satchless Documentation

Release 1.0.4

Mirumee Software

May 09, 2017

Contents

A pythonic way to deal with lower levels of e-commerce.

CHAPTER 1

Overview

Satchless is a low-level library that provides base classes, interfaces and patterns to work with while implementing a store. Its goal is to provide you with well-tested core logic to build business logic and UIs upon.

Satchless is **not a framework** and it will not give you a full-fledged store to work with. If that is what you want and you're working with Django, check the Saleor project instead.

Modules provided by `satchless`

`satchless.item` — abstract priceables

Available Types

All of the following types are abstract and meant to be subclassed to implement their missing methods.

Note: Implementations provided by Satchless expect to work with price objects as implemented by the prices library.

class `satchless.item.`**`Item`**
A priceable item. Usually a single product or a single product variant.

class `satchless.item.`**`ItemRange`**
A range of priceables that vary in price. An example is a product with multiple variants.

class `satchless.item.`**`ItemLine`**
A certain amount of an *`Item`*. Contains a priceable and its quantity. An example is a cart line.

class `satchless.item.`**`ItemSet`**
A collection of *`ItemLine`* objects that have a total price. Examples include a bundle, an order or a shopping cart.

class `satchless.item.`**`ItemList`**
An *`ItemSet`* that subclasses the Python `list`.

class satchless.item.**Partitioner**
An object capable of partitioning an *ItemSet* into multiple sets for purposes such as split delivery.

class satchless.item.**ClassifyingPartitioner**
A *Partitioner* that automatically splits based on a classifying function.

class satchless.item.**StockedItem**
A stocked *Item*. Introduces the concept of stock quantities.

class satchless.item.**InsufficientStock**
Exception class that is raised by *StockedItem* when trying to exceed the stock quantity.

Available functions

satchless.item.**partition**(*subject*, *keyfunc*[, *partition_class=ItemList*])
Returns a *Partitioner* objects that splits *subject* based on the result of *keyfunc(item)*.

Item Objects

class satchless.item.**Item**
An *Item* instance represents a priceable item.

Instance methods:

Item.**get_price**(**kwargs*)
Returns a prices.Price object representing the price of the priceable.

The default implementation passes all keyword arguments to *get_price_per_item()*. Override to implement discounts and such.

For subclassing:

Item.**get_price_per_item**(**kwargs*)
Returns a prices.Price object representing the price for a single piece of the priceable.

The default implementation will raise a NotImplementedError exception.

Example use:

```
>>> import prices
>>> from satchless.item import Item
>>> class Coconut(Item):
...     def get_price_per_item(self): return prices.Price(10, currency='USD')
...
>>> coconut = Coconut()
>>> coconut.get_price()
Price('10', currency='USD')
```

ItemRange Objects

class satchless.item.**ItemRange**
An *ItemRange* instance represents a range of priceables.

Instance methods:

ItemRange.**__iter__**()
Returns an iterator yielding priceable objects that implement a get_price() method.

The default implementation will raise a NotImplementedError exception.

ItemRange.**get_price_range**(***kwargs*)
Returns a prices.PriceRange object representing the price range of the priceables included in the range object. Keyword arguments are passed to *get_price_per_item()*.

Calling this method on an empty range will raise an AttributeError exception.

For subclassing:

ItemRange.**get_price_per_item**(*item*, ***kwargs*)
Return a prices.Price object representing the price of a given item.

The default implementation will pass all keyword arguments to item.get_price(). Override to implement discounts or caching.

Example use:

```
>>> import prices
>>> from satchless.item import Item, ItemRange
>>> class SpanishInquisition(Item):
...     def get_price_per_item(self): return prices.Price(50, currency='BTC')
...
>>> class LaVache(Item):
...     def get_price_per_item(self): return prices.Price(15, currency='BTC')
...
>>> class ThingsNobodyExpects(ItemRange):
...     def __iter__(self):
...         yield SpanishInquisition()
...         yield LaVache()
...
>>> tne = ThingsNobodyExpects()
>>> tne.get_price_range()
PriceRange(Price('15', currency='BTC'), Price('50', currency='BTC'))
```

ItemLine Objects

class satchless.item.**ItemLine**
An *ItemLine* instance represents a certain quantity of a particular priceable.

Instance methods:

ItemLine.**get_total**(***kwargs*)
Return a prices.Price object representing the total price of the line. Keyword arguments are passed to both *get_quantity()* and *get_price_per_item()*.

For subclassing:

ItemLine.**get_quantity**(***kwargs*)
Returns an int or a decimal.Decimal representing the quantity of the item.

The default implementation will ignore all keyword arguments and always return 1.

ItemLine.**get_price_per_item**(***kwargs*)
Returns a prices.Price object representing the price of a single piece of the item.

The default implementation will raise a NotImplementedError exception.

Example use:

```
>>> import prices
>>> from satchless.item import ItemLine
>>> class Shrubberies(ItemLine):
```

```
...     def __init__(self, qty): self.qty = qty
...     def get_quantity(self): return self.qty
...     def get_price_per_item(self): return prices.Price(11, currency='GBP')
...
>>> shrubberies = Shrubberies(7)
>>> shrubberies.get_total()
Price('77', currency='GBP')
```

`ItemSet` Objects

class `satchless.item.`**`ItemSet`**

An *`ItemSet`* instance represents a set of *`ItemLine`* or other *`ItemSet`* objects that has a total price.

Instance methods:

`ItemSet.`**`__iter__`**`()`

Returns an iterator yielding objects that implement a `get_total()` method. Good candidates include instances of *`ItemLine`* and *`ItemSet`* itself.

The default implementation will raise a `NotImplementedError` exception.

`ItemSet.`**`get_total`**`(`*`**kwargs`*`)`

Return a `prices.Price` object representing the total price of the set. Keyword arguments are passed to *`get_subtotal()`*.

Calling this method on an empty set will raise an `AttributeError` exception.

For subclassing:

`ItemSet.`**`get_subtotal`**`(`*`item`*`, `*`**kwargs`*`)`

Returns a `prices.Price` object representing the total price of `item`.

The default implementation will pass keyword arguments to `item.get_total()`. Override to implement discounts or caching.

Example use:

```
>>> import prices
>>> from satchless.item import Item, ItemLine, ItemSet
>>> class Product(Item):
...     def get_price_per_item(self): return prices.Price(10, currency='EUR')
...
>>> class CartLine(ItemLine):
...     def __init__(self, product, qty): self.product, self.qty = product, qty
...     def get_price_per_item(self): return self.product.get_price()
...     def get_quantity(self): return self.qty
...
>>> class Cart(ItemSet):
...     def __iter__(self):
...         yield CartLine(Product(), 5)
...
>>> cart = Cart()
>>> cart.get_total()
Price('50', currency='EUR')
```

`Partitioner` Objects

class `satchless.item.`**`Partitioner`**(*subject*)

A *`Partitioner`* instance is an iterable view of the `subject` that partitions it for purposes such as split delivery.

Instance methods:

`Partitioner.`**`__iter__`**`()`

Returns an iterator that yields *`ItemSet`* objects representing partitions of `self.subject`.

The default implementation will yield a single `ItemList` containing all the elements of `self.subject`. Override to implement your partitioning scheme.

Example use:

```
>>> from satchless.item import ItemList, Partitioner
>>> class EvenOddSplitter(Partitioner):
...     def __iter__(self):
...         yield ItemList(it for n, it in enumerate(self.subject) if not n % 2)
...         yield ItemList(it for n, it in enumerate(self.subject) if n % 2)
...
>>> splitter = EvenOddSplitter(['a', 'b', 'c', 'd', 'e', 'f'])
>>> list(splitter)
[ItemList(['a', 'c', 'e']), ItemList(['b', 'd', 'f'])]
```

A more advanced example could split an imaginary cart object into groups of objects that can be delivered together:

```
from satchless.item import ItemList, Partitioner

class DeliveryPartitioner(Partitioner):

    def __iter__(self):
        """
        Yield single-product groups for products that need to be shipped
        separately. Yield a separate group for digital products if present.
        Everything else can be shipped together.
        """
        digital = []
        remaining = []
        for it in self.subject:
            if it.ship_separately:
                yield ItemList([it])
            elif it.is_digital:
                digital.append(it)
            else:
                remaining.append(it)
        if digital:
            yield ItemList(digital)
        if remaining:
            yield ItemList(remaining)
```

`ClassifyingPartitioner` Objects

class `satchless.item.`**`ClassifyingPartitioner`**(*subject*)

A *`Partitioner`* subclass that splits the subject into groups based on the result of the classifying method.

Instance methods:

ClassifyingPartitioner.**classify**(*item*)
Returns a classification key that groups together items that are meant for the same group.

The default implementation will raise a NotImplementedError exception.

Example use:

```
>>> from satchless.item import ItemList, ClassifyingPartitioner
>>> class ClassNameSplitter(ClassifyingPartitioner):
...     def classify(self, item):
...         return type(item).__name__
...
>>> splitter = ClassNameSplitter(['a', 'b', 1, ['one'], 2, ['two']])
>>> list(splitter)
[ItemList([1, 2]), ItemList([['one'], ['two']]), ItemList(['a', 'b'])]
```

StockedItem Objects

class satchless.item.**StockedItem**
A *StockedItem* object is subclass of *Item* that allows you to track stock quantities and guard against excess allocation.

Instance methods:

ItemSet.**get_stock**()
Returns the current stock quantity of the item.

The default implementation will raise a NotImplementedError exception.

StockedItem.**check_quantity**(*quantity*)
Makes sure that at least *quantity* of the object are in stock by comparing the value with the result of *self.get_stock()*. If there is not enough, an *InsufficientStock* exception will be raised.

Example use:

```
>>> from satchless.item import InsufficientStock, StockedItem
>>> class LimitedShrubbery(StockedItem):
...     def get_stock(self):
...         return 1
...
>>> shrubbery = LimitedShrubbery()
>>> try:
...     shrubbery.check_quantity(2)
... except InsufficientStock as e:
...     print('only %d remaining!' % (e.item.get_stock(),))
...
only 1 remaining!
```

InsufficientStock Exception

class satchless.item.**InsufficientStock**(*item*)
Informs you that a stock quantity check failed against *item*. Raised by *StockedItem.check_quantity()*.

satchless.cart — shopping carts

Available Types

All of the following types are abstract and meant to be subclassed to implement their missing methods.

Note: Implementations provided by Satchless expect to work with price objects as implemented by the prices library.

class satchless.cart.**CartLine**
A single line in a shopping cart. Describes a product, its quantity and (optinal) data. Attributes: `product`, `quantity`, `data`.

Suitable for pickling.

class satchless.cart.**Cart**
A shopping cart. Contains a number of *CartLine* objects.

Suitable for pickling.

`CartLine` Objects

class satchless.cart.**CartLine**(*product*, *quantity*, *data=None*)
A *CartLine* object represents a single line in a shopping cart. It subclasses the *ItemLine*.

You should not have to create instances of this class manually. Use the *Cart.add()* method and let it construct lines as needed instead.

Instance methods:

CartLine.**get_total**(***kwargs*)
See *ItemLine*.

For subclassing:

CartLine.**get_price_per_item**(***kwargs*)
See *ItemLine*.

The default implementation passes all keyword arguments to `self.product.get_price()`. Override to implement discounts or caching.

`Cart` Objects

class satchless.cart.**Cart**(*items=None*)
A *Cart* object represents a shopping cart. It subclasses the *ItemSet*.

Instance attributes:

satchless.cart.**modified**
`True` if the object was modified since it was created/deserialized. `False` otherwise.

Useful if you need to persist the cart.

Instance methods:

Cart.**__iter__**()
Returns an iterator that yields *CartLine* objects contained in the cart.

See *ItemSet*.

`Cart.add`(*product*, *quantity=1*, *data=None*, *replace=False*)
If `replace` is `False`, increases quantity of the given product by `quantity`. If given product is not in the cart yet, a new line is created.

If `replace` is `True`, quantity of the given product is set to `quantity`. If given product is not in the cart yet, a new line is created.

If the resulting quantity of a product is zero, its line is removed from the cart.

Products are considered identical if both `product` and `data` are equal. This allows you to customize two copies of the same product (eg. choose different toppings) and track their quantities independently.

`Cart.get_total`(***kwargs*)
Return a `prices.Price` object representing the total price of the cart.

See *ItemSet*.

For subclassing:

`Cart.check_quantity`(*product*, *quantity*, *data*)
Checks if given quantity is valid for the product and its data.

Default implementation will call *product.check_quantity(quantity)* if such a method exists. This is useful when working with *satchless.item.StockedItem* objects.

`Cart.create_line`(*product*, *quantity*, *data*)
Creates a *CartLine* given a product, its quantity and data. Override to use a custom line class.

Example use:

```
>>> import prices
>>> from satchless.item import Item
>>> from satchless.cart import Cart
>>> class Taco(Item):
...     def __repr__(self): return 'Taco()'
...     def get_price_per_item(self): return prices.Price(5, currency='CHF')
...
>>> cart = Cart()
>>> veggie_taco = Taco()
>>> cart.add(veggie_taco, quantity=3, data=['extra cheese'])
>>> cart.add(veggie_taco, data=['very small rocks'])
>>> cart.add(veggie_taco, data=['very small rocks'])
>>> list(cart)
[CartLine(product=Taco(), quantity=3, data=['extra cheese']),
 CartLine(product=Taco(), quantity=2, data=['very small rocks'])]
>>> cart.get_total()
Price('25', currency='CHF')
```

`satchless.process` — multi-step processes

The purpose of this module is to aid with creation of multi-step processes such as online store checkout. We solve this by providing a step class and a process class that are typically bound to data that is persisted between the steps. Each time a single step is completed you can ask the manager which step needs to be completed next until the whole process is considered complete.

We do not define what a step is. It could be a Django view or a call to an external API. It can be anything as long as it can determine its own state (usually passed to its constructor by the process manager) and decide whether it's valid or not.

Available Types

exception satchless.process.**InvalidData**
An exception raised when a step is invalid.

class satchless.process.**Step**
A single step in a multi-step process.

class satchless.process.**ProcessManager**
A multi-step process manager.

Step Objects

class satchless.process.**Step**
A *Step* instance is a single step in a process. For example a single checkout step.

Instance methods:

satchless.process.**__str__**()
Returns the step ID. The same step must always return the same ID and the ID has to be unique within the process.

Default implementation will raise an NotImplementedError exception.

satchless.process.**validate**()
Validates the step and raises *InvalidData* if the step requires further attention.

Default implementation will raise an NotImplementedError exception.

Example use:

```
>>> from satchless.process import InvalidData, Step
>>> class LicenseAcceptance(Step):
...     def __init__(self, data): self.data = data
...     def __str__(self): return 'license'
...     def validate(self):
...         if not self.data.get('license_accepted'):
...             raise InvalidData('Nice try')
...
>>> data = {}
>>> step = LicenseAcceptance(data)
>>> str(step)
'license'
>>> step.validate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in validate
satchless.process.InvalidData: Nice try
>>> data['license_accepted'] = True
>>> step.validate()
```

ProcessManager Objects

class satchless.process.**ProcessManager**
A *ProcessManager* instance represents a single process. For example the checkout process in a typical online store.

Instance methods:

ProcessManager.**__iter__**()
Returns an iterator that yields *Step* objects that define the process.

Default implementation will raise an NotImplementedError exception.

ProcessManager.**__getitem__**(*step_id*)
Returns the *Step* object whose Step.__str__() method returns step_id.

If no such step is found, KeyError will be raised.

ProcessManager.**is_complete**()
Returns *True* if all the steps of the process are valid. *False* otherwise.

ProcessManager.**get_next_step**()
Returns the first step that does not validate.

If all steps are valid, None is returned.

ProcessManager.**get_errors**()
Returns a dict whose keys are IDs of steps that did not validate and the values are the exceptions raised by the steps.

Example use:

```
>>> from satchless.process import InvalidData, ProcessManager, Step
>>> class LicenseAcceptance(Step):
...     def __init__(self, data): self.data = data
...     def __str__(self): return 'license'
...     def __repr__(self): return 'LicenseAcceptance(%r)' % (self.data, )
...     def validate(self):
...         if not self.data.get('license_accepted'):
...             raise InvalidData('Nice try')
...
>>> class SingleStepProcessManager(ProcessManager):
...     def __init__(self, data): self.data = data
...     def __iter__(self):
...         yield LicenseAcceptance(self.data)
...
>>> data = {}
>>> process = SingleStepProcessManager(data)
>>> list(process)
[LicenseAcceptance({})]
>>> process['license']
LicenseAcceptance({})
>>> process.is_complete()
False
>>> process.get_next_step()
LicenseAcceptance({})
>>> process.get_errors()
{'license': InvalidData('Nice try',)}
>>> data['license_accepted'] = True
>>> process.is_complete()
True
>>> process.get_next_step()
>>> process.get_errors()
{}
```

Python Module Index

S

Index

Symbols

A

C

G

I

M

P

S

V