Lorikeet Documentation

Release 0.0.1

Adam Brenecki

Sep 17, 2017

Getting Started

Note: Lorikeet is a work in progress. It may change at any time, and you shouldn't use it in production yet.

Lorikeet is a simple, generic, API-only shopping cart framework for Django.

Lorikeet currently supports Django 1.8 to 1.10 on Python 3.4+. New versions of Lorikeet will support Django versions that currently have extended support.

CHAPTER 1

Why use Lorikeet?

E-commerce apps are divided into two types: simple ones that work well so long as what you're selling is simple, and complex ones that try to be all things to all people by way of a maze of checkboxes and dropdowns.

Lorikeet isn't an e-commerce app; it's a shopping cart framework. With Lorikeet, you define models for line items (the things that go in your cart), delivery addresses and payment methods yourself. For complex shops, this means you can model exactly the functionality you need without fighting the system. For simple shops, this is a simple process that requires way less code than you'd expect, and gives you a system without unnecessary bloat, but with room to grow.

Lorikeet only cares about the cart itself; everything outside of that, including your navigation and product pages, is directly under your control, so you're free to use a simple `ListView` and `DetailView`, Wagtail, Mezzanine, Django CMS, or something totally bespoke. There's not a single line of HTML or CSS in Lorikeet's codebase either, so Lorikeet gives you total control over your visuals too.

Lorikeet line items, delivery addresses and payment methods are designed to be orthogonal, so you can package them as reusable apps and share them internally between sites in your company, or with the world as open-source packages. In fact, **Lorikeet already includes an optional Stripe payment method plugin**, totally separate from the rest of the codebase and written against the same public API as your own apps.

Because most modern payment providers require JavaScript anyway, **Lorikeet is API-only**. This lets you build a fast, frictionless shopping experience where users can add to and change their shopping carts without the entire page refreshing each time, and Lorikeet's API is designed to allow logged-in repeat users to check out in a single click.

CHAPTER 2

Why use something else?

- **Lorikeet isn't turnkey.** For simple sites, you won't need to write much Python code; for complex ones, the time it takes to get up and running will probably be comparable to the time it takes to figure out how to bend e-commerce apps to your will. But the total control over the frontend that Lorikeet gives you means you'll need to write a fair bit of HTML, CSS and JavaScript to get up and running, so if you need to go from zero to shop quickly, it's best to look somewhere else.
- **Lorikeet sites will require JavaScript.** Lorikeet doesn't provide regular HTML-form-based views for adding items to the cart and checking out; if you need this, Lorikeet isn't for you.

Installation

This tutorial assumes you have an existing Django project set up. If you don't, you can create one with startproject.

1. Install Lorikeet, by running `pip install https://gitlab.com/abre/lorikeet.git`.
2. Add `'lorikeet'` to `INSTALLED_APPS`.
3. Add `'lorikeet.middleware.CartMiddleware'` to `MIDDLEWARE_CLASSES`.
4. Add a line that looks like `url(r'^_cart/', include('lorikeet.urls', namespace='lorikeet')),` to `urls.py`. (You don't have to use `_cart` in your URL—anything will do.)

You're all set! If you run `python manage.py runserver` and visit `http://localhost:8000/_cart/`, you should see a JSON blob with a few properties. Now you're ready to start building your backend!

Building The Backend

Before You Start

This guide assumes you already have a Django-based website where your users can browse around whatever it is you're selling. If you haven't, go ahead and build one, we'll be here when you get back!

For the examples in this guide, we'll assume the products in your store are modelled using the following model.

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    unit_price = models.DecimalField(max_digits=7, decimal_places=2)
```

Note: Lorikeet doesn't require that you create a `Product` model. You can model your products however you like; this is just how we've chosen to do it in this tutorial.

Line Items

In Lorikeet, a shopping cart is made up of line items; subclasses of *lorikeet.models.LineItem*, with a `get_total` method that returns how much they cost. Here's a simple one:

```
from django.db import models
from lorikeet.models import LineItem

class MyLineItem(LineItem):
    product = models.ForeignKey(Product, on_delete=models.PROTECT)
    quantity = models.PositiveSmallIntegerField()

    def get_total(self):
        return self.quantity * self.product.unit_price
```

Every line item type needs a serializer, so that your frontend can create new LineItems and add things to your users' carts. You should subclass these from *lorikeet.api_serializers.LineItemSerializer*, but otherwise write them as you would a normal Django REST Framework serializer.

```
from rest_framework import fields
from lorikeet.api_serializers import (LineItemSerializer,
                                      PrimaryKeyModelSerializer)

from . import models


class ProductSerializer(PrimaryKeyModelSerializer):
    class Meta:
        model = models.Product
        fields = ('id', 'name', 'unit_price')


class MyLineItemSerializer(LineItemSerializer):
    product = ProductSerializer()
    class Meta:
        model = models.MyLineItem
        fields = ('product', 'quantity',)
```

Note: We've also made a simple serializer for our `Product` class. Notice that we've subclassed *lorikeet.api_serializers.PrimaryKeyModelSerializer*; we'll talk about what this serializer class does when we get to the frontend.

The last thing we need to do is link the two together when Django starts up. The easiest place to do this is in the *ready* method of your app's *AppConfig*:

```
from django.apps import AppConfig

class MyAppConfig(AppConfig):
    # ...

    def ready(self):
        from . import models, api_serializers
        from lorikeet.api_serializers import registry

        registry.register(models.MyLineItem,
                          api_serializers.MyLineItemSerializer)
```

Warning: If you're newly setting up an app config for use with Lorikeet, make sure Django actually loads it!

You can do this by either changing your app's entry in *INSTALLED_APPS* to the dotted path to your AppConfig (e.g. `myapp.apps.MyAppConfig`), or by adding a line like `default_app_config = "myapp.apps.MyAppConfig"` in your app's `__init__.py`.

For more on app configs, check out the Django documentation.

Delivery Addresses

Now that Lorikeet knows about the things you're selling, it needs to know where you plan to send them after they've been sold, whether that's a postal address, an email, or something totally different.

Note: There are plans to eventually add an optional pre-built postal addressing plugin, which will mean you'll be able to skip this section in the future if you're delivering to postal addresses.

Just like with line items, we need a model subclassing *`lorikeet.models.DeliveryAddress`*, a serializer, and a `registry.register` call to connect the two. Delivery addresses are even eaiser, though; there's no special methods you need to define.

```
class AustralianDeliveryAddress(DeliveryAddress):
    addressee = models.CharField(max_length=255)
    address = models.TextField()
    suburb = models.CharField(max_length=255)
    state = models.CharField(max_length=3, choices=AUSTRALIAN_STATES)
    postcode = models.CharField(max_length=4)
```

```
class AustralianDeliveryAddressSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.AustralianDeliveryAddress
        fields = ('addressee', 'address', 'suburb', 'state', 'postcode')
```

```
registry.register(models.AustralianDeliveryAddress,
                  api_serializers.AustralianDeliveryAddressSerializer)
```

Payment Methods

Now Lorikeet knows what we're buying, and where it's going, but it needs to be able to collect payment. By now, you probably won't be surprised to find that you need to provide a model subclassing *lorikeet.models.PaymentMethod*, a serializer, and link the two with `registry.register`.

Tip: If you're planning to accept payments via Stripe, you can skip this section; Lorikeet comes built-in with an optional Stripe payment method. See the section on stripe.

For this example, we'll use the fictional payment provider Pipe, which just so happens to have a similar API to Stripe, although slightly simplified.

```
class PipeCard(PaymentMethod):
    card_id = models.CharField(max_length=30)
```

With most payment providers, the data you want to send to the server on creation is totally different to the data you want to receive when viewing the payment method. Usually, you have some sort of opaque token returned by a JavaScript library, which you want to pass to your payment provider and store the result; when you read it back you want to know that it's a Visa that ends in 4242.

We've accomplished that by using a `write_only` field and a pair of `SerializerMethodField` instances (which defualt to read-only), and a `create()` method to communicate with the payment provider.

```
class PipeCardSerializer(serializers.ModelSerializer):
    card_token = fields.CharField(max_length=30, write_only=True)
    brand = fields.SerializerMethodField()
    last4 = fields.SerializerMethodField()

    class Meta:
        model = models.PipeCard
        fields = ('card_token', 'brand', 'last4')

    def get_brand(self, object):
        return pipe.get_card(object.card_id)['brand']

    def get_last4(self, object):
        return pipe.get_card(object.card_id)['last4']

    def create(self, validated_data):
        card_token = validated_data.pop('card_token')
        validated_data['card_id'] = pipe.create_card(card_token)['id']
        return super().create(validated_data)
```

Now that we can create payment methods, let's look at what happens when we charge them. We'll need a model to store details about a charge, which should be a subclass of `lorikeet.models.Payment`.

```
class PipePayment(Payment):
    payment_id = models.CharField(max_length=30)
```

Then, we'll need to add a `make_payment` method to our `PipeCard` class. This should either return an instance of our `PipePayment` class, or raise *lorikeet.exceptions.PaymentError* if the payment is unsuccessful.

Note: The `Payment` model has a mandatory `method` field, which you'll need to fill with `self` when you create new instances.

```
class PipeCard(PaymentMethod):
    card_id = models.CharField(max_length=30)

    def make_payment(self, order, amount):
        try:
            payment_id = pipe.charge_card(self.card_id, amount)
        except pipe.ChargeError as e:
            raise PaymentError(e.user_info)
        else:
            return PipePayment.objects.create(method=self, payment_id=payment_id)
```

Building The Frontend

Todo

Write this part of the tutorial :S

Building the Post-Checkout Experience

Our users can make purchases and check out! Now, though, we need to make sure they can view their existing orders.

This is just a matter of setting up a regular Django `ListView` and `DetailView`, both with Lorikeet's `OrderMixin`.

```
from django.views.generic import DetailView, ListView
from lorikeet.mixins import OrderMixin

class OrderListView(OrderMixin, ListView):
    template_name = "products/order_list.html"

class OrderDetailView(OrderMixin, DetailView):
    template_name = "products/order.html"
```

We'll make sure the views have URLs.

```
from django.conf.urls import url
from . import views

urlpatterns = [
    # ...
    url(r'^orders/$', views.OrderListView.as_view(), name='order-list'),
    url(r'^orders/(?P<pk>\d+)/$', views.OrderDetailView.as_view(), name='order'),
]
```

All we need to do now is write up templates, consulting the documentation for *`Order`* to find out what we can access, the same as a standard `DetailView`. (The `ListView` template is left as an exercise for the reader, but is just as straightforward. You'll also want to make sure that your delivery address and payment method models have `__str__` methods.)

```
<h1>Order {{ object.invoice_id }}</h1>
```

```
<h2>Shipped To</h2>
{{ object.delivery_address_subclass }}

<h2>Paid With</h2>
{{ object.payment_method_subclass }}

<table>
    <tr>
        <th>Product</th>
        <th>Quantity</th>
        <th>Subtotal</th>
    </tr>
    {% for item in object.items.select_subclasses %}
        <tr>
            <td>{{ item.product.name }}</td>
            <td>{{ item.quantity }}</td>
            <td>{{ item.get_total }}</td>
        </tr>
    {% endfor %}
    <tr>
</table>
```

Todo

emails, tokenised URLs

Note: The tokenised URLs generated by Lorikeet don't expire. So, if you include them in your order emails, be sure that your order detail pages don't contain any information that's not safe to include in an email to the user.

Line Items

In Lorikeet, a user's shopping cart is made up of one or more line items, each of which models a particular thing in the cart. Each different kind of line item needs to have two things: a model to define what can be stored in it, and a serializer so that your frontend can do the actual storing.

Building a Line Item Model

You might remember from the *backend tutorial* that line items are subclasses of *lorikeet.models.LineItem*, with a `get_total` method that returns how much they cost. That's really all there is to it! Here's the model we made in the tutorial:

```
from django.db import models
from lorikeet.models import LineItem

class MyLineItem(LineItem):
    product = models.ForeignKey(Product, on_delete=models.PROTECT)
    quantity = models.PositiveSmallIntegerField()

    def get_total(self):
        return self.quantity * self.product.unit_price
```

It's worth reiterating that the only two things Lorikeet cares about are the fact that it's a *lorikeet.models.LineItem* subclass, and the fact that it defines a `get_total` method. All these other things:

- The details of what's in your `Product` model,
- Whether you have a single `Product` model, two or more different models (`TShirt` and `Mug`, maybe?), or no product model at all,
- What fields are on your `LineItem` subclass, or what their types are (for instance, if you're selling T-shirts you might need fields for colour and size, or if you're selling goods by the kilogram you might make `quantity` a `DecimalField`),

Lorikeet doesn't care about those, and you can structure them how you like.

Lorikeet also **isn't limited to one type of line item**. If you sell multiple different kinds of products, like in the `TShirt` and `Mug` example before, you might need to store different kinds of data on their respective line items; mugs don't come in different sizes and cuts, after all. Lorikeet will let you define a `TShirtLineItem` and a `MugLineItem`, and your users can add a combination of both into their cart.

Building a Line Item Serializer

You might also remember from the *backend tutorial* that every line item serializer is a subclass of *lorikeet.api_serializers.LineItemSerializer*. Lorikeet will use this serializer both to populate new line items, and to render existing ones into JSON.

```
from rest_framework import fields
from lorikeet.api_serializers import (LineItemSerializer,
                                      PrimaryKeyModelSerializer)

from . import models


class ProductSerializer(PrimaryKeyModelSerializer):
    class Meta:
        model = models.Product
        fields = ('id', 'name', 'unit_price')


class MyLineItemSerializer(LineItemSerializer):
    product = ProductSerializer()
    class Meta:
        model = models.MyLineItem
        fields = ('product', 'quantity',)
```

Tip: If you have any application logic you need to run when you add an item to the cart, you can do it inside the `create()` method on the line item's serializer.

Linking it all together

Once you've written your model and serializer, link them together in *The Registry*.

The Registry

Lorikeet's *Line Items*, Delivery Addresses and Payment Methods are all made up of two components: a model and a serializer. In order to use them, Lorikeet needs to know about each serializer that's available, and which model each one corresponds to.

There's various ways that Lorikeet could associate the two automatically, but they're all failure-prone and difficult to debug. Instead, Lorikeet exposes a registry at *lorikeet.api_serializers.registry* for you to declare those mappings manually.

Mappings can be declared by calling *lorikeet.api_serializers.registry.register()*. You can do this anywhere as long as it gets run when Django starts up, but the best place to do it is the `ready()` method of an `AppConfig` for your app:

```
from django.apps import AppConfig

class MyAppConfig(AppConfig):
    # ...

    def ready(self):
        from . import models, api_serializers
        from lorikeet.api_serializers import registry

        registry.register(models.MyLineItem,
                          api_serializers.MyLineItemSerializer)
        registry.register(models.MyDeliveryAddress,
                          api_serializers.MyDeliveryAddressSerializer)
        registry.register(models.MyPaymentMethod,
                          api_serializers.MyPaymentMethodSerializer)
```

Warning: If you're newly setting up an app config for use with Lorikeet, make sure Django actually loads it!

You can do this by either changing your app's entry in *INSTALLED_APPS* to the dotted path to your AppConfig (e.g. `myapp.apps.MyAppConfig`), or by adding a line like `default_app_config = "myapp.apps.MyAppConfig"` in your app's `__init__.py`.

For more on app configs, check out the Django documentation.

Cart Checkers

Cart checkers are functions that determine whether or not a cart can be checked out in its current state. If a cart is ready to be checked out, and all cart checkers pass, it is said to be **complete**.

If you think of the entire cart as being like a form, cart checkers are like validators. (We don't actually call them that, because Django REST Framework validators perform a separate function within Lorikeet; ensuring that individual instances of *LineItem*, *DeliveryAddress* and so on are valid.)

Cart checkers are run in two places. One is in the *GET /_cart/* endpoint, where any checkers that fail are listed in `incomplete_reasons`, so the client user interface can show details. The other is in the *POST /_cart/checkout/* endpoint, where any checkers that fail will prevent checkout from happening, resulting in a **422** response with a reason of `"incomplete"`.

Writing a Cart Checker

Cart checkers are functions that accept a *Cart* instance as an argument; they should either raise a *IncompleteCartError* if the cart is not ready to be checked out, or return successfully if it is.

Here's one that's built in to Lorikeet:

```
def payment_method_required(cart):
    """Checks that a payment method is set on the cart."""

    if cart.payment_method is None:
        raise IncompleteCartError(code='not_set',
                                  message='A payment method is required.',
                                  field='payment_method')
```

If your cart checker identifies multiple different reasons the cart can't be checked out, it should instead raise a *IncompleteCartErrorSet*, which can be passed a list of *IncompleteCartError* instances.

Once you've written your cart checker, add it to the *LORIKEET_CART_COMPLETE_CHECKERS* setting.

Warning: The default value for *LORIKEET_CART_COMPLETE_CHECKERS* contains important built-in checkers that you probably don't want to disable, because they prevent things like going through checkout with an empty cart. If you override this setting, make sure you include them!

Built-in Cart Checkers

The built-in cart checkers are documented in the *Cart Checkers* section of the API documentation.

Handling an Incomplete Cart on the Client

Todo

Document the API side of things

Python API

Models

class lorikeet.models.**Cart**(**args*, ***kwargs*)

An in-progress shopping cart.

Carts are associated with the user for an authenticated request, or with the session otherwise; in either case it can be accessed on `request.cart`.

delivery_address_subclass

Get the delivery address instance selected for this cart.

Returns an instance of one of the registered *DeliveryAddress* subclasses.

get_grand_total()

Calculate the grand total for this cart.

is_complete(*raise_exc=False*, *for_checkout=False*)
Determine if this cart is able to be checked out.

If this function returns `False`, the `.errors` attribute will be set to a *IncompleteCartErrorSet* containing all of the reasons the cart cannot be checked out.

Parameters **raise_exc** (*bool*) – If `True` and there are errors, raise the resulting *IncompleteCartErrorSet* instead of just returning `False`.

Returns Whether this cart can be checked out.

Return type bool

payment_method_subclass
Get the payment method instance selected for this cart.

Returns an instance of one of the registered *PaymentMethod* subclasses.

class lorikeet.models.**LineItem**(**args*, ***kwargs*)
An individual item that is either in a shopping cart or on an order.

This model doesn't do anything by itself; you'll need to subclass it as described in the Getting Started Guide.

check_complete(*for_checkout=False*)
Checks that this line item is ready to be checked out.

This method should raise *IncompleteCartError* if the line item is not ready to be checked out (e.g. there is insufficient stock in inventory to fulfil this line item). By default it does nothing.

Parameters **for_checkout** (*bool*) – Set to `True` when the cart is about to be checked out. See the documentation for `prepare_for_checkout()` for more details. is going to be called within the current transaction, so you should use things like select_for_update.

get_total()
Returns the total amount to charge on this LineItem.

By default this raises `NotImplemented`; subclasses of this class need to override this.

If you want to know the total for this line item from your own code, use the *total()* property rather than calling this function.

prepare_for_checkout()
Prepare this line item for checkout.

This is called in the checkout process, shortly before the payment method is charged, within a database transaction that will be rolled back if payment is unsuccessful.

This function shouldn't fail. (If it does, the transaction will be rolled back and the payment won't be processed so nothing disastrous will happen, but the user will get a 500 error which you probably don't want.)

The `check_complete()` method is guaranteed to be called shortly before this method, within the same transaction, and with the `for_checkout` parameter set to `True`. Any checks you need to perform to ensure checkout will succeed should be performed there, and when `for_checkout` is true there you should ensure that those checks remain valid for the remainder of the database transaction (e.g. using select_for_update).

total
The total cost for this line item.

Returns the total actually charged to the customer if this item is attached to an *Order*, or calls *get_total()* otherwise.

class lorikeet.models.**DeliveryAddress**(**args*, ***kwargs*)
An address that an order can be delivered to.

This model doesn't do anything by itself; you'll need to subclass it as described in the Getting Started Guide.

class lorikeet.models.**PaymentMethod**(**args*, ***kwargs*)
A payment method, like a credit card or bank details.

This model doesn't do anything by itself; you'll need to subclass it as described in the Getting Started Guide.

class lorikeet.models.**Order**(**args*, ***kwargs*)
A completed, paid order.

delivery_address_subclass
Get the delivery address instance selected for this cart.

Returns an instance of one of the registered *DeliveryAddress* subclasses.

get_absolute_url(*token=False*)
Get the absolute URL of an order details view.

Parameters **token** (*bool*) – If true, include in the URL a token that allows unauthenticated access to the detail view.

See the documentation for the LORIKEET_ORDER_DETAIL_VIEW setting.

invoice_id
The ID of the invoice.

If custom_invoice_id is set, it will be returned. Otherwise, the PK of the order object will be returned.

payment_method_subclass
Get the delivery address instance selected for this cart.

Returns an instance of one of the registered *DeliveryAddress* subclasses.

payment_subclass
Get the payment method instance selected for this cart.

Returns an instance of one of the registered *PaymentMethod* subclasses.

Serializers

lorikeet.api_serializers.**registry**
Registers serializers with their associated models.

This is used instead of discovery or a metaclass-based registry as making sure the classes to be registered actually get imported can be fragile and non-obvious to debug.

The registry instance is available at lorikeet.api_serializers.registry.

registry.**register**(*model*, *serializer*)
Associate model with serializer.

class lorikeet.api_serializers.**PrimaryKeyModelSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)
A serializer that accepts the primary key of an object as input.

When read from, this serializer works exactly the same as ModelSerializer. When written to, it accepts a valid primary key of an existing instance of the same model. It can't be used to add or edit model instances.

This is provided as a convenience, for the common use case of a *LineItem* subclass that has a foreign key to a product model; see the Getting Started Guide for a usage example.

get_queryset()
Returns a queryset which the model instance is retrieved from.

By default, returns `self.Meta.model.objects.all()`.

class lorikeet.api_serializers.**LineItemSerializer**(*instance=None*, **args*, ***kwargs*)
Base serializer for LineItem subclasses.

Mixins

Template Tags

lorikeet.templatetags.lorikeet.**lorikeet_cart**(*context*)
Returns the current state of the user's cart.

Returns a JSON string of the same shape as a response from *GET /_cart/*. Requires that the current request be in the template's context.

Cart Checkers

lorikeet.cart_checkers.**delivery_address_required**(*cart*)
Prevents checkout unless a delivery address is selected.

lorikeet.cart_checkers.**payment_method_required**(*cart*)
Prevents checkout unless a payment method is selected.

lorikeet.cart_checkers.**cart_not_empty**(*cart*)
Prevents checkout of an empty cart.

lorikeet.cart_checkers.**email_address_if_anonymous**(*cart*)
Prevents anonymous users checking out without an email address.

Exceptions

class lorikeet.exceptions.**PaymentError**(*info=None*)
Represents an error accepting payment on a PaymentMethod.

Parameters **info** – A JSON-serializable object containing details of the problem, to be passed to the client.

class lorikeet.exceptions.**IncompleteCartError**(*code*, *message*, *field=None*)
Represents a reason that a cart is not ready for checkout.

Similar to a Django `ValidationError`, but not used to reject a change based on submitted data.

Parameters

- **code** (*str*) – A consistent, non-localised string to identify the specific error.
- **message** (*str*) – A human-readable message that explains the error.
- **field** (*str, NoneType*) – The field that the error relates to. This should match one of the fields in the cart's serialized representation, or be set to `None` if a specific field does not apply.

to_json()
Returns the error in a JSON-serializable form.

Return type dict

class lorikeet.exceptions.**IncompleteCartErrorSet**(*errors=()*)
Represents a set of multiple reasons a cart is not ready for checkout.

You can raise this exception instead of *IncompleteCartError* if you would like to provide multiple errors at once.

This class is iterable.

Parameters **errors** (*Iterable[IncompleteCartError]*) – All of the errors that apply.

add(*error*)
Add a new error to the set.

Parameters **error** (IncompleteCartError, IncompleteCartErrorSet) – The error to add. If an *IncompleteCartErrorSet* instance is passed, it will be merged into this one.

to_json()
Returns the list of errors in a JSON-serializable form.

Return type dict

Settings

Lorikeet's behaviour can be altered by setting the following settings in your project's `settings.py` file.

LORIKEET_CART_COMPLETE_CHECKERS
Default value:

```
[
    'lorikeet.cart_checkers.delivery_address_required',
    'lorikeet.cart_checkers.payment_method_required',
    'lorikeet.cart_checkers.cart_not_empty',
    'lorikeet.cart_checkers.email_address_if_anonymous',
]
```

Checkers that validate whether or not a cart is ready for checkout. For more detail on these, including how to write your own, refer to the guide on *Cart Checkers*.

Warning: The default value for *LORIKEET_CART_COMPLETE_CHECKERS* contains important built-in checkers that you probably don't want to disable, because they prevent things like going through checkout with an empty cart. If you override this setting, make sure you include them!

LORIKEET_ORDER_DETAIL_VIEW
Default value: `None`

The name of a URL pattern that points to a view describing a single *Order* object. The regex for this URL pattern must have an `id` kwarg that matches the numeric ID of the order object; custom invoice IDs in URLs are not yet supported.

This value should be the same as the string you'd pass as the first argument to `django.core.urlresolvers.reverse()`, e.g. `'products:order'`.

If set, it will be used in *lorikeet.models.Order.get_absolute_url()* and *POST /_cart/checkout/*.

LORIKEET_SET_CSRFTOKEN_EVERYWHERE
Default value: `True`

The Lorikeet JavaScript library expects the CSRF token cookie to be set, but it isn't always (see the warning in the Django CSRF docs). For convenience, Lorikeet tells Django to set the cookie on every request (the equivalent of calling ensure_csrf_cookie() on every request). If you wish to handle this yourself, you can set this setting to `False` to disable this behaviour.

LORIKEET_INVOICE_ID_GENERATOR
Default value: `None`

Todo

Document this here as well as in recipes

Signals

`lorikeet.signals.`**order_checked_out**
Fired when a cart is checked out and an order is generated.

Parameters:

- `order` - the *Order* instance that was just created.

Signal handlers can return a dictionary, which will be merged into the response returned to the client when the checkout happens. They can also return `None`, but should not return anything else.

If signals raise an exception, the exception will be logged at the `warning` severity level; it's up to you to be able to report this and respond appropriately.

Note: This signal is fired synchronously during the checkout process, before the checkout success response is returned to the client. If you don't need to return data to the client, try to avoid doing any long-running or failure-prone processes inside handlers for this signal.

For example, if you need to send order details to a fulfilment provider, you could use a signal handler to enqueue a task in something like Celery, or you could have a model with a one-to-one foreign key which you create in a batch process.

HTTP API

GET /_cart/
The current state of the current user's cart. An example response body looks like this:

```
{
    "items": [/* omitted */],
    "new_item_url": "/_cart/new/",
    "delivery_addresses": [

    ],
    "new_address_url": "/_cart/new-address/",
    "payment_methods": [/* omitted */],
```

```
    "new_payment_method_url": "/_cart/new-payment-method/",
    "grand_total": "12.00",
    "generated_at": 1488413101.985875,
    "is_complete": false,
    "incomplete_reasons": [
        {
            "code": "not_set",
            "field": "payment_method",
            "message": "A payment method is required."
        }
    ],
    "checkout_url": "/_cart/checkout/",
    "is_authenticated": true,
    "email": null
}
```

The meaning of the keys is as follows:

- `items` - The list of items in the cart. Each entry in this list is a JSON blob with the same structure as the *`GET /_cart/(id)/`* endpoint.
- `delivery_addresses` - The list of all delivery addresses available to the user. Each entry in this list is a JSON blob with the same structure as the *`GET /_cart/address/(id)/`* endpoint.
- `email` - The email address attached to the cart, as set by *`PATCH /_cart/`*.

`PATCH /_cart/`

Set an email address on this cart. This API call is useful for sites that allow anonymous checkout. Note that you **must** use the `PATCH` method, and you cannot update any fields other than `email`.

An example request body looks like this:

```
{"email": "joe.bloggs@example.com"}
```

The email value can also be `null` to un-set the value.

Status Codes

- 200 OK – The email was changed successfully.
- 400 Bad Request – The supplied email was invalid.

`GET /_cart/` (*id*) **`/`**

Details about a particular item in the cart. An example response body looks like this:

```
{
    "type": "WineLineItem",
    "data": {
        "product": {
            "id": 11,
            "name": "Moscato 2016",
            "photo": "/media/moscato.png",
            "unit_price": "12.00"
        },
        "quantity": 1
    },
    "total": "12.00",
    "url": "/_cart/77/"
}
```

GET /_cart/address/(*id*)**/**

Details about a particular delivery address that is available for the user. An example response body looks like this:

```
{
    "type": "AustralianDeliveryAddress",
    "data": {
        "addressee": "Joe Bloggs",
        "address": "123 Fake St",
        "suburb": "Adelaide",
        "state": "SA",
        "postcode": "5000"
    },
    "selected": true,
    "url": "/_cart/address/55/"
}
```

POST /_cart/checkout/

Finalise the checkout process; process the payment and generate an order.

Status Codes

- 200 OK – Checkout succesful; payment has been processed and order has been generated.
- 422 Unprocessable Entity – Checkout failed, either because the cart was not ready for checkout or the payment failed.

This endpoint should be called without any parameters, but the user's cart should be in a state that's ready for checkout; that is the `is_complete` key returned in *GET /_cart/* should be `true`, and `incomplete_reasons` should be empty.

If checkout was successful, the response body will look like this:

```
{
    "id": 7,
    "url": "/products/order/7/",
}
```

where the returned `id` is the ID of the *Order* instance that was created, and the `url` is a URL generated from the `LORIKEET_ORDER_DETAIL_VIEW` setting (or `null` if that setting is not set).

If the cart was not ready for checkout, the endpoint will return a 422 response with a body that looks like this:

```
{
    "reason": "incomplete",
    "info": [
        {
            "message": "There are no items in the cart.",
            "field": "items",
            "code": "empty"
        }
    ]
}
```

In this case, the `reason` is always the string `"incomplete"`, and the `info` is the same list of values as in the `incomplete_reasons` key returned in *GET /_cart/*.

If processing the payment failed, the endpoint will return a 422 response with a body that looks like this:

```
{
    "reason": "payment",
    "payment_method": "StripeCard",
    "info": {
        "message": "Your card was declined.",
        // ...
    }
}
```

In this case, the `reason` is always the string `"payment"`; `payment_method` is the name of the *`PaymentMethod`* subclass that handled the payment. `info` is data returned by the payment method itself; consult its documentation for its meaning.

Todo

describe the other endpoints

Why does Lorikeet's API work like this?

By now, you'll have noticed that Lorikeet's API isn't structured like most REST APIs, with different endpoints returning a bunch of paginated collections of resources you can query from. Instead, there's one endpoint that returns one object containing the entire contents of the API. That resource contains sub-resources which do have their own endpoints, but they're only really useful for making modifications with `POST`, `PUT` and `PATCH`.

This design is inspired by Facebook's GraphQL, as well as web frontend state management libraries like Redux. In GraphQL, an entire API is conceptually a single object, which can be filtered and have parameters passed to its properties. In Lorikeet, the entire API is *literally* a single object, with no filtering or parameterisation, because the amount of data an individual user cares about is compact and practical to return all at once. The `POST`, `PUT` and `PATCH` endpoints, on the other hand, can be thought of as roughly analogous to Redux actions; there's not much to gain by merging these into a single endpoint.

JavaScript API

Lorikeet comes with a small JavaScript library to make manipulating the cart from client JavaScript a little easier. It provides convenience creation, update and delete methods for line items, delivery addresses and payment methods, and also keeps the state of the shopping cart in sync if it's open in multiple tabs using `localStorage`.

It supports IE11, the latest versions of Safari, Edge and Firefox, and the two latest versions of Chrome. It requires a `window.fetch` polyfill for IE and Safari.

Installation

The JavaScript component of Lorikeet can be installed via NPM (to be used with a bundler like Webpack). In the future, it will also be provided as a CDN-hosted file you can reference in a `<script>` tag. To install it, run `npm install https://gitlab.com/abre/lorikeet`.

Usage

If you're using the NPM version, `import CartClient from 'lorikeet'` or `var CartClient = require('lorikeet')` as appropriate for your setup.

Use the CartClient constructor to instantiate the client. This is the object you'll use to interact with the API.

```
var client = new CartClient('/_cart/')
```

You can now access the current state of the cart on `client.cart`, which exposes the entire contents of the main endpoint of the *HTTP API*.

```
console.log(client.cart.grand_total) // "123.45"
console.log(client.cart.items.length) // 3
```

You can listen for changes using the `addListener` and `removeListener` events.

```
var listenerRef = client.addListener(function(cart){console.log("Cart updated", cart)}
↪)
client.removeListener(listenerRef)
```

All of the members of the lists at `client.cart.items`, `client.cart.delivery_addresses` and `client.cart.payment_methods` have a `delete()` method. Members of `client.cart.items` also have `update(data)` method, which performs a partial update (`PATCH` request) using the data you pass, and members of the other two have a `select()` method that, makes them the active delivery address or payment method.

```
client.cart.items[0].update({quantity: 3})
client.cart.items[1].delete()
client.cart.delivery_addresses[2].select()
client.cart.payment_methods[3].delete()
```

There's also `addItem`, `addAddress` and `addPaymentMethod` methods, which take a type of line item, address or payment method as their first item, and a blob in the format expected by the corresponding serializer as the second.

```
client.addItem("MyLineItem", {product: 1, quantity: 2})
client.addAddress("AustralianDeliveryAddress", {
  addressee: "Adam Brenecki",
  address: "Commercial Motor Vehicles Pty Ltd\nLevel 1, 290 Wright St",
  suburb: "Adelaide", state: "SA", postcode: "5000",
})
client.addPaymentMethod("PipeCard", {card_token: "tok_zdchtodladvrcmkxsgvq"})
```

Reference

class `CartClient` (*cartUrl*, *cartData*)

A client that interacts with the Lorikeet API.

Arguments

- **`cartUrl`** (*string*) – URL to the shopping cart API endpoint.
- **`cartData`** (*object*) – Current state of the cart. If provided, should match the expected strcuture returned by the cart endpoint.

`CartClient.`**`addItem`**(*type*, *data*)

Add an item to the shopping cart.

Arguments

- **`type`** (*string*) – Type of LineItem to create
- **`data`** (*object*) – Data that the corresponding LineItem serializer is expecting.

CartClient.**addAddress**(*type*, *data*)
Add a delivery address to the shopping cart.

Arguments

- **type** (*string*) – Type of DeliveryAddress to create
- **data** (*object*) – Data that the corresponding DeliveryAddress serializer is expecting.

CartClient.**addPaymentMethod**(*type*, *data*)
Add a delivery address to the shopping cart.

Arguments

- **type** (*string*) – Type of PaymentMethod to create
- **data** (*object*) – Data that the corresponding PaymentMethod serializer is expecting.

CartClient.**setEmail**(*address*)
Set an email address for the shopping cart.

Arguments

- **address** (*string|null*) – Email address to set. Use null to clear the address field.

CartClient.**addListener**(*listener*)
Register a listener function to be called every time the cart is updated.

Arguments

- **listener** (*CartClient~cartCallback*) – The listener to add.

Returns **CartClient~cartCallback** – Returns the listener function that was passed in, so you can pass in an anonymous function and still have something to pass to removeListener later.

CartClient.**removeListener**(*listener*)

Arguments

- **listener** (*CartClient~cartCallback*) – The listener to remove.

class **CartItem**(*client*, *data*)
A single item in a cart.

CartItem.**update**(*newData*)
Update this cart item with new data, e.g. changing a quantity count. Note that calling this method will not update the current CartItem; you'll have to retrieve a new CartItem from the client's cart property or from an event handler.

Arguments

- **newData** (*object*) – The data to patch this cart item with. Can be a partial update (i.e. something you'd send to a HTTP PATCH call).

class **AddressOrPayment**(*client*, *data*)
A single delivery address, or a single payment method. (Both have the same shape and methods, so they share a class.)

AddressOrPayment.**select**()
Make this the active address or payment method.

Promise Behaviour

All of the methods that modify the cart (*CartClient.addItem()*, *CartClient.addAddress()*, *CartClient.addPaymentMethod()*, *CartItem.update()*, and *AddressOrPayment.select()*) return Promises, which have the following behaviour.

If the request **succeeds**, the promise will *resolve* with the JSON-decoded representation of the response returned by the relevant API endpoint.

If the request **fails with a network error**, the promise will *reject* with an object that has the following shape:

```
{
    reason: 'network',
    error: TypeError("Failed to fetch"), // error from fetch() call
}
```

If the request is made, but **receives an error response from the server**, the promise will *reject* with an object that has the following shape:

```
{
    reason: 'api',
    status: 422,
    statusText: 'Unprocessable Entity',
    body: "{\"suburb\":[\"This field is...", // Raw response body
    data: {
        suburb: ["This field is required."],
        // ...
    }, // JSON-decoded response body
}
```

If an error response is returned from the server, and **the response is not valid JSON**, such as a 500 response with `DEBUG=True` or a 502 from a reverse proxy, the promise will instead *reject* with an object that has the following shape:

```
{
    reason: 'api',
    status: 502,
    statusText: 'Bad Gateway',
    body: "<html><body><h1>Bad Gateway...", // Raw response body
    decodeError: SyntaxError("Unexpected token < in JSON at position 0"),
}
```

Reducing Round Trips

The *CartClient()* constructor takes an optional second argument `cart`, which it will use instead of hitting the API if there's no data already in local storage. (Even if there is, it'll update it if it's stale, so it's always a good idea.)

You can use it alongside the *lorikeet_cart()* template tag like this:

```
{% load lorikeet %}
{# ... #}
<body data-cart="{% lorikeet_cart %}">
```

```
var cart = JSON.parse(document.body.attributes['data-cart'].value)
var client = new CartClient('/_cart/', cart)
```

React

Lorikeet also comes with optional support for React. To use it, wrap your React app's outermost component in `CartProvider`, providing your Lorikeet client instance as the `client` prop.

```
import { CartProvider } from 'lorikeet/react'

class App extends Component {
  render() {
    return <CartProvider client={myClient}>
        // ...
    </CartProvider>
  }
}
```

Then, in any component where you want to use the client, decorate it with `cartify`, and you'll have access to the client as `props.cartClient`, as well as a shortcut to the cart itself on `props.cart`.

```
import cartify from 'lorikeet/react'

class MyCart extends Component {
  handleAddItem(item){
    this.props.cartClient.addItem('ItemType', item)
  }
  render(){
    return <div>My cart has {this.props.cart.items.length} items!</div>
  }
}

MyCart = cartify(MyCart)
```

Email Invoicing

Installation

1. Make sure Lorikeet is installed with the `email_invoice` extra, by running `pip install https://gitlab.com/abre/lorikeet.git[email_invoice]`.
2. Add `'lorikeet.extras.email_invoice'` to your `INSTALLED_APPS`.

3. Set the `LORIKEET_EMAIL_INVOICE_SUBJECT` variable in `settings.py` to a subject line. 3. Set the `LORIKEET_EMAIL_INVOICE_TEMPLATE_HTML` variable in `settings.py` to a HTML template. 4. Set the `LORIKEET_EMAIL_INVOICE_TEMPLATE_TEXT` variable in `settings.py` to a plain text template. 3. Set the `LORIKEET_EMAIL_INVOICE_FROM_ADDRESS` variable in `settings.py` to an email address.

Usage

Set the `LORIKEET_EMAIL_INVOICE_SUBJECT` setting to the subject line you want your emails to have. You can use the Python new-style format string syntax to reference the *`Order`* object, e.g. `"Your invoice for order {order.invoice_id}"`.

Create a HTML template at the path you set `LORIKEET_EMAIL_INVOICE_TEMPLATE_HTML` to. It will recieve the *`Order`* instance in its context as `order`, and `order_url` will be set to the absolute URL to your order details view,

The template will be run through premailer, so you can safely use `<style>` and `<link rel="stylesheet">` tags. Of course, you can still only use CSS properties supported by the email clients you're targeting.

```
<html><body>
<p><a href="{{ order_url }}">To find out the current status of your order, click here.
↪</a></p>
<h1>Tax Invoice</h1>
<p>ACME Corporation Pty Ltd<br />ABN 84 007 874 142</p>

<h2>Invoice {{ order.invoice_id }}</h2>

<h3>Shipped To</h3>
{{ order.delivery_address_subclass }}

<h3>Order Details</h3>
<table>
    <tr>
        <th>Product</th>
        <th>Quantity</th>
        <th>Subtotal</th>
    </tr>
    {% for item in order.items.select_subclasses %}
        <tr>
            <td>{{ item.product.name }}</td>
            <td>{{ item.quantity }}</td>
            <td>{{ item.total }}</td>
        </tr>
    {% endfor %}
    <tr>
    <tr>
        <td colspan='2'>Total</td>
        <td>{{ order.grand_total }}</td>
    </tr>
</table>
</body></html>
```

Then, create a plain-text template at the path you set `LORIKEET_EMAIL_INVOICE_TEMPLATE_TEXT` to. It will recieve the same context.

Stripe

Installation

1. Make sure Lorikeet is installed with the `stripe` extra, by running `pip install https://gitlab.com/abre/lorikeet.git[stripe]`.
2. Add `'lorikeet.extras.stripe'` to your `INSTALLED_APPS`.
3. Set the `STRIPE_API_KEY` variable in `settings.py` to your Stripe API key.

Usage

Note: These examples use the js, but everything works the same if you're using the api directly, since the JavaScript API is only a thin wrapper.

On creation, the Stripe payment method takes only one parameter, the token that you get from Stripe.js.

```
Stripe.card.createToken(form, function(status, response){
    if (status == 200){
        client.addPaymentMethod("StripeCard", {token: response.id})
    }
})
```

Once they're created, in the cart they'll show up with their brand and last 4 digits.

```
console.log(client.cart.payment_methods)
// [{
//     type: "StripeCard",
//     url: "/_cart/payment-methods/1/",
//     selected: true,
//     data: {brand: "Visa", last4: "4242"},
// }]
```

If you try to charge the card and the charge fails,

StarShipIT

Deprecated since version 0.1.5: Projects that wish to continue using this integration should vendor the `lorikeet.extras.starshipit` package within their own projects; it will be removed from Lorikeet before version 1.0 is released.

This integration posts orders to StarShipIT, a cloud-based fulfilment software provider.

Installation

1. Make sure Lorikeet is installed with the `starshipit` extra, by running `pip install https://gitlab.com/abre/lorikeet.git[starshipit]`.
2. Add `'lorikeet.extras.starshipit'` to your `INSTALLED_APPS`.
3. Set the `STARSHIPIT_API_KEY` variable in `settings.py` to your StarShipIT API key.
4. Configure your site to call `lorikeet.extras.starshipit.submit.submit_orders()` periodically (using e.g. a management command, django-cron or Celery Beat). If you're using Celery, there's a task at `lorikeet.extras.starshipit.tasks.submit_orders` that you can add to your `CELERYBEAT_SCHEDULE`.

Serialisation

To work with StarShipIT, all of your cart item and delivery address models should implement a `.starshipit_repr()` method. These methods should return a dictionary with keys expected by StarShipIT's Create Order API endpoint: the **ShipmentItem** parameters for a cart item, and **DestinationDetails** for a delivery address.

If you can't do this (for instance, you have cart items or delivery addresses provided by a third-party package), create a `STARSHIPIT_REPR` setting in your `settings.py`. This setting should be a dictionary where the keys are cart item or delivery address model names, and the values are functions that take an instance of that model and return the appropriate value.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

/_cart

Index

T