A simple shopping cart implementation for Laravel

157 commits | 2 branches | 19 releases | 21 contributors | MIT

Branch: master | New pull request | Find file | Clone or download

Crinsane Merge pull request #425 from publiux/master ... Latest commit 7ff157b on Feb 17

config	Set default number format in config file	2 years ago
database/migrations	update migration	10 months ago
src	Update CartItem.php	8 months ago
tests	Updated PHPUnit to work with version 6.0 required by Laravel 5.5	8 months ago
.gitignore	Large update for version 2.0	2 years ago
.travis.yml	No need to test on PHP55 anymore.	11 months ago
LICENSE	Large update for version 2.0	2 years ago
README.md	Update README.md	8 months ago
composer.json	Updated composer.json for Laravel 5.6 support	2 months ago
phpunit.xml	Large update for version 2.0	2 years ago

README.md

LaravelShoppingcart

build passing | downloads 311.64 k | stable 2.5.0 | unstable dev-master | license MIT

A simple shoppingcart implementation for Laravel.

Installation

Install the package through Composer.

Run the Composer require command from the Terminal:

```
composer require gloudemans/shoppingcart
```

If you're using Laravel 5.5, this is all there is to do.

Should you still be on version 5.4 of Laravel, the final steps for you are to add the service provider of the package and alias the package. To do this open your `config/app.php` file.

Add a new line to the `providers` array:

```
Gloudemans\Shoppingcart\ShoppingcartServiceProvider::class
```

And optionally add a new line to the `aliases` array:

```
'Cart' => Gloudemans\Shoppingcart\Facades\Cart::class,
```

Now you're ready to start using the shoppingcart in your application.

As of version 2 of this package it's possibly to use dependency injection to inject an instance of the Cart class into your controller or other class

Overview

Look at one of the following topics to learn more about LaravelShoppingcart

Usage

The shoppingcart gives you the following methods to use:

Cart::add()

Adding an item to the cart is really simple, you just use the `add()` method, which accepts a variety of parameters.

In its most basic form you can specify the id, name, quantity, price of the product you'd like to add to the cart.

```
Cart::add('293ad', 'Product 1', 1, 9.99);
```

As an optional fifth parameter you can pass it options, so you can add multiple items with the same id, but with (for instance) a different size.

```
Cart::add('293ad', 'Product 1', 1, 9.99, ['size' => 'large']);
```

The `add()` method will return an CartItem instance of the item you just added to the cart.

Maybe you prefer to add the item using an array? As long as the array contains the required keys, you can pass it to the method. The options key is optional.

```
Cart::add(['id' => '293ad', 'name' => 'Product 1', 'qty' => 1, 'price' => 9.99, 'options' => ['size' => 'large']]);
```

New in version 2 of the package is the possibility to work with the `Buyable` interface. The way this works is that you have a model implement the `Buyable` interface, which will make you implement a few methods so the package knows how to get the id, name and price from your model. This way you can just pass the `add()` method a model and the quantity and it will automatically add it to the cart.

As an added bonus it will automatically associate the model with the CartItem

```
Cart::add($product, 1, ['size' => 'large']);
```

As an optional third parameter you can add options.

```
Cart::add($product, 1, ['size' => 'large']);
```

Finally, you can also add multipe items to the cart at once. You can just pass the `add()` method an array of arrays, or an array of Buyables and they will be added to the cart.

When adding multiple items to the cart, the `add()` method will return an array of CartItems.

```
Cart::add([
  ['id' => '293ad', 'name' => 'Product 1', 'qty' => 1, 'price' => 10.00],
  ['id' => '4832k', 'name' => 'Product 2', 'qty' => 1, 'price' => 10.00, 'options' => ['size' => 'large']]
]);

Cart::add([$product1, $product2]);
```

Cart::update()

To update an item in the cart, you'll first need the rowId of the item. Next you can use the `update()` method to update it.

If you simply want to update the quantity, you'll pass the update method the rowId and the new quantity:

```
$rowId = 'da39a3ee5e6b4b0d3255bfef95601890afd80709';

Cart::update($rowId, 2); // Will update the quantity
```

If you want to update more attributes of the item, you can either pass the update method an array or a `Buyable` as the second parameter. This way you can update all information of the item with the given rowId.

```
Cart::update($rowId, ['name' => 'Product 1']); // Will update the name

Cart::update($rowId, $product); // Will update the id, name and price
```

Cart::remove()

To remove an item for the cart, you'll again need the rowId. This rowId you simply pass to the `remove()` method and it will remove the item from the cart.

```
$rowId = 'da39a3ee5e6b4b0d3255bfef95601890afd80709';

Cart::remove($rowId);
```

Cart::get()

If you want to get an item from the cart using its rowId, you can simply call the `get()` method on the cart and pass it the rowId.

```
$rowId = 'da39a3ee5e6b4b0d3255bfef95601890afd80709';

Cart::get($rowId);
```

Cart::content()

Of course you also want to get the carts content. This is where you'll use the `content` method. This method will return a Collection of CartItems which you can iterate over and show the content to your customers.

```
Cart::content();
```

This method will return the content of the current cart instance, if you want the content of another instance, simply chain the calls.

```
Cart::instance('wishlist')->content();
```

Cart::destroy()

If you want to completely remove the content of a cart, you can call the destroy method on the cart. This will remove all CartItems from the cart for the current cart instance.

```
Cart::destroy();
```

Cart::total()

The `total()` method can be used to get the calculated total of all items in the cart, given there price and quantity.

```
Cart::total();
```

The method will automatically format the result, which you can tweak using the three optional parameters

```
Cart::total($decimals, $decimalSeperator, $thousandSeperator);
```

You can set the default number format in the config file.

If you're not using the Facade, but use dependency injection in your (for instance) Controller, you can also simply get the total property `$cart->total`

Cart::tax()

The `tax()` method can be used to get the calculated amount of tax for all items in the cart, given there price and quantity.

```
Cart::tax();
```

The method will automatically format the result, which you can tweak using the three optional parameters

```
Cart::tax($decimals, $decimalSeperator, $thousandSeperator);
```

You can set the default number format in the config file.

If you're not using the Facade, but use dependency injection in your (for instance) Controller, you can also simply get the tax property `$cart->tax`

Cart::subtotal()

The `subtotal()` method can be used to get the total of all items in the cart, minus the total amount of tax.

```
Cart::subtotal();
```

The method will automatically format the result, which you can tweak using the three optional parameters

```
Cart::subtotal($decimals, $decimalSeperator, $thousandSeperator);
```

You can set the default number format in the config file.

If you're not using the Facade, but use dependency injection in your (for instance) Controller, you can also simply get the subtotal property `$cart->subtotal`

Cart::count()

If you want to know how many items there are in your cart, you can use the `count()` method. This method will return the total number of items in the cart. So if you've added 2 books and 1 shirt, it will return 3 items.

```
Cart::count();
```

Cart::search()

To find an item in the cart, you can use the `search()` method.

This method was changed on version 2

Behind the scenes, the method simply uses the filter method of the Laravel Collection class. This means you must pass it a Closure in which you'll specify you search terms.

If you for instance want to find all items with an id of 1:

```
$cart->search(function ($cartItem, $rowId) {
        return $cartItem->id === 1;
});
```

As you can see the Closure will receive two parameters. The first is the CartItem to perform the check against. The second parameter is the rowId of this CartItem.

The method will return a Collection containing all CartItems that where found

This way of searching gives you total control over the search process and gives you the ability to create very precise and specific searches.

Collections

On multiple instances the Cart will return to you a Collection. This is just a simple Laravel Collection, so all methods you can call on a Laravel Collection are also available on the result.

As an example, you can quicky get the number of unique products in a cart:

```
Cart::content()->count();
```

Or you can group the content by the id of the products:

```
Cart::content()->groupBy('id');
```

Instances

The packages supports multiple instances of the cart. The way this works is like this:

You can set the current instance of the cart by calling `Cart::instance('newInstance')`. From this moment, the active instance of the cart will be `newInstance`, so when you add, remove or get the content of the cart, you're work with the `newInstance` instance of the cart. If you want to switch instances, you just call `Cart::instance('otherInstance')` again, and you're working with the `otherInstance` again.

So a little example:

```
Cart::instance('shopping')->add('192ao12', 'Product 1', 1, 9.99);

// Get the content of the 'shopping' cart
Cart::content();

Cart::instance('wishlist')->add('sdjk922', 'Product 2', 1, 19.95, ['size' => 'medium']);

// Get the content of the 'wishlist' cart
Cart::content();

// If you want to get the content of the 'shopping' cart again
Cart::instance('shopping')->content();

// And the count of the 'wishlist' cart again
Cart::instance('wishlist')->count();
```

N.B. Keep in mind that the cart stays in the last set instance for as long as you don't set a different one during script execution.

N.B.2 The default cart instance is called `default`, so when you're not using instances, `Cart::content();` is the same as `Cart::instance('default')->content()`.

Models

Because it can be very convenient to be able to directly access a model from a CartItem is it possible to associate a model with the items in the cart. Let's say you have a `Product` model in your application. With the `associate()` method, you can tell the cart that an item in the cart, is associated to the `Product` model.

That way you can access your model right from the `CartItem`!

The model can be accessed via the `model` property on the CartItem.

If your model implements the `Buyable` interface and you used your model to add the item to the cart, it will associate automatically.

Here is an example:

```
// First we'll add the item to the cart.
$cartItem = Cart::add('293ad', 'Product 1', 1, 9.99, ['size' => 'large']);

// Next we associate a model with the item.
Cart::associate($cartItem->rowId, 'Product');

// Or even easier, call the associate method on the CartItem!
$cartItem->associate('Product');

// You can even make it a one-liner
Cart::add('293ad', 'Product 1', 1, 9.99, ['size' => 'large'])->associate('Product');

// Now, when iterating over the content of the cart, you can access the model.
foreach(Cart::content() as $row) {
        echo 'You have ' . $row->qty . ' items of ' . $row->model->name . ' with description: "' . $row->model->descr
}
```

Database

- Config
- Storing the cart
- Restoring the cart

Configuration

To save cart into the database so you can retrieve it later, the package needs to know which database connection to use and what the name of the table is. By default the package will use the default database connection and use a table named `shoppingcart`. If you want to change these options, you'll have to publish the `config` file.

```
php artisan vendor:publish --provider="Gloudemans\Shoppingcart\ShoppingcartServiceProvider" --tag="config"
```

This will give you a `cart.php` config file in which you can make the changes.

To make your life easy, the package also includes a ready to use `migration` which you can publish by running:

```
php artisan vendor:publish --provider="Gloudemans\Shoppingcart\ShoppingcartServiceProvider" --tag="migrations"
```

This will place a `shoppingcart` table's migration file into `database/migrations` directory. Now all you have to do is run `php artisan migrate` to migrate your database.

Storing the cart

To store your cart instance into the database, you have to call the `store($identifier)` method. Where `$identifier` is a random key, for instance the id or username of the user.

```
Cart::store('username');

// To store a cart instance named 'wishlist'
Cart::instance('wishlist')->store('username');
```

Restoring the cart

If you want to retrieve the cart from the database and restore it, all you have to do is call the `restore($identifier)` where `$identifier` is the key you specified for the `store` method.

```
Cart::restore('username');

// To restore a cart instance named 'wishlist'
Cart::instance('wishlist')->restore('username');
```

Exceptions

The Cart package will throw exceptions if something goes wrong. This way it's easier to debug your code using the Cart package or to handle the error based on the type of exceptions. The Cart packages can throw the following exceptions:

Exception	Reason
CartAlreadyStoredException	When trying to store a cart that was already stored using the specified identifier
InvalidRowIDException	When the rowId that got passed doesn't exists in the current cart instance
UnknownModelException	When you try to associate an none existing model to a CartItem.

Events

The cart also has events build in. There are five events available for you to listen for.

Event	Fired	Parameter
cart.added	When an item was added to the cart.	The `CartItem` that was added.
cart.updated	When an item in the cart was updated.	The `CartItem` that was updated.
cart.removed	When an item is removed from the cart.	The `CartItem` that was removed.
cart.stored	When the content of a cart was stored.	-
cart.restored	When the content of a cart was restored.	-

Example

Below is a little example of how to list the cart content in a table:

```


// Add some items in your Controller.
Cart::add('192ao12', 'Product 1', 1, 9.99);
Cart::add('1239ad0', 'Product 2', 2, 5.95, ['size' => 'large']);

// Display the content in a View.
<table>
        <thead>
        <tr>
                <th>Product</th>
                <th>Qty</th>
                <th>Price</th>
                <th>Subtotal</th>
        </tr>
        </thead>
```

```
        <tbody>

                <?php foreach(Cart::content() as $row) :?>

                <tr>
                        <td>
                        <p><strong><?php echo $row->name; ?></strong></p>
                        <p><?php echo ($row->options->has('size') ? $row->options->size : ''); ?></p>
                        </td>
                        <td><input type="text" value="<?php echo $row->qty; ?>"></td>
                        <td>$<?php echo $row->price; ?></td>
                        <td>$<?php echo $row->total; ?></td>
                </tr>

                <?php endforeach;?>

        </tbody>
```

```
        <tfoot>
                <tr>
                        <td colspan="2"> </td>
                        <td>Subtotal</td>
                        <td><?php echo Cart::subtotal(); ?></td>
                </tr>
                <tr>
                        <td colspan="2"> </td>
                        <td>Tax</td>
                        <td><?php echo Cart::tax(); ?></td>
                </tr>
                <tr>
                        <td colspan="2"> </td>
                        <td>Total</td>
                        <td><?php echo Cart::total(); ?></td>
                </tr>
        </tfoot>
</table>
```