Views

Creating Views

Looking for more information on how to write Blade templates? Check out the full Blade documentation to get started.

Views contain the HTML served by your application and separate your controller / application logic from your presentation logic. Views are stored in the `resources/views` directory. A simple view might look something like this:

```
<!-- View stored in resources/views/greeting.blade.php -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

Since this view is stored at `resources/views/greeting.blade.php`, we may return it using the global `view` helper like so:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

As you can see, the first argument passed to the `view` helper corresponds to the name of the view file in the `resources/views` directory. The second argument is an array of data that should be made available to the view. In this case, we are passing the `name` variable, which is displayed in the view using Blade syntax.

Views may also be nested within sub-directories of the `resources/views` directory. "Dot" notation may be used to reference nested views. For example, if your view is stored at `resources/views/admin/profile.blade.php`, you may reference it like so:

```
return view('admin.profile', $data);
```

Determining If A View Exists

If you need to determine if a view exists, you may use the `View` facade. The `exists` method will return `true` if the view exists:

```
use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {
    //
}
```

Creating The First Available View

Using the `first` method, you may create the first view that exists in a given array of views. This is useful if your application or package allows views to be customized or overwritten:

```
return view()->first(['custom.admin', 'admin'], $data);
```

You may also call this method via the `View` facade:

```
use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);
```

Passing Data To Views

As you saw in the previous examples, you may pass an array of data to views:

```
return view('greetings', ['name' => 'Victoria']);
```

When passing information in this manner, the data should be an array with key / value pairs. Inside your view, you can then access each value using its corresponding key, such as `<?php echo $key; ?>`. As an alternative to passing a complete array of data to the `view` helper function, you may use the `with` method to add individual pieces of data to the view:

```
return view('greeting')->with('name', 'Victoria');
```

Sharing Data With All Views

Occasionally, you may need to share a piece of data with all views that are rendered by your application. You may do so using the view facade's `share` method. Typically, you should place calls to `share` within a service provider's `boot` method. You are free to add them to the `AppServiceProvider` or generate a separate service provider to house them:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        View::share('key', 'value');
    }
}
```

View Composers

View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organize that logic into a single location.

For this example, let's register the view composers within a service provider. We'll use the `View` facade to access the underlying `Illuminate\Contracts\View\Factory` contract implementation. Remember, Laravel does not include a default directory for view composers. You are free to organize them however you wish. For example, you could create an `app/Http/View/Composers` directory:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ViewServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }
```

```
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        // Using class based composers...
        View::composer(
            'profile', 'App\Http\View\Composers\ProfileComposer'
        );

        // Using Closure based composers...
        View::composer('dashboard', function ($view) {
            //
        });
    }
}
```

Remember, if you create a new service provider to contain your view composer registrations, you will need to add the service provider to the `providers` array in the `config/app.php` configuration file.

Now that we have registered the composer, the `ProfileComposer@compose` method will be executed each time the `profile` view is being rendered. So, let's define the composer class:

```
<?php

namespace App\Http\View\Composers;

use App\Repositories\UserRepository;
use Illuminate\View\View;

class ProfileComposer
{
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * Create a new profile composer.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Dependencies automatically resolved by service container...
        $this->users = $users;
    }

    /**
     * Bind data to the view.
     *
     * @param  View  $view
     * @return void
     */
    public function compose(View $view)
    {
        $view->with('count', $this->users->count());
    }
}
```

Just before the view is rendered, the composer's `compose` method is called with the `Illuminate\View\View` instance. You may use the `with` method to bind data to the view.

All view composers are resolved via the service container, so you may type-hint any dependencies you need within a composer's constructor.

Attaching A Composer To Multiple Views

You may attach a view composer to multiple views at once by passing an array of views as the first argument to the `composer` method:

```
View::composer(
    ['profile', 'dashboard'],
    'App\Http\View\Composers\MyViewComposer'
);
```

The `composer` method also accepts the `*` character as a wildcard, allowing you to attach a composer to all views:

```
View::composer('*', function ($view) {
    //
});
```

View Creators

View **creators** are very similar to view composers; however, they are executed immediately after the view is instantiated instead of waiting until the view is about to render. To register a view creator, use the `creator` method:

```
View::creator('profile', 'App\Http\View\Creators\ProfileCreator');
```

architected project.

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.