Portões e Políticas em Laravel

por [Sajal Soni](https://tutsplus.com/authors/sajal-soni)15 Nov 2017

Hoje, discutiremos o sistema de autorização da estrutura da web do Laravel. A estrutura do Laravel implementa a autorização na forma de portões e políticas. Após uma introdução aos portões e políticas, demonstrarei os conceitos implementando um exemplo personalizado.

Suponho que você já esteja ciente do sistema de autenticação interno do Laravel, pois isso é essencial para entender o conceito de autorização. Obviamente, o sistema de autorização trabalha em conjunto com o sistema de autenticação para identificar a sessão legítima do usuário.

Se você não conhece o sistema de autenticação Laravel, eu recomendo consultar a [documentação oficial](https://laravel.com/docs/5.5/authentication) , que fornece informações práticas sobre o assunto.

**Abordagem do Laravel à Autorização**

Até agora, você já deve saber que o sistema de autorização do Laravel vem em dois sabores: portões e políticas. Embora possa parecer um assunto complicado, eu diria que é muito fácil implementá-lo quando você pegar o jeito!

Gates permitem definir uma regra de autorização usando uma abordagem simples baseada em fechamento. Em outras palavras, quando você deseja autorizar uma ação que não está relacionada a nenhum modelo específico, o portão é o lugar perfeito para implementar essa lógica.

Vamos dar uma olhada rápida em como é a autorização baseada em portas:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ...  ...  Gate::define('update-post', function ($user, $post) {    return $user->id == $post->user\_id;  });  ...  ... |

O snippet acima define a regra de autorização update-postque você pode chamar de qualquer lugar no seu aplicativo.

Por outro lado, você deve usar políticas quando desejar agrupar a lógica de autorização de qualquer modelo. Por exemplo, digamos que você tenha um modelo Post em seu aplicativo e deseje autorizar as ações CRUD desse modelo. Nesse caso, é a política que você precisa implementar.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class PostPolicy  {    public function view(User $user, Post $post) {}    public function create(User $user) {}    public function update(User $user, Post $post) {}    public function delete(User $user, Post $post) {}  } |

Como você pode ver, é uma classe de política bastante simples que define a autorização para as ações CRUD do Postmodelo.

Então isso foi uma introdução aos portões e políticas do Laravel. A partir da próxima seção, faremos uma demonstração prática de cada elemento.

**Portões (Gates)**

Nesta seção, veremos um exemplo do mundo real para entender o conceito de portões.

Na maioria das vezes, você acaba consultando o provedor de serviços Laravel quando precisa registrar um componente ou serviço. Seguindo essa convenção, vamos em frente e defina nosso portão personalizado no app/Providers/AuthServiceProvider.phpconforme mostrado no seguinte snippet.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28.  29 | <?php  namespace App\Providers;    use Illuminate\Support\Facades\Gate;  use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;  use Illuminate\Http\Request;    class AuthServiceProvider extends ServiceProvider  {    /\*\*     \* The policy mappings for the application.     \*     \* @var array     \*/    protected $policies = [      'App\Model' => 'App\Policies\ModelPolicy',    ];      /\*\*     \* Register any authentication / authorization services.     \*     \* @return void     \*/    public function boot()    {      $this->registerPolicies();        Gate::define('update-post', function ($user, $post) {        return $user->id == $post->user\_id;      });    }  } |

No bootmétodo, definimos nosso portão personalizado:

|  |  |
| --- | --- |
| 1  2  3 | Gate::define('update-post', function ($user, $post) {    return $user->id == $post->user\_id;  }); |

Ao definir um portão, é necessário um fechamento que retorne VERDADEIRO ou FALSO, com base na lógica de autorização definida na definição do portão. Além da função de fechamento, existem outras maneiras de definir portas.

Por exemplo, a seguinte definição de porta chama a ação do controlador em vez da função de fechamento.

|  |  |
| --- | --- |
| 1 | Gate::define('update-post', 'ControllerName@MethodName'); |

Agora, vamos em frente e adicione uma rota personalizada para que possamos passar por uma demonstração de como funciona a autorização baseada em portas. No arquivo de rotas routes/web.php, vamos adicionar a seguinte rota.

|  |  |
| --- | --- |
| 1 | Route::get('service/post/gate', 'PostController@gate'); |

Vamos criar um arquivo de controlador associado app/Http/Controllers/PostController.phptambém.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <?php  namespace App\Http\Controllers;    use App\Http\Controllers\Controller;  use App\Post;  use Illuminate\Support\Facades\Gate;    class PostController extends Controller  {    /\* Make sure you don't user Gate and Policy altogether for the same Model/Resource \*/    public function gate()    {      $post = Post::find(1);        if (Gate::allows('update-post', $post)) {        echo 'Allowed';      } else {        echo 'Not Allowed';      }        exit;    }  } |

Na maioria dos casos, você acabará usando o  método allowsou deniesda Gatefachada para autorizar uma determinada ação. No exemplo acima, usamos o allowsmétodo para verificar se o usuário atual é capaz de executar a update-postação.

Usuários com olhos afiados teriam notado que apenas passamos o segundo argumento $postpara o fechamento. O primeiro argumento, o usuário conectado atual, é injetado automaticamente pela Gatefachada.

Então é assim que você deve usar portões para autorizar ações em seu aplicativo Laravel. A próxima seção é sobre como usar políticas, caso você deseje implementar a autorização para seus modelos.

**Políticas (Polices)**

Como discutimos anteriormente, quando você deseja agrupar logicamente suas ações de autorização para qualquer modelo ou recurso específico, é a política que você está procurando.

Nesta seção, criaremos uma política para o modelo Post que será usada para autorizar todas as ações CRUD. Presumo que você já tenha implementado o modelo Post em seu aplicativo; caso contrário, algo semelhante fará.

O artisancomando Laravel é seu melhor amigo quando se trata de criar código stubbed. Você pode usar o seguinte comando artesanal para criar uma política para o modelo Post.

|  |  |
| --- | --- |
| 1 | $php artisan make:policy PostPolicy --model=Post |

Como você pode ver, fornecemos o --model=Postargumento para que ele crie todos os métodos CRUD. Na ausência disso, ele criará uma classe de política em branco. Você pode localizar a classe de política recém-criada em app/Policies/PostPolicy.php.

Vamos substituí-lo pelo seguinte código.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28.  29  30  31  32.  33  34  35  36.  37.  38.  39.  40.  41.  42.  43  44  45  46.  47  48.  49.  50. | <?php  namespace App\Policies;    use App\User;  use App\Post;  use Illuminate\Auth\Access\HandlesAuthorization;    class PostPolicy  {    use HandlesAuthorization;      /\*\*     \* Determine whether the user can view the post.     \*     \* @param  \App\User  $user     \* @param  \App\Post  $post     \* @return mixed     \*/    public function view(User $user, Post $post)    {      return TRUE;    }      /\*\*     \* Determine whether the user can create posts.     \*     \* @param  \App\User  $user     \* @return mixed     \*/    public function create(User $user)    {      return $user->id > 0;    }      /\*\*     \* Determine whether the user can update the post.     \*     \* @param  \App\User  $user     \* @param  \App\Post  $post     \* @return mixed     \*/    public function update(User $user, Post $post)    {      return $user->id == $post->user\_id;    }      /\*\*     \* Determine whether the user can delete the post.     \*     \* @param  \App\User  $user     \* @param  \App\Post  $post     \* @return mixed     \*/    public function delete(User $user, Post $post)    {      return $user->id == $post->user\_id;    }  } |

Para poder usar nossa classe de política, precisamos registrá-la usando o provedor de serviços Laravel, conforme mostrado no seguinte trecho.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | <?php  namespace App\Providers;    use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;  use Illuminate\Http\Request;  use App\Post;  use App\Policies\PostPolicy;    class AuthServiceProvider extends ServiceProvider  {    /\*\*     \* The policy mappings for the application.     \*     \* @var array     \*/    protected $policies = [      'App\Model' => 'App\Policies\ModelPolicy',      Post::class => PostPolicy::class    ];      /\*\*     \* Register any authentication / authorization services.     \*     \* @return void     \*/    public function boot()    {      $this->registerPolicies();    }  } |

Adicionamos o mapeamento de nossa política na  $policiespropriedade Diz ao Laravel para chamar o método de política correspondente para autorizar a ação CRUD.

Você também precisa registrar as políticas usando o registerPoliciesmétodo, como fizemos no bootmétodo.

Indo além, vamos criar algumas rotas personalizadas no routes/web.phparquivo para que possamos testar nossos métodos de política lá.

|  |  |
| --- | --- |
| 1  2  3  4 | Route::get('service/post/view', 'PostController@view');  Route::get('service/post/create', 'PostController@create');  Route::get('service/post/update', 'PostController@update');  Route::get('service/post/delete', 'PostController@delete'); |

Por fim, vamos criar um controlador associado em app/Http/Controllers/PostController.php.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28.  29  30  31  32.  33  34  35  36.  37.  38.  39.  40.  41.  42.  43  44  45  46.  47  48.  49.  50.  51  52  53  54  55  56.  57  58.  59.  60 | <?php  namespace App\Http\Controllers;    use App\Http\Controllers\Controller;  use App\Post;  use Illuminate\Support\Facades\Auth;    class PostController extends Controller  {    public function view()    {      // get current logged in user      $user = Auth::user();        // load post      $post = Post::find(1);        if ($user->can('view', $post)) {        echo "Current logged in user is allowed to update the Post: {$post->id}";      } else {        echo 'Not Authorized.';      }    }      public function create()    {      // get current logged in user      $user = Auth::user();        if ($user->can('create', Post::class)) {        echo 'Current logged in user is allowed to create new posts.';      } else {        echo 'Not Authorized';      }        exit;    }      public function update()    {      // get current logged in user      $user = Auth::user();        // load post      $post = Post::find(1);        if ($user->can('update', $post)) {        echo "Current logged in user is allowed to update the Post: {$post->id}";      } else {        echo 'Not Authorized.';      }    }      public function delete()    {      // get current logged in user      $user = Auth::user();        // load post      $post = Post::find(1);        if ($user->can('delete', $post)) {        echo "Current logged in user is allowed to delete the Post: {$post->id}";      } else {        echo 'Not Authorized.';      }    }  } |

Existem diferentes maneiras de autorizar suas ações usando Políticas. No exemplo acima, usamos o Usermodelo para autorizar nossas Postações de modelo.

O modelo de usuário fornece dois métodos úteis para fins de autorização - cane cant. O canmétodo é usado para verificar se o usuário atual é capaz de executar uma determinada ação. E a contraparte do canmétodo, o cantmétodo, é usada para determinar a incapacidade da execução da ação.

Vamos pegar o trecho do viewmétodo no controlador para ver o que exatamente ele faz.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14 | public function view()  {    // get current logged in user    $user = Auth::user();      // load post    $post = Post::find(1);      if ($user->can('view', $post)) {      echo "Current logged in user is allowed to update the Post: {$post->id}";    } else {      echo 'Not Authorized.';    }  } |

Primeiro, carregamos o usuário conectado no momento, o que nos fornece o objeto do modelo de usuário. Em seguida, carregamos uma postagem de exemplo usando o modelo Post.

Seguindo em frente, usamos o canmétodo do modelo Usuário para autorizar a viewação do Postmodelo. O primeiro argumento do canmétodo é o nome da ação que você deseja autorizar e o segundo argumento é o objeto de modelo que você deseja obter autorização.

Essa foi uma demonstração de como usar o Usermodelo para autorizar as ações usando políticas. Como alternativa, você também pode usar o *Controller Helper* , se estiver no controlador enquanto autoriza uma determinada ação.

|  |  |
| --- | --- |
| 1  2  3 | …  $this->authorize('view', $post);  … |

Como você pode ver, não é necessário carregar o modelo de usuário se você usar o Controller Helper.

Esse era o conceito de políticas à sua disposição e é realmente útil ao autorizar um modelo ou um recurso, pois permite agrupar a lógica de autorização em um único local.

Apenas certifique-se de não usar portas e políticas para as mesmas ações do modelo, caso contrário, isso criará problemas. É isso do meu lado por hoje, e eu vou encerrar o dia!