E-Learning – Automatisierte Analyze von Webseiten mithilfe der Google Page-Speed-Insights Api

Inhaltsverzeichnis

Das Projekt

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.

Vorraussetzungen

Für das Frontend sollte das Framework React bereits beherrscht werden. Erfahrungen mit NextJs oder Typescript sind nicht notwendig.

Im Backend sind lediglich Erfahrungen mit MySQL empfehlenswert. Die Programmiersprache PHP sollte vor allem in Bezug auf Objektorientierung bekannt sein. Die Architektur des MVC-Prinzips ist von Vorteil.

Tech-Stack

Der Tech-Stack dieses Projektes ist eine eher untypische Kombination. Im Backend wird Laravel (V9) ein PHP-Framework verwendet. Im Frontend wird NextJs in Kombination mit Typescript verwendet.

1. Einrichtung Backend

Installation Laravel

Nachdem ein passender Ordner mithilfe des Terminals ausgewählt wurde, kann Laravel mit folgendem Befehl installiert werden:

```
composer create-project laravel/laravel [server_name]
```

Grundaufbau / Ordner-Struktur

Erste Schritte

Diese vier Begriffe bzw. Methoden können als Kern von Laravel angesehen werden. Mithilfe dieser können so gut wie alle Vorstellungen umgesetzt werden.

Model

Das Model definiert die Grundstruktur eines Objekts. Ebenfalls werden hier Relations definiert, sowie Grund-Funktionen eines Objektes hinterlegt.

Controller

Der Controller verknüpft das Model mit den Routes. Innerhalb der einzelnen Controller-Funktionen können

Migration

Über Migration werden die einzelnen Tabellen in der Datenbank erstellt. Innerhalb dieser werden die Tabellen nach folgendem Prinzip definiert:

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    /**
     * diese Funktion wird zum Erstellen der Tabellen ausgeführt.
     */
    public function up()
    {
        Schema::create('testing', function (Blueprint $table) {
            $table->string('name');
            $table->integer('number')->nullable();
            $table->boolean('error')->default('false');
        });
    }

    /**
     * Diese Funktion macht das Mirgrieren von Tabellen rückgängig
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('domains');
    }
};
```

Innerhalb der `Schema::create` function wird durch `Blueprint $table` die Tabelle definiert. Dabei stehen string, integer, boolean noch viele weitere Attribute zur Verfügung:

- `id()` : Legt einen Unique-Identifier an
- `foreingId()` : Definiert eine 'Relation'
- `timestamps()` : Legt automatisch die Spalten `updated_at` und `created_at` an
- weitere Methoden

Routing

Im Routing wird typischerweise eine Request-URL mit einer Controller-Methode verbunden. Diese Verknüpfung wird innerhalb der `routes/api.php` für alle Api-Endpunkte festgelegt.

Eine Verknüpfung ist nach folgendem Standard aufgebaut:

```
Route::get('test-url',[ControllerName::class,'controller-method']);
```

Neben einem Get-Request können hier alle Request-Methoden angelegt werden: `PUT`, `POST`, `DESTROY`. Zusätzlich können mit einem Befehl direkt alle Routes für die CRUD-Operationen angelegt werden:

```
Route::apiRessource('crud-url-prefix',ControllerName::class)
```

Die dadurch erstellten Routes können mit `php artisan route:list` überprüft 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@updat
DELETE   api/domain/{domain} .....  domain.destroy › DomainController@dest
```

Die Parameter innerhalb der geschweiften Klammern müssen dabei als ID des jeweiligen Models übergeben werden. Der Vorteil: Über das sogenannte **Route-Model-Binding** werden nicht vorhandenen Records direkt abgefangen und durch einen 404-Error direkt an den Nutzer zurückgegeben.

Json-Response

Laravel hat als Framework verschiedenste Einsatzzwecke. Für dieses Projekt wird Laravel ausschließlich als API genutzt. Damit Laravel auf alle Requests mit einem Json-Response geantwortet wird zunächst ein neuer Request erstellt.

1. Request erstellen

```
php artisan make:Request JsonRequest
```

2. Anpassen der Request-Datei in `app/Http/Requests/JsonRequest.php`

```
public function expectsJson(): bool
    {
        return true;
    }
```

```
    public function wantsJson(): bool
    {
        return true;
    }
```

3. Registrieren des neuen Requests als Standard in `public/index.php`

```
$response = $kernel->handle(
$request = \App\Http\Requests\JsonRequest::capture()
)->send();
```

CRUD-Optionen am Beispiel der Domain

Folgend werden am Beispiel des Domain-Models CRUD-Operationen in Laravel gezeigt. Dabei wird zunächst die Grund-Struktur erstellt und im Anschluss erweitert.

Erstellen der Migration, Controller, Routes und Model

Zunächst wird das Model und die damit verbunden Migration erstellt. Dies geschieht über den folgenden Befehl:

`php artisan make:model Domain -m` Über die Flag `-m` wird neben dem Model direkt eine Migrations-Datei erstellt.

Eine Domain hat folgende Eigenschaften:

- `id` integer als UID
- `name` string, einzigartig
- `favicon` string als Icon der Website (optional)
- `url` string, Standard-Link der Webseite, einzigartig
- `sitemap` string, Link zur Sitemap der Webseite, einzigartig
- `sitemapFound` boolean, default:true
- `timestamps` Laravel-Funktion, erstellt die Spalten updated_at und created_at und aktualisiert diese automatisch

Für diese Eigenschaften schaut die dazugehörige Migrations-Datei wie foglt aus:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
```

```
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('domains', function (Blueprint $table) {
            $table->id();

            $table->string('name')->unique();
            $table->string('favicon')->default(null);
            $table->string('sitemap')->unique();
            $table->string('url')->unique();

            $table->boolean('sitemapFound')->default(true);


            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('domains');
    }
};
```

Nachdem die Migrations-Datei angepasst wurde, kann die Tabelle durch den Befehl `php artisan migrate` erstellt werden. Aus der Migrations-Datei lässt sich dann das Model ableiten.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

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

```
}
```

Dabei wird über den Artisan bereits eine Grund-Struktur erstellt. Die geschütze variable `$fillable` beschreibt dabei alle Eigenschaften des Models, welche über Nutzer-Requests bearbeitet beziehungsweise beeinflusst werden können.

Nachdem das Model definiert und die Tabelle erstellt wurde, können die Routes sowie der Controller ergänzt werden. Der Controller wird mit folgendem Befehl erstellt `php artisan make:controller DomainController --api` . Die Flag `--api` das folgende Methoden bereits in den Controller implementiert werden:

```
<?php

namespace App\Http\Controllers;
use App\Models\Domain;

class DomainController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
    }
    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
    }

    /**
     * Update the specified resource in storage.
```

```
     *
     * @param  Request  $request
     * @param  int $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request,  $id)
    {
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
    }
}
```

Diese stellen die typischen CRUD-Operationen eines Api-Endpunktes dar. Die Methode `index()` soll alle Records, `find()` ein Record eines Models zurückgeben. `store()` dient zum Erstellen eines neuen Records, sowie `update()` zum Aktualisieren. Durch die `destroy()` Methode können Records gelöscht werden.

Innerhalb der `api.php` kann der Controller durch den `Route::apiRessource` -Befehl mit den Endpunkten verknüpft werden.

Im Folgenden werden Techniken beschrieben, die das Daten-Handling vereinfachen können.

Route-Model-Binding

Durch das Route-Model-Binding (**RMB**) überprüft Laravel selbständig, ob ein Record mit der angegebenen id überhaupt existiert. Existiert dieser nicht, wird automatisch ein 404 Response zurückgegeben.

Das RMB bietet sich für alle Methoden an, die eine id als Parameter erwarten. Dazu zählen die `update()` , `destroy()` , `find()` Methoden. Das RMB wird durch die Parameter einer Methode genutzt. Dabei wird anstatt des Parameters `int $id` direkt ein Model, bspws `Domain $domain` , erwartet. Die Methoden können deshalb wie folgt angepasst werden:

```
/**
     * Display the specified resource.
     *
```

```
     * @param  Domain  $domain
     * @return \Illuminate\Http\Response
     */
    public function show(Domain $domain)
    {
        return response($domain);
    }
```

Da das RMB bereits überprüft, ob eine Ressource mit der angegeben id existiert, kann die `$domain` direkt durch einen Response zurückgegeben werden.

Custom Requests

Es ist essenziell, Eingaben eines Nutzers zu validieren. Laravel besitzt hierfür bereits eine integrierte Validierungs-Funktion. Diese Validierung kann durch Custom-Requests aus dem Controller ausgegliedert werden. Analog zum bereits erstellten JsonRequest können wir hier beispielsweise einen DomainRequest über `php artisan make:Request DomainRequest` erstellen. Dabei ensteht eine Datei mit folgender Struktur:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class DomainRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string, mixed>
     */
    public function rules()
    {
        return [
            'name' => ['required', 'string', 'unique:domains'],
            'favicon' => ['string'],
```

```
            'sitemap' => ['required', 'string'],
            'url' => ['required', 'string', 'unique:domains'],
        ];
    }
}
```

Die Methode `authorize()` kann dafür genutzt werden, zu Bestimmen, ob der Nutzer die nötigen Berechtigungen für den Request hat. Authorization wird in diesem Projekt vernachlässigt, weshalb diese Methoden immer den Wert `true` zurückgibt.

Über die Methode `rules()` können die Validierungs-Regeln definiert werden. Diese sind in der Laravel-Dokumentation hier nachzulesen.

Sobald eine der Validierungs-Regeln nicht erefolgreich war, wird automaitsch ein 422-Response an den Nutzer zurückgeschickt. Zum Nutzen des neu erstellten Requests kann dieser als entsprechender Parameter in einer Methode zugewiesen werden. Wichtig ist, das dies nur innerhalb eines POST-Requests funktioniert.

Am Beispiel der Store und Update Methode schauen diese nun wie folgt aus:

```
/**
     * Store a newly created resource in storage.
     *
     * @param  DomainRequest  $request
     * @return \Illuminate\Http\Response
     */
    public function store(DomainRequest $request)
    {
       $domain=Domain::create($request->all());
       return response($domain,201);
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  UpdateDomainRequest  $request
     * @param  Domain  $domain
     * @return \Illuminate\Http\Response
     */
    public function update(UpdateDomainRequest $request, Domain $domain)
    {
        $domain->update($request->all());
        return response($domain, 200);
    }
```

Die Daten innerhalb der `store` -Methode sind bereits validiert, weshalb diese direkt dazu genutzt werden können, eine neue Ressource zu erstellen. Nachdem diese erstellt wurde, wird diese mit dem Status-Code 201 an den Nutzer zurückgegeben.

Der Angepasste Controller schaut nun wie folgt aus:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests\DomainRequest;
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  DomainRequest  $request
     * @return \Illuminate\Http\Response
     */
    public function store(DomainRequest $request)
    {
        $domain = Domain::create($request->all());
        return response($domain, 201);
    }

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        return response($domain);
    }

    /**
```

```
     * Update the specified resource in storage.
     *
     * @param  DomainRequest  $request
     * @param  Domain  $domain
     * @return \Illuminate\Http\Response
     */
    public function update(DomainRequest $request, Domain $domain)
    {
        $domain->update($request->all());
        return response($domain, 200);
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  Domain  $domain
     * @return \Illuminate\Http\Response
     */
    public function destroy(Domain $domain)
    {
        $domain->delete();
        return response(['message' => 'Record deleted'], 200);
    }
}
```

Relations in Laravel

Relations bezeichnen Abhängigkeiten von einzelnen Models untereinander. Dabei können diese in folgenden Varianten auftretten:

- One to One
- One to Many
- Many to Many

So stehen beispielsweise die Seiten einer Domain beziehungsweise Webseite in einem One to Many Verhältnis. Eine Seite hat gehört immer zu einer Domain, hingegen kann eine Domain mehrere Seiten haben.

One To Many Relation am Beispiel Page

Abhängigkeit von Models werden im Model selbst, sowie der Migrations-Datei definiert. Eine One To Many Abhängigkeit wird dabei in Migration des 'Many'-Models gekennzeichnet. Dies geschieht über die `foreignId([modelname]_id)` Methode. Innerhalb dieser wird das ' One'-Model zugewiesen. Neben der Abhängigkeit besitzt

das Page model noch eine `url`, sowie den boolean `error`, welche standardmäßig `0` * ist.

> *Laravel verwendet für die Darstellung von Booleans nicht `true` oder `false` sondern `0` und `1`

Das Page- und Domain-Model kann neben der bekannten `$fillable` nun mit Methoden erweitert werden. Diese Methoden helfen dabei, die Abhängigkeiten von Ressourcen abzufragen:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class Page extends Model{
    protected $fillable = ['url', 'domain_id', 'error'];
    protected $touches=['domain'];

    //default order by last update
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('order', function (Builder $builder) {
            $builder->orderBy('updated_at', 'DESC');
        });
    }


    public function domain(){
        return $this->belongsTo(Domain::class);
    }
}


class Domain extends Model{

    protected $fillable = ['name', 'favicon', 'sitemap', 'sitemapFound', '
    protected $with=['pages'];

    public function pages(){
        return $this->hasMany(Pages::class);
    }
}
```

Neben der `$fillable` Variable bieten Models weitere nützliche Variablen, die im Laufe des Projektes genutzt werden:

- `protected $with` : Standardmäßig werden Ressourcen ohne Abhängigkeiten zurückgegeben. Alle Abhängigkeiten, die in der `$with` angegeben sind, werden nun mit zurückgegeben.
- `protected $touches` : Alle abhängigen Ressourcen innerhalb der `$touches` Variable aktualsieren dabei die Spalte `updated_at` . Standardmäßig wird bei einer Veränderung keine Ressource 'aktualisiert'.