. Was passiert? In dem Moment, in dem du das Passwort am PC änderst, wirst du am Handy abgemeldet. Das ist genau das richtige Verhalten. Du musst ja das neue Passwort im Mobilgerät eingeben, um dich wieder anzumelden.

Mit Laravel ist auch das ein Kinderspiel. Kommentiere im Kernel die Middleware *AuthenticateSession* in der Route Group *web* ein. Nachdem du dies getan hast, kannst du die Methode `logoutOtherDevices` auf der *Auth*-Facade oder Helper-Funktion nutzen. Als Parameter muss das neue Passwort mitgegeben werden. Am besten eignet sich ein Formular-Input. Die Methode macht alle anderen angemeldeten Sessions ungültig. Dadurch werden alle anderen angemeldeten Sessions / Geräte abgemeldet.

https://s.webmasters.de/2R

E-Mail-Verifizierung

Es kann schon manchmal echt nervig sein. Es ist aber echt wichtig. Entweder muss man nach dem Registrieren auf einen Link in der E-Mail klicken oder einen erhaltenen Zahlencode in der Website eingeben. Wie aber kannst du dir als Entwickler sonst sicher sein, dass es auch wirklich die E-Mail unseres Nutzers ist, die er da gerade angibt? Weil die E-Mail-Verifizierung mittlerweile Standard ist, hat auch Laravel diese Funktionalität integriert. Hier zeige ich dir, mit welchen Schritten du diese nutzen kannst: Zuerst implementierst du im User-Model den bereits eingebundenen Contract *MustVerifyEmail*.

```
<?

//vorher
class User extends Authenticatable
//nachher
class User extends Authenticatable implements MustVerifyEmail
```

Codebeispiel 11

Anschließend müssen wir sicherstellen, dass die Spalte *email_verified_at* in unserer User-Tabelle vorhanden ist. Das ist normalerweise der Fall, außer du hast die Migration diesbezüglich bearbeitet. Schon können wir die Funktion einsetzen. Die Logik sitzt im *VerificationController*, den wir bereits mit dem *laravel/ui*-Package hinzugefügt haben. Dadurch wird der Link generiert und die E-Mail gesendet. Was noch

fehlt, ist die Registrierung der notwendigen Routes. Füge zu `Auth::routes()` den Key *verify* mit dem Value *true* hinzu. `Auth::routes(['verify'=>true]);`. Außerdem kannst du die Middleware *verified* nutzen, um ausschließlich angemeldeten und verifizierten Nutzern die Anfrage auf eine bestimmte Route zu erlauben. Gut zu wissen: Auch im *VerificationController* gibst du die Weiterleitung nach der erfolgreichen Verifizierung der E-Mail mittels der Eigenschaft *redirectTo* an.

Wichtig: Das Verifizieren der E-Mails funktioniert an dieser Stelle nicht, da wir die E-Mail-Funktion noch nicht eingeführt haben. Du musst dich daher noch ein wenig gedulden. Gleiches gilt für die E-Mail der Passwortzurücksetzen-Funktion, die ich im Folgenden zeige. Beide Funktionen werden wir in unserer Applikation nach der Lektion **Benachrichtigungen** nutzen.

Neben der E-Mail-Verifizierung ist vor allem bei Banken und Finanzdienstleistern die Verifizierung deiner Handynummer erforderlich. Dies dient der Zwei-Faktor-Authentifizierung, die das Sicherheitslevel noch einmal anhebt. Ich zeige dir das in einem kurzen Video. Aber Vorsicht, ich verwende dabei die APIs von Twilio, die kostenpflichtig sind! Ich will dich damit nicht zum Kauf dieser Dienstleistung ermutigen, sondern dir den Vorgang vor dem möglichen Einsatz in einem beruflichen Projekt beispielhaft vorführen.

https://s.webmasters.de/2S

Passwörter zurücksetzen

Aus meiner Aufzählung von vorhin fehlt noch das Zurücksetzen von Passwörtern. Dass diese Funktion wichtig und nützlich ist, kann jeder bestätigen, der schon einmal sein Passwort vergaß. Umso glücklicher ist man, wenn man einen Link zum Zurücksetzen via E-Mail anfordern konnte: Draufklicken und einfach, schnell und unkompliziert ein neues Passwort festlegen. In Laravel bewerkstelligen wir das ohne auch nur eine Zeile Code schreiben zu müssen. Natürlich haben wir die Möglichkeit verschiedener Anpassungen wie z. B. die Weiterleitung nach dem Zurücksetzen des Passworts. Diese kannst du im *ResetPasswordController* vornehmen. Außerdem legst du in der Auth Config die Ablaufzeit der via E-Mail zugesandten Tokens fest, wobei eine Stunde als Standard vorgegeben ist. Ich persönlich stelle die Ablaufzeit gern auf 24 Stunden ein, um den Nutzer nicht zu hetzen. Außerdem lassen sich die E-Mails, die bei der E-Mail-Verifizierung und dem Passwortzurücksetzen gesendet werden, anpassen. Dazu aber später mehr.

Übung 6: Schreibe selbst Code und implementiere das Login System

Baue in deine Laravel-Applikation das Login-System ein, falls du dies noch nicht gemacht hast. Ein Nutzer soll sich anmelden, registrieren und abmelden können. E-Mail-Verifizierung und Zurücksetzen des Passworts sind nicht Teil der Übung. Ich habe dazu kein Lösungsvideo gedreht. Vielmehr sind die einzelnen Videos dieser Lektion die Lösung. :) Nutze diese Übung als Selbsttest, ob du das Thema verstanden hast.

4.2 Social Login

Wir verwenden Laravel Socialite, um einen Login via Github in unserer Applikation zu ermöglichen. Socialite verwendet das OAuth-Protokoll, das einer sicheren, standardisierten Anmeldung dient und von den meisten Unternehmen genutzt wird. Für den Nutzer hat es den Vorteil, sich mit vorhandenen Accounts anmelden zu können und sich keine neuen Login-Daten ausdenken und merken zu müssen. Jedes mal, wenn du dich via Facebook oder Google auf einer anderen Website anmeldest, verwendest du, wahrscheinlich unbewusst, OAuth. Du kannst bei Socialite auch Google, Facebook, Twitter usw. ver-

wenden. Beachte aber, dass du Zugangsdaten für den OAuth-Service von den Providern benötigst. Hier eine Liste aller verfügbaren Socialite-OAuth-Provider: https://socialiteproviders.netlify.com/about.html.

Zuerst fügen wir Socialite via Composer hinzu: `composer require laravel/socialite`.

Nachdem wir dies erfolgreich getan haben, müssen wir unseren Service der gleichnamigen Config-Datei hinzufügen. Nachfolgenden Codeschnipsel habe ich von der Website des Socialite-Providers kopiert:

```
<?php

//config/services.php
'github' => [
    'client_id' => env('GITHUB_KEY'),
    'client_secret' => env('GITHUB_SECRET'),
    'redirect' => env('GITHUB_REDIRECT_URI')
],
```

Codebeispiel 12

Wir müssen jetzt nur noch die Zugangdaten erhalten. Dazu meldest du dich mit deinem vorhandenen Account bei Github an. Anschließend gehst du oben rechts auf deinen Namen und dann auf *Settings*. In den Einstellung klickst du ganz links unten auf *Developer Settings*, dann auf OAuth Apps und *Register a new Application*. Alternativ kannst du auch den folgenden Link verwenden: https://github.com/settings/applications/new. Die Callback URL bei der Registrierung von Github muss gleich der *Redirect URI* unserer Service Config des Github-Providers in Laravel sein. Diese Route erstellen wir aber gleich. Stell dir die Anmeldungsschritte folgendermaßen vor: Der Nutzer klickt in unserer Applikation auf einen Link *anmelden mit Github*. Unsere Laravel-App leitet zu Github weiter und übermittelt den Github Key und unser Secret. Dadurch kann Github unsere Applikation identifizieren und authorisieren. Anschließend meldet der Nutzer sich bei Github an und bestätigt die Anmeldung auf unserer Seite. Github leitet dann an die Callback URL unserer Applikation weiter und übermittelt dabei die Daten des bei Github angemeldeten Nutzers.

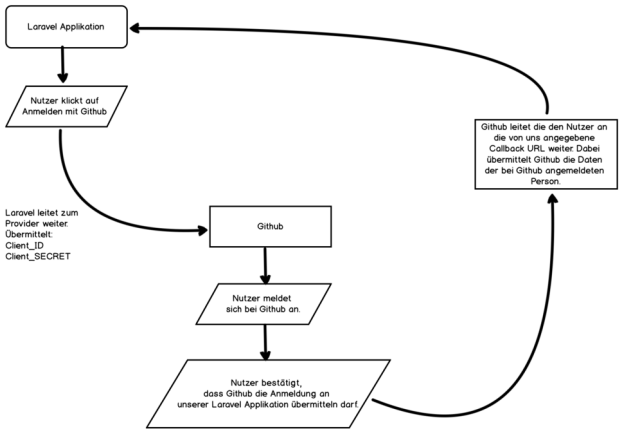

***Abb. 2** OAuth-Workflow in Laravel Socialite*

Ich verwende die Callback Route *auth/github/callback*, die ich auch sofort erstelle. Das ausgefüllte Formularfeld beim Hinzufügen einer OAuth App zu Github sieht in meinem Fall folgendermaßen aus:

- Application Name: *advanced-laravel*
- Homepage URL: *http://advanced-laravel.test*
- Authorization Callback URL: *http://advanced-laravel.test/auth/github/callback*

Nach dem Hinzufügen der OAuth App zu Github wird dir die Client ID und Client Secret angezeigt. Weise beide den entsprechenden ENV-Variablen der Service Config zu. Die Konfiguration ist damit endlich abgeschlossen. Nun benötigen wir die zwei angesprochenen Routes. Zum einen die Route, die den Nutzer zu Github weiterleitet, zum anderen die Callback Route, um die Antwort von Github nach der Anmeldung zu empfangen. Für diese zwei Routes erstellen wir noch entsprechende Actions in unserem Login Controller. Dabei nutzen wir die *Socialite* Facade, um auf die Methoden von Socialite zugreifen zu können. Anschließend registrieren wir die zwei Routes. Beachte bitte dabei, dass die definierte Callback Route mit der im Routing übereinstimmt.

```
<?
//login controller
public function redirectToProvider()
{
    return Socialite::driver('github')->redirect();
}
public function handleProviderCallback()
{
    $oauthUser = Socialite::driver('github')->user();
}
```

Codebeispiel 13

```
<?php

Route::get('auth/github', 'Auth\LoginController@redirectToProvider');
Route::get('auth/github/callback', 'Auth\LoginController@handleProviderCallback');
```

Codebeispiel 14

Zeit für den ersten Test. Füge am besten noch ein Anchor-Tag zu deiner Welcome View hinzu, der auf die Route zur Weiterleitung zu Github verweist. Damit wir das Ergebnis von Socialite sichtbar machen, gebe ich mit `dump()` den Nutzer in der Action `handleProviderCallback` aus. Dieser Wert sieht bei einer Anmeldung mit meinem Github Account folgendermaßen aus (einige Werte habe ich aus Datenschutzgründen mit … überschrieben):

```
<?

Laravel\Socialite\Two\User {#307 ▼
  +token: "8b7db20d31a9aae4c8ee887ca0bcb45c455efd20"
  +refreshToken: null
  +expiresIn: null
  +id: 1234
  +nickname: "Cosnavel"
  +name: "Cosnavel"
  +email: "..."
  +avatar: "..."
  +user: array:31 [▼
    "login" => "..."
    "id" => ...
    "node_id" => "..."
    "avatar_url" => "..."
    "gravatar_id" => ""
    "url" => "https://api.github.com/users/Cosnavel"
    "html_url" => "https://github.com/Cosnavel"
    "followers_url" => "https://api.github.com/users/Cosnavel/followers"
    "following_url" => "https://api.github.com/users/Cosnavel/following{/other_user}"
    "gists_url" => "https://api.github.com/users/Cosnavel/gists{/gist_id}"
    "starred_url" => "https://api.github.com/users/Cosnavel/starred{/owner}{/repo}"
    "subscriptions_url" => "https://api.github.com/users/Cosnavel/subscriptions"
    "organizations_url" => "https://api.github.com/users/Cosnavel/orgs"
    "repos_url" => "https://api.github.com/users/Cosnavel/repos"
    "events_url" => "https://api.github.com/users/Cosnavel/events{/privacy}"
```

```
    "received_events_url" => "https://api.github.com/users/Cosnavel/received_events"
    "type" => "User"
    "site_admin" => ...
    "name" => "Cosnavel"
    "company" => "..."
    "blog" => "www.niclaskahlmeier.de"
    "location" => "currently based in Nuremberg"
    "email" => "..."
    "hireable" => false
    "bio" => """
      Software Dev
      ❤? for React
      """
    "public_repos" => 55
    "public_gists" => 0
    "followers" => 0
    "following" => 2
    "created_at" => "2018-08-14T22:09:32Z"
    "updated_at" => "2020-03-22T18:47:47Z"
  ]
}
```

Codebeispiel 15

Wir haben jetzt zwar eine ganze Menge Informationen, aber noch keinen Nutzer in unserer Applikation erstellt. Du kannst ein paar selbsterklärende Methoden nutzten, mit denen die Inhalte einer OAuth-Antwort abgerufen werden können:

- `$user->getId();`
- `$user->getNickname();`
- `$user->getName();`
- `$user->getEmail();`
- `$user->getAvatar();`

Wir haben beim Erstellen des Nutzers jedoch ein Problem: Entweder möchte sich der Nutzer mit Github registrieren und ist noch nicht in unserer Datenbank vorhanden, oder unser Nutzer ist in der Datenbank vorhanden und möchte sich nur erneut anmelden. Du kennst das bestimmt von schlecht programmierten Applikationen, wo es für Login und Registrierung verschiedene Social Logins gibt, die dann bei falscher Verwendung eine kryptische Fehlermeldung ausgeben.

Um dieses Problem zu lösen, erstellen wir eine kleine Funktion, deren Aufgabe es ist zu prüfen, ob der Nutzer schon vorhanden ist. Dazu erstellen wir ein Model mit Migration für die sozialen Identitäten, da wir dabei mitbedenken müssen, dass mehrere Provider von Socialite genutzt werden könnten. Zwischen diesen Providern sollten wir differenzieren. In der Tabelle muss die *user_id* mit der ID des Providers und dem Provider-Typ gespeichert werden.

```
php artisan make:model SocialAuth –m
```

Migration & migrate:

```
<?php

Schema::create('social_auths', function (Blueprint $table) {
    $table->id();
    $table->bigInteger('user_id');
    $table->string('provider_name');
    $table->string('provider_id')->unique();
    $table->timestamps();
});
```

Codebeispiel 16

Im User-Model und im SocialAuth-Model füge ich noch eine Beziehung der beiden hinzu. Du kannst auch eine One-To-Many-Beziehung anstatt der One-To-One-Beziehung nutzen, wenn der Nutzer sich mit mehreren Diensten verbinden darf.

```
<?
//user model
public function oauth()
{
    return $this->hasOne('App\SocialAuth');
}
```

Codebeispiel 17

```
<?
//socialauth model
protected $fillable = ['user_id', 'provider_name', 'provider_id'];

public function user()
{
    return $this->belongsTo('App\User');
}
```

Codebeispiel 18

Anschließend modifiziere ich die `handleProviderCallback()`-Methode im *LoginController* so, dass wir einen neuen Nutzer erstellen, wenn dieser noch nicht verfügbar ist, oder einen existierenden Nutzer abrufen.

```
<?
public function handleProviderCallback()
{
    $oauthUser = Socialite::driver('github')->user();

    $oauthUser = $this->findOrCreateUser($oauthUser, 'github');

    Auth::login($oauthUser, true);

    return redirect($this->redirectTo);
}
public function findOrCreateUser($oauthUser, $provider)
{
    $existingOAuth = SocialAuth::where('provider_name', $provider)
        ->where('provider_id', $oauthUser->getId())
        ->first();

    if ($existingOAuth) {
        return $existingOAuth->user;
    } else {
        $user = User::whereEmail($oauthUser->getEmail())->first();

        if (!$user) {
```

```
            $user = User::create([
                'email' => $oauthUser->getEmail(),
                'name'  => $oauthUser->getName(),
            ]);
        }

        $user->oauth()->create([
            'provider_id'   => $oauthUser->getId(),
            'provider_name' => $provider,
        ]);

        return $user;
    }
}
```

Codebeispiel 19

Super! Probier jetzt doch mal aus, ob die Anmeldung via Github funktioniert. Ich bin mir ziemlich sicher, dass jetzt bei dir ein Fehler bezüglich des Passworts auftaucht. Richtig, wir müssen in der User-Migration noch die Passwortspalte auf *nullable* setzen, da der Nutzer ja kein Passwort definiert. Anschließend kannst du dich via Github anmelden :)

Zwei Dinge sind mir bei diesem Code noch ein Dorn im Auge. Vielleicht ist es dir auch aufgefallen, aber wir geben keinen Fehler wieder, wenn bei der Anmeldung bei Github etwas schief geht oder der Nutzer diese abbricht. Github ist zudem als Provider fest im Code definiert. Wenn wir einen neuen Provider hinzufügen wollen, müssen wir an mehreren Stellen den Code ändern. Mit diesem Verhalten verstoßen wir jedoch gegen die zweite SOLID-Regel. Das Open-Closed-Prinzip besagt, dass sich neue Funktionen ohne Modifikation des vorhandenen Codes hinzufügen lassen sollen. Damit wir das Open-Closed-Prinzip einhalten, ändere ich den Code dahingehend. Außerdem füge ich Gitlab als Provider zum Anmelden hinzu. All dem kannst du in meinem Video folgen. Den Code lade ich dir ebenfalls hoch. Viel Spaß!

Übung 7: Refactoring der Socialite Anmeldung

Ich habe mir überlegt, dass du den Code so ändern könntest, dass er mir kein Dorn mehr im Auge ist. Die Vorraussetzungen dazu stehen über der Übung. Selbstverständlich bestehe ich nicht auf Gitlab als zusätzlichen Provider. Suche dir einfach irgendeinen Socialite Provider aus und implementiere diesen. Achte vor allem darauf, das Open-Closed-Prinzip einzuhalten.

4.3 Teste dein Wissen

1. Warum wird das Passwort als Hash gespeichert?

   Bitte ankreuzen:

   - ☐ Damit das Passwort nicht in reiner Textform in der Datenbank gespeichert wird.
   - ☐ Da sich der Hash nicht mehr in das ursprüngliche Passwort zurück entschlüsseln lässt.
   - ☐ Da das Passwort verschlüsselt wird und sich einfach wieder entschlüsseln lässt.
   - ☐ Der Hash wird nur verwendet, da ein vom Nutzer gewähltes, kurzes Passwort dadurch länger wird und dadurch sicherer aussieht.

2. Markieren Sie die Codeschnipsel als richtig, mit denen die Nutzer ID erfolgreich in der Variable *$id* gespeichert wird.

Bitte ankreuzen:

☐
```
<?php

$id = auth()->user()->id;
```

☐
```
<?
 //controller
 use Illuminate\Http\Request;
 ...
function store(Request $request)
{
$id = $request->user()->id;
}
```

☐
```
<?php
$id = request()->user()->id;
```

☐
```
<?php

$id = Illuminate\Support\Facades\Auth::user()->id;
```

3. Ändern Sie die Route, zu der der Nutzer weitergeleitet wird, wenn er ohne angemeldet zu sein versucht, eine durch die Middleware *auth* gesicherte Route zu besuchen, auf *login-first*.

4. Wie funktioniert der Anmeldevorgang mit dem offiziellen OAuth-Package Socialite? Definieren Sie die korrekte Reihenfolge der Schritte. Verwenden sie hierfür die Zahlen 1 bis 6.

Index