إنشاء النماذج Forms

بالاستفادة من laravelcollective

السلام عليكم و رحمة الله و بركاته

في هذه المحاضرة ستتعرف على الطريقة المثلى لكتابة وسوم html الخاصة بإنشاء النماذج **Forms** ، و أيضاً ستتعلم الطرق المختلفة للتحقق من المدخلات Validation في لارافل .

ضمن الصفحة index و ضعت كود html لإنشاء النموذج Form الخاص بإدخال البيانات ، يمكنك إنشاء هذا النموذج بكتابة كود html بطريقة أخرى أكثر اختصاراً و تنظيماً .

فمثلاً لفتح Form سيُستبدل هذا السطر

```
<form action="{{route('posts.store')}}" method="POST">
```

بالتالي

```
{!! Form::open(['route' => 'posts.store','method' => 'POST']) !!}
```

أما وسم إغلاق الفورم فيمكن كتابته بهذا الشكل

```
{!! Form::close() !!}
```

حالياً النتيجة ستكون ظهور هذا الخطأ

```
ErrorException (E_ERROR)
Class 'Form' not found
```

السبب

لاستخدام هذه الميزة تحتاج أولاً إلى تثبيت حزمة Laravelcollective/html ، و هي غير مثبته حالياً .. لهذا يظهر لنا هذا الـerror ، سنستخدم composer لتثبيتها عن طريق كتابة الأمر التالي على موجه الأوامر :

```
composer require "laravelcollective/html":"^5.5"
```

الآن يمكنك استخدام المزايا التي تقدمها هذه الحزمة

لإنشاء عناصر html بالاستفادة من laravelcollective ستبدأ دائماً بكتابة Form يتبعه نوع العنصر و بين القوسين ستحدد خصائص هذا العنصر كالاسم و القيمة و كذلك تنسيقات CSS .

لنقم بإضافة عنصر label بالمعرف title_itm ليعرض النص "title:"

أول معامل سيكون معرف العنصر ، و ثاني معامل هو النص الذي سيتم عرضه .

```
{{ Form::label('title_itm', 'title :') }}
```

بالتأكيد تعرف الآن كيف ستضيف أي عناصر من أي نوع آخر ، فلإضافة حقل إدخال textbox سنكتب التالي

```
{{ Form::text('title_itm',null) }}
```

أول معامل هو معرف العنصر ، و الثاني يحدد القيمة الافتراضية للعنصر ، و في حالة لن تحدد أي قيم ستضع null .

و إذا أردت تحديد خصائص HTML إضافية يمكنك تحديدها كمعامل ثالث بشكل مصفوفة

فمثلاً لإضافة الخاصية placeholder سأضيف هذا الجزء

```
{{ Form::text('title_itm',null,['placeholder'=>'Enter Title']) }}
```

و كذلك تستخدم الخاصية class بنفس الطريقة .. لنرى النتائج الآن

```
{{Form::text('id',null,['placeholder'=>'Enter Title','class'=>'form-control'])}}
```

سنكتب التالي لإضافة الحقل الخاص باختيار التاريخ

```
{{  Form::date('name', \Carbon\Carbon::now())  }}
```

المعامل الأول هو معرف الحقل و الثاني هو التاريخ الذي سيظهر افتراضياً على الحقل ، سنستخدم الحزمة Carbon (الخاصة بالتعامل مع التاريخ و الوقت) لإظهار التاريخ الحالي .

نفس الاسلوب السابق سيطبق مع العناصر الأخرى كزر الإرسال و مربعات التأشير و القائمة المنسدلة و غيرها

```
Form::label('email', 'E-Mail Address', ['class' => 'awesome'])

Form::submit('Click Me!')

Form::password('password', ['class' => 'awesome'])
```

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

Form::radio('name', 'value')

Form::file('image')

Form::select('size', ['L' => 'Large', 'S' => 'Small'])
```

كما ذكرنا في المحاضرة السابقة فإننا نحتاج أحياناً لإرسال حقل مخفي ضمن الفورم لمعالجة الطلبات من نوع PUT و DELETE ، يمكنك الاستغناء عن هذا السطر

```
<input name="_method" type="hidden" value="PUT">
```

بكتابة هذا فقط

```
{{ method_field('PUT') }}
```

للاستزادة

https://laravelcollective.com/docs/5.4/html

التحقق من المدخلات Validation

عند ملء نماذج الويب Forms قد يخطئ المستخدم في إدخال القيم إلى الحقول ، فمثلاً قد يقوم بإدخال الايميل بصيغة خاطئة ، أو إدخال رقم الهاتف بعدد خانات أقل أو أكثر من المطلوب ، لذلك لا بد من تنبيه المستخدم بالأخطاء التي ارتكبها قبل إرسال هذه المدخلات إلى قاعدة البيانات ، لنضمن بذلك الحصول على بيانات في صورة صحيحة و كذلك تحسين تجربة المستخدم .

في لارافل توجد عدة أساليب للتحقق من صحة البيانات ، و سنرى الآن بشكل عملي كيف يتم التحقق من البيانات قبل حفظها في قاعدة البيانات .

الفورم الظاهر أمامك خاص بإضافة مواضيع ، و يتألف الفورم من حقل لعنوان الموضوع و آخر لمحتواه .

إذا أدخلت الآن أي قيم بأي حجم فسيتم قبولها مباشرةً ، أما في حال تركت أحد الحقول فارغة فسيعود بصفحة الخطأ الظاهرة أمامك ، و كما تلاحظ فالسبب هو كون الحقل title في قاعدة البيانات لا يقبل القيمة null ، لنتفادى هذا علينا اختبار صحة المدخلات قبل إرسالها كما سنرى الآن .

في داخل الدالة الخاصة بحفظ البيانات store() سنعمل على إضافة قواعد التحقق قبل عملية الحفظ

تُستخدم الدالة **validate()** المزودة من الكائن Request بالشكل التالي

```
public function store(Request $request)
{
   $validatedData = $request->validate([
       'title' => 'required',
       'body' => 'required',
   ]);

  Post::create($request->all());
  return back()->with('success', 'The post has been successfully created');
}
```

كما رأيت يتم تحديد القواعد الخاصة بكل حقل بداخل الدالة **validate()**

قمنا بتقييد الحقل title كحقل مطلوب required و كذلك بالنسبة للحقل body

في حالة تطابق القيم المدخلة مع هذه القواعد سيتم تمرير الطلب بنجاح و إرجاع البيانات إلى **$validatedData** بشكل مصفوفة .

من الممكن أن نمرر **$validatedData** مباشرةً إلى الدالة create() ليتم إضافتها كسجل إلى قاعدة البيانات .

```
        Post::create($validatedData);
```

إذا لم تتوافق القيم المدخلة مع هذه القواعد فسوف يُعاد توجيه المستخدم لنفس الصفحة مرة أخرى ..

لنقم بإضافة شرط آخر لحقل title بحيث لا يسمح بإدخال عنوان أطول من ٥٠ حرف .

```
        $validatedData = $request->validate([
            'title' => 'required|max:50',
            'body' => 'required',
        ]);
```

ستجد الآن أنه يرفض إدخال أي سلسلة نصية بطول أكبر من ٥٠ حرف .

قم بإضافة أي شروط أخرى بنفس هذا النمط .

عرض أخطاء التحقق من الصحة :

كما لاحظت .. لا رسائل خطأ تظهر .. فقط يكتفي بإعادة التوجيه لنفس الصفحة السابقة ، ماذا إذا أردت عرض رسائل للمستخدم توضح الأخطاء بالتحديد .

في حالة لم ينجح التحقق فإن لارافل يقوم بتخزين رسائل خطأ ضمن session بالطريقة Flash()
تخزين البيانات في Session باستخدام هذه الطريقة يعني أنها ستكون مؤقتة حتى الطلب Request التالي للصفحة .

ما سنفعله الآن هو عرض هذه الرسائل على الصفحة ، نتحقق أولاً إذا ما كان هناك أخطاء مسجلة ضمن المتغير $errors ، ثم نقوم بعرض جميعا باستخدام حلقة foreach

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

انشاء Form Request Validation

لتكون الشيفرة البرمجية في ملف الـController نظيفة و واضحة يمكن وضع قواعد التحقق في ملف منفصل ، تسمى هذه الملفات Form requests . و يتم إنشاءها بداخل المجلد Requests الموجود ضمن الدليل App/HTTP .

راقب الآن كيف سأقوم باستخدام Form request لقواعد التحقق التي أضفناها بدلاً من كتابتها ضمن الدالة ()store

سنبدأ بإنشاء الملف و سنستخدم الـArtisan و ننفذ الأمر التالي

```
php artisan make:request PostRequest
```

ستلاحظ ظهور ملف جديد بداخل الدليل Requests بنفس الاسم الذي اخترناه PostRequest

تكتب قواعد التحقق ضمن الدالة rules

```
    public function rules()
    {
        return [
            'title' => 'required|max:50',
            'body' => 'required',
        ];
    }
```

الدالة ()authorize تُستخدم لوضع شروط للتحقق من كون المستخدم يمتلك أساساً صلاحية تحديث هذه البيانات أم لا ، هذه الدالة تعيد false في حالة أن المستخدم لا يمتلك الصلاحية .

إذا لم تكن هناك أية شروط أو أنك قمت بتحديدها في جزء آخر ضمن التطبيق فاجعلها تعيد true مباشرةً .

لنعد الآن للمتحكم PostController .. سنحدث تغيير بسيط على الدالة ()store .

سنقوم باستبدال Request بإسم الكلاس الذي يشمل القواعد للمدخلات و هو PostRequest

```
public function store(PostRequest $request)
{
   Post::create($request->all());
   return back()->with('success', __('The post has been successfully created'));
}
```

ماذا يعني هذا ؟

هذا سيعمل على تمرير الطلب إلى PostRequest أولاً ثم بعدها ينتقل إلى تنفيذ الدالة store() .

و لنتمكَّن من استخدام الكلاس PostRequest ضمن المتحكم ، لا بد من تعريفه في أعلى الصفحة بالشكل :

```
use App\Http\Requests\PostRequest;
```

الآن جميع الأمور جاهزة ، فلنقم بتجربة إضافة سجل جديد بدون إدخال عنوان

كما هو واضح فهو ينفذ القواعد التي وضعناها بشكل جيد ، و يعرض تنبيهات توضح نوعية الأخطاء .. رسائل الخطأ التي تُعرض مسجلة ضمن الملف validation.php الموجود ضمن الدليل resources/lang/en/validation.php

و لإنشاء رسائل مخصصة Custom Error Messages يمكنك إنشائها في هذا الملف بإضافتها ضمن المصفوفة custom بالشكل التالي :

```
    'custom' => [
        'title' => [
            'required' => 'Title can\'t be empty',
        ],
    ],
```

هناك خيار آخر لإنشاء رسائل خطأ مخصصة

و تكون بتضمينها في ملف الـ**Form request** ، و إعادة كتابة (overriding) الدالة messages()

بالشكل التالي :

```
    public function messages()
    {
        return [
            'title.required' => 'Title can\'t be empty',
            'title.max' => 'Maximum characters are 50',
            'body.required'  => 'Content can\'t be empty',
        ];
    }
```

ستجد الآن أنه يعتمد الرسائل التي حددتها بداخل الدالة messages() .

هناك العديد من قواعد التحقق الأخرى المتاحة ، فيمكنك مثلاً استخدام email للتحقق من كون الصيغة المدخلة تطابق صيغة عنوان البريد الإلكتروني ، و integer للتحقق من كون المدخلات عبارة عن أرقام فقط ، و العديد من قواعد التحقق الأخرى الموضحة هنا :

https://laravel.com/docs/5.5/validation#available-validation-rules

ماذا إذا أردت استخدام قاعدة تحقق غير موجودة ضمن هذه اللائحة ، مثلاً التحقق من أن الرقم المدخل يبدأ بمفتاح بلدك ، يمكنك في هذه الحالة إنشاء قاعدة تحقق مخصصة كما سنرى تالياً .

قواعد التحقق المخصصة
Custom Validation Rules

ستتعلم الآن كيف تنشئ قاعدة تحقق مخصصة ، نلجأ لإنشاء هذه القواعد عندما نحتاج إلى التحقق من أن المدخلات تطابق صيغة معينة غير متوفرة ضمن القواعد Rules الجاهزة التي تقدمها لارافل .

سنعرض مثال بسيط حول إنشاء Custom Validation للتحقق من كون أول حرف مدخل هو حرف كبير upper case

أول خطوة هي إنشاء Rule جديد لتحدد بداخله القواعد الخاصة بك ، و يتم إنشاؤه عن طريق الـartisan بكتابة :

```
php artisan make:rule UpperCaseFirst
```

سيظهر الملف بداخل مجلد Rules ضمن الدليل app

في الملف الناتج UpperCaseFirst

تحدد القواعد بداخل الدالة passes() ، هذه الدالة تستقبل اسم العنصر و القيمة المطلوب التحقق منها و تعيد true في حالة صحة البيانات و تماشيها مع الشروط ، عدا ذلك ستعيد القيمة false .

أما الدالة message() ستعيد رسالة خطأ عند فشل التحقق .

نبدأ بتحديد القواعد بداخل الدالة passes ، سنتحقق من القيمة التي أدخلها المستخدم و الممثلة في المتغير $value ،

```
public function passes($attribute, $value)
{
    $str=trim($value);
    $first_letter= substr($str,0,1);
    return ctype_upper($first_letter);
}
```

و الآن الدالة جاهزة ، ففي حالة كان أول حرف هو حرف كبير ستعيد القيمة true مالم ستعيد false .

سنحدد الرسالة التي نرغب في ظهورها عند عدم التطابق بداخل الدالة message().

```
public function message()
{
    return 'first letter must be upper case';
}
```

الآن قمنا بإنشاء rule جديد بالاسم `UpperCaseFirst` و هي عبارة عن كائن object

ما نريده الآن هو تطبيق هذه القاعدة على الحقل title ، و الذي أسندنا له عدة قواعد مسبقاً ، سنمرر القاعدة الجديدة `UpperCaseFirst` بالشكل التالي :

```
        return [
            'title' => ['required','max:10',new UpperCaseFirst],
            'body' => 'required',
        ];
```

لا تنس وضع السطر التالي أعلى الملف **PostRequest** ليستطيع الوصول للكلاس **`UpperCaseFirst`**

```
use App\Rules\UppercaseFirst;
```

جرب الآن إدخال عنوان يبدأ بحرف صغير Lower case و لاحظ أنه يستجيب للقواعد التي وضعتها و يعرض رسالة الخطأ التي قمت بتحديدها ضمن الدالة ()message.