django-easycart Documentation

Release 0.1.0

nevimov

May 28, 2016

Contents

Easycart is a flexible session-based shopping cart application for Django. It provides plenty of hooks for overriding and extending the way it works.

By installing this app you get:

- Highly-customizable `BaseCart` and `BaseItem` classes to represent the user cart and items in it.
- A handy set of reusable components (views, urls and a context processor) for the most common tasks. These components are completely optional.

Requirements: Python 3.4+ Django 1.8+

CHAPTER 1

Table of Contents

1.1 Quickstart

This document demonstrates how you can use Easycart to implement the shopping cart functionality in your django project.

1.1.1 Install the app

Before you do anything else, ensure that Django Session Framework is enabled and configured.

Use pip to install Easycart:

```
$ pip install django-easycart
```

Add the app to your INSTALLED_APPS setting:

```
INSTALLED_APPS = [
    ...
    'easycart',
]
```

1.1.2 Define your cart class

First, create a new django application:

```
$ python manage.py startapp cart
```

It will contain things not provided by Easycart, such as templates and static files. Those are unique to each project, so it's your responsibility to provide them.

Next, we need to create a customized cart class. Don't worry, it's really easy, just subclass *BaseCart* and override its *get_queryset()* method:

```
# cart/views.py
from easycart import BaseCart

# We assume here that you've already defined your item model
# in a separate app named "catalog".
from catalog.models import Item
```

```
class Cart(BaseCart):

    def get_queryset(self, pks):
        return Item.objects.filter(pk__in=pks)
```

Now, our class knows how to communicate with the item model.

Note: For simplicity's sake, the example above supposes that a single model is used to access all database information about items. If you use multi-table inheritance, see *this link*.

There are many more customizations you can make to the cart class, check out *Cookbook* and *Reference*, after you complete this tutorial.

1.1.3 Plug in ready-to-use views

Every cart needs to perform tasks like adding/removing items, changing the quantity associated with an item or emptying the whole cart at once. You can write your own views for that purpose, using the cart class we've created above, but what's the point in reinventing the wheel? Just use *the ones* shipped with Easycart.

Add the following to your project settings:

```
EASYCART_CART_CLASS = 'cart.views.Cart'
```

Create `cart/urls.py`:

```
from django.conf.urls import url

urlpatterns = [
    # This pattern must always be the last
    url('', include('easycart.urls'))
]
```

Include it in the root URLconf:

```
url(r'^cart/', include('cart.urls')),
```

Now, the cart can be operated by sending POST-requests to Easycart urls:

URL name	View
cart-add	*AddItem*
cart-remove	*RemoveItem*
cart-change-quantity	*ChangeItemQuantity*
cart-empty	*EmptyCart*

Tip: It would be wise to create a javascript API to handle these requests. Here's an oversimplified example of such an API that can serve as a starting point. It uses a bit of jQuery and assumes that CSRF-protection has already been taken care of.

```
var cart = {
    add: function (pk, quantity) {
      quantity = quantity || 1
      return $.post(URLS.addItem, {pk: pk, quantity: quantity}, 'json')
    }

    remove: function (itemPK) {
      return $.post(URLS.removeItem, {pk: itemPK}, 'json')
```

```
    }

    changeQuantity: function (pk, quantity) {
      return $.post(URLS.changeQuantity, {pk: pk, quantity: quantity}, 'json')
    }

    empty: function () {
      $.post(URLS.emptyCart, 'json')
    }
}
```

Inline a script similar to the one below in your base template, so you don't have to hardcode the urls.

```
    <script>
    var URLS = {
      addItem:        '{% url "cart-add" %}',
      removeItem:     '{% url "cart-remove" %}',
      changeQuantity: '{% url "cart-change-quantity" %}',
      emptyCart:      '{% url "cart-empty" %}',
    }
    </script>
```

1.1.4 Access the cart from templates

To enable the built-in cart context processor, add `context_processors.cart` to your project settings:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # other context processors
                'easycart.context_processors.cart',
            ],
        },
    },
]
```

Now, the cart can be accessed in any template through context variable `cart` like this:

```
{{ cart.item_count }}
{{ cart.total_price }}

{% for item in cart.list_items %}
<div>
    {# Access the item's model instance using its "obj" attribute #}
    {{ item.obj.name }}
    <img src="{{ item.obj.picture.url }}">
    {{ item.price }}
    {{ item.quantity }}
    {{ item.total }}
</div>
{% endfor %}
```

The name of the variable can be changed using the *EASYCART_CART_VAR* setting.

Well, that's all. Of course, you still need to write some front-end scripts and create additional views (for instance, for order processing), but all of this is far beyond the scope of this document.

1.2 Cookbook

1.2.1 Adapting to multiple item models

If you use multi-table inheritance in your item models, then you will likely want that cart items were associated with instances of their respective child models. This can be achieved by overriding the *process_object()* method of the *BaseCart* class.

Let's assume we have the following models:

```
# catalog/models.py
from django.db import models

class Item(models.Model):
    name = models.CharField(max_length=40)
    price = models.PositiveIntegerField()

class Book(Item):
    author = models.CharField(max_length=40)

class Magazine(Item):
    issue = models.CharField(max_length=40)
```

Instances of `Item` can access their respective child model through attributes *book* and *magazine*. The problem is, we don't know in advance which one to use. The easiest way to circumvent it is to use a tryexcept block to access each attribute one by one:

```
from django.core.exceptions import ObjectDoesNotExist
from easycart import BaseCart

CATEGORIES = ('book', 'magazine')


class Cart(BaseCart):

    def get_queryset(self, pks):
        return Item.objects.filter(pk__in=pks).select_related(*CATEGORIES)

    def process_object(self, obj):
        for category in CATEGORIES:
            try:
                return getattr(obj, category)
            except ObjectDoesNotExist:
                pass
```

Alternatively, just store the name of the right attribute in a separate field on `Item`:

```
class Item(models.Model):
    name = models.CharField(max_length=40)
    price = models.PositiveIntegerField()
    category = models.CharField(max_length=50, editable=False)

    def save(self, *args, **kwargs):
        if not self.category:
```

```
            self.category = self.__class__.__name__.lower()
        super().save(*args, **kwargs)
```

In this case, your cart class may look something like this:

```
class Cart(BaseCart):

    def get_queryset(self, pks):
        return Item.objects.filter(pk__in=pks).select_related(*CATEGORIES)

    def process_object(self, obj):
        return getattr(obj, obj.category)
```

Attention: Whatever technique you choose, be sure to use select_related() to avoid redundant queries to the database.

1.2.2 Associating arbitrary data with cart items

You can associate arbitrary data with items by passing extra keyword arguments to the cart's method *add()*.

As an example, we will save the date and time the item is added to the cart. Having a timestamp may be handy in quite a few scenarios. For example, many e-commerce sites have a widget displaying a list of items recently added to the cart.

To implement such functionality, create a cart class similar to the one below:

```
import time
from easycart import BaseCart

class Cart(BaseCart):

    def add(self, pk, quantity=1):
        super(Cart, self).add(pk, quantity, timestamp=time.time())

    def list_items_by_timestamp(self):
        return self.list_items(sort_key=lambda item: item.timestamp, reverse=True)
```

Now, in your templates, do something like:

```
{% for item in cart.list_items_by_timestamp|slice:":6" %}
    {{ item.name }}
    {{ item.price }}
{% endfor %}
```

1.2.3 Adding per item discounts and taxes

To change the way the individual item prices are calculated, you need to override the *total()* method of the *BaseItem* class.

Assume we have the following models.py:

```
from django.db import models

class Item(models.Model):
    price = models.DecimalField(decimal_places=2, max_digits=8)
    # Suppose discounts and taxes are stored as percentages
```

```
    discount = models.IntegerField(default=0)
    tax = models.IntegerField(default=0)
```

In this case, your item class may look like this:

```
class CartItem(BaseItem):

    @property
    def total(self):
        discount_mod = 1 - self.obj.discount/100
        tax_mod = 1 + self.obj.tax/100
        return self.price * discount_mod * tax_mod


class Cart(BaseCart):
    # Point the cart to the new item class
    item_class = CartItem
```

1.2.4 Limiting the maximum quantity allowed per item

You may want to limit the maximum quantity allowed per item, for example, to ensure that the user can't put more items in his cart than you have in stock.

See the *max_quantity* attribute of the *BaseCart* class.

1.3 Settings

Most of the Easycart behavior is customized by overriding and extending the *BaseCart* and *BaseItem* classes, however, a few things are controlled through the settings below:

EASYCART_CART_CLASS
A string pointing to *your cart class*.

Has no default value, must always be set, if you want to use *built-in views*.

EASYCART_CART_VAR
default: 'cart'

The name for the context variable providing *access to the cart from templates*.

EASYCART_SESSION_KEY
default: 'easycart'

Key in `request.session` under which to store the cart data.

1.4 Reference

1.4.1 easycart.views module

A set of views every cart needs.

On success, each view returns a JSON-response with the cart representation. For the details on the format of the return value, see the *encode()* method of the *BaseCart* class.

Note: All of the views in this module accept only POST requests.

class easycart.views.**AddItem**(***kwargs*)
Add an item to the cart.

This view expects *request.POST* to contain:

key	value
pk	the primary key of an item to add
quantity	a quantity that should be associated with the item

class easycart.views.**RemoveItem**(***kwargs*)
Remove an item from the cart.

Expects *request.POST* to contain key *pk*. The associated value should be the primary key of an item you wish to remove.

class easycart.views.**ChangeItemQuantity**(***kwargs*)
Change the quantity of an item.

This view expects *request.POST* to contain:

key	value
pk	the primary key of an item
quantity	a new quantity to associate with the item

class easycart.views.**EmptyCart**(***kwargs*)
Remove all items from the cart.

1.4.2 easycart.cart module

Core classes to represent the user cart and items in it.

BaseCart

class easycart.cart.**BaseCart**(*request*)
Base class representing the user cart.

In the simplest case, you just subclass it in your views and override the *get_queryset()* method.

If multi-table inheritance is used to store information about items, then you may also want to override *process_object()* as well.

Parameters **request** (*django.http.HttpRequest*)

Variables

- **items** (*dict*) – A map between item primary keys (converted to strings) and corresponding instances of *item_class*. If, for some reason, you need to modify *items* directly, don't forget to call *update()* afterwards.
- **item_count** (*int*) – The total number of items in the cart. By default, only unique items are counted.
- **total_price** (*same as the type of item prices*) – The total value of all items in the cart.
- **request** – A reference to the *request* used to instantiate the cart.

item_class
Class to use to represent cart items.

alias of *BaseItem*

add(*pk*, *quantity=1*, ***kwargs*)
Add an item to the cart.

If the item is already in the cart, then its quantity will be increased by *quantity* units.

Parameters

- **pk** (*str or int*) – The primary key of the item.
- **quantity** (*int-convertible*) – A number of units of to add.
- ****kwargs** – Extra keyword arguments to pass to the item class constructor.

Raises *ItemNotInDatabase* – Database doesn't contain an item with the given primary key.

change_quantity(*pk*, *quantity*)
Change the quantity of an item.

Parameters

- **pk** (*str or int*) – The primary key of the item.
- **quantity** (*int-convertible*) – A new quantity.

Raises *ItemNotInCart* – Attempting to change the quantity of an item that is not in the cart.

remove(*pk*)
Remove an item from the cart.

Parameters **pk** (*str or int*) – The primary key of the item.

Raises *ItemNotInCart* – Attempting to delete an item that is not in the cart.

empty()
Remove all items from the cart.

list_items(*sort_key=None*, *reverse=False*)
Return a list of cart items.

Parameters

- **sort_key** (*func*) – A function to customize the list order, same as the 'key' argument to the built-in `sorted()`.
- **reverse** (*bool*) – If set to True, the sort order will be reversed.

Returns *list* – List of *item_class* instances.

Examples

```
>>> cart = Cart(request)
>>> cart.list_items(lambda item: item.obj.name)
[<CartItem: obj=bar, quantity=3>,
 <CartItem: obj=foo, quantity=1>,
 <CartItem: obj=nox, quantity=5>]
>>> cart.list_items(lambda item: item.quantity, reverse=True)
[<CartItem: obj=nox, quantity=5>,
 <CartItem: obj=bar, quantity=3>,
 <CartItem: obj=foo, quantity=1>]
```

encode (*formatter=None*)

Return a representation of the cart as a JSON-response.

Parameters **formatter** (*func, optional*) – A function that accepts the cart representation and returns its formatted version.

Returns *django.http.JsonResponse*

Examples

Assume that items with primary keys "1" and "4" are already in the cart.

```
>>> cart = Cart(request)
>>> def format_total_price(cart_repr):
...     return intcomma(cart_repr['totalPrice'])
...
>>> json_response = cart.encode(format_total_price)
>>> json_response.content
b'{
    "items": {
        '1': {"price": 100, "quantity": 10, "total": 1000},
        '4': {"price": 50, "quantity": 20, "total": 1000},
    },
    "itemCount": 2,
    "totalPrice": "2,000",
}'
```

get_queryset (*pks*)

Construct a queryset using given primary keys.

The cart is pretty much useless until this method is overriden. The default implementation just raises `NotImplementedError`.

Parameters **pks** (*list of str*)

Returns *django.db.models.query.QuerySet*

Examples

In the most basic case this method may look like the one below:

```
def get_queryset(self, pks):
    return Item.objects.filter(pk__in=pks)
```

process_object (*obj*)

Process an object before it will be used to create a cart item.

This method provides a hook to perform arbitrary actions on the item's model instance, before it gets associated with the cart item. However, it's usually used just to replace the passed model instance with its related object. The default implementation simply returns the passed object.

Parameters **obj** (*item model*) – An item's model instance.

Returns *item model* – A model instance that will be used as the *obj* argument to *item_class*.

handle_stale_items (*pks*)

Handle cart items that are no longer present in the database.

The default implementation results in silent removal of stale items from the cart.

Parameters **pks** (*set of str*) – Primary keys of stale items.

create_items(*session_items*)
Instantiate cart items from session data.

The value returned by this method is used to populate the cart's *items* attribute.

Parameters **session_items** (*dict*) – A dictionary of pk-quantity mappings (each pk is a string). For example: `{'1': 5, '3': 2}`.

Returns

dict – A map between the *session_items* keys and instances of *item_class*. For example:

```
{'1': <CartItem: obj=foo, quantity=5>,
 '3': <CartItem: obj=bar, quantity=2>}
```

update()
Update the cart.

First this method updates attributes dependent on the cart's *items*, such as *total_price* or *item_count*. After that, it saves the new cart state to the session.

Generally, you'll need to call this method by yourself, only when implementing new methods that directly change the *items* attribute.

count_items(*unique=True*)
Count items in the cart.

Parameters **unique** (*bool-convertible, optional*)

Returns *int* – If *unique* is truthy, then the result is the number of items in the cart. Otherwise, it's the sum of all item quantities.

count_total_price()
Get the total price of all items in the cart.

BaseItem

class easycart.cart.**BaseItem**(*obj*, *quantity=1*, ***kwargs*)
Base class representing the cart item.

Parameters

- **obj** (*subclass of django.db.models.Model*) – A model instance holding database information about the item. The instance is required to have an attribute containing the item's price.
- **quantity** (*int, optional*) – A quantity to associate with the item.

Variables

- **obj** – A reference to the *obj* argument.
- **price** (same as *obj.price*) – The price of the item (a reference to the corresponding attribute on the *obj*).

Raises *InvalidItemQuantity* – Argument *quantity* doesn't pass the validation by the *clean_quantity()* method.

PRICE_ATTR = 'price'
str – The name of the *obj* attribute containing the item's price.

max_quantity = None
The maximum quantity allowed per item.

Used by the *clean_quantity()* method. Should be either a positive integer or a falsy value. The latter case disables the check. Note that you can make it a property to provide dynamic values.

Examples

If you want to ensure that the user can't put more items in his cart than you have in stock, you may write something like this:

```
class CartItem(BaseItem):
    @property
    def max_quantity(self):
        return self.obj.stock
```

quantity
int – The quantity associated with the item.

A read/write property.

New values are checked and normalized to integers by the *clean_quantity()* method.

total
same as obj.price – Total price of the item.

A read-only property.

The default implementation simply returns the product of the item's price and quantity. Override to adjust for things like an individual item discount or taxes.

clean_quantity(*quantity*)
Check and normalize the quantity.

The following checks are performed:

- the quantity can be converted to an integer
- it's positive
- it's doesn't exceed the value of *max_quantity*

Parameters **quantity** (*int-convertible*)

Returns *int* – The normalized quanity.

Raises *InvalidItemQuantity* – The quantity can't be cleaned due to one of the reasons listed above.

Module Exceptions

class easycart.cart.**CartException**
Bases: Exception

Base class for cart exceptions.

class easycart.cart.**InvalidItemQuantity**
Bases: *easycart.cart.CartException*

Provided item quantity is invalid.

class easycart.cart.**ItemNotInCart**(*pk*, **args*)
Bases: *easycart.cart.CartException*

Item with the given pk is not in the cart.

class easycart.cart.**ItemNotInDatabase**
Bases: *easycart.cart.CartException*

Database doesn't contain an item with the given primary key.

e