basky

Release 0.9.8

July 11, 2014

Contents

basky is a lightweight, robust basket application for Django projects.

Contents:

CHAPTER 1

Getting Started

1.1 License

basky is licensed under The BSD 3-Clause License

1.2 What is basky not?

basky is not a full stack e-commerce solution, it only handles the basket part of the ecommerce stack.

There is no products application because nearly all businesses have their own subtle requirements when it comes to products. Whilst being able write software that handles 99% of use cases displays fine talent, often each project will only use a portion of such code, so we decided to leave that part to you; to reduce complexity.

1.3 What is basky?

basky is lightweight (but robust) basket application for Django projects. basky is essentially a collection of urls, views, forms, signals and middleware. The goal of basky is that you can use it with zero fuss and minimal head scratching.

basky doesn't care what you put into it, it only cares that the 'thing' you put into the basket has two properties

- **Name** - a unicode string
- **Total** - a decimal formatted to two places

The experience of using the basket has been modelled on how you would interact with a shopping basket in the real world and the development has been largely lead by designers and clients. With this in mind some of the behavior make seem a little 'cooky', but it's really easy to get your head around.

1.4 Installing

You can install basky using pip

```
pip install django-basky
```

or you can clone/fork it directly from the main basky repository on Github.

Add `basky` to INSTALLED_APPS in your settings.py file

```
INSTALLED_APPS = (
        # ...
        'basky',
        # ...
)
```

Add `basky.middleware.BasketInSessionMiddleware` into MIDDLEWARE_CLASSES in your settings.py file:

```
MIDDLEWARE_CLASSES = (
        # ...
        'basky.middleware.BasketInSessionMiddleware',
        # ...
)
```

Add `basky.urls` into your urls.py file:

```
urlpatterns = patterns('',
        url(r'^basket/', include('basky.urls', namespace='basky')),
)
```

Note: The basky namespace is non-negotiable. If the namespace is not basky then tests will fail, the sky will turn blood red and all humanity as you know it will cease to be. You haz had da warnings.

Now do a syncdb to create the basket tables:

```
./manage.py syncdb
```

Basky comes with a management command that will delete baskets older than `settings.BASKY_AGE`. You can either call this via cron or whap it into a task queue like Celery. You'll probably want to enable this as each time a new request is made without a basket in the session a new basket is created.

Note: I am aware how much this approach sucks, and I'll be implementing a LazyBasket approach for the 1.0 version of basky.

Finally, (and you don't have to do this), you can add `basky.context_processors.basket` into the tuple of `CONTEXT_PROCESSORS` in your settings.py. Adding `basky.context_processors.basket` will inject a variable into all templates called `basket` or whatever you've set `BASKY_SESSION_KEY_NAME` to.

```
TEMPLATE_CONTEXT_PROCESSORS = (
        #...
        'basky.context_processors.basket'
)
```

If you've not added to the tuple of defaults then you may not see this setting in your settings file

And that's that, you're set. So let's jump in with a few examples.

1.5 Meet The Basket

1.5.1 Registering A Product With The Basket

Basky has a registration pattern and this is used to register products with the basket.

If the thing that you want to use in the basket has properties called `name` and `total` then you don't have to register the product with the basket. A default configuration will be applied and you're good to go.

The default configuration looks this:

```
class BasketConfig(object):
    """Basic configuration"""
    # the property used for the name
    name = 'name'
    # the property used to get the total
    total = 'total'
    # the form that is used to post information to the basket
    form = BasketForm
```

If your model does not have a property called name or total, or you'd like to provide some custom configuration then you need to create a file called `basket.py` in the same directory as your `models.py` and it will be automatically found and applied.

A custom configuration may look like this:

```
class ProductBasketConfig(object)::
    name = 'catalogue_name'
    total = 'cost'
    form = MyBasketForm
```

1.5.2 Getting The Basket

Remember that you put the middleware into your settings? Great, because that's what ensures that all users have a basket attached to their session. We can access the the basket from the request object like this:

```
basket = request.session.get('basket')
```

Note: If you want the basket to be called something other than `basket` you can set this in the *Configuration* options.

The basket is easy to interogate, we can see how many items we have in the basket by asking it:

```
>>> basket.total_items()
>>> 0
```

This will return an integer, and obviously because we've not put anything in it, it will return 0.

If you wanted to know how much the total price of all of the items in the basket comes to, just ask it:

```
>>> basket.total_price()
>>> 0.00
```

This will return a decimal number formatted to two decimal places. Again, we haven't got anything in the basket so we'll get a big fat `0.00`

To prove this let's ask the basket to give us all of the items:

```
>>> basket.basketitem_set.all()
>>> []
```

It returned an empty BasketItem() queryset. Let's put something in the basket and then ask it again so we can explore the BasketLine()

1.5.3 Adding To The Basket

Note: basky doesn't provide any models, because that bit is up to you – It'll be largely dependant on the project

you're working in.

Let's assume that you have the following model:

```
class SimpleProduct(models.Model):
        name = models.CharField(max_length=255)
        price = models.DecimalField(max_digits=8, decimal_places=2)

        def __unicode__(self):
                return u'%s' % self.name

        @property
        def total(self):
                return self.price
```

Let's also assume that the following object exists:

```
book = SimpleProduct(
        name='Colour Theory',
        price=Decimal('9.99')
)
```

We can add `book` to the basket easy like this:

```
>>> basketitem = basket.add(book)
```

When we add to a basket we will always get a *BasketItem* object back, it's up to you if you want to keep it around or not. There's no reason not to really.

We didn't specify a quantity so a quantity of one was assumed. A few things happened behind the scenes as well. Firstly two *Signals* were emitted – `pre_add_to_basket` and `post_add_to_basket`. You could use these signals to attach listeners for weird and wonderful product behavior.

Now we have an item in the basket we can once again get some information back from it:

```
>>> basket.total_items()
>>> 1
```

Awesome, what about price:

```
>>> basket.total_price()
>>> 9.99
```

Splendid, what about getting the the basketitems? Easy peasy, it's just the standard Django ORM API with no shenanigans:

```
>>> basket.basketitem_set.all()
>>> ['BasketItem: 1 × Colour Theory']
```

Well isn't that smashing. Let's add another book into the basket:

```
>>> basketitem = basket.add(book)
```

Now we have two items

```
>>> basket.total_items()
>>> 2
```

With a total of 19.98

```
>>> basket.total_price()
>>> 19.98
```

And the items

```
>>> basket.items()
>>> ['BasketItem: 2 × Colour Theory']
```

1.5.4 Emptying The Basket

Alright this is all getting a bit contrived, let's wrap this up by removing the book from the basket

```
>>> basketitem = basket.add(book)
>>> basket.remove(basketitem)
```

So what's left in the basket?

```
>>> basket.basketitem_set.all()
>>> []
```

Nothing is left in basket. See, that was nice and easy.

1.5.5 Signals

In all this kerfuffle about emptying the basket we missed the emission of some *Signals*. The first *Signals* were `pre_remove_from_basket` and `post_remove_from basket` – No prizes for guessing when they were sent. These two *Signals* were sent each time we removed items from the basket.

However, there were also two more *Signals* that was sent and that was `basket_is_now_not_empty` and `basket_is_now_empty`.

`basket_is_now_not_empty` is sent when someones basket was empty and now isn't. `basket_is_now_empty` is sent when someones basket had items, but now doesn't.

CHAPTER 2

How It Works

2.1 Basket

The `basketitems` are foreignkey'd to the `basket`.

Every item in the `basket` is a `basketitem`.

If you have two loaves of bread in your `basket`, then there will be two `basketitem` objects to represent this.

Basketitems each have an md5 hash that allows quick and accurate comparison of basketitems. This is useful if you want to group the items for display back to the user.

CHAPTER 3

Models

3.1 Why no product models?

There is no models production ready models bundled with this application because in our humble experience, the nature of a what constitutes a products varies wildly from one project to the next.

The goal of basky is to be lightweight and robust, so bundling complex ‘’and the kitchen sink” style product models is not inline with our aims for the application.

If however, you’re looking for a starting point you can find some very contrived examples in the test project.

3.2 Defining Your Own Product Models

basky doesn’t use the registry pattern, because it doesn’t care what you add into the basket. The only requirement is that the instance you add into the basket have a:

- Property called `name` - **String**: used to make the basketitem description
- Property called `total` - **Decimal**: formatted to two decimals places.

This is the only requirement. Nice and easy lemon squeezy.

3.3 Basket

3.3.1 Basket.add(self, item, **kwargs)

Accepts the following arguments

- `quantity` - **Integer**: defaults to `1`. Is the quantity of items to add to the basket.
- `price` - **Decimal**: defaults to `None`. Overrides the price of the item
- `description` - **String**: defaults to `None`. Overrides the name of the item
- `silent` - **Boolean**: defaults to `False`. If `True` the `pre_add_to_basket` & `post_add_to_basket` signals will not be sent.
- `locked` - **Boolean**: defaults to `False`. If `True` then the basket item will not be editable by user, so they won’t be able to remove it or update the quantity

- `append` - **Boolean**: defaults to `True`. If `True` then the quantity will be added onto the current quantity. If `False` it will replace the current quantity.

3.3.2 Basket.remove(self, basketitem, **kwargs)

- `basketitem` - **BasketItem**: The basket item that you want to remove from the basket.
- `silent` - **Boolean**: defaults to `False`. If `True` the `pre_add_to_basket` & `post_add_to_basket` signals will not be sent.

CHAPTER 4

Managers

CHAPTER 5

Views

5.1 updatebasket

5.2 Supports Ajax Calls

All of the provided views support ajax calls using jsonp. See the ajax section for more information on this.

CHAPTER 6

Forms

The forms provided by basky are really just for convenience . They don't do anything special

6.1 BasketForm

This form will return a form that can be used to POST to the *views#updatebasket*

CHAPTER 7

Signals

basky emits a number of useful signals at certain points when interacting with the basket.

7.1 request_total_for_product

Before a product can be added to the basket we need to be aware of the total price for the item that is being added into the basket. For this reason the `request_total_for_product` will be sent.

Sends arguments of

- `sender` - **Basket**
- `instance` - **the users basket instance**
- `item` - **the item added to the basket**

Can receive a single argument of `total` that will override the total from the item and be used as the total for the basketitem.

7.2 Adding To Basket

Two signals are provided for listening for products being added to the basket:

```
pre_add_to_basket()
post_add_to_basket()
```

They are similar in nature, but they are different. `pre_add_to_basket` will accept return values that can modify the basketitem being added to the basket whereas `post_add_to_basket` does not.

7.2.1 pre_add_to_basket

Sends arguments of

- `sender` - **Basket**
- `instance` - **the users basket instance**
- `item` - **the item added to the basket**
- `quantity` - **quantity of items to be added to the basket**

Can optionally receives a dictionary containing the following keys

- `price` - **Decimal** : will override the price of the item
- `description` - **String**: will override the default basketitem description
- `quantity` - **Integer**: will override the quantity of items added to the basket
- `locked` - **Boolean**: if `True` will prevent the basket line from being editable by the user. Very useful for items added as part of promotions.
- `do_not_add` - **Boolean**: if `True` will prevent the item from being added to the basket.

If `do_not_add` is returned by any of your connected receivers and it has a value of `True` then the item will not be added to basket and the Basket.add() method will return False. This is very useful for performing stock checks or eligibility criteria.

Your receiver functions can return a dictionary that can override the basketitem quantity, description and price. This is done without affecting the actual product instance. This is very useful for running promotions (2 for 1, buy one get one free and any other elaborate promotion you can think of).

Note: Be aware that that if you have multiple receivers modifying product information then it will be the last receiver called that overrides the product information. This shouldn't be an issue, but it is best noted as a possible reason for head scratching.

---

pre_add_to_basket example

Let's assume the following product model:

```
class SimpleProduct(models.Model):
        name = models.CharField(max_length=255)
        price = models.DecimalField(max_digits=8,
                        decimal_places=2)

        def __unicode__(self):
                return u'%s' % self.name

        @property
        def total(self):
                return self.price
```

and from that model we've made a product called "hammer":

```
hammer = SimpleProduct(
                        name="Hammer",
                        price="4.99")
```

and that we're running a promotion of buy one hammer and get one free. This is very simple to achieve with basky. Here's a rather crude receiver function:

```
def example_buy_one_get_one_free(sender, instance, item, quantity, **kwargs):
        # if item is a hammer, insert another hammer
        # with title 'Free Hammer! Buy One Get One Free'
        # and a price of 0.00
        if item.name == "Hammer":
                instance.add(hammer,
                        price="0.00",
                        description="Free Hammer! Buy One Get One Free",
                        locked=True,
                        silent=True)
```

Note: In this example we're passing the `silent=True` argument because we're adding another item to the basket. By passing `silent=True` the `pre_add_to_basket` and `post_add_to_basket` signals will not be sent. If we didn't do this we'd end up reaching a maximum recursion error. And that's just not cricket.

So let's connect the signal and add something into our basket:

```
>>> pre_add_to_basket.connect(example_buy_one_get_one_free, dispatch_uid='bogof')
>>> basket.add(hammer)
```

Now let's ask the basket what's in the basket:

```
>>> basket.basketitem_set.all()
>>> ['<BasketItem: Free Hammer! Buy One Get One Free>', '<BasketItem: Hammer>']
```

And we're only charging our customer for one

```
>>> basket.total_price()
>>> 4.99
```

That's only the tip of the iceburg for the `pre_add_to_basket` and not immediately very useful, but it gives you an idea of where you can go with the `pre_add_to_basket` signal.

7.2.2 post_add_to_basket

Sends arguments of

- `sender` - **Basket**
- `instance` - **the users basket instance**
- `basketitem` - **the item added to the basket**
- `quantity` - **quantity of items to be added to the basket**

post_add_to_basket example

Let's assume the following product model:

```
class SimpleProduct(models.Model):
        name = models.CharField(max_length=255)
        price = models.DecimalField(max_digits=8, decimal_places=2)

        def __unicode__(self):
                return u'%s' % self.name

        @property
        def total(self):
                return self.price
```

and from that model we've saved two products: a hammer and a bag of nails:

```
hammer = SimpleProduct(
                        name="Hammer",
                        price="4.99")
nails = SimpleProduct(
                        name="Bag of 500 Nails",
                        price="1.99")
```

And that we have a special offer of a free bag of nails with every hammer. To do this we're going to attach a rather crude receiver function:

```
def free_nails_with_every_hammer(sender, instance, basketitem, **kwargs):
        # if item being added is a hammer, add some free nails!
        if basketitem.content_object.title == 'Hammer':
                nails = SimpleProduct.objects.get(title="Bag of 500 Nails")
                instance.add(nails,
                        price=Decimal("0.00"),
                        description="Free nails with every hammer",
                        silent=True,
                        locked=True)
```

So let's connect the signal and add something into our basket:

```
>>> pre_add_to_basket.connect(free_nails_with_every_hammer, dispatch_uid='freenails')
>>> basket.add(hammer)
```

We now have two items:

```
>>> basket.basketitem_set.all()
>>> ['<BasketItem: Hammer>', '<BasketItem: Free nails with every hammer>']
```

And the customer is paying for one, the hammer.

```
>>> basket.total_price()
>>> 4.99
```

7.3 Removing From Basket

Two signals are provided for listening for products being removed to the basket:

```
pre_remove_from_basket()
post_remove_from_basket()
```

7.3.1 pre_remove_from_basket

Sends arguments of

- `sender` - **Basket**
- `instance` - **the users basket instance**
- `basketitem` - **the basketitem to be removed from the basket**

Can optionally receives a dictionary containing the following keys

- `do_not_remove` - **Boolean** : if `True` the item will not be removed from the basket.

This signal is sent directly before the item is to be removed from the users basket. If any of the connected receivers return `do_not_remove = True` then the item will not be removed from the basket.

7.3.2 post_remove_from_basket

Sends arguments of

- `sender` - **Basket**

- `instance` - **the users basket instance**
- `basketitem` - **the basketitem to be removed from the basket**

This signal is sent directly after the item is to be removed from the users basket. It cannot modify the removing of the item from the basket in anyway.

post_remove_from_basket example

Let's assume we were selling individually numbered products such as collectable figurines. With our collectable figurines we can only ever sell one of each number. For instance, we have to make sure that when someone adds figurine number 23 to their basket, then nobody else can add number 23 to their basket. To this we might have the following model:

```
class DistastefullCollectableFigurine(models.Model):
        AVAILABLE = 1
        RESERVED = 2
        SOLD = 3
        STATUS_CHOICES = (
                (AVAILABLE, 'Available'),
                (RESERVED, 'Reserved'),
                (SOLD, 'Sold'),
        )
        name = models.CharField(max_length=255)
        number = models.PositiveIntegerField()
        price = models.DecimalField(
                        max_digits=8,
                        decimal_places=2)
        status = models.PositiveIntegerField(
                        default=1,
                        choices=STATUS_CHOICES )

        def __unicode__(self):
                return u'%s' % self.name

        @property
        def total(self):
                return self.price
```

Firstly we're going to need a receiver to listen for the `post_add_to_basket` signal that will change the status of the instance to `DistastefullCollectableFigurine.RESERVED` when it is placed into someones basket. It might look like this:

```
def take_off_sale(sender, instance, **kwargs):
        item = kwargs['basketitem'].content_object
        item.status = item.RESERVED
        item.save()
```

At this point it would be trivial to create a custom manager method on your product model to only return items that had a status of `DistastefullCollectableFigurine.AVILABLE`, but that's outside the scope of this example.

Now we have to make sure that if someone has a `DistastefullCollectableFigurine` in their basket and they decide that they'd rather not buy it then we can use the `post_remove_from_basket` signal to connect a receiver that will change the items status back to `DistastefullCollectableFigurine.AVAILABLE`:

```
def put_back_on_sale(sender, instance, **kwargs):
        item = kwargs['basketitem'].content_object
        item.status = item.AVAILABLE
        item.save()
```

All you'd have to do now is connect them:

```
post_add_to_basket.connect(take_off_sale)
post_remove_from_basket.connect(put_back_on_sale)
```

And without too much hassle you can now handle numbered products with basky.

7.4 Other Signals

7.4.1 basket_is_now_not_empty

Sends arguments of

- `sender` - **Basket**
- `instance` - **the users basket instance**

This signal is sent immediately after the `post_add_to_basket` signal if the users basket had no items in it previous to adding the current item. It cannot modify the contents of the basket in any way.

Note: This signal will not be sent if you've passed silent=True to the add method on the basket

7.4.2 basket_is_now_empty

Sends arguments of

- `sender` - **Basket**
- `instance` - **the users basket instance**

This signal is sent immediately after the `post_remove_from_basket` signal if the users basket had items in it previous to removing the current item. It cannot modify the contents of the basket in any way.

Note: This signal will not be sent if you've passed silent=True to the add method on the basket

CHAPTER 8

Template Tags & Filters

8.1 Filters

8.2 Tags

CHAPTER 9

Context Processors

The context_processor supplied by basky will inject a template variable into your templates, the name of which will be *basket*. If you've set `BASKY_SESSION_KEY_NAME` to customise your installation then the template variable will whatever you've set `BASKY_SESSION_KEY_NAME` to.

For example:

```
<p>
        You have {{basket.total_items}}
        items{{basket.total_items|pluralize}}
        in your basket.
</p>
```

To use the `basky.context_processor.basket` make sure that it's in the list of context_processors in your settings file.:

```
TEMPLATE_CONTEXT_PROCESSORS = (
        #...
        'basky.context_processors.basket'
)
```

CHAPTER 10

Caching

basky makes use of the cache to save on queries here and there. Here's where the queries are cached

CHAPTER 11

Configuration

The following configuration options are available

11.1 BASKY_AGE

Integer.

The timedelta at which baskets will be deleted by the management command.

Default value is `settings.SESSION_COOKIE_AGE`

11.2 BASKY_SESSION_KEY_NAME

String.

Defines the name of the key that will be used to attach the basket object to the session.

Default value is `'basket'`

11.3 BASKY_MAX_ITEMS

Integer.

Defines a maximum amount of items that a basket can hold.

Default value is `9999999`

11.4 BASKY_SINGLE_QUANTITY

Boolean

If set to `True` the basket will only allow a quantity of one on each item and will not add another product to the basket if it already exists.

Default value is `False`

11.5 BASKY_CACHE_PREFIX

String

The prefix for all of the cache keys used in the basket.

Default value is `BASKY`

11.6 BASKY_CACHE_TIMEOUT

Integer

The length of time the basky cache should live.

Default value is 0 - unlimited

CHAPTER 12

Logging

basky logs output to a logger called `basky`

CHAPTER 13

Use Cases

Examples of use cases ....

CHAPTER 14

Background to basky

CHAPTER 15

Feedback

We welcome all feedback, you can find C&C Design Consultants LTD on twitter or you can drop by our website.

15.1 Reporting Bugs

Bugs can be reported on the basky issue page, just be sure to label your issue as bug.

15.2 Feature Requests

Feature requests can be submitted on the basky issue page, just be sure to label your issue as a feature request.

CHAPTER 16

Indices and tables

- *genindex*
- *modindex*
- *search*