django-plans Documentation

Release 0.7-alpha

Krzysztof Dorosz

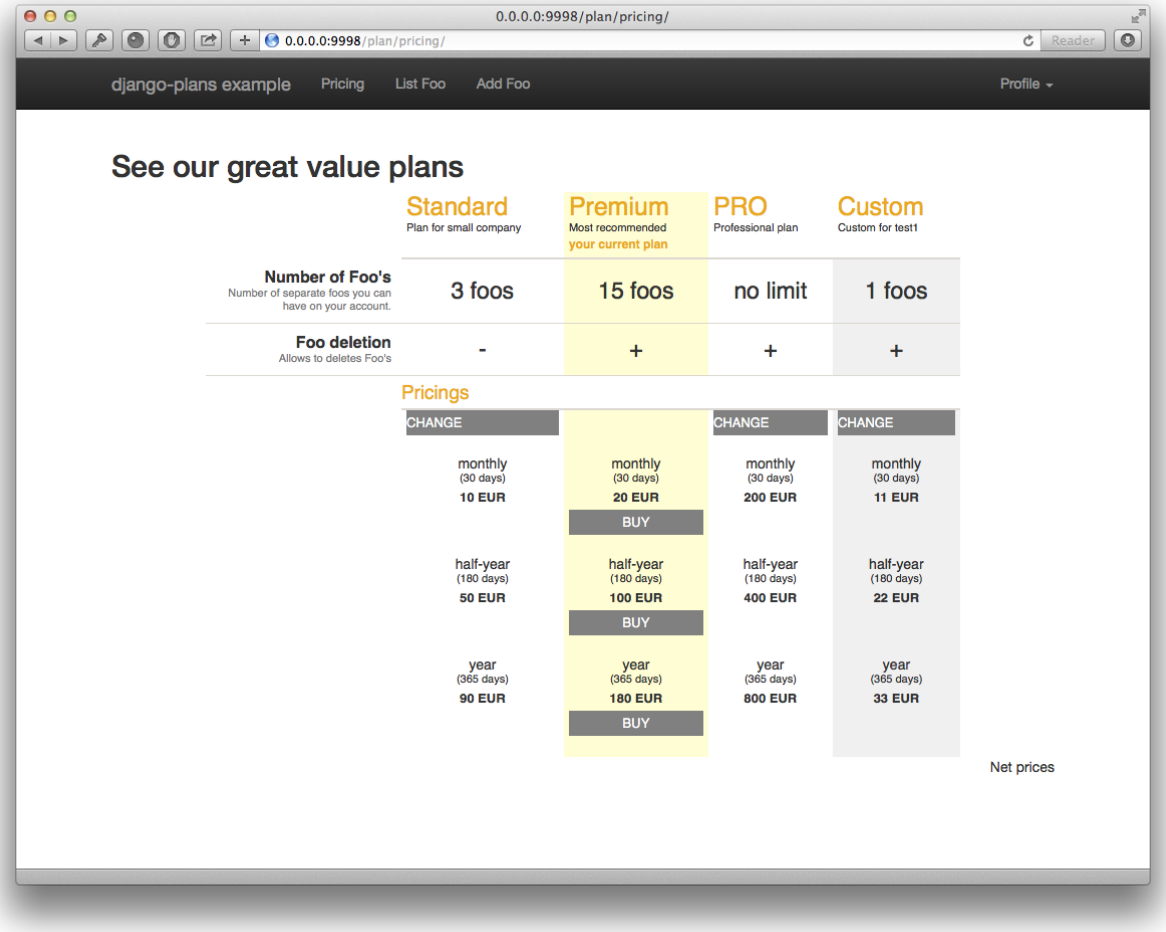
Sep 27, 2017

Contents

1 Contents: **3**
- 1.1 Installation . . . 3
- 1.2 Integration . . . 4
- 1.3 Configuration via *settings* . . . 5
- 1.4 Plans and pricing definition . . . 9
- 1.5 Validation of quota . . . 13
- 1.6 Plan change policies . . . 16
- 1.7 Taxation Policies . . . 17
- 1.8 Invoicing . . . 18
- 1.9 Templates . . . 20
- 1.10 Working with South migrations . . . 21
- 1.11 Caveats . . . 22

2 Indices and tables **25**
- 2.1 Developing . . . 25

Django-plans is a pluggable app for managing pricing plans with quotas and accounts expiration.



CHAPTER 1

Contents:

Installation

Installing module code

You can install app using package manager directly from github:

```
$ pip install -e git://github.com/cypreess/django-plans.git#egg=django-plans
```

For integration instruction please see section *Integration*.

Running example project

Clone git repository to your current directory:

```
$ git clone git://github.com/cypreess/django-plans.git
```

Optionally create virtual environment and get required packages to run example project:

```
$ cd django-plans/demo/
$ pip install -r requirements.txt
```

Initialize example project database:

```
$ cd ..
$ python manage.py syncdb
[...]
Would you like to create one now? (yes/no): no
[...]
```

Initial example data will be loaded automatically.

Start development web server:

```
$ python manage.py runserver
```

Visit http://localhost:8000/

Integration

TODO: write complete description

This section describes step by step integration of django-plans with your application.

Enable plans application in django

Add this app to your `INSTALED_APPS` in django settings.py:

```
INSTALLED_APPS += ('plans', 'ordered_model',)
```

Note: The app 'ordered_model' is required to display the assets used in the django admin to manage the plan model ordering

You should also define all other variables in `settings.py` marked as **required**. They are described in detail in section *Configuration via settings*.

Don't forget to run:

```
$ python manage.py syncdb
```

If you are going to use South migrations please read section *Working with South migrations*.

Enable context processor

Section *Templates* describes a very helpful content processor that you should definitely enable in your project settings in this way:

```
from django.conf import global_settings

TEMPLATE_CONTEXT_PROCESSORS = global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
    'plans.context_processors.account_status'
    )
```

Send signal when user account is fully activated

You need to explicitly tell django-plans that user has fully activated account. django-plans provides a special signal that it listen to.

`plans.signals.activate_user_plan(user)`

You should send this signal providing `user` argument as an object of `auth.User`. django-plans will use this information to initialize plan for this user, i.e. set account expiration date and will mark the default plan as active for this account.

Note: If you use django-registration app for managing user registration process, you are done. django-plans automagically integrates with this app (if it is available) and will activate user plan when django-registration send it's signal after account activation.

Configuration via *settings*

PLANS_CURRENCY

Required

Three letter code for system currency. Should always be capitalized.

Example:

```
PLANS_CURRENCY = 'EUR'
```

DEFAULT_FROM_EMAIL

Required

This is the default mail `FROM` value for sending system notifications.

PLANS_INVOICE_COUNTER_RESET

Optional

Default: `monthly`

This settings switches invoice counting per days, month or year basis. It requires to provide one of the value:

- Invoice.NUMBERING.DAILY
- Invoice.NUMBERING.MONTHLY
- Invoice.NUMBERING.ANNUALY

Example:

```
PLANS_INVOICE_COUNTER_RESET = Invoice.NUMBERING.MONTHLY
```

Warning: Remember to set `PLANS_INVOICE_NUMBER_FORMAT` manually to match preferred way of invoice numbering schema. For example if you choose reset counter on daily basis, you need to use in `PLANS_INVOICE_NUMBER_FORMAT` at least `{{ invoice.issued|date:'d/m/Y' }}` to distinguish invoice's full numbers between days.

PLANS_INVOICE_NUMBER_FORMAT

Optional

Default: `"{{ invoice.number }}/{% ifequal invoice.type invoice.INVOICE_TYPES.PROFORMA %}PF{% else %}FV{% endifequal %}/{{ invoice.issued|date:'m/Y' }}"`

A django template syntax format for rendering invoice full number. Within this template you can use one variable `invoice` which is an instance of `Invoice` object.

Example:

```
PLANS_INVOICE_NUMBER_FORMAT = "{{ invoice.number }}/{{ invoice.issued|date='m/FV/Y' }}
↪"
```

This example for invoice issued on `March 5th, 2010`, with sequential number `13`, will produce the full number `13/03/FV/2010` or `13/03/PF/2010` based on invoice type.

Warning: Full number of an invoice is saved with the Invoice object. Changing this value in settings will affect only newly created invoices.

PLANS_INVOICE_LOGO_URL

Optional

Default: `None`

URL of logo image that should be placed in an invoice. It will be available in invoice template as `{{ logo_url }}` context variable.

Example:

```
from urllib.parse import urljoin
PLANS_INVOICE_LOGO_URL = urljoin(STATIC_URL, 'my_logo.png')
```

PLANS_INVOICE_TEMPLATE

Optional

Default: `'plans/invoices/PL_EN.html'`

Template name for displaying invoice.

Warning: Invoices are generated on the fly from database records. Therefore changing this value will affect all previously created invoices.

Example:

```
PLANS_INVOICE_TEMPLATE = 'plans/invoices/PL_EN.html'
```

PLANS_INVOICE_ISSUER

Required

You need to define a dictionary that will store information needed to issue an invoice. Fill dict fields as in an example.

Example:

```
PLANS_INVOICE_ISSUER = {
    "issuer_name": "Joe Doe Company",
    "issuer_street": "Django street, 34",
    "issuer_zipcode": "123-3444",
    "issuer_city": "Djangoko",
    "issuer_country": "Djangoland",
    "issuer_tax_number": "1222233334444555",
}
```

PLANS_ORDER_EXPIRATION

Optional

Default: `14`

A number of days that an Order is valid (e.g. to start a payment) counting from order creation date. This value is only used in `is_ready_for_payment()` method for django-getpaid integration. This value has no effect on processing already paid orders before `PLANS_ORDER_EXPIRATION` period, even if confirmation for this payment will came after `PLANS_ORDER_EXPIRATION` period.

Example:

```
PLANS_ORDER_EXPIRATION = 14
```

PLANS_EXPIRATION_REMIND

Optional

Application is responsible for expiring user accounts. Before account became expired it is able to send expiration warnings to the users. This setting should contain a list of numbers, that corresponds to days before expiration period. User will receive expiration warning at each moment from that list.

Default: `[]`

Example:

```
PLANS_EXPIRATION_REMIND = [1, 3 , 7]
```

User will receive notification before 7 , 3 and 1 day to account expire.

PLANS_CHANGE_POLICY

Optional

Default: `'plans.plan_change.StandardPlanChangePolicy'`

A full python to path that should be used as plan change policy.

PLANS_DEFAULT_GRACE_PERIOD

Optional

Default: `30`

New account default plan expiration period counted in days.

Example:

```
PLANS_DEFAULT_GRACE_PERIOD = 30
```

Note: Default plan should be selected using site admin. Set default flag to one of available plans.

PLANS_VALIDATORS

Optional

Default: `{}`

A dict that stores mapping `"Quota codename" : "validator object"`. Validators are used to check if user account can be activated for the given plan. Account cannot exceed certain limits introduced by quota.

Given account will be activated only if calling all validators that are defined with his new plan does not raise any ValidationError. If account cannot be activated user will be noticed after logging with information that account needs activation.

Example:

```
PLANS_VALIDATORS = {
    'CAN_DO_SOMETHING' : 'myproject.validators.can_do_something_validator',
    'MAX_STORAGE' : 'myproject.validators.max_storage_validator',
}
```

Further reading: *Validation of quota*

SEND_PLANS_EMAILS

Optional

Default: `True`

Boolean value for enabling (default) or disabling the sending of plan related emails.

PLANS_TAX

Required

Decimal or integer value for default TAX (usually referred as VAT).

Example:

```
from decimal import Decimal
PLANS_TAX = Decimal('23.0')  # for 23% VAT
```

Default: `None`

Warning: The value `None` means "TAX not applicable, rather than value `Decimal('0')` which is 0% TAX.

PLANS_TAXATION_POLICY

Required

Class that realises taxation of an order.

Example:

```
PLANS_TAXATION_POLICY='plans.taxation.eu.EUTaxationPolicy'
```

Further reading: *Taxation Policies*

PLANS_TAX_COUNTRY

Optional

Two letter ISO country code. This variable is used to determine origin issuers country. Taxation policy uses this value to determine tax amount for any order.

Example:

```
PLANS_TAX_COUNTRY = 'PL'
```

Note: `settings.PLANS_TAX_COUNTRY` is a separate value from `settings.PLANS_INVOICE_ISSUER.issuer_country` on purpose. `PLANS_INVOICE_ISSUER` is just what you want to have printed on an invoice.

Plans and pricing definition

All definition of an offer (like plans, options, pricing, etc...) is made by django admin panel. That means that there no need to make any hardcoded definitions of plans, they are stored in the database.

Engine allows for the following customisation:

- Many plans can be defined, plan can be considered as a named group of account features for a specific price in specific period.
- Many pricing periods can be defined (e.g. monthly, annually, quarterly or any other), pricing is basically named amount of days.
- Many types of account feature (called quotas) can be defined (eg. maximum number of some items, account transfer limit, does the account is allowed to customize something).
- After defining quotas, each plan can define its own set of quotas with given values.
- I18n is supported in every aspect (in database text name fields also)

Plan

Plan stores information about single plan that is offered by a service. It is defined by the following properties:

name

type: text

Plan name that is visible in headers, invoice, etc. This should be a short name like: "Basic", "Premium", "Pro".

Note: This field supports i18n. In admin view you will be able to input this name in all available languages.

description

type: text

Stores a short description for the plan that will be used in various places mostly for marketing purposes, eg. "For small groups", "Best value for medium companies", etc.

Note: This field supports i18n. In admin view you will be able to input this name in all available languages.

available

type: boolean

Only plans that are marked as `available` will be enabled to buy for the customers.

Warning: You should never delete once created `Plan` unless you are sure that nobody is using it. If you want to stop offering some plan to customers, just mark it `unavailable` and create other plan (even with the same name; plan name is not unique in the system). Users will be asked to switch to the other plan when they will try to extend their accounts bound to Plan which is not available.

customized

type: `User`

Setting `customized` value to a specific users creates a special `Plan` that will be available only for that one user. This allows to setup a tailored `Plans` that are not available for public.

Note: Plan that is customized for a user need to be also `available` if user need to be able to buy this plan.

Note: It is not possible to share one customized plan for two users. Even if plans are the same, there must be two identical custom plans for both users.

List of `pricing` periods

type: Many-to-many with `Pricing` by `PlanPricing`

Many pricing periods can be defined for a given plan. For each entry there is a need of defining price. The currency of price is defined by `settings.PLANS_CURRENCY`.

Warning: It is not possible to define multiple price currencies in the system. You can define only one type of currency and it will describe a currency of all amounts in the system.

Note: Not all plans need necessarily to define all available pricing periods. Therefore a single plan need to define at least single pricing period, because it will be not possible to buy one without it.

List of `quotas`

type: Many-to-many with `Quota` by `PlanQuota`

Account that uses a given `Plan` can have various restrictions. Those restrictions are realised by `Quota` parameter. Each plan can have defined multiple set of `Quota` parameters with theirs corresponding values.

Please refer to `Quota` documentation for description of parameters types.

Warning: Unless you know what you are doing all available plans should have defined the same set of quotas.

Note: Omitting value for integer type quota is interpreted as "no limit".

Quota

Quota represents a single named parameter that can be given to restrict functionality in the system. Parameters can have two types:

- integer type - `is_boolean` is off, then the value for a `Quota` will be interpreted as numerical (integer) restriction (e.g. "number of photos").
- boolean type - `is_boolean` is on, the value will be interpreted as boolean flag (e.g. "user can add photos").

Warning: Making actual restrictions based on that values is a part of development process and is not covered here. In admin module you can only define any named quotas, but of course it will not magically affect anything unless any part of code implement some restrictions based on that.

Quota is made of following fields:

`codename`

type: string

This is a name for internal use by developers. They can use this name to identity quotas in the system and fetch their values.

name

type: string

Human readable name of restriction (e.g. “Total number of photos”)

Note: This field supports i18n. In admin view you will be able to input this name in all available languages.

unit

type: string

For displaying purposes you can define a unit that will be displayed after value (e.g. “MB”).

Note: This field supports i18n. In admin view you will be able to input this name in all available languages.

description

type: string

Short description of the restriction (e.g. “This is a limit of total photos that you can have in your account”)

Note: This field supports i18n. In admin view you will be able to input this name in all available languages.

is_boolean

type: boolean

This field flags this restriction as boolean type field. Value of this quota will be evaluated to `True` or `False` to determine provided option.

Pricing

Pricing defines a single period of time that can be billed and account can be extended for this period. Because periods can be named differently in many languages you can provide following properties for this objects:

name

type: string

Pricing period name (e.g. “Monthly”, “Month”, “Full 30 days”, “Annually”, etc.)

Note: This field supports i18n. In admin view you will be able to input this name in all available languages.

period

type: integer

Number that is representing a period in days (e.g. for month - `30`, for annual - `365`, etc.)

Validation of quota

The model of plans introduced in this application make use of quota, which are just some arbitrarily given limits. Quota definition is quite flexible and allows to define many different types of restrictions. Conceptually you may need one of following types of quota in your system.

- **limiting resources** - limiting number of some entities on user account (typically an entities is a single django model instance); e.g. an image gallery system could limit number of images per account as one of the system parameter. Each image is represented by one instance of e.g. UploadedImage model.
- **limiting states** - limiting that some entities on user account can be in a given state (typically some model instance attributes can have specific values)
- **limiting actions** - limiting if user can perform some kind of action (typically an action is a specific POST request which creates/updates/delete a model instance)

Note: Presented list of quota types is only a conceptual classification. It may not be directly addressed in django-plans API, however django-plans aims to support those kind of limitations.

Account complete validation

Complete account validation is needed when user is switching a plan (or in a general - activating a plan). The reason is that user account can be in the state that exhausting limits of new plan (e.g. on downgrade). Plan should not be activated on the user account until user will not remove over limit resources until the account could validate in limits of the new plan.

In django-plans there is a common validation mechanism which requires defining `PLANS_VALIDATORS` variable in `settings.py`.

The format of `PLANS_VALIDATORS` variable is given as a dict:

```
PLANS_VALIDATORS = {
    '<QUOTA_CODE_NAME>' :  '<full.python.path.to.validator.class>',
    [...]
}
```

First of all this variable defines all quota that should be validated on any plan activation.

Note: Please note that the only quota that can be added to `PLANS_VALIDATORS` are "limiting resources quota" and "limiting states" quota. Those are the kind of quota that conceptually can be validated within the database state. The third kind of quota ("limiting actions quota") are to be checked on to go when user is just using it's account and performing certain actions.

Secondly each quota has a specific validator defined that is custom to your need of validations.

Quota validators

Each validator should inherit from *plans.validators.QuotaValidator*.

class plans.validators.**QuotaValidator**
Base class for all Quota validators needed for account activation

code

default_quota_value = None

get_error_message(*quota_value*, ***kwargs*)

get_quota_value(*user*, *quota_dict=None*)
Returns quota value for a given user

on_activation(*user*, *quota_dict=None*, ***kwargs*)
Hook for any action that validator needs to do while successful activation of the plan Most useful for validators not required to activate, e.g. some "option" is turned ON for user but when user downgrade plan this option should be turned OFF automatically rather than stops account activation

required_to_activate = True

Validator should have defined `__call__(self, user, **kwargs)` method which should raise `django.core.exceptions.ValidationError` if account does not meet limits requirement.

Model count validator

Currently django-plans is shipped with one handy validator. It can easily validate number of instances of any model for a given user.

class plans.validators.**ModelCountValidator**
Validator that checks if there is no more than quota number of objects given model

get_error_message(*quota_value*, ***kwargs*)

get_queryset(*user*)

model

We recommend to create `validators.py` in your application path with your own custom validators.

E.g. this limits number of `Foo` instances in the example project, in `foo/validators.py`:

```
from example.foo.models import Foo
from plans.validators import ModelCountValidator


class MaxFoosValidator(ModelCountValidator):
    code = 'MAX_FOO_COUNT'
    model = Foo

    def get_queryset(self, user):
        return super(MaxFoosValidator, self).get_queryset(user).filter(user=user)

max_foos_validator = MaxFoosValidator()
```

You can easily re-use it also in create model form for this object to check if user can add a number of instances regarding his quota, in `foo/forms.py`:

```
from django.forms import ModelForm, HiddenInput
from example.foo.models import Foo
from example.foo.validators import max_foos_validator


class FooForm(ModelForm):
    class Meta:
        model = Foo
        widgets = {'user' : HiddenInput,}

    def clean(self):
        cleaned_data = super(FooForm, self).clean()
        max_foos_validator(cleaned_data['user'], add=1)
        return cleaned_data
```

Model attribute validator

This validator can validate that every object returned from a queryset have correct value of attribute.

class `plans.validators.`**`ModelAttributeValidator`**

Validator checks if every obj.attribute value for a given model satisfy condition provided in check_attribute_value() method.

> **Warning:** ModelAttributeValidator requires *get_absolute_url()* method on provided model.

`attribute`

`check_attribute_value`(*attribute_value*, *quota_value*)

`get_error_message`(*quota_value*, ***kwargs*)

E.g.:

```
from example.foo.models import Foo
from plans.validators import ModelCountValidator


class MaxFooSizeValidator(ModelAttributeValidator):
    code = 'MAX_FOO_SIZE'
    model = Foo
    attribute = 'size'

    def get_queryset(self, user):
        return super(MaxFoosValidator, self).get_queryset(user).filter(user=user)

max_foo_size_validator = MaxFooSizeValidator()
```

This validator will ensure that user does not have any object with attribute 'size' which is greater then the quota. If you need to provide any custom comparison other than "greater than" just override method `check_attribute_value(attribute_value, quota_value)`.

Plan change policies

Changing (upgrading or downgrading) plan is another thing that can be highly customizable. You can choose which ChangePlanPolicy should be used via `PLANS_CHANGE_POLICY` settings variable.

Plan change policy is a class that derives from `plans.plan_change.PlanChangePolicy` which should implement `get_change_price(plan_old, plan_new, period)`. This method returns should return total price of changing current plan to new one, assuming that a given active period left on the account.

class `plans.plan_change.`**PlanChangePolicy**

get_change_price(*plan_old*, *plan_new*, *period*)
Calculates total price of plan change. Returns None if no payment is required.

There are some default change plan policies already implemented.

StandardPlanChangePolicy

class `plans.plan_change.`**StandardPlanChangePolicy**

This plan switch policy follows the rules:

- user can downgrade a plan for free if the plan is cheaper or have exact the same price (additional constant charge can be applied)
- user need to pay extra amount depending of plans price difference (additional constant charge can be applied)

Change percent rate while upgrading is defined in `StandardPlanChangePolicy.UPGRADE_PERCENT_RATE`

Additional constant charges are:

- `StandardPlanChangePolicy.UPGRADE_CHARGE`
- `StandardPlanChangePolicy.FREE_UPGRADE`
- `StandardPlanChangePolicy.DOWNGRADE_CHARGE`

Note: Example

User has PlanA which costs monthly (30 days) 20 €. His account will expire in 23 days. He wants to change to PlanB which costs monthly (30 days) 50€. Calculations:

```
PlanA costs per day 20 €/ 30 days = 0.67 €
PlanB costs per day 50 €/ 30 days = 1.67 €
Difference per day between PlanA and PlanB is 1.00 €
Upgrade percent rate is 10%
Constant upgrade charge is 0 €
Switch cost is:
          23 *            1.00 € *                  10% +                     0␣
↪€ = 25.30 €
    days_left * cost_diff_per_day * upgrade_percent_rate + constant_upgrade_charge
```

Note: Values of `UPGRADE_CHARGE`, `DOWNGRADE_CHARGE`, `FREE_UPGRADE` and `UPGRADE_PERCENT_RATE` can be customized by creating a custom change plan class that derives from `StandardPlanChangePolicy`.

Taxation Policies

Creating new order is a process that apart from counting item values depends also on specific rules how to apply a tax to the order. Django-plans is designed with internationalization in mind, therefore the way that the module calculates additional tax for an order is highly customizable and depends in general on locale.

For each country, or more generally for each specific use, there need to be created specific taxation policy which defines what rate of tax is suitable for an order depending on issuer country and customer billing data.

Taxation policy can be defined as a simple class that should inherit from `plans.taxation.TaxationPolicy` and provide `get_default_tax(vat_id, country_code)` method. Having arguments like customer

class `plans.taxation.`**`TaxationPolicy`**

Abstract class for defining taxation policies. Taxation policy is a way to handle what tax rate should be put on the order, this depends on user billing data.

Custom taxation policy should implement only method `get_default_tax(vat_id, country_code)`. This method should return a percent value of tax that should be added to the Order, or None if tax is not applicable.

classmethod `get_default_tax``()`

Gets default tax rate. Simply returns `settings.PLANS_TAX`

Returns Decimal()

classmethod `get_issuer_country_code``()`

Gets issuers country. Simply returns `settings.PLANS_TAX_COUNTRY`

Returns unicode

classmethod `get_tax_rate`(*tax_id*, *country_code*)

Methods

Parameters

- **`tax_id`** – customer tax id
- **`country_code`** – customer country in ISO 2-letters format

Returns Decimal()

Django-plans application is shipped with some default taxation policies. You can choose them via settings-PLANS_TAXATION_POLICY variable.

`EUTaxationPolicy`

class `plans.taxation.eu.`**`EUTaxationPolicy`**

This taxation policy should be correct for all EU countries. It uses following rules:

- if issuer country is not in EU - assert error,
- for buyer of the same country as issuer - return issuer tax,
- for company buyer from EU (with VIES) returns VAT n/a reverse charge,

- for non-company buyer from EU returns VAT from buyer country,
- for non-EU buyer return VAT n/a.

This taxation policy was updated at 1 Jan 2015 after new UE VAT regulations. You should also probably register in MOSS system.

Note: This taxation policy requires `suds` (we use suds-jurko) and `vatnumber` python modules (connecting to VIES). If you want them automatically installed please remember to insert extra depedencies for pip:

```
$ pip install django-plans[eu]
```

RussianTaxationPolicy

FIXME: under developement

class `plans.taxation.ru.`**RussianTaxationPolicy**

FIXME: description needed

get_tax_rate(*tax_id*, *country_code*)

Invoicing

There is a built in support for creating invoices. This functionality brings powerful features like:

- invoices are linked to orders,
- invoices can have different shipping info,
- invoices can be marked as "requiring shipment"
- invoices can be previewed as HTML or PDF

Changing values of VAT tax and PLANS_INVOICE_ISSUER in a living system

Your system can be running for a while. You can have a multiple orders and you could have issued a multiple invoices already. There can be a situation that you need to change after a while a tax or your company. This can be easily done by changing those data in django settings. This will **not** affect any already created payment, order or invoice. System is designed in such way, that those information are duplicated and stored within proper object in the moment of those object creation.

After changing those settings every new order, payment, invoice will use those new values.

Warning: Remember that orders can be payed in some time window (e.g. 14 days). This mean that even if you change VAT tax rate, all your already created orders but not yet paid will have old tax. If this is what you don't want you need to cancel those orders manually and remember to contact your client that theirs orders were cancelled!

This however is not a case with `PLANS_INVOICE_ISSUER` change, because those data are taken in the same moment of issuing invoice. Even an old order will use new `PLANS_INVOICE_ISSUER` when invoicing a new payment.

Billing data

First of all you should provide a way to input a billing data by the customer. Billing data are stored as model `BillingInfo`.

class `plans.models.`**`BillingInfo`**(**args*, ***kwargs*)
Stores customer billing data needed to issue an invoice

There are four class-based views to manage deleting and adding billing data:

class `plans.views.`**`BillingInfoRedirectView`**(***kwargs*)
Checks if billing data for user exists and redirects to create or update view.

class `plans.views.`**`BillingInfoCreateView`**(***kwargs*)
Creates billing data for user

class `plans.views.`**`BillingInfoUpdateView`**(***kwargs*)
Updates billing data for user

class `plans.views.`**`BillingInfoDeleteView`**(***kwargs*)
Deletes billing data for user

Described views are pointed by following urls name patterns:

- `billing_info`,
- `billing_info_create`,
- `billing_info_update`,
- `billing_info_delete`.

Described views require creating following templates:

- `billing_info`,
- `plans/billing_info_create.html`,
- `plans/billing_info_update.html`,
- `plans/billing_info_delete.html`.

Basically you need only to manage `{{ form }}` displaying and sending within these templates.

Invoice model class

class `plans.models.`**`Invoice`**(**args*, ***kwargs*)
Single invoice document.

class `NUMBERING`
Used as a choices for settings.PLANS_INVOICE_COUNTER_RESET

`Invoice.`**`copy_from_order`**(*order*)
Filling orders details likes totals, taxes, etc and linking provided `Order` object with an invoice

Parameters **`order`** (*`Order`*) – Order object

`Invoice.`**`get_full_number`**()
Generates on the fly invoice full number from template provided by `settings.PLANS_INVOICE_NUMBER_FORMAT`. `Invoice` object is provided as `invoice` variable to the template, therefore all object fields can be used to generate full number format.

> **Warning:** This is only used to prepopulate `full_number` field on saving new invoice. To get invoice full number always use `full_number` field.

Returns string (generated full number)

`Invoice.`**`set_buyer_invoice_data`**`(`*billing_info*`)`
Fill buyer invoice billing and shipping data by copy them from provided user's `BillingInfo` object.

Parameters **`billing_info`** (`BillingInfo`) – BillingInfo object

`Invoice.`**`set_issuer_invoice_data`**`()`
Fills models object with issuer data copied from `settings.PLANS_INVOICE_ISSUER`

Raise ImproperlyConfigured

Templates

Account expiration warnings

Via the `plans.context_processors.account_status` this module allows to get information in any template about:

- user account has expired - `{{ ACCOUNT_EXPIRED }}`,
- user account is not active - `{{ ACCOUNT_NOT_ACTIVE }}`,
- user account will expire soon - `{{ EXPIRE_IN_DAYS }}`,
- an URL of account extend action - `{{ EXTEND_URL }}`,
- an URL of account activate action - `{{ ACTIVATE_URL }}`.

First you need to add a context processor to your settings, e.g.:

```
TEMPLATE_CONTEXT_PROCESSORS = global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
    'plans.context_processors.account_status',
    )
```

The context processor is defined as follows:

`plans.context_processors.`**`account_status`**`(`*request*`)`
Set following `RequestContext` variables:

- `ACCOUNT_EXPIRED = boolean`, account was expired state,
- **`ACCOUNT_NOT_ACTIVE = boolean`, set when account is not expired, but it is over quotas so it is** not active
- `EXPIRE_IN_DAYS = integer`, number of days to account expiration,
- `EXTEND_URL = string`, URL to account extend page.
- `ACTIVATE_URL = string`, URL to account activation needed if account is not active

What you might want to do now is to create a custom `expiration_messages.html` template:

```
{% load i18n %}

{% if ACCOUNT_EXPIRED %}
    <div class="messages_permanent error">
        {% blocktrans with url=EXTEND_URL %}
            Your account has expired. Please <a href="{{ url }}">extend your account</
↪a>.
        {% endblocktrans %}
    </div>
{% else %}

    {% if ACCOUNT_NOT_ACTIVE %}
        <div class="messages_permanent warning">
        {% blocktrans with url=ACTIVATE_URL %}
            Your account is not active. Possibly you are over some limits.
            Try to <a href="{{ url }}">activate your account</a>.
        {% endblocktrans %}
        </div>
    {% endif %}

    {% if EXPIRE_IN_DAYS >= 0 and EXPIRE_IN_DAYS <= 14 %}
        <div class="messages_permanent warning">
            {% blocktrans with extend_url=EXTEND_URL days_to_expire=EXPIRE_IN_DAYS %}
                Your account will expire soon (in {{ days_to_expire }} days).
                We recommend to <a href="{{ extend_url }}">extend your account now.</
↪a>
            {% endblocktrans %}
        </div>
    {% endif %}

{% endif %}
```

and put `{% include "expiration_messages.html" %}` in suitable places (for example in base template of every user logged pages). Here in template you can customize when exactly you want to display notifications (e.g. how many days before expiration).

Working with South migrations

Because this project is designed with i18n and l10n in mind it supports translating some of model fields (e.g. plans names and descriptions). This feature is implemented using django-modeltranslation. Unfortunately this approach generate models on the fly - i.e. depending on activated translations in django settings.py it generate appropriate list of translated fields for every text field marked an translatable.

This bring a problem that south migrations cannot be made for an app itself due to lack of possibility to frozen such dynamically generated model. However you can still benefit from south migrations using django plans using an apporach presented in this document. We will use a great feature of South module, which isaccesible via SOUTH_MIGRATION_MODULES setting.

This option allows you to overwrite default South migrations search path and create your own project dependent migrations in scope of your own project files. To setup custom migrations for your project follow these simple steps.

Step 1. Add SOUTH_MIGRATION_MODULES setting

You should put your custom migrations somewhere. The good place seems to be path PROJECT_ROOT/migrations/plans directory.

Note: Remember that PROJECT_ROOT/migrations/plans path should be a python module, i.e. it needs to be importable from python.

Then put the following into settings.py:

```
SOUTH_MIGRATION_MODULES = {
    'plans' : 'yourproject.migrations.plans',
}
```

Step 2. Create initial migration

From now on, everything works like standard South migrations, with the only difference that migrations are kept in scope of your project files - not plans module files.

```
$ python migrate.py schemamigration --initial plans
```

Step 3. Migrate changes on deploy

```
$ python migrate.py migrate plans
```

Step 4. Upgrading to new a version of plans

When there is a new version of django-plans, you can upgrade your module by simply using South to generate custom migration:

```
$ python migrate schemamigration --auto plans
```

and then:

```
$ python migrate.py migrate plans
```

Caveats

Problem with generic Suds client

Suds client is used by vatnumber module to query VIES system for VAT ID numbers. The problem was it was making an error (exception that NoneType does not have str method). This error was shown only with django (with possibly django-debug-toolbar enabled). As we can read here http://stackoverflow.com/questions/9664705/django-and-suds-nonetype-object-has-no-attribute-str-in-suds this is a bug in Suds that is caused by some logging problem. In console this bug fails silently, but when called from django make an Exception.

As stackoverflow answers, the solution is to use fixed version of Suds that unfortunately is not in PyPi. Working suds version can be clone from: https://github.com/cypreess/suds-htj.git

Version: 0.4.1-htj is reported to be working.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Developing

Project leader:

- Krzysztof Dorosz <cypreess@gmail.com>

Contributors:

- Victor Safronovich <vsafronovich@gmail.com>
- Dominik Kozaczko <http://dominik.kozaczko.info>

Source code: https://github.com/cypreess/django-plans

You are very welcome to join the development team of django-plans. Contribution via github fork and pull requests. Here are some ideas what is needed:

- more tests,
- more precise documentation,
- documentation proofreading and copyediting,
- taxation backends for you country/area,
- translations,
- other then default change plan policies.

Index

A

account_status() (in module plans.context_processors), 20
attribute (plans.validators.ModelAttributeValidator attribute), 15

B

BillingInfo (class in plans.models), 19
BillingInfoCreateView (class in plans.views), 19
BillingInfoDeleteView (class in plans.views), 19
BillingInfoRedirectView (class in plans.views), 19
BillingInfoUpdateView (class in plans.views), 19

C

check_attribute_value() (plans.validators.ModelAttributeValidator method), 15
code (plans.validators.QuotaValidator attribute), 14
copy_from_order() (plans.models.Invoice method), 19

D

default_quota_value (plans.validators.QuotaValidator attribute), 14

E

EUTaxationPolicy (class in plans.taxation.eu), 17

G

get_change_price() (plans.plan_change.PlanChangePolicy method), 16
get_default_tax() (plans.taxation.TaxationPolicy class method), 17
get_error_message() (plans.validators.ModelAttributeValidator method), 15
get_error_message() (plans.validators.ModelCountValidator method), 14
get_error_message() (plans.validators.QuotaValidator method), 14
get_full_number() (plans.models.Invoice method), 19
get_issuer_country_code() (plans.taxation.TaxationPolicy class method), 17
get_queryset() (plans.validators.ModelCountValidator method), 14
get_quota_value() (plans.validators.QuotaValidator method), 14
get_tax_rate() (plans.taxation.ru.RussianTaxationPolicy method), 18
get_tax_rate() (plans.taxation.TaxationPolicy class method), 17

I

Invoice (class in plans.models), 19
Invoice.NUMBERING (class in plans.models), 19

M

model (plans.validators.ModelCountValidator attribute), 14
ModelAttributeValidator (class in plans.validators), 15
ModelCountValidator (class in plans.validators), 14

O

on_activation() (plans.validators.QuotaValidator method), 14

P

PlanChangePolicy (class in plans.plan_change), 16

Q

QuotaValidator (class in plans.validators), 14

R

required_to_activate (plans.validators.QuotaValidator attribute), 14
RussianTaxationPolicy (class in plans.taxation.ru), 18

S

set_buyer_invoice_data() (plans.models.Invoice method), 20

set_issuer_invoice_data() (plans.models.Invoice method), 20
StandardPlanChangePolicy (class in plans.plan_change), 16

T

TaxationPolicy (class in plans.taxation), 17