Module Development

Module Development

- How do I register a custom Vue component for my module
- Anatomy of a custom module
- How do I create a custom module?
- How do I setup routes for my custom module?

How do I register a custom Vue component for my module

To register a Vue component for our module, we will need to create a JavaScript file for our components and in order to initialize Vue.

Link assets

In order to link all the assets, which is a necessary step of registering a Vue component, we need to add a function to the module **Service Provider**. This is the code we need in the service provider:

```
public function assetLinks()
{
     return [
         'vendor/module-name' => __DIR__ . '/../public',
     ];
}
```

Bear in mind this is the exact path we have to supply when loading components into a view.

Initialize components

```
import NewComponent from './components/NewComponent


window.newModule = {
    install(Vue) {
        Vue.component('new-component', NewComponent)
    },
}
```

The above code gives us access to the `<new-component></new-component>` tag, provided we've loaded the components into a view.

Loading components into a view

```
@push('scripts')
    <script src="{{ asset(mix(new-module.js', 'modules/aerocargo/new-module')) }}"></script>
    <script>
        window.AeroAdmin.vue.use(window.newModule);
    </script>
@endpush
```

Provided all of the namespacing in the two files matches correctly, the Vue components that we have declared will now be accessible in our view.

If there is a problem with the mix of our assets, remember to run `php artisan aero:link` in the **root directory** of your Aero store.

Enabling Vue devtools

In order to enable Vue devtools for our module, we simply add the following line of code into the file where we initialize our components:

```
import NewComponent from './components/NewComponent

window.new-module = {
    install(Vue) {
        Vue.config.devtools = true;

        Vue.component('new-component', NewComponent)
    },
}
```

Anatomy of a custom module

```
└─ module
    └─ database
        └─ migrations
    └─ public
    └─ resources
        └─ css
        └─ js
       └─ views
           └─ admin
          └─ store
    └─ routes
    └─ src
        └─ Http
       └─ Controllers
       └─ Responses
           └─ Steps
       └─ Requests
       └─ Models
    ServiceProvider.php
.gitignore
composer.json
README.md
```

Database

The database folder is only required for purposes of migrating, seeding, or storing data in formats such as JSON.

Migrations

Migrations is a folder within the database directory structure used for storing migration files which allow for populating the database with missing tables, or updating them.

Running `php artisan migrate` in the **root project directory** after installing a module detects migrations from all modules installed on Aero, which have not yet been migrated, and migrates them into Aero.

Public

All files within the public directory of a module can be published, to then be accessed anywhere in Aero. Some modules might require the public folder published before it can be used.

This might include images or files used to instantiate a JavaScript file for the use of Vue or any other JS framework.

Resources

The resources are required by the module for rendering views and storing custom CSS and JS files. Separating views into sub-folders allows for more readability and clarity on where particular views are used.

For example, the **admin** and **store** sub-folders of `resources/views` can be used for separating the different routing the module has.

Routes

Modules can have custom routes which are then served to the rest of the app. These might serve as callback URLs for APIs in the module or rendering views.

Example:

```
<?php

use Aerocargo\Csv\Http\Controllers\CsvController;
use Illuminate\Support\Facades\Route;

Route::group(['prefix' => 'csv'], function ($route) {
      $route->get('/', [CsvController::class, 'index'])->name('admin.csv.import');
      $route->get('/mapping/{hash?}', [CsvController::class, 'mapping'])-
>name('admin.csv.mapping');
      $route->get('/search', [CsvController::class, 'aeroFieldsSearch'])-
>name('admin.csv.search.aero');
      $route->get('/search-csv', [CsvController::class, 'csvFieldsSearch'])-
```

```
>name('admin.csv.search.csv');
      $route->get('/processing', [CsvController::class, 'processing'])-
>name('admin.csv.processing');

      $route->post('/importing', [CsvController::class, 'importing'])-
>name('admin.csv.importing');
      $route->post('/process', [CsvController::class, 'process'])-
>name('admin.csv.process');
      $route->post('/import', [CsvController::class, 'import'])-
>name('admin.csv.process.import');
      $route->post('/ping-import', [CsvController::class, 'pingImport'])-
>name('admin.csv.process.ping');
      $route->post('/get-mapping/{id?}', [CsvController::class, 'getMappingData'])-
>name('admin.csv.get.mapping');
});
```

Source (src)

The source folder contains all of the backend serving the entire module, whether it is controllers, models, or traits.

Service Provider

The module **Service Provider**is used to instantiate the module, and usually contains configuration for the resources the module uses. This could range from routes, views to being able to display the module and its UI within the Modules section of Aero.

The scaffolded Service Provider has several functions that are hidden from view.

These include:

- **setup()**- **assetLinks()**- **boot()**- **seed()**- **getSeeds()**

setup()

Every scaffolded Service Provider will include the setup() function, which has to be configured to include all necessary resources.

Example:

```
public function setup()
{
     AdminSlot::inject('catalog.products.index.header.buttons', function () {
         return view('csv::buttons.import');
     });

     Router::addAdminRoutes(__DIR__.'/../routes/admin.php');

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

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

     BulkAction::create(ExportProducts::class, ProductsResourceList::class)
                 ->notRunnable()
                 ->permissions('products.export')
                 ->title('Export products');
}
```

assetLinks()

Asset links are used to publish the resources included in the public folder of the module so that they can be used anywhere on the Aero platform.

Example:

```
public function assetLinks()
{
     return [
         'vendor/module-name' => __DIR__ . '/../public',
     ];
}
```

boot()

The boot() method is run in the background if the setup() method is present within the Service Provider. This is also the case for the booted() method.

seed()

With the **seed()**method, it is possible to add a seed to a collection of seeds for the module.

getSeeds()

Returns a unique collection of seeds that have been added to the module.

If we require our module to listen to certain events, they can be added into an array property of the class:

Listen property

The listen property is an array of events for each given module. The syntax for adding a listener to an event looks like so:

```
protected $listen = [
        \Aero\Cart\Events\OrderSuccessful::class => [
            \Aerocargo\Testing\Listeners\ExportOrder::class,
        ],
];
```

Custom directories

It is possible to create custom directories for different types of classes, for example, **Traits**, **Jobs**, or **Helpers**.

How do I create a custom module?

When interacting with the admin, especially when making a custom module, it is important to require `aerocommerce/admin` in the modules' `composer.json` file.

Scaffolding modules

Modules can be created using artisan, a command line interface supplied with Laravel. The easiest way to create a new module is to open the terminal, navigate to the root folder of the project and paste in:

```
php artisan make:module vendor/module-name
```

It is important to stick to module **naming conventions** and include the **vendor** (such as aerocommerce or aerocargo) and the **module name**, with dashes replacing any space in the name.

Running the above command in the terminal results in a couple of brief messages, reporting on the status of the creation and installation process of our module. After all is complete, the module can be accessed in the root directory of Aero under "modules".

Configuring modules

As mentioned above, each module that interacts with the admin needs to require `aerocommerce/admin` in the modules' `composer.json` file.

Composer.json

```
{
    "name": "aerocargo/new-module",
    "description": "",
    "require": {
        "php": "^7.2|^8.0",
        "aerocommerce/core": "^0",
        "aerocommerce/admin": "^0"
```

```
    },
    "autoload": {
        "psr-4": {
            "Aerocargo\\NewModule\\": "src/"
        }
    },
    "extra": {
        "laravel": {
            "providers": [
                "Aerocargo\\NewModule\\ServiceProvider"
            ]
        }
    }
}
```

Service Provider

The module **Service Provider**ensures that our modules are connected to the rest of the platform through the **setup()**method that is automatically scaffolded into the class.

Adding modules to a visible list in the admin

Some modules might not require any interaction with the admin in terms of UI, so they don't need to necessarily be listed in the modules section of the admin. If we wish to give our module an interface and allow users to access the module through the admin interface, we have to add the following code to our **Service Provider's setup()**function:

```
AdminModule::create('new-module')
            ->title('New Aero Module')
            ->summary('A brand new Aero module.')
            ->routes(__DIR__.'/../routes/admin.php')
            ->route('admin.new-module.index');
```

This module should now be listed in the admin.

Loading migrations

In order for the module to be able to detect all the migrations that a module has, it has to have a specified path in the modules' **Service Provider setup()**function.

If the migration folder follows the original anatomy of module structure, all we have to do is paste the following code into the **Service Provider setup()**function:

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

You should now be able to call `php artisan migrate` in the root Aero directory to migrate data from the module.

Loading views

Loading views works the same way as loading migrations, all we have to do it to specify the path to our views and also a namespace.

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

The second parameter in the above function is the namespace assigned to all of the module views. This is extra useful as we can then use that namespace to render views in the controller:

```
return view('new-module::index');
```

Loading routes

There are two different types of routes that the module has access to. One of them is the **admin routes** which only give access to routes provided the user is an administrator. The other is the **store routes** which affect the store/frontend side of the system.

Admin routes

There are two ways of setting up **admin routes**, depending on whether we choose to use the AdminModule facade and load the module into the admin interface.

If we do choose to add our module to a list in the admin, apply the following chained function to the AdminModule facade in the **setup()**function in the **Service Provider**:

```
AdminModule::create('new-module')
                    ->title('New Aero Module')
                    ->summary('A brand new Aero module.')
                    ->routes(__DIR__.'/../routes/admin.php');
```

This allows us to create the necessary admin routes, for navigating the module in the admin.

If we don't choose to add our module to a list in the admin, we simply add the following piece of code to the **setup()**function in the module **Service Provider**:

```
Router::addAdminRoutes(__DIR__.'/../routes/admin.php');
```

Store routes

In order to add store (frontend) routes to the module, we simply add a piece of code to the **setup()**function of the module **Service Provider**:

```
Router::addStoreRoutes(__DIR__.’/../routes/store.php');
```

Asset Linking

In order to publish any resources from our module to the rest of Aero, we have to specify the path for those resources in a special function. This function is added to the module **Service Provider**:

```
public function assetLinks()
{
        return [
                'aerocargo/new-module' => __DIR__.'/../public',
        ];
}
```

After defining any asset paths to link, you'll need to run the command:

```
php artisan aero:link
```

How do I setup routes for my custom module?

There are a few necessary steps to create and properly configure module routing.

Creating routes

The first step to creating module routes is to create a directory named **routes**on the same level as the **src**directory – for more information refer to "Anatomy of a custom module**.php**file in the **routes**directory. If my module only requires admin routes, I'll create a file called admin.php in the routes directory:

```
└─ module
    └─ database
        └─ migrations
    └─ public
    └─ resources
        └─ css
        └─ js
       └─ views
           └─ admin
          └─ store
    └─ routes
        └─ admin.php
    └─ src
        └─ Http
       └─ Controllers
       └─ Responses
           └─ Steps
       └─ Requests
       └─ Models
    ServiceProvider.php
.gitignore
composer.json
```

```
README.md
```

The contents of the file become:

```
<?php

use Illuminate\Support\Facades\Route;
```

The file can then be populated with the necessary routes the module needs. For more information on routes see the laravel documentaiton on routing

Loading routes in the Service Providers

The module needs to be made aware of all the routes that are available for it, and this is how it can be done:

Single routes file

```
$this->loadRoutesFrom(__DIR__.'/../routes/routes.php');
```

Store routes

```
Router::addStoreRoutes(__DIR__ . '/../routes/web.php');
```

Admin routes

```
Router::addAdminRoutes(__DIR__ . '/../routes/admin.php');
```

Using AdminModule facade

Routes can also be added through the **AdminModule**facade which allows the chaining for both **route()**and **routes()**. This should only be used with modules that have an interface/access point in the admin modules section. It can be done like so:

```
AdminModule::create('csv')
  ->title('CSV Import & Export')
  ->summary('Used to import and export a variety of platform data.')
  ->routes(__DIR__.'/../routes/admin.php')
  ->route('admin.csv.index');
```

Where the **route()**accesses a declared route that returns a view or **routes()**, with the same principle as the above examples.