Extending Core Functionality

Extending Core Functionality

What are models and how do I use them?

All models provided as part of the Aero core platform are Eloquent models

For example, if a module provided `reviews` for products, the `reviews` method can be added to the Product model using a `macro`:

```
\Aero\Catalog\Models\Product::macro('reviews', function () {
    return $this->hasMany(\Acme\MyModule\Models\Review::class);
});
```

The relationship query builder can be accessed by referencing the method:

```
$approvedReviewCount = $product->reviews()->where('approved', true)->count();
```

Just like a typical Eloquent relationship on a model, the resulting `Collection` of reviews can be accessed through the magic property:

```
$reviews = $product->reviews;
```

```
{% for review in product.reviews %}
    ...
{% endfor %}
```

How do I add a custom field to the new and edit product page?

In this mini tutorial we will add a notes field to the product new and edit page for the product and each variant.

This mini tutorial assumes that you have a module setup or you’re happy working from the app service provider (using the boot method). You can see how to set up a module here (link).

Adding the Database Migration

To get started we will add a migration file that will update the products and variants tables to have a notes column. To do this we’ll need to create the migration file and load them from within the modules service provider.

```
<?php

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

class AddNotesFieldsToProductsAndVariants extends Migration
{
   /**
    * Run the migrations.
    *
    * @return void
    */
   public function up()
   {
       Schema::table('products', function (Blueprint $table) {
```

```
            $table->string('notes')->nullable()->after('description');
        });

        Schema::table('variants', function (Blueprint $table) {
            $table->string('notes')->nullable()->after('name');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('products', function (Blueprint $table) {
            $table->dropColumn('notes');
        });

        Schema::table('variants', function (Blueprint $table) {
            $table->dropColumn('notes');
        });
    }
}
```

Adding the Slot View

Now we'll create two views, one that will be injected for the product and one that will be injected for the variants. These views will have a text area in them for the notes input.

product-notes-field.blade.php

```
<div class="card mt-4">
    <label for="notes" class="block mb-2">Notes</label>
    <textarea name="notes" id="notes" v-model="product.notes" class="w-full"></textarea>
</div>
```

variant-notes-field.blade.php

```
<div class="p-4" v-if="!isSimpleProduct">
   <label :for="'variant-notes-' + key" class="block mb-2">Notes</label>
   <textarea :name="'variants[' + key + '][notes]'" :id="'variant-notes-' + key" v-
model="variants[key].notes" class="w-full"></textarea>
</div>
```

After creating the views we will load them in the module service provider using the `$this->loadViewsFrom` method and inject them into the relevant slots using the `Aero\Admin\AdminSlot::inject` method.

```
<?php

namespace Acme\MyModule;

use Aero\Admin\AdminSlot;
use Aero\Common\Providers\ModuleServiceProvider;

class ServiceProvider extends ModuleServiceProvider
{
   public function setup()
   {
       if ($this->app->runningInConsole()) {
           $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
       }

       $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');

       AdminSlot::inject('catalog.product.new.cards', 'my-module::product-notes-field');
       AdminSlot::inject('catalog.product.edit.cards', 'my-module::product-notes-field');
       AdminSlot::inject('catalog.product.new.variant', 'my-module::variant-notes-field');
       AdminSlot::inject('catalog.product.edit.variant', 'my-module::variant-notes-field');
   }
}
```

Making the Notes Save

Adding Notes as a Fillable

We need to make the notes fillable as Laravel will only mass assign fillable attributes (`Aero\Catalog\Models\Product and Aero\Catalog\Models\Variant`).

```
<?php

namespace Acme\MyModule;

use Aero\Admin\AdminSlot;
use Aero\Catalog\Models\Product;
use Aero\Catalog\Models\Variant;
use Aero\Common\Providers\ModuleServiceProvider;

class ServiceProvider extends ModuleServiceProvider
{
   public function setup()
   {
       if ($this->app->runningInConsole()) {
           $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
       }

       $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');

       AdminSlot::inject('catalog.product.new.cards', 'my-module::product-notes-field');
       AdminSlot::inject('catalog.product.edit.cards', 'my-module::product-notes-field');
       AdminSlot::inject('catalog.product.new.variant', 'my-module::variant-notes-field');
       AdminSlot::inject('catalog.product.edit.variant', 'my-module::variant-notes-field');

       Product::makeFillable('notes');
       Variant::makeFillable('notes');
   }
}
```

Adding Notes to the Validators

To let the notes input data get through validation it needs to be added to the `Aero\Admin\Http\Requests\Catalog\CreateProductRequest` and

`Aero\Admin\Http\Requests\Catalog\UpdateProductRequest` validators. To do this we need to use the `expects` method on the two validators and pass in the notes field and its rules.

```
<?php

namespace Acme\MyModule;

use Aero\Admin\AdminSlot;
use Aero\Admin\Http\Requests\Catalog\CreateProductRequest;
use Aero\Admin\Http\Requests\Catalog\UpdateProductRequest;
use Aero\Catalog\Models\Product;
use Aero\Catalog\Models\Variant;
use Aero\Common\Providers\ModuleServiceProvider;

class ServiceProvider extends ModuleServiceProvider
{
   public function setup()
   {
       if ($this->app->runningInConsole()) {
           $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
       }

       $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');

       AdminSlot::inject('catalog.product.new.cards', 'my-module::product-notes-field');
       AdminSlot::inject('catalog.product.edit.cards', 'my-module::product-notes-field');
       AdminSlot::inject('catalog.product.new.variant', 'my-module::variant-notes-field');
       AdminSlot::inject('catalog.product.edit.variant', 'my-module::variant-notes-field');

       Product::makeFillable('notes');
       Variant::makeFillable('notes');

       CreateProductRequest::expects('notes', 'nullable|string');
       UpdateProductRequest::expects('notes', 'nullable|string');
   }
}
```

Adding Notes to the Transformers

Adding notes to the transformers will ensure that it's available in our views through Vue. We need to use the add method on the `Aero\Admin\Transformers\BaseVariantTransformer`, `Aero\Admin\Transformers\ProductTransformer`, and `Aero\Admin\Transformers\VariantTransformer` transformers.

```
<?php

namespace Acme\MyModule;

use Aero\Admin\AdminSlot;
use Aero\Admin\Http\Requests\Catalog\CreateProductRequest;
use Aero\Admin\Http\Requests\Catalog\UpdateProductRequest;
use Aero\Admin\Transformers\BaseVariantTransformer;
use Aero\Admin\Transformers\ProductTransformer;
use Aero\Admin\Transformers\VariantTransformer;
use Aero\Catalog\Models\Product;
use Aero\Catalog\Models\Variant;
use Aero\Common\Providers\ModuleServiceProvider;

class ServiceProvider extends ModuleServiceProvider
{
   public function setup()
   {
       if ($this->app->runningInConsole()) {
           $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
       }

       $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');

       AdminSlot::inject('catalog.product.new.cards', 'my-module::product-notes-field');
       AdminSlot::inject('catalog.product.edit.cards', 'my-module::product-notes-field');
       AdminSlot::inject('catalog.product.new.variant', 'my-module::variant-notes-field');
       AdminSlot::inject('catalog.product.edit.variant', 'my-module::variant-notes-field');

       Product::makeFillable('notes');
       Variant::makeFillable('notes');

       CreateProductRequest::expects('notes', 'nullable|string');
       UpdateProductRequest::expects('notes', 'nullable|string');
```

```
        ProductTransformer::add(function ($data) {
            return [
                'notes' => $data['product']->notes ?? '',
            ];
        });

        VariantTransformer::add(function ($data) {
            return [
                'notes' => $data['variant']->notes ?? '',
            ];
        });

        BaseVariantTransformer::add(function ($data) {
            return [
                'notes' => '',
            ];
        });
    }
}
```

What are the available commands?

php artisan make:module

This command scaffolds a module for you (you need to provide a valid name for the module such as **aerocargo/my-module**). This involves setting up composer and publishing some default module stubs.

php artisan aero:configure

This command walks you through configuring your store. This includes things such as setting the store's name, database connection, and Elasticsearch connection.

php artisan aero:install

This command runs you through the installation process for your store. This includes completing migrations, configuring Elasticsearch, linking storage, seeding data, and clearing the applications cache.

php artisan aero:link

This command calls the patch command, links theme/module assets, clears the Twig views cache, and clears the Laravel views cache.

php artisan module:install

This command will install a specific module after you have composer required the module. This command seeds any data for the module, optionally clears the application cache, and links the module assets. You should note that this command doesn’t have to be used and is automatically run when composer requiring a module.

php artisan aero:patch

This command patches your store's database. This command is run automatically when updating and ensures that your database is kept up to date with Aeros latest updates.

php artisan aero:search:install

This command sets up the necessary environment for your search driver. By default when using Elasticsearch this command will create the required indexes.

php artisan aero:search:rebuild

This command resets up and reindexes your search by running the search install and reindex commands.

php artisan aero:search:reindex

This command will reindex your database data into your searches index. By default when using Elasticsearch this command will clear the index and then index your database data.

php artisan aero:seed:emails

This command will seed some default email templates to your mail notifications.

php artisan aero:sitemap:cms

This command will generate the pages-sitemap.xml sitemap file in your public storage directory. This sitemap covers any pages you create through the admin.

php artisan aero:sitemap:combinations

This command will generate the combinations-sitemap.xml file in your public storage directory. This sitemap covers any combinations you create through the admin.

php artisan aero:sitemap:generate

This command will generate a complete sitemap for your store. This generates a sitemap.xml file in your public storage directory that references individual sitemap files for products, combinations, and pages.

php artisan aero:sitemap:products

This command will generate the products-sitemap.xml sitemap file in your public storage directory. This sitemap covers any products you create through the admin.

php artisan aero:subscriptions:alert-upcoming

This command will look for upcoming subscriptions and emit the **Aero\Subscription\Events\SubscriptionUpcoming **event for them.

php artisan aero:subscriptions:check-cards

This command will look for expired cards on subscriptions and then cancel the subscription and emit the **Aero\Subscription\Events\SubscriptionCardExpired**event for them.

php artisan aero:subscriptions:process

This command will process any subscriptions that need processing.

How to add a custom field to the address forms?

This short tutorial will walk you through adding an address line 3 field to all of the address forms (found in the admin, account-area, and checkout).

Migrations

The first step is to add your field to all of the address tables so that it can be stored. To do this we'll create and register a migration that adds a `line_3` field to the `addresses`, `order_addresses`, and `fulfillment_addresses` tables.

Migration Code

```
<?php

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

class AddLine3ToAddressTables extends Migration
{
   /**
    * Run the migrations.
    *
    * @return void
    */
   public function up()
   {
       Schema::table('addresses', function (Blueprint $table) {
           $table->string('line_3')->nullable()->after('line_2');
       });
```

```
        Schema::table('order_addresses', function (Blueprint $table) {
            $table->string('line_3')->nullable()->after('line_2');
        });

        Schema::table('fulfillment_addresses', function (Blueprint $table) {
            $table->string('line_3')->nullable()->after('line_2');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('fulfillment_addresses', function (Blueprint $table) {
            $table->dropColumn('line_3');
        });

        Schema::table('order_addresses', function (Blueprint $table) {
            $table->dropColumn('line_3');
        });

        Schema::table('addresses', function (Blueprint $table) {
            $table->dropColumn('line_3');
        });
    }
}
```

Service Provider Code

```
<?php

namespace Acme\MyModule;

use Aero\Common\Providers\ModuleServiceProvider;
```

```
class ServiceProvider extends ModuleServiceProvider
{
   public function setup()
   {
       if ($this->app->runningInConsole()) {
           $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
       }
   }
}
```

Views

The next step is to create a view for the line 3 field. To achieve this we'll create and register a Twig view for the field. The view will include the `forms::components.input` view to get a simple input field.

View Code

```
{% include "forms::components.input" with {
   type: 'text',
   error: errors.first((inputName ?? group) ~ '.line_3'),
   half: false,
   label: 'Address Line 3 (Optional)',
   class: (inputName ?? group) ~ '-' ~ 'line-3',
   name: (inputName ?? group) ~ '[line_3]',
   value: old((inputName ?? group) ~ '.line_3', address.line_3),
   autocomplete: type ? type ~ ' line_3' : 'line_3',
   required: true
} only %}
```

Service Provider Code

```
<?php

namespace Acme\MyModule;

use Aero\Common\Providers\ModuleServiceProvider;
```

```
class ServiceProvider extends ModuleServiceProvider
{
   public function setup()
   {
       if ($this->app->runningInConsole()) {
           $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
       }

       $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');
   }
}
```

Adding the Field to the Forms

Now we need to add the field to the address forms. To do this we'll make the field fillable and add it to the validation requests using the `Aero\Common\Helpers\Address::addField()` helper method. After that we'll extend `Aero\Forms\AddressForm` to add the field to the frontend address forms (checkout and account-area) and `Aero\Admin\Http\Forms\AdminAddressForm` to add the field to the admin address forms.

```
<?php

namespace Acme\MyModule;

use Aero\Admin\Http\Forms\AdminAddressForm;
use Aero\Common\Helpers\Address;
use Aero\Common\Providers\ModuleServiceProvider;
use Aero\Forms\AddressForm;

class ServiceProvider extends ModuleServiceProvider
{
   public function setup()
   {
       if ($this->app->runningInConsole()) {
           $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
       }
```

```
        $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');

        Address::addField('line_3');

        AddressForm::extend(function ($form) {
            $form->addSectionAfter('line3', 'my-module::line-3-field', 'line2');
        });

        AdminAddressForm::extend(function ($form) {
            $form->addSectionAfter('line3', 'my-module::line-3-field', 'line2');
        });
    }
}
```

Adding the Field to the Rendered Addresses

This step is optional and will add the line 3 field to the address when it's rendered (such as when viewing an order or viewing your addresses in the account area). To do this we'll extend `Aero\Address\Pipelines\AddressFormatter` and `Aero\Address\Pipelines\AddressStringFormatter`.

```
<?php

namespace Acme\MyModule;

use Aero\Address\Pipelines\AddressFormatter;
use Aero\Address\Pipelines\AddressStringFormatter;
use Aero\Admin\Http\Forms\AdminAddressForm;
use Aero\Common\Helpers\Address;
use Aero\Common\Providers\ModuleServiceProvider;
use Aero\Forms\AddressForm;

class ServiceProvider extends ModuleServiceProvider
{
    public function setup()
    {
        if ($this->app->runningInConsole()) {
```

```
            $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
        }

        $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');

        Address::addField('line_3');

        AddressForm::extend(function ($form) {
            $form->addSectionAfter('line3', 'my-module::line-3-field', 'line2');
        });

        AdminAddressForm::extend(function ($form) {
            $form->addSectionAfter('line3', 'my-module::line-3-field', 'line2');
        });

        AddressFormatter::extend(function ($formatter) {
            $formatter->putAfter('line3', $formatter->fields['line_3'], 'line2');
        });

        AddressStringFormatter::extend(function ($formatter) {
            $formatter->putAfter('line3', $formatter->fields['line_3'], 'line2');
        });
    }
}
```

How do I extend the address forms?

Aero has address forms in the checkout, account-area, and admin that can be extended.

Storefront Address Forms

The storefront address forms are address forms found on your storefront (the checkout and the account-area). These forms are customer facing and all extend `Aero\Forms\AddressForm`.

Structure

In all of the address forms there's a `top_sections`, `sections`, and `bottom_sections` section (some forms have an additional section). These sections build up the form. The reason that there are sections is so when the form is in lookup mode, the address fields can be hidden as they're all in the `sections` section.

Top Sections

Key	View
first_name	forms::address.fields.first-name
last_name	forms::address.fields.last-name
lookup	forms::address.lookup

Sections

Key	View
company	forms::address.fields.company
line1	forms::address.fields.line1

Key	View
line2	forms::address.fields.line2
city	forms::address.fields.city
zone	forms::address.fields.zone
postcode	forms::address.fields.postcode
country	forms::address.fields.country

Bottom Sections

Key	View
phone	forms::address.fields.phone

Additional Sections

Some storefront address forms have some additional sections on top of the ones above. To adjust these sections you'd have to specifically extend the specific form class instead of the generic **Aero\Forms\AddressForm**class.

Aero\Checkout\Http\Forms\CustomerAddressForm

This address form has additional customer sections.

Key	View
customer	account-area::forms.fields.address.name

Aero\AccountArea\Http\Forms\AccountAddressForm

This address form is extended by all of the other account area address forms. This form injects 2 additional things to the top and bottom sections.

The top sections has this injected right at the top of all of the top sections:

Key	View
address_name	checkout::sections.customer

The bottom sections has this injected right at the bottom of all of the bottom sections:

Key	View
is_default	account-area::forms.fields.address.is_default

Admin Address Forms

The admin address forms are address forms found on your admin. These forms all extend **Aero\Admin\Http\Forms\AdminAddressForm**.

Structure

The structure for the admin forms is the same as the structure for the storefront forms but the views are different.

Top Sections

Key	View
first_name	admin::forms.fields.address.first-name
last_name	admin::forms.fields.address.last-name
lookup	admin::forms.partials.address-lookup

Sections

Key	View
company	admin::forms.fields.address.company
line1	admin::forms.fields.address.line1
line2	admin::forms.fields.address.line2
city	admin::forms.fields.address.city
zone	admin::forms.fields.address.zone
postcode	admin::forms.fields.address.postcode
country	admin::forms.fields.address.country

Bottom Sections

Key	View
phone	admin::forms.fields.address.phone

Address Form Validation

All address forms take validation from **Aero\Forms\Validation\AddressRules**. Instead of adding rules to this class through the **extends**static method you should make use of the **Aero\Common\Helpers\Address::addField()**method. This method accepts the same parameters as if you were extending a validator (you can learn more about that here, link). The difference is that it adds the rules but also makes your field fillable on all of the address models. These models are **Aero\Account\Models\Address**, **Aero\Cart\Models\OrderAddress**, and **Aero\Fulfillment\Models\FulfillmentAddress**.

How do I add settings to my model?

This tutorial will explain the steps required to add settings to your model and assumes you have your model setup with create and edit pages.

Adding a Trait to your Model

The first step is to add the `Aero\Common\Traits\CanHaveSettings` trait to your model.

```
<?php

  namespace Acme\MyModule\Models;

  use Aero\Common\Models\Model;
  use Aero\Common\Traits\CanHaveSettings;

  class MyModel extends Model
  {
     use CanHaveSettings;
  }
```

Defining the Settings for your Model

Now that your model has the `Aero\Common\Traits\CanHaveSettings` trait you can use the static `settings()` function on your models class to define your settings.

The settings are defined in the same way as when you create a normal setting group, the only difference being that you use your models class instead of the normal settings facade. You can read more about settings here

```php
<?php

  namespace Acme\MyModule;

  use Acme\MyModule\Models\MyModel;
  use Aero\Common\Providers\ModuleServiceProvider;
  use Aero\Common\Settings\SettingGroup;

  class ServiceProvider extends ModuleServiceProvider
  {
     public function setup()
     {
         MyModel::settings(function (SettingGroup $group) {
             $group->encrypted('password');
             $group->boolean('require_password_to_view')->default(false);
         });
     }
  }
```

Adding the Settings Fields to your Create/Edit Pages

Now that you have defined the settings for your model you can update your models create/edit pages to let users update the settings.

To add a card that lets user edit the models settings to your create/edit pages you need to include the `admin::settings.model-settings` view and pass in your model, like this:

```
@include('admin::settings.model-settings', ['model' => $myModel])
```

If you don’t have a model to pass in (because you’re on a create page so the model hasn’t been created yet), pass in a fresh instance of your model, like this:

```
@include('admin::settings.model-settings', ['model' => new \Acme\MyModule\Models\MyModel()])
```

It’s important to ensure that the include is inside of your form. A more “complete” example may look something like this:

```
@extends('admin::layouts.main')


  @section('content')
     <div class="max-w-2xl mx-auto">
         <div class="flex w-full justify-between">
             <h2><a href="{{ route('admin.modules', request()->all()) }}" class="btn mr-
4">@include('admin::icons.back') Back</a> Managing My Model</h2>
         </div>
         @include('admin::partials.alerts')
         <form action="#" method="post" class="flex flex-wrap">
             @csrf
             @method('put')
             <fieldset class="w-full">
  {{--                Your other fields etc--}}

                 @include('admin::settings.model-settings', ['model' => $myModel])
                 <div class="form-buttons fieldset-disabled-hide">
                     <div class="card w-full">
                         <button class="btn btn-secondary" type="submit">Save</button>
                     </div>
                 </div>
             </fieldset>
         </form>
     </div>
  @endsection
```

Adding the Settings Validation to your Create/Edit Requests

You need to update your request validators so that the settings data will be correctly validated and formatted.

Adding the Rules

You need to update your validators `rules` method to merge in the settings rules with your current rules. To do this you need to `array_merge` your rules with the array of rules returned by the `Aero\Admin\Utils\SettingHelpers::getRulesForModel` method. The

`Aero\Admin\Utils\SettingHelpers::getRulesForModel` method expects you to pass the class string of your model.

```
<?php

namespace Acme\MyModule\Requests;

use Acme\MyModule\Models\MyModel;
use Aero\Admin\Utils\SettingHelpers;
use Aero\Common\Requests\AeroRequest;

class StoreMyModelRequest extends AeroRequest
{
   public function rules(): array
   {
       return array_merge([
           'name' => 'required|max:255', // You can add your models settings here
       ], SettingHelpers::getRulesForModel(MyModel::class));
   }
}
```

Adding the Attributes

You need to do the same thing for the attributes.

```
<?php

namespace Acme\MyModule\Requests;

use Acme\MyModule\Models\MyModel;
use Aero\Admin\Utils\SettingHelpers;
use Aero\Common\Requests\AeroRequest;

class StoreMyModelRequest extends AeroRequest
{
   public function attributes(): array
   {
       return array_merge([], SettingHelpers::getRuleAttributesForModel(MyModel::class));
   }
```

```
    public function rules(): array
    {
        return array_merge([
            'name' => 'required|max:255', // You can add your models settings here
        ], SettingHelpers::getRulesForModel(MyModel::class));
    }
}
```

Formatting the Data for Validation

You need to add a `prepareForValidation` method to your validator and ensure it calls the `SettingHelpers::formatRequestDataForModel` method like shown below. This method will format the incoming settings request data so that it is ready to be validated.

```
<?php

namespace Acme\MyModule\Requests;

use Acme\MyModule\Models\MyModel;
use Aero\Admin\Utils\SettingHelpers;
use Aero\Common\Requests\AeroRequest;

class StoreMyModelRequest extends AeroRequest
{
    public function prepareForValidation()
    {
        $this->replace(
            SettingHelpers::formatRequestDataForModel(MyModel::class, $this->all())
        );
    }

    public function attributes(): array
    {
        return array_merge([], SettingHelpers::getRuleAttributesForModel(MyModel::class));
    }

    public function rules(): array
    {
```

```
        return array_merge([
            'name' => 'required|max:255', // You can add your models settings here
        ], SettingHelpers::getRulesForModel(MyModel::class));
    }
}
```

Saving the Validated Settings for your Model

To save the settings you need to pass your model and data into the `Aero\Admin\Utils\SettingHelpers::saveForModel` method, like this:

```
<?php

namespace Acme\MyModule\Http\Controllers;

use Acme\MyModule\Models\MyModel;
use Acme\MyModule\Requests\UpdateMyModelRequest;
use Aero\Admin\Http\Controllers\Controller;
use Aero\Admin\Utils\SettingHelpers;

class MyModelController extends Controller
{
    public function update(UpdateMyModelRequest $request, MyModel $model)
    {
        $model->update($data = $request->validated());

        SettingHelpers::saveForModel($model, $request->validated()['settings'] ?? []);

        return redirect(route('my-model.index'))->with([
            'message' => __('Your changes have been saved'),
        ]);
    }
}
```

How do I add a custom price to a product with a module?

In this mini tutorial we will add some checkboxes to the product page that when checked add an additional charge to the product. The additional charge will be shown dynamically on the product page as it changes or the product's price changes (due to different variants being selected).

This mini tutorial assumes that you have a module setup or you’re happy working from the app service provider (using the boot method). You can see how to set up a module here (link).

Adding the Prices Data

To get started we will add a `$prices` array to our module service provider that will be the source of truth for our extra prices. We are using a simple array instead of database data due to this being a mini tutorial. The `$prices` array will have id, name, and price (this will also have inc and ex values and be in pence) keys.

```
<?php

namespace Acme\MyModule;

use Aero\Common\Providers\ModuleServiceProvider;

class ServiceProvider extends ModuleServiceProvider
{
   protected $prices = [
       [
           'id' => 1,
           'name' => 'Pay £1 more',
           'price' => [
               'inc' => 100,
               'ex' => 83.33,
           ],
       ],
```

```
        [
            'id' => 2,
            'name' => 'Pay £5 more',
            'price' => [
                'inc' => 500,
                'ex' => 416.6,
            ],
        ],
        [
            'id' => 3,
            'name' => 'Pay £10 more',
            'price' => [
                'inc' => 1000,
                'ex' => 833.33,
            ],
        ],
    ];

    public function setup()
    {
        //
    }
}
```

Creating a Javascript View and Adding the Prices Data and Javascript to the Product Page

Now we're going to create a Twig file called javascript where we will put the frontend javascript that will be used on the product page. This view is registered using the `$this->loadViewsFrom` method in the module service providers `setup` method.

Next we're going to use the `Aero\Store\Http\Responses\ProductPage` response builder to extend the product page. We will inject the $prices data using the setData method and then we'll use the `Aero\Store\Pipelines\ContentForBody` pipeline to add our javascript view to the product page.

```php
<?php

namespace Acme\MyModule;

use Aero\Common\Providers\ModuleServiceProvider;
use Aero\Store\Http\Responses\ProductPage;
use Aero\Store\Pipelines\ContentForBody;

class ServiceProvider extends ModuleServiceProvider
{
   protected $prices = [
       [
           'id' => 1,
           'name' => 'Pay £1 more',
           'price' => [
               'inc' => 100,
               'ex' => 83.33,
           ],
       ],
       [
           'id' => 2,
           'name' => 'Pay £5 more',
           'price' => [
               'inc' => 500,
               'ex' => 416.6,
           ],
       ],
       [
           'id' => 3,
           'name' => 'Pay £10 more',
           'price' => [
               'inc' => 1000,
               'ex' => 833.33,
           ],
       ],
   ];

   public function setup()
   {
```

```
        $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');

        ProductPage::extend(function (ProductPage $page) {
            $page->setData('extra_prices', $this->prices);

            ContentForBody::extend(function (&$content) {
                $content .= view('my-module::javascript');
            });
        });
    }
}
```

Adding the Extra Prices to the Product Page

In the `product.twig` file we're going to add some Vue code to show the total price and the total extra price. Then we'll add some Twig code to loop over the `extra_prices` variable that we injected into the product page earlier. This loop will display the name as a label and then add a checkbox with a value of the extra price id and some data attributes for the price inc and ex values.

```
Total Price
<p v-text="total_price.inc"></p>

Total Extra Price
<p v-text="additional_prices['extra-prices'].inc" v-if="additional_prices['extra-
prices']"></p>

{% for extra in extra_prices %}
    <div>
        <input type="checkbox" id="extraPrice{{ extra.id }}" value="{{ extra.id }}" data-extra-
price-inc="{{ extra.price.inc }}" data-extra-price-ex="{{ extra.price.ex }}">
        <label for="extraPrice{{ extra.id }}"> {{ extra.name }}</label>
    </div>
{% endfor %}
```

Coding the Javascript for the Product Page

Now we're going to add javascript code to the javascript twig view we previously created. This javascript code will be responsible for setting the additional price when the checkboxes are checked and sending any checked extra prices to the server when the product is added to the cart.

To update the additional price when the checkboxes are checked we'll set some code to be executed on the `product.loaded` Aero Event. This code will loop over all input checkbox elements that have the `data-extra-price-inc` attribute and add an input event listener. We'll add a `updateAndSetAdditionalPrices` function that will be executed when the input changes and also initially once the product has loaded (in case you have some extra prices that are checked by default). This function loops over all input checked checkboxes that have the `data-extra-price-inc` attribute and adds up their inc and ex prices. Then it runs the product.set-additional-price Aero Event to tell Aero the new total additional price.

To send the checked extra prices to the server we'll set some code to be executed on the `product.add-to-cart` Aero Event. This code loops over all input checked checkboxes that have the `data-extra-price-inc` attribute and adds their values (the extra price id) to an array. This array is then added to the payload that will be sent to the server.

```
<script>
   window.AeroEvents.on('product.loaded', function () {
       var extraPriceElements = document.querySelectorAll('input[type="checkbox"][data-extra-
price-inc]');

       for (var i = 0; i < extraPriceElements.length; i++) {
           extraPriceElements[i].addEventListener('input', updateAndSetAdditionalPrices);
       }

       function updateAndSetAdditionalPrices() {
           var price = { key: 'extra-prices', inc: 0, ex: 0 };
           var extraPriceElements =
document.querySelectorAll('input[type=checkbox]:checked[data-extra-price-inc]');

           for (var i = 0; i < extraPriceElements.length; i++) {
               price.inc += parseInt(extraPriceElements[i].dataset.extraPriceInc);
               price.ex += parseInt(extraPriceElements[i].dataset.extraPriceEx);
           }
```

```
            window.Aero.runEvent('product.set-additional-price', price);
        }

        updateAndSetAdditionalPrices();
    });

    window.AeroEvents.on('product.add-to-cart', function (data) {
        var extras = [];
        var extraPriceElements = document.querySelectorAll('input[type=checkbox]:checked[data-extra-price-inc]');

        for (var i = 0; i < extraPriceElements.length; i++) {
            extras.push(extraPriceElements[i].value);
        }

        data.extras = extras;

        return data;
    });
</script>
```

Adding any Selected Extra Prices to the Cart Item

We'll add some code in the module service provider below the current code that extends the `Aero\Store\Pipelines\CartItemBuilder` and gets the extra price ids from the request, maps them to the relevant extra price, filters out any that are null (because they were not valid extra price ids), and then adds a `Aero\Cart\CartItemOption` to the `Aero\Cart\CartItem` with the extra price details.

```
<?php

namespace Acme\MyModule;

use Aero\Cart\CartItem;
use Aero\Cart\CartItemOption;
use Aero\Common\Providers\ModuleServiceProvider;
use Aero\Store\Http\Responses\ProductPage;
```

```
use Aero\Store\Pipelines\CartItemBuilder;
use Aero\Store\Pipelines\ContentForBody;

class ServiceProvider extends ModuleServiceProvider
{
   protected $prices = [
       [
           'id' => 1,
           'name' => 'Pay £1 more',
           'price' => [
               'inc' => 100,
               'ex' => 83.33,
           ],
       ],
       [
           'id' => 2,
           'name' => 'Pay £5 more',
           'price' => [
               'inc' => 500,
               'ex' => 416.6,
           ],
       ],
       [
           'id' => 3,
           'name' => 'Pay £10 more',
           'price' => [
               'inc' => 1000,
               'ex' => 833.33,
           ],
       ],
   ];

   public function setup()
   {
       $this->loadViewsFrom(__DIR__.'/../resources/views', 'my-module');

       ProductPage::extend(function (ProductPage $page) {
           $page->setData('extra_prices', $this->prices);

           ContentForBody::extend(function (&$content) {
               $content .= view('my-module::javascript');
```

```
            });
        });

        CartItemBuilder::extend(function (CartItem $item) {
            collect(request()->input('extras', []))->map(function ($extra) {
                return collect($this->prices)->firstWhere('id', $extra);
            })->filter()->each(function ($extra) use ($item) {
                $option = CartItemOption::create($extra['name']);
                $option->setPriceInc($extra['price']['inc']);

                $item->addOption($option);
            });
        });
    }
}
```