Response Builders

Response Builders

How do I access the properties of the response builder?

All response builders hold properties that were assigned to them upon creation from the request. These properties are typically the arguments that would be passed to a conventional Laravel controller and subsequently used in the code within the controller.

For example on the **ProductPage**, the **Aero\Catalog\Models\Product model**for the requested URL can be accessed as follows:

```
\Aero\Store\Http\Responses\ProductPage::extend(function ($page) {
    $product = $page->product;

    // do things with the $product model
});
```

How do I pass data to a response?

The most common task performed when extending responses, is setting additional data. Typically, for HTML responses that render from a view, this involves sending data to the view to be used as a variable. If the response returns JSON, it may be adding additional key/value pairs.

For example, if you needed to obtain customer reviews for a given product and output them on the product page, you could do the following, which passes a product to a 3rd party reviews API and then adds the reviews it returns to the response data, which is then passed to the product.twig view file ready to be loop around and outputted.

```
\Aero\Store\Http\Responses\ProductPage::extend(function ($page) {
    $reviews = \Acme\ReviewsAPI::forProduct($page->product);

    $page->setData('reviews', $reviews);
});
```

Another example is getting the latest 5 blog posts from a blog module and sending them to the homepage to be displayed:

```
\Aero\Store\Http\Responses\Homepage::extend(function ($page) {
    $posts = \Acme\BlogPosts::latest()->limit(5)->get();

    $page->setData('posts', $posts);
});
```

You can obtain the existing data set against the response builder using the **getData()**method and remove data using the **removeData()**method.

How do I redirect a response?

Sometimes you may wish to redirect the user to another location, either to an external site or within the store. To do so, pass the URL, **route()**, or an **Illuminate\Http\RedirectResponse** instance to the **setRedirect()**method. For example, if you stored the age of customers (computed from their provided date of birth), and needed to prevent access to those under the age of 18:

```
\Aero\Store\Http\Responses\ProductPage::extend(function ($page) {
    if ($page->request->customer()->age < 18) {
        $page->setRedirect('/denied');
    }
});
```

As extension code is evaluated in the order it's added to the response builder, code that runs after this step may override the redirect. To force a redirect and prevent execution of the code that follows in the pipeline, you should return an instance of the **redirect()**helper:

```
\Aero\Store\Http\Responses\ProductPage::extend(function ($page) {
    if (! $page->product->canBeViewedByIp($page->request->ip())) {
        return redirect('/denied');
    }
});
```

How do I set the response status code?

There may be situations where the HTTP response status needs to be altered. By default, the returned status is **200**, however, this can be changed by passing the new status code to the **setStatus()**method:

```
\Aero\Store\Http\Responses\ProductPage::extend(function ($page) {
    $page->setStatus(403);
});
```

How do I set response headers?

When extending the response builder, you can set HTTP headers to be sent with the response:

```
\Aero\Store\Http\Responses\ProductPage::extend(function ($page) {
    $page->setHeader('foo', 'bar');
});
```

How do I attach middlewear to a response builder?

Since Aero does not expose the underlying routes and controllers, middleware can be attached directly to the response builder. This allows for code to be executed before the main pipeline is run.

For example, to restrict all products from being accessible to guest visitors, you could register the **auth**middleware to the **ProductPage**response:

```
\Aero\Store\Http\Responses\ProductPage::middleware('auth');
```

When assigning middleware, you can also pass the fully qualified class name:

```
\Aero\Store\Http\Responses\ProductPage::middleware(\App\Http\Middleware\CheckAge::class);
```

Alternatively, you can provide a **Closure**. For example, middleware could be added to the homepage to set a tracking cookie from the referrer:

```
\Aero\Store\Http\Responses\Homepage::middleware(function ($request, $next) {
    return tap($next($request), function ($response) use ($request) {
        $response->cookie('referer', $request->header('Referer'));
    });
});
```

Refer to the Laravel documentation for more information on middleware.

How do I extend responses?

Adding additional code to run on the response is as simple as calling the **extend()**method on the response builder class. This is typically done from within a **Service Provider**. If the code is unique to a particular store it could be placed in the boot method of the **AppServiceProvider**. If it is to be included as part of a module, then it can be added to the setup method of the module's Service Provider.

For example, if a module needs to send a variable **foo**to the **product.twig**view file for it to be outputted in the HTML, the **ProductPage**response builder class should be extended in the module's **Service Provider**as shown below:

```
<?php

namespace Acme\MyModule;

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

class ServiceProvider extends ModuleServiceProvider
{
    public function setup()
    {
        ProductPage::extend(function ($page) {
            $page->setData('foo', 'bar');
        });
    }
}
```

When passing a **Closure**to the extend method, the **$page**argument is the response builder instance that will process the response.

For more advanced use cases, you can pass the response builder on for further processing in order to return the **Illuminate\Http\Response**instance. This is useful in situations when you may wish to modify the response before it is passed to the browser. A common use case for this is to add cookies to the response:

```
\Aero\Store\Http\Responses\ProductPage::extend(function ($page, $next) {
    $response = $next($page);

    $response->cookie('foo', 'bar');

    return $response;
});
```

Alternatively, a fully qualified class name can be passed to the **extend()**method, which reduces bloat in the **Service Provider**and provides a better indication of what the code is responsible for. When doing so, the class must implement a **handle()**method:

```
<?php

namespace App\Http\Extensions;

class AddFooVariableToProductPage
{
    public function handle($page)
    {
        $page->setData('foo', 'bar');
    }
}
```

```
\Aero\Store\Http\Responses\ProductPage::extend(AddFooVariableToProductPage::class);
```

Note that this code will only evaluate when Aero routes a request to the particular response builder, i.e. code that is set to run on the **ProductPage**will not be processed on a request for the **Homepage**.

How do I change the response view?

For responses that return HTML, a Twig view file is used. Whilst these are originally defined, the view can be changed. For example, to change the product page view based on product data:

```
\Aero\Store\Http\Responses\ProductPage::extend(function ($page) {
    if ($view = $page->product->additional('view')) {
        $page->setView($view);
    }
});
```

Setting the view to **null**will result in a JSON response.

You can get the current view for a response using **$page->getView()**.

What are the available responses from the response builder?

Core

Class	Description	View
Homepage	The homepage of the store.	homepage.twig
ProductPage	The product page providing information on a particular product.	product.twig
ListingsPage	The listings page which displays available products for the particular criteria defined against the URL	listings.twig
ListingsJson	The JSON response of listings for the particular criteria defined against the URL.	-
SearchPage	The search results for a particular search term.	search.twig
SearchJson	The JSON response of search results for a particular search term.	-

Class	Description	View
CartPage	The cart summary page, where customers can see an overview of the items in their cart and update quantities.	cart.twig
CartItemAdd	Actions to take when an item is added to the cart.	-
CartItemUpdate	Actions to take when an item is updated in the cart.	-
CartEmpty	Actions to take when the cart is emptied.	-
CartDiscountCodeApply	Actions to take when a discount code is applied to the cart.	-
CartDiscountCodeRemove	Actions to take when a discount code is removed from the cart.	
InformationPage	A static page used to display information, for example the About Us or Terms & Conditions.	information-page.twig
Error404Page	The 404 page that displays when a page does not exist.	errors/404.twig
FormSubmit	Actions to take when a form is submitted.	-
AccountOverviewPage	A static page used to display account overview information.	account/account-overview.twig

Admin

Catalog

Class	Description	View
AdminDashboardPage	Used to display the admin dashboard.	admin/dashboard.blade.php
AdminCategoryCreatePage	Used to display the category creation page in the admin.	admin/catalog/categories/new.blade.php
AdminCategoryEditPage	Used to display the category edit page in the admin	admin/catalog/categories/edit.blade.php
AdminCategoryStore	Actions for creating a new category.	-
AdminCategoryUpdate	Actions for updating an existing category.	-
AdminCollectionCreatePage	Used to display the collection creation page in the admin.	admin/catalog/collections/index.blade.php
AdminCollectionEditPage	Used to display the collection edit page in the admin.	admin/catalog/collection/edit.blade.php
AdminCollectionStore	Actions for creating a new collection.	-
AdminCollectionUpdate	Actions for updating an existing collection.	-
AdminManufacturerCreatePage	Used to display the manufacturer creation page in the admin.	admin/catalog/manufacturers/new.blade.php
AdminProductCreatePage	Used to display the product creation page in the admin.	admin/catalog/products/new.blade.php
AdminProductStore	Action for creating a new product.	-
AdminProductUpdate	Actions for updating an existing product.	-

Discounts

Class	Description	View
AdminDiscountCreatePage	Used to display the discount creation page in the admin.	
AdminDiscountEditPage	Used to display the discount edit page in the admin.	admin/discounts/edit.blade.php
AdminDiscountStore	Actions for creating a new discount.	-
AdminDiscountUpdate	Actions for updating an existing discount.	-

Orders

Class	Description	View
AdminOrderViewPage	Used to display the order view page in the admin.	admin/orders/view.blade.php