# Erstellung einer API mit Laravel

## Inhalte

1. Einleitung
2. Erste Schritte
   1. Erstellen des Domain-Model und Migration
   2. Crud-Operation der Domain
   3. Erstellen von Pages inklusive Abhängigkeit zur Domain
   4. Automatisches Erstellen der Seiten durch Observer
3. Automatisierungen und Optimierungen mit Jobs und Queues
   1. Die Seitenanalyse
      1. Regelmäßgie Analyse von Seiten
      2. Durchschnitts-Bewertung von Seiten
   2. Domainbewertung und Sitemap
      1. Durchschnittsbewertung der Domain
      2. Aktualisierung der Sitemap
   3. Automatische Aktualisierung des Frontends durch Web-Sockets

## Einleitung

Die Zuverlässigkeit von Webseiten ist einer der wichtigstens Aspekte von Agenturen oder selbständigen Web-Entwicklern. Ein regelmäßiger Check der Webseite kann dabei helfen, frühzeitig Bugs oder Fehler zu erkennen und anzupassen.

Je größer die Anzahl der erstellten Webseiten oder selbst die Anzahl der Unterseiten einer Webseite, desto mehr Zeit ist für einen solchen Check notwendig. Mit diesem Projekt soll dieser Prozess optimiert werden können. Im Rahmen dieses E-Learnings wird eine Anwendung erstellt, die automatisiert Webseiten in Bezug auf Leistung, Barriere-Freiheit und SEO bewertet.

Dazu kann ein späterer Nutzer über eine Oberfläche gewünschte Webseiten hinzufügen. Basierend auf der Sitemap einer Webseite werden alle Seiten analysiert und diese dem Nutzer in einem Dashboard angezeigt.

Laravel ist eines der am weitesten verbreiten PHP-Frameworks. Im Rahmen dieses Projektes wird Laravel mit eine API-Schnittstelle erstellt. Ebenso wurde ein Frontend erstellt, welches aber hier nicht näher beschrieben wird. Das fertige Projekt kann <hier> eingesehen werden.

Für das Projekt wurden folgende Ziele definiert werden:

* Speichern von Domains durch Nutzer-Eingabe.
* Analyse der Domain-Sitemap, sowie automatisches erstellen der dazugehörigen Seiten.
* Regelmäßige Analyse der Seiten durch die Google Pagespeed Api.
* Automatische Aktualisierung der Frontends nach neuer Bewertung.
* *[Erstellung eines Dashboard]*

Folgend wir die Herangehenweise des Projektes dargestellt.

## Erste Schritte

### Erstellen des Domain-Model und Migration

Nach der Installation ([Installation Laravel](Installation%20von%20Laravel)) sollen zunächst ein Model mit Controller und Route angelegt werden. Laravel liefert hierfür eine Unterstützung: den Artisan. Der Artisan stellt eine CLI dar, die Entwickler dabei unterstütz neue Dateien innerhalb der Laravel-Applikation zu erstellen. Hierdurch werden bereits die richtigen Namespaces gesetzt, die Ordner-Struktur bleibt übersichtlich und eine Boilerplate für notwendige Funktionen erstellt.

Mit dem Artisan werden nun folgende Dateien erstellt:

* php artisan make:controller DomainController --api: Erstellt den DomainController inklusive der CRUD-Methoden.
* php artisan make:model Domain -m: Erstellt das Domain-Model inklusive dazugehöriger Datenbank-Migration.

*Laravel arbeitet mit dem Prinzip der* ***Datenbankmigration****. Dies hat den Vorteil, dass Tabellen-Strukturen innerhalb einer Datei definiert werden können und diese nicht per Hand in einer Datenbankoberfläche erstellt werden müssen. Zusätzlich können diese Dateien auch bspws. auf Github hochgeladen werden und so die Tabellen-Struktur versioniert oder mit anderen geteilt werden.*

Nun können in der Migration, innerhalb der up-Methode, die Eigenschaften des Models wie folgt definiert werden:

public function up()  
 {  
 Schema::create('domains', function (Blueprint $table) {  
 $table->id();  
 $table->timestamps();  
  
 $table->string('name')->unique();  
 $table->string('favicon')->default(null);  
 $table->string('sitemap')->unique();  
 $table->string('url')->unique();  
 $table->boolean('sitemapFound')->default(true);  
 });  
 }

Innerhalb des Models bestimmen wir nun die Felder, die durch eine Nutzer-Eingabe gesetzt werden dürfen:

class Domain extends Model  
{  
 use HasFactory;  
 protected $fillable = ['name', 'favicon', 'sitemap', 'sitemapFound', 'url'];  
}

Für die Migration muss zunächst Laravel die Datenbank-Verbindung mitgeteilt werdne. Dies kann in der .env-Datei definiert werden:

DB\_CONNECTION=mysql  
DB\_HOST=[DATENBANK\_SERVER]  
DB\_PORT=3306  
DB\_DATABASE=[DATENBANK\_NAME]  
DB\_USERNAME=[NUTZER\_DATENBANK]  
DB\_PASSWORD=[NUTZER\_PASSWORT\_DATENBANK]

Über den Artisan php artisan migrate wird die Migration in die Datenbank integriert.

### Crud-Operation der Domain

Folgend werden für die Domain die klassischen CRUD-Operationen und Routes definiert.

Über das Modell können Querys erstellt werden. So können zum Beispiel über die Methode ::all() alle vorhandenen Domains abgefragt werden:

public function index(){  
 $domains = Domain::all();  
 return response($domains)  
}

Einzelne Domains, basierend auf einer ID, können wie folgt abgefragt werden:

public function show($id){  
 $domain = Domain::find($id);  
 return response ($domain);  
}

Auf diesem Eintrag können dann weitere Operationen ausgeführt werden, wie Beispielsweise das löschen:

public function destroy($id){  
 $domain = Domain::find($id)->delete();  
 return response(['message' => 'Record deleted'], 200);  
}

Sobald mit Nutzereingaben gearbeitet wird, müssen diese unbedingt validiert werden. Hierfür wird der Laravel Validator verwendet. Die Validierung kann direkt innerhalb des Controllers stattfinden:

public function store(Request $request){  
 $request->validate([  
 'name' => ['required', 'string', 'unique:domains'],  
 'favicon' => ['string',],  
 'sitemap' => ['required', 'string',],  
 'url' => ['required', 'string', 'unique:domains'],  
 ]);  
}

Zur Wiederverwendbarkeit, sowie zur Übersicht, kann diese Logik in eine Request Datei ausgelagert werden. Über php artisan make:request CreateDomainRequest wird diese erstellt und die Validierungs-Logik darin integriert:

public function authorize(){  
 return true;  
}  
   
public function rules(){  
 return [  
 'name' => ['required', 'string', 'unique:domains'],  
 'favicon' => ['string',],  
 'sitemap' => ['required', 'string',],  
 'url' => ['required', 'string', 'unique:domains'],  
 ];  
}

Wichtig ist, das die authorize-Methode true zurückgibt. Diese bestimmt, ob der Request autorisiert ist. Damit der Controller weiß, dass er den CreateDomainRequest verweden soll, muss der request Parameter entsprechend angepasst werden. Durch diese Middleware wissen wir, dass alle Daten innerhalb der store-Methode validiert wurden. Nun kann mit den Daten eine neue Domain erstellt und zurückgegeben werden:

public function store(CreateDomainRequest $request){  
 $domain = Domain::create($request->all());  
 return response ($domain,201);  
}

Ähnlich kann die update-Methode, inklusive Request, angepasst werden:

public function update(UpdateDomainRequest $request, $id){  
 $domain = Domain::find($id)->update($request->all());  
 return response($domain, 200);  
}

Mit folgendem Request:

public function rules(){  
 return [  
 'name' => ['string'],  
 'favicon' => ['string',],  
 'sitemap' => ['string',],  
 'url' => ['string'],  
 'sitemapFound' => ['boolean',]  
 ];}

Falls eine Methode ohne passender ID aufgerufen wird, entsteht aktuell ein Server-Error. Um dies zu verhindern, kann die ::findOrFail Methode verwendet werden. Schöner hingegen ist jedoch die ‘Dependency Injection’. Hierbei wird dem Parameter ein Typ mitgegeben. Laravel erkennt selbständig, ob der Typ mit der angegebenen ID existiert und gibt entweder einen 404-Error zurück oder die Methode wird ausgeführt. Abschließend schaut unser Controller also so aus:

// ORM

<?php  
  
namespace App\Http\Controllers;  
use App\Http\Requests\CreateDomainRequest;  
use App\Http\Requests\UpdateDomainRequest;  
use App\Models\Domain;  
  
class DomainController extends Controller  
{  
 /\*\*  
 \* Display a listing of the resource.  
 \*  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function index(){  
 $domains = Domain::all();  
 return response($domains);  
 }  
   
  
 /\*\*  
 \* Store a newly created resource in storage.  
 \*  
 \* @param CreateDomainRequest $request  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function store(CreateDomainRequest $request){  
 $domain = Domain::create($request->all());  
 return response($domain,201);  
 }  
 /\*\*  
 \* Display the specified resource.  
 \*  
 \* @param int $id  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function show(Domain $domain)  
 {  
 return response($domain);  
 }  
  
 /\*\*  
 \* Update the specified resource in storage.  
 \*  
 \* @param UpdateDomainRequest $request  
 \* @param Domain $domain  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function update(UpdateDomainRequest $request, Domain $domain)  
 {  
 $domain = $domain->update($request->all());  
 return response($domain);   
 }  
  
 /\*\*  
 \* Remove the specified resource from storage.  
 \*  
 \* @param Domain $domain  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function destroy($domain)  
 {  
 $domain->delete();  
 return response(['message' => 'Record deleted'], 200);  
 }  
}

Abschließend wird der Controller nun mit den Routes verknüpft. Für eine Api-Schnittstelle werden diese innerhalb der api.php definiert. Für Api-Crud-Operationen bietet Laravel hier ebenfalls als Unterstützung die apiResource:

Route::apiResource('domain', DomainController::class);

mit php artisan route:list können alle Routes angezeigt werden:

GET|HEAD api/domain ........... domain.index › DomainController@index  
POST api/domain ........... domain.store › DomainController@store  
GET|HEAD api/domain/{domain} .. domain.show › DomainController@show  
PUT|PATCH api/domain/{domain} .. domain.update › DomainController@update  
DELETE api/domain/{domain} .. domain.destroy › DomainController@destroy

### Erstellen von Pages inklusive Abhängigkeit zur Domain

Eine Domain soll eine one-to-many Beziehung zu den entsprechenden Unterseiten ( Page) haben. Darüber hinaus besitzt eine Page eine url, sowie eine error Eigenschaft. Hierfür werden analog zur Domain zunächst Migration, Model, Controller und Routing erstellt:

1. Migration:  
   php artisan make:model Page -m  
   Bereits bei der Migration wird die Abhängigkeit definiert. Über eine foreignId wird das ‘Eltern’-Model festgelegt. Über constrained geht Laravel von den Standard-Namen aus. In diesem Fall: Tabelle *domains*, Spalte *id*. Alternativ können diese auch manuell, als weitere Parameter von foreignId, definiert werden.

public function up()  
 {  
 Schema::create('pages', function (Blueprint $table) {  
 $table->id();  
 $table->foreignId('domain\_id')->constrained()->cascadeOnDelete();  
 $table->boolean('error')->default(false);  
 $table->string('url')->unique();  
 $table->timestamps();  
 });  
 }

1. Model

class Page extends Model{  
 use HasFactory;  
 protected $fillable = ['url', 'domain\_id', 'error'];  
}

1. Controller

<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Requests\PageRequest;  
use App\Models\Page;  
use App\Models\Rating;  
use Illuminate\Support\Facades\Http;  
  
class PageController extends Controller  
{  
  
  
 /\*\*  
 \* Display a listing of the resource.  
 \*  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function index()  
 {  
 $pages = Page::all();  
 return response($pages);  
 }  
  
 /\*\*  
 \* Store a newly created resource in storage.  
 \*  
 \* @param PageRequest $request  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function store(PageRequest $request)  
 {  
 $page = Page::create($request->all());  
 return response($page, 201);  
 }  
  
 /\*\*  
 \* Display the specified resource.  
 \*  
 \* @param Page $page  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function show(Page $page)  
 {  
 return response($page);  
 }  
  
 /\*\*  
 \* Update the specified resource in storage.  
 \*  
 \* @param PageRequest $request  
 \* @param Page $page  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function update(PageRequest $request, Page $page)  
 {  
 $page->update($request->all());  
 return response($page);  
 }  
  
 /\*\*  
 \* Remove the specified resource from storage.  
 \*  
 \* @param Page $page  
 \* @return \Illuminate\Http\Response  
 \*/  
 public function destroy(Page $page)  
 {  
 $page->delete();  
 return response(['message' => 'Recorded deleted'], 200);  
 }  
}

1. Routing  
   Die api.php wird wie folgt ergänzt:

Route::apiResource('domain', DomainController::class);  
Route::apiResource('pages', PageController::class);

Damit die Abhängigkeiten auch abrufbar sind, werden die Models wie folgt aktualisiert:

class Page extends Model  
{  
 protected $fillable = ['url', 'domain\_id', 'error'];  
  
 public function domain(){  
 return $this->belongsTo(Domain::class);  
 }  
}  
  
class Domain extends Model  
{  
 protected $fillable = ['name', 'favicon', 'sitemap', 'sitemapFound', 'url'];  
  
 public function pages(){  
 return $this->hasMany(Page::class);  
 }  
}

Es gibt es mehrere Optionen, die Abhängikeiten abzufragen beziehungsweise diese zurückzugeben. Die drei häufigsten Varianten sind: 1. Domain wird standardmäßig mit den Seiten zurückgegeben. Hierfür muss das Domain Model angepasst werden: protected $with=['pages']; 2. Pages werden bei bestehender Domain-Collection nachgeladen $domain->load('pages'); 3. Domain und Pages werden gleichzeitig angefragt `Domain::with(‘pages’)

> ->get()`

Die show Methode des DomainControllers soll standardmäßig Domain inklusive Pages zurückgeben. Hierfür wird diese wie folgt angepasst:

public function show(Domain $domain){  
 $domain->load('pages');  
 return response($domain);  
}

### Automatisches Erstellen der Seiten durch Observer

Seiten sollen nicht nur manuell erstellt werden können, sondern automatisch nach dem Erstellen der Domain. Hierfür soll die Sitemap analysiert werden und basierend darauf die entsprechenden Seiten erstellt werden. Hierfür bieten sich Observer an. Diese geben eine Möglichkeit auf Events zu reagieren. So wird zum Beispiel beim Erstellen einer Domain die *created* Methode des Observers aufgerufen. Unabhängig davon, wodurch die Domain erstellt wurde.

Zunächst erstellen wir den Observer und verknüpfen diesen mit dem Domain-Model: php artisan make:observer DomainObserver --model=Domain, sowie innerhalb der EventServiceProvider:

public function boot(){  
 Domain::observe(DomainObserver::class);  
}

Innerhalb der created Methode des Observer integrieren wir nun die Logik zum Erstellen der Page Records:

public function created(Domain $domain)  
 {  
 try {  
 $pages = $this->fetchSitemap($domain->sitemap);  
  
 //create Pages  
 foreach ($pages as $sitemapPage) {  
 Page::firstOrCreate([  
 'domain\_id' => $domain->id,  
 'url' => $sitemapPage,  
 ]);  
 $domain->update(['sitemapFound' => '1']);  
 }  
 } catch (\e) {  
 $domain->update(['sitemapFound' => '0']);  
 }  
 }  
  
 public function fetchSitemap(string $url)  
 {  
  
 $xml = simplexml\_load\_file($url);  
 $pages = [];  
  
 // get page urls from sitemap  
 foreach ($xml as $element) {  
 $pages[] = $element->loc[0];  
 }  
 return $pages;  
 }

## Automatisierung und Optimierungen mit Jobs und Queues

### Die Seitenanalyse

#### Model, Migration und Controller

Im Rating Model werden die Ergebnisse der Seiten-Analyse in Form von Integern für *performance*, *seo* und *accessibility* gespeichert. Das Domain- und Page-Model sollen Abhängigkeiten zum Rating-Model haben, weshalb hierfür polimorphische Beziehungen benutzt werden. Dabei kann eine Page mehrere besitzen, die Domain lediglich eins. Hierfür bieten sich polimorphische Abhängigkeiten an. Diese ermöglichen es, dass ein Model in einer Abhängigkeit mit verschiedensten Models steht. Zunächst wird das Model inklusive Migration mit php artisan make:model Rating -m erstellt. Die Migrations-Datei des Models wird wie folgt angepasst:

public function up(){  
 Schema::create('ratings', function (Blueprint $table) {  
 $table->id();  
 $table->timestamps();  
 $table->morphs('ratable');  
  
 $table->integer('seo');  
 $table->integer('performance');  
 $table->integer('accessibility');  
});

Die morphs-Methode des Blueprints erstellt dabei die Spalten *ratable\_id* und *ratable\_type*. Die Id verweist auf die id des abhängigen Records, im type wird der Model-Typ angegeben. In diesem entweder *App* oder *App*. Das Model schaut nun so aus:

class Rating extends Model{  
 use HasFactory;  
 protected $fillable = ['seo', 'performance', 'accessibility'];  
   
 public function ratable()  
 {  
 return $this->morphTo();  
 }  
}

Die Page und Domain Models werden ebenfalls erweitert:

class Page extends Model{  
 protected $fillable = ['url', 'domain\_id', 'error'];  
  
 public function domain(){  
 return $this->belongsTo(Domain::class);  
 }  
   
 public function ratings(){  
 return $this->morphMany(Rating::class, 'ratable')  
 ->orderBy('created\_at','DESC');  
 }  
   
 public function averageRating(){  
 return $this->morphOne(Rating::class, 'ratable');  
 }  
}  
  
class Domain extends Model  
{  
 protected $fillable = ['name', 'favicon', 'sitemap', 'sitemapFound', 'url'];  
  
 public function pages(){  
 return $this->hasMany(Page::class);  
 }  
   
 public function rating(){  
 return $this->morphOne(Rating::class, 'ratable');  
 }  
}

Da es für das Frontend möglich sein soll, die letzten Ratings abzufragen, wird ein Controller erstellt, diesmal ohne Flag: php artisan make:controller RatingController. Innerhalb des Controllers werden folgende Methoden implementiert:

class RatingController extends Controller{  
 public function index(): Response{  
 $ratings = Rating::all();  
 return response($ratings);  
 }  
 // All Ratings related to Page  
 public function pages(){  
 $ratings = Rating::where('ratable\_type', 'App\\Models\\Page')  
 ->with(['ratable', 'ratable.domain'])  
 ->get();  
 return response($ratings);  
 }  
 // All today created Ratings  
 public function today(): Response{  
 $ratings = Rating::where('created\_at', '>=', Carbon::today())->get();  
 return response($ratings);  
 }  
}

Diese werden entsprechend verknüpft:

Route::group(['prefix' => 'ratings'], function () {  
 Route::get('', [RatingController::class, 'index']);  
 Route::get('today', [RatingController::class, 'today']);  
 Route::get('pages', [RatingController::class, 'pages']);  
});

#### Das Erstellen eines Ratings

Ratings sollen nicht durch Nutzer-Eingaben erstellt werden, sondern basierend auf den Ergebnissen der PageSpeed-Insights Analyse. Für die Analyse wird ein Api-Key benötigt, dieser ist hier erhältlich. Dieser wird unter dem Key GOOGLE\_CWV\_KEY in der .env integriert.

Das Rating einer Seite soll von überall ausgelöst werden können. Hierfür bieten sich Laravel Jobs an. Durch den Artisan wird der erste Job erstellt: php artisan make:job RatePage

Der Job wird in zwei Schritte aufgeteilt:

1. Finde die zu bewertende Page. Dabei werden zunächst alle Seiten ohne Rating bevorzugt. Haben alle Seiten ein Rating, werden alle Seiten abgefragt, die heute noch nicht bewertet wurden. Sollten hier ebenfalls keine mehr vorhanden sein, wird *false* zurückgegeben. Grundsätzlich werden Seiten mit einer *error* ausgeschlossen.
2. Analysiere die Seite und gebe das Rating zurück. Sollte die Bewertung fehlschlagen, wird der Error-Status der Seite auf 1 gestellt

Laravel benutzt als Boolean-Werte nicht true oder false, sondern 1 und 0

1. Füge das Rating zur Seite hinzu. Falls die Seite bereits mehr als 10 Ratings besitzt, lösche das letzte.

Der entsprechende Job sieht dann so aus:

class RatePage implements ShouldQueue{  
  
 use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;  
   
 public Page $page;  
 private string $key;  
  
  
 /\*\*  
 \* Create a new job instance.  
 \*  
 \* @return void  
 \*/  
 public function \_\_construct(){  
 $this->page = $this->getRatingPage();  
 $this->key = env('GOOGLE\_CWV\_KEY');  
 }  
  
 /\*\*  
 \* Execute the job.  
 \*  
 \* @return void  
 \*/  
 public function handle(){  
 if (!$this->page) {  
 return;  
 }  
  
 try {  
 $rating = $this->getRating();  
 $this->setRating($rating);  
 } catch (\Exception $e) {  
 $this->page->update(['error' => '1']);  
 }  
 }  
   
 protected function getRatingPage(): Page|bool  
 {  
 $pages = Page::doesntHave('ratings')  
 ->where('error', '=', '0')  
 ->get();  
  
 if (count($pages) === 0) {  
 $pages = Page::where('error', '=', '0')  
 ->andWhereHas('ratings', function (Builder $query) {  
 $query->where('created\_at', '<=', Carbon::today());  
 })  
 ->orderBy('updated\_at', 'ASC')  
 ->get();  
 }  
 if (count($pages) === 0) {  
 return false;  
 }  
 return $pages[0];  
 }  
   
 protected function getRating(): Rating{  
 $url = 'https://www.googleapis.com/pagespeedonline/v5/runPagespeed?url='.$url.'&category=PERFORMANCE&category=ACCESSIBILITY&category=SEO&strategy=MOBILE&key='.$key;  
 $response = Http::get($url);  
 $json = $response->json();  
 $ratings = $json['lighthouseResult']['categories'];  
   
 return new Rating([  
 'seo' => round($ratings['seo']['score'] \* 100),  
 'performance' => round($ratings['performance']['score'] \* 100),  
 'accessibility' => round($ratings['accessibility']['score'] \* 100),  
 ]);  
 }  
   
 protected function setRating(Rating $rating): Rating{  
 $this->page->loadCount('ratings');  
 $ratings\_count = $this->page->ratings\_count;  
 if ($ratings\_count > 10) {  
 //Delete older Ratings  
 $ratings = $this->page->load('ratings')->ratings;  
 for ($i = 9; $i < count($ratings); $i++) {  
 $ratings[$i]->delete();  
 }  
 }  
 $this->page->ratings()->save($rating);  
 $this->page->touch();  
 return $rating;  
 }  
}

Damit die Seiten-Auswahl richtig funktioniert, müssen im Page- und Rating-Model einige Anpassungen vorgenommen werden. Zunächst wollen wir, dass sich die *updated\_at* Spalte der Seiten aktualisiert, sobald ein neues Rating für diese erstellt wurde. Dies kann entweder durch das Aufrufen der touches-Methode auf der entsprechenden Page-Instanz, oder wie in diesem Fall durch das Anpassen der $touches Eigenschaft des Rating-Models:

class Rating extends Model{  
/\* … \*/  
 protected $touches = ['ratable'];  
/\* … \*/  
}

Zusätzlich sollen die Seiten standardmäßig nach der *updated\_at* Spalte absteigend sortiert werden. Dies kann in der statischen boot-Methode des Models definiert werden:

protected static function boot(){  
 parent::boot();  
 static::addGlobalScope('order', function (Builder $builder) {  
 $builder->orderBy('updated\_at', 'DESC');  
 });  
}

#### Regelmäßige Analyse von Seiten

Nachdem die Bewertung der Seiten funktioniert, soll der Job nun regelmäßig ausgeführt werden. Hierfür werden sogennante Cron-Jobs verwendet. Unser Ziel ist, jede Minute eine Seite zu bewerten. > **Cron-Jobs** sind wiederkehrende Aufgaben, die zu einer definierten Zeit oder > in einem festgelegten Rhythmus, automatisch vom Server ausgeführt werden. > Dabei ist keine weitere Nutzer-Interaktion notwendig.

Innerhalb der *Console.php* können bei Laravel Cron-Jobs festgelegt werden:

protected function schedule(Schedule $schedule){  
 $schedule->job(new RatePage)->everyMinute();  
}

Über den Artisan können die Cron-Jobs gestartet werden:

* In einer Entwicklungsumgebung: php artisan schedule:work
* In der Produktion: php artisan schedule:run

#### Durchschnitts-Bewertung von Seiten

Damit Laravel bei den Seiten unterscheiden kann, ob ein Rating die Durchschnitts-Bewertung oder eine Standard-Bewertung ist, müssen zunächst einige Anpassungen vorgenommen werden. Zunächst wir der Rating-Tabelle eine weitere Spalte *variant* hinzugefügt. In dieser sollen dann entweder *rating* für eine Standard-Bewertung oder *average* für eine Durchschnitts-Bewertung stehen. Zusätzlich wird das Page-Model angepasst, sodass die Methoden *ratings* und *averageRating* die passenden Werte zurückgeben, sowie der RatePage Job, welcher nun den entsprechenden Wert setzen muss: > Das Ändern einer Migration ist nur möglich, falls diese nocht nicht > migriert wurde. Sollten Änderungen danach vorgenommen werden, können diese > nur mit dem Befehl php artisan migrate:refresh übernommen werden. > Hierdurch gehen alle Einträge der gesamten Datenbank verloren, da diese > zurückgesetzt wird.  
> Alternativ kann eine neue Migrations-Datei erstellt werden, welche die > Tabelle aktualisiert. Allerdings musst hier die table-Methode und nicht > die create-Methode von Schema genutzt werden. *Weitere Infos >* [*hier*](https://laravel.com/docs/9.x/migrations#updating-tables).

// Rating Migration  
public function up(){  
 Schema::create('ratings', function (Blueprint $table) {  
 /\* … \*/  
 $table->string('variant')->nullable();  
 /\* … \*/  
 });  
}  
// Page Model  
public function averageRatings()  
{  
 return $this->morphOne(Rating::class, 'ratable')  
 ->where('variant', '=', 'average');  
}  
  
public function ratings(){  
 return $this->morphMany(Rating::class, 'ratable')  
 ->orderBy('created\_at', 'DESC')  
 ->where('variant', '=', 'rating');  
}  
// RatePage Job  
protected function getRating(): Rating{  
 /\* … \*/  
 return new Rating([  
 'seo' => $ratings['seo']['score'] \* 100,  
 'performance' => $ratings['performance']['score'] \* 100,  
 'accessibility' => $ratings['accessibility']['score'] \* 100,  
 'variant' => 'rating',  
 ]);  
}

Nachdem ein neues Rating erstellt wurde, soll sich die Durchschnitts-Bewertung einer Seite anpassen. Hierfür wird ein Observer, sowie ein Job genutzt.

1. **UpdateDefaultPageRatings Job**: Dieser Job berrechnet den Durchschnitt aus allen Bewertungen einer Seite. Im Anschluss erstellt oder aktualisiert er die entsprechende Durchschnitts-Bewertung einer Seite.

class UpdateDefaultPageRatings implements ShouldQueue  
{  
 use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;  
  
 public Page $page;  
  
  
 /\*\*  
 \* Create a new job instance.  
 \*  
 \* @return void  
 \*/  
 public function \_\_construct(Page $page){  
 $this->page = $page;  
 }  
  
 /\*\*  
 \* Execute the job.  
 \*  
 \* @return void  
 \*/  
 public function handle(){  
 $rating = $this->getAverage($this->page->ratings);  
 $this->page->averageRatings()->updateOrCreate($rating);  
 }  
  
 protected function getAverage($ratings): array{  
 $count = count($ratings);  
 $average\_ratings = [  
 'performance' => 0,  
 'seo' => 0,  
 'accessibility' => 0  
 ];  
   
 foreach ($ratings as $rating) {  
 $average\_ratings['performance'] += $rating->performance;  
 $average\_ratings['seo'] += $rating->seo;  
 $average\_ratings['accessibility'] += $rating->accessibility;  
 }  
  
 return [  
 'performance' => round($average\_ratings['performance'] / $count),  
 'seo' => round($average\_ratings['seo'] / $count),  
 'accessibility' => round($average\_ratings['accessibility'] / $count),  
 'variant' => 'average'  
 ];  
 }  
}

1. **PageObserver**: Der *UpdateDefaultPageRatings* Job soll immer ausgeführt werden, sobald eine neue Seiten-Bewertung erstellt wurde. Da innerhalb des Rating-Models bereits die $touches Variable angepasst wurde, wird bei einer Aktualisierung die updated-Methode des Observers aufgerufen. Nach der Erstellung mit php artisan make:Observer PageObserver --model=Page und der Registrierung innerhalb der EventServiceProvider boot-Methode:

* public function boot(){  
   Domain::observe(DomainObserver::class);  
   Page::observe(PageObserver::class);  
   }
* kann innerhalb der updated-Methode des Observers, der Job aufgerufen werden: php public function updated(Page $page){ UpdateDefaultPageRatings::dispatch($page); }

### Domainsbewertung und Sitemaps

#### Durchschnittsbewertung

Analog zur durchschnitts Seiten-Bewertung sollen Domains ebenfalls eine Durchschnitts-Bewertung besitzen. Hierfür wird der Job *UpdateDomainRatings* mit php artisan make:job UpdateDomainRatings erstellt und wie folgt angepasst:

class UpdateDomainRatings implements ShouldQueue  
{  
 use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;  
  
  
 public Domain $domain;  
 public Rating $domain\_rating;  
  
 /\*\*  
 \* Create a new job instance.  
 \*  
 \* @return void  
 \*/  
 public function \_\_construct(Domain $domain) {  
 $this->domain = $domain;  
 }  
  
 /\*\*  
 \* Execute the job.  
 \*  
 \* @return void  
 \*/  
 public function handle() {  
 $pages = $this->domain->pages()  
 ->where('error', '=', '0')  
 ->whereHas('averageRatings')  
 ->get();  
  
 if (count($pages) === 0) {  
 return;  
 }  
 $ratings = $this->getAverageRatings($pages);  
 $this->setDomainRatings($ratings);  
 }  
  
 protected function setDomainRatings($ratings): void {  
 $this->domain->rating()->updateOrCreate(  
 ['ratable\_id' => $this->domain->id], $ratings);  
 $this->domain\_rating = $this->domain->rating;  
 }  
  
 protected function getAverageRatings($pages){  
 $seo = 0;  
 $performance = 0;  
 $accessibility = 0;  
 $count = count($pages);  
  
 foreach ($pages as $page) {  
 $seo += $page->averageRatings->seo;  
 $performance += $page->averageRatings->performance;  
 $accessibility += $page->averageRatings->accessibility;  
 }  
   
 return [  
 'seo' => round($seo / $count),  
 'performance' => round($performance / $count),  
 'accessibility' => round($accessibility / $count),  
 ];  
 }  
}

Aus Kapazitätsgründen, soll der Job nicht nach jedem neuen Rating, sondern entweder über eine Nutzeranfrage oder alle 15 Minuten durch einen Cron-Job ausgeführt werden:

1. Nutzeranfrage:

* // api  
  Route::get('domain/{domain}/average', [DomainController::class, 'average']);  
    
  // DomainController  
  public function average(Domain $domain){  
   UpdateDomainRatings::dispatch($domain);  
  }

1. Cron-Job: Hierfür wird einer neuer Job erstellt php artisan make:job UpdateAllDomainRatings. Dieser dispatched lediglich den UpdateDomainRatings Job für jede Domain:

* // UpdateAllDomainRatings Job  
  public function handle(){  
   $domains = Domain::all();  
   foreach ($domains as $domain) {  
   dispatch(new UpdateDomainRatings($domain));  
   }  
  }  
  // Kernel  
  protected function schedule(Schedule $schedule){  
   $schedule->job(new RatePage)->everyMinute();  
   $schedule->job(new UpdateAllDomainRatings)->everyFifteenMinutes();  
  }

#### Aktualisierung und Überprüfung Sitemaps

Analog zur Seitenbewertung sollen die Sitemaps regelmäßig oder durch eine Nutzeingabe überprüft werden können. Hierfür wird ein neuer Job php artisan make:job CreateSitemap erstellt. Die Logik existiert bereits im DomainObserver und wird entsprechend angepasst:

1. CreateSitemap Job

* class CreateSitemap implements ShouldQueue  
  {  
   use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;  
   public Domain $domain;  
    
   /\*\*  
   \* Create a new job instance.  
   \*  
   \* @return void  
   \*/  
   public function \_\_construct(Domain $domain){  
   $this->domain = $domain;  
   }  
    
   /\*\*  
   \* Execute the job.  
   \*  
   \* @return void  
   \*/  
   public function handle(){  
   try {  
   $pages = $this->fetchSitemap();  
    
   //create Pages  
   foreach ($pages as $sitemapPage) {  
   Page::firstOrCreate([  
   'domain\_id' => $domain->id,  
   'url' => $sitemapPage,  
   ]);  
   $this->domain->update(['sitemapFound' => '1']);  
   }  
   } catch (\e) {  
   $this->domain->update(['sitemapFound' => '0']);  
   }  
   }  
    
   protected function fetchSitemap(){  
   $xml = simplexml\_load\_file($this->domain->url);  
   $pages = [];  
    
   // get page urls from sitemap  
   foreach ($xml as $element) {  
   $pages[] = $element->loc[0];  
   }  
   return $pages;  
   }  
  }

1. CheckSitemaps Job

* class CheckSitemaps implements ShouldQueue  
  {  
   use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;  
   /\*\*  
   \* Execute the job.  
   \*  
   \* @return void  
   \*/  
   public function handle(){  
   $domains = Domain::all();  
   foreach ($domains as $domain) {  
   CreateSitemap::dispatch($domain);  
   }  
   }  
  }

1. DomainObserver php class DomainObserver{ /\* … \*/ public function created(Domain $domain){ CreateSitemap::dispatch($domain); } /\* … \*/ }
2. Api-Route php /\* … \*/ Route::get('domain/{domain}/sitemap', [DomainController::class, 'sitemap']); /\* … \*/
3. DomainController ```php class DomainController extends Controller{ /\* … \*/ public function sitemap(Domain $domain){ foreach ($domain->pages as $page) { $page->delete(); }

* $domain->update(['sitemapFound' => 0]);  
   CreateSitemap::dispatch($domain);  
   $domain->loadCount('pages');  
   return response($domain);   
   }  
   /\* … \*/
* } ```

### Broadcasting mit Web-Sockets

Einige Jobs benötigen relative lange, bevor sie abgeschlossen sind. In dieser Zeit muss der Nutzer warten. Besser wäre es, den Nutzer darüber zu informieren, dass der Job dispatched wurde und ausgeführt werden wird. Nach der Ausführung kann der Nutzer wieder darüber informiert werden.

Hierfür stellt Laravel eine Queue- und ein Broadcasting-System zur Verfügung.