**Manual**

**Larus**

**V1.0.1**

**CONTENIDO**

Objetivo 2

Menú ………………………………………………………………………………………...2

Módulos ……………………………………………………………………………………..3

Widgets ……………………………………………………………………………………...6

# Objetivo

El objetivo del presente documento es ayudar al programador

# ¿Qué trae?

* Admin LTE => <https://almsaeedstudio.com/preview>
* Pingpong modules => <http://sky.pingpong-labs.com/docs/2.0/modules>
* PingPong Menus =><http://sky.pingpong-labs.com/docs/2.0/menus>
* PingPong Widget => <http://sky.pingpong-labs.com/docs/2.0/widget>
* Sweet Alert => <https://github.com/uxweb/sweet-alert>
* Flash => <https://github.com/laracasts/flash>
* Sentinel => <https://cartalyst.com/manual/sentinel/2.0>

# Menú

Larus utiliza PingPong Menus para administrar los menús de la aplicación. Por defecto se utiliza 1 sección o espacios de menú. **El siderbar-menu.**

# Agregando un ítem al menú

Para agregar un ítem de algún menú utilizamos el Service Provider del módulo que va agregar el menú. En dicho service provider utilizando el método boot, debemos obtener la instancia del menú al que le queremos agregar el ítem usando el FACADE Menú de la siguiente forma

$menu = Menu::instance('sidebar-menu');

En ese caso estamos obteniendo la instancia del menú sidebar-menu. Usando los métodos ofrecidos por Pingpong agregaremos un nuevo ítem:

$menu->url(

'module', // URL

'Nuevo Módulo', // Titulo

['icon' => 'fa fa-user'] // Icono

);

$menu->route(

**'dashboard'**, // Route ( php artisan route:list vez las rutas )

**'Dashboard'**, // Titulo

[],

1, // orden

[**'icon'** => **'fa fa-dashboard'**] // icono

);

# Usando Dropdowns

Si queremos hacer un ítem que sea dropdown con varias opciones, podemos usar el método dropdown de la siguiente manera:

// siderbar-menu es a la clase que estamos intanciando

$menu = Menu::instance('siderbar-menu');

// la función dropdown es para crear un menú con submenús

// titulo del menu

$menu->dropdown('Gestión de usuarios', function ($sub) {

$sub->route(

'users.index', // Ruta del controlador (php artisan route:list)

'Usuarios', // titulo del sub menu

[],

1, // posición

['icon' => 'fa fa-circle'] // icono

);

$sub->route('roles.index', 'Roles',[],2, ['icon' => 'fa fa-fa-circle']);

},2,[‘icon’=>’fa fa-users’]);

# Módulos

Larus utiliza Pingpong Modules para manejar una estructura modular, lo cual nos facilita la ventaja de crear módulos para la aplicación y compartirlo entre instalaciones.

Por defecto Larus trae un modulo de usuarios, roles, permisos, alertas, notificaciones y tareas.

# Creando Módulos

Para crear un módulo es hacerlo como un paquete, esto hace que sea más fácil de mantener y de transportar entre instalaciones. Usando la documentación de PingPong Modules podemos ver que tenemos un comando para crear el módulo.

php artisan module:make <module-name>

Este comando nos creara todo el scafold del módulo



# Service Provider

Cada módulo tiene un Service provider que se declara en el archivo module.json que se crea con el módulo. Este service provider será cargado automáticamente y es acá donde podrá registrar cualquier cosa que necesite como menus o cualquier otro bind.

/\*\*

\* Boot the application events.

\*

\* @return void

\*/

public function boot() {

$this->registerConfig();

$this->registerTranslations();

$this->registerViews();

$this->setMenu();

$this->widget();

}

# Widgets

Larus cuenta con un módulo de dashboard que permite el registro y visualización de widgets.

# Registrar un Widget

Para registración de un widget, en el service provider usando el método boot puede usar la clase de widgets que se resuelve del contenedor de laravel y usar el método registerWidget que recibe un arreglo de arreglos donde cada uno de esos arreglo es un widget con los siguientes parámetros:

* name : Nombre del widget ( sin espacios, ni caracteres especiales)
* class : Nombre de la clase que usara el widget, debe de ser el nombre completo sin espacios.
* order: El orden del widget dentro del gurpo
* group: el grupo que pertene el widget ( demo es el que usaremos)

use Modules\User\Widgets\TestBasciWidget;

use Modules\User\Widgets\TestMiddleWidget;

use Modules\User\Widgets\TestAdvancedWidget;

$widgets = app('app.widgets');

$widgets->registerWidget([

[

'name' => 'testBasic',

'class' => TestBasicWidget::class,

'order' => 1,

'group' => 'demo'

],

[

'name' => 'testMiddle',

'class' => TestMiddlwWidget::class,

'order' => 2,

'group' => 'demo'

] ,

[

'name' => 'testAdvanced',

'class' => TestAdvancedWidget::class,

'order' => 3,

'group' => 'demo'

]

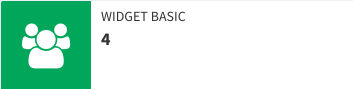
]);

# La clase del widget

La clase que se registra como widget debe de tener un método register quien es el que va a regresar el contenido del mismo, ese contenido por lo general es HTML y puede venir de una vista:

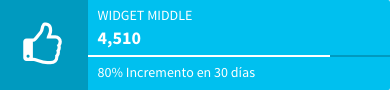
**Ejemplos:**

**Widget Basico Larus default**

****

**<?php namespace** Modules\User\Widgets;  
**use** Modules\User\Entities\User;  
*/\*\*  
 \* Class TotalUsersWidget  
 \* Display Total Users Statistics  
 \** ***@package*** *Modules\Users\Widgets  
 \*/***class** TotalUsersWidget {  
  
 */\*\*  
 \** ***@return*** *string  
 \*/* **public function** register()  
 {  
 $number = User::*count*(); // numero  
 $text = **"Widget basic"**; //texto  
 $bgcolor = **"bg-green"**; // color  
 $faicon = **"fa-users"**; // icono  
 **return** view(**'dashboard::widgets.widget-basic'**, *compact*(**'number'**,**'text'**,**'bgcolor'**,**'faicon'**))->render();  
 }  
}

**Widget Medio Larus core default**

****

**<?phpnamespace** Modules\User\Widgets;  
**class** TestMiddleWidget  
{  
 */\*\*  
 \** ***@return*** *string  
 \*/* **public function** register()  
 {  
 $text = **"Widget Middle"**; // texto   
 $bgcolor = **"bg-aqua"**; //color  
 $faicon = **"fa-thumbs-o-up"**; //icono  
 $progress = **"80"**; //progreso  
 $description = **"Incremento en 30 días"**; // descripcion  
 $number = **"4,510"**; // numero  
 **return** view(**'dashboard::widgets.widget-middle'**,*compact*(**'text'**,**'bgcolor'**,**'faicon'**,**'progress'**,**'description'**,**'number'**))->render();  
 }  
}

Widget Avanzado default Larus Core



**<?php namespace** Modules\User\Widgets;  
  
  
**class** TestAdvancedWidget  
{  
 **public function** register()  
 {  
 $number = **"41,130"**; // cantidad  
 $text = **"Widget Advanced"**; // Titulo  
 $bgcolor = **"bg-yellow"**; // color de fondo  
 $faicon = **"fa fa-shopping-cart"**; // icono  
 $url = **"#"**; // url  
 **return** view(**'dashboard::widgets.widget-advanced'**, *compact*(**'number'**,**'text'**,**'bgcolor'**,**'faicon'**,**'url'**))->render();  
 }  
}

Nota: ese ejemplo de clase de un widget es solo un ejemplo utilizando el widget middle que esta por default en el core de larus , los parámetros, pueden variar, dependiendo de su widget que usted crea y que es lo que quiera que tenga.

Nota:

1. los widgets por default, se localiza en /modules/Dashboard/Resources/views/widgets/
2. para crear tus propios widgets en tu modulo, puede tener la siguiente ruta .

/modules/NOMBRE\_DE\_TU\_MOdulo/Resources/views/widgets/nombre\_de\_tu\_widget.blade.php

1. en views puedes crear una carpeta llamada widgets.
2. Para llamar un widget de la siguiente forma.

**return** view(**'modelo::widgets.nombre\_de\_la\_vista '**, *compact*(**'parametros’**))->render();

# Registrar modulo y permisos

Cada modulo necesita tener sus permisos para en el cual se pueda asignar a cada role y se maneje el uso de accesos y restricciones.

*// insertamos en la tabla modules, el nombre de nuestro paquete*DB::*table*(**'modules'**)->insert(  
 **array**(  
 **'module\_name'** => **'Nombre del paquete'**,)  
);

// obtenemos el ultimo id   
$module\_id = DB::*getPdo*()->lastInsertId();

*// Insertamos los permisos que tendra nuestros modulo*DB::*table*(**'module\_permissions'**)->insert([  
 [  
 **'module\_id'** => $module\_id,  
 **'display\_name'** => **'ver test'**,  
 **'permission'** => **'test.view'** ],  
 [  
 **'module\_id'** => $module\_id,  
 **'display\_name'** => **'create test'**,  
 **'permission'** => **'test.create'** ],  
 [  
 **'module\_id'** => $module\_id,  
 **'display\_name'** => **'actualizar test'**,  
 **'permission'** => **'test.update'** ],  
 [  
 **'module\_id'** => $module\_id,  
 **'display\_name'** => **'delete test'**,  
 **'permission'** => **'test.delete'** ],  
 [  
 **'module\_id'** => $module\_id,  
 **'display\_name'** => **'widget test'**,  
 **'permission'** => **'TestWidget.view'** ]  
  
]);

# Sweet Alert

Para mandar alertas se utiliza SweetAlert, aquí unos ejemplos sencillos de cómo usarlo, lo utilizo para mandar el aviso de Acceso restringido

**public function** index(){  
 **if**(Sentinel::*hasAccess*(**'user.view'**)){  
 $users = $this->**model**->all();  
 **return** view(**'user::index'**,*compact*(**'users'**));  
 }  
 alert()->error(**'No tiene permisos para acceder a esta area.'**, **'Oops!'**)->persistent(**'Cerrar'**);  
 **return** back();  
  
}

## Uso

### Con Facade

* Alert::message('Message', 'Optional Title');
* Alert::basic('Basic Message', 'Mandatory Title');
* Alert::info('Info Message', 'Optional Title');
* Alert::success('Success Message', 'Optional Title');
* Alert::error('Error Message', 'Optional Title');
* Alert::warning('Warning Message', 'Optional Title');

### Con el Helper

* alert($message = null, $title = '')
* alert()->message('Message', 'Optional Title')

# Flash

## Uso

En el controlador antes de hacer el redirect()

public function store() {

Flash::message('Welcome Aboard!');

return Redirect::home();

}

Con este mensaje dirigió a la sesión, es posible que ahora lo mostrará en la vista (s)

**@include('flash::message')**

# Tasks

Para usar el modelo de tasks se necesito hacer el llamado de la siguiente manera

**use** Modules\Tasks\Entities\Task;

De esta forma podemos usar todo el eloquent del modelo y además con sus métodos

**private function** roles($id)

Busca los usuarios de un rol en especifico.

$task->addUser($user\_id)

Agrega un usuario a una tarea.

$task->addRole($role\_id)

Agrega todos los usuarios de ese role a la tarea

**private function** today()

Regresa la fecha del día

**public function** scopeNoDone($query)

Hace una consulta donde regresa todos los que no tengan estatus DONE

**public function** scopeDone($query)

Hace una consulta en la cual regresa todos los de estatus DONE

**public function** scopeOutTime($query)

Hace la consulta que regresa la tarea que expiro

# Tabla Tasks

**role\_id**: Id del role , es necesario guardarlo cuando se agrega por ID role sino va null.  
**flow\_id:** cada tarea esta enlazada a un flujo de trabajo, el flujo de trabajo de una tarea por default es el Id 1  
**name:** nombre de la tarea  
**description:** descripción de la tarea  
**start\_date**: fecha de inicio   
**due\_date:** fecha de vencimiento  
**status**: estatus de la tarea, por de ley es importante terminar con el estatus DONE, para terminar una tarea  
**progress**: para medir el progreso %  
**user\_id\_done:** el id del usuario quien la termino  
**done\_date:** Fecha de cuando se termino la tarea.

# Flows

Para poder crear un flujo de trabajo para un modulo o paquete, se creo unas tablas en el cual pueden ser utilizadas para poder controlar el flujo de trabajo de una tarea ( ya depende de su forma de lógica de trabajar ).

# Tabla Flow

Name: *Nombre del flujo*

Description: *Descripcion del flujo*

Module: *Nombre del modulo*

Active: *estado del modulo*

# Tabla Flows\_steps

En esta tabla vamos agregar los pasos del flujo de trabajo

Name: *nombre del paso de flujo*

Description: *descripción del paso de flujo*

Order: *orden*

Is\_last: es el ultimo paso

# Tabla flow\_steps\_transition

Flow\_id : *id del flujo de trabajo*

Step\_from\_id : primer paso

Step\_to\_id: segundo paso

Para poder usar el modelo de flows o step tenemos que llamarlo para poder usar su eloquent o query buildier de ese modelo

**use** Modules\Tasks\Entities\Flow;

**use** Modules\Tasks\Entities\Step;

# Notificaciones

Para poder crear notificaciones es importante llamar al modelo notificaciones de la siguiente manera:

**use** Modules\Notifications\Entities\Notification;

Una vez llamando el modelo, podemos usuar el eloquent de ese modelo, y sus métodos de ahí.

**protected $fillable** = [

**'role\_id'**, // Cuando se crea una nueva notificación por role, es importante guardar el rol

**'icon'**, // Agregar el icono de su modulo ej. “$fields[**'icon'**] = **"fa-envelope-o"**;

**'name'**, // nombre de la notificacion

**'description'**, // descripción de la notificacion

**'url', link**

];

*/\*\*  
 \* Search users in that role  
 \** ***@param*** *$role\_id  
 \** ***@return*** *mixed  
 \*/***private function** roles($id)

*/\*\*  
 \* Add task\_id and user\_id table task\_user  
 \** ***@param*** *$user\_id  
 \*/***public function** addUser($id)

*/\*\*  
 \* Add task\_id and array (user\_id) table task\_user  
 \** ***@param*** *$role\_id  
 \*/***public function** addRole($id)

*/\*\*  
 \*marcar como leida una notificacion   
 \** ***@param*** *$user\_id  
 \*/***public function** addRead($id)

*/\*\*  
\* si no es leida  
 \** ***@param*** *$user\_id  
 \*/***public function** addNoRead($id)

las siguiente funciones se usan para esconder si la notificación es eliminada por un usuario o no

Si el campo active es 1 o true, ya no le aparece al usuario la notificación, le aparece como si fuera eliminada, si es 0, es como si no hubiera sido eliminada

*/\*\*  
 \* if Active is 1, its like to delete  
 \** ***@param*** *$user\_id*

*\*  
 \*/***public function** addActive($id)

*/\*\*  
 \* if Active is 0 its no delete  
 \** ***@param*** *$user\_id  
 \*/***public function** addNoActive($id)

*/\*\*  
 \* Notifications no read and not delete  
 \** ***@param*** *$query  
 \** ***@return*** *mixed  
 \*/***public function** scopeNoRead($query)

*/\*\*  
 \* Notifications read and not delete  
 \** ***@param*** *$query  
 \*/***public function** scopeRead($query)