django-flexible-subscriptions Documentation

Release 0.15.1

Joshua Robert Torrance

Sep 11, 2020

Contents

Django Flexible Subscriptions provides subscription and recurrent billing for Django applications. Any payment provider can be quickly added by overriding the placeholder functions. You can view the source code on GitHub.

CHAPTER 1

Getting started

1.1 Installation and Setup

1.1.1 Install django-flexible-subscriptions and its dependencies

Install `django-flexible-subscriptions` (which will install Django as a dependency). It is strongly recommended you use a virtual environment for your projects. For example, you can do this easily with Pipenv:

```
$ pipenv install django-flexible-subscriptions
```

1.1.2 Add django-flexible-subscriptions to your project

1. Update `django-flexible-subscriptions` to your settings file. While not mandatory, it is very likely you will also want to include the `django.contrib.auth` and `django.contrib.admin` apps as well (see Understanding a Description Plan for details).

```
INSTALLED_APPS = [
    # Django applications
    'django.contrib.auth',
    'django.contrib.admin',
    ...
    # Your third party applications
    'subscriptions',
    ...
]
```

2. Run the package migrations:

```
$ pipenv run python manage.py migrate
```

3. Add the `django-flexible-subscriptions` URLs to your project:

```
import subscriptions

from django.contrib import admin # Optional, but recommended
from django.urls import include, urls


urlpatterns = [
    ...
    path('subscriptions/', include('subscriptions.urls')),
    path('admin/', include(admin.site.urls), # Optional, but recommended
    ...
]
```

4. You can test that the project is properly setup by running the server (`pipenv run python manage.py runserver`) and visiting `http://127.0.0.1:8000/subscriptions/subscribe/`.

1.2 Configuration

While not required, you are able to customize aspects of Django Flexible Subscriptions in your settings file. At a minimum, you will probably want to set the following settings:

```
# Set your currency type
DFS_CURRENCY_LOCALE = 'en_us'

# Specify your base template file
DFS_BASE_TEMPLATE = 'base.html'
```

A full list of settings and their effects can be found in the *settings documentation*.

1.3 Understanding a Subscription Plan

Django Flexible Subscriptions uses a `Plan` model to describe a subscription plan. A `Plan` describes both billing details and user permissions granted.

User permissions are dictacted by the Django `Group` model, which is included as part of the authentication system. Django Flexible Subscriptions will add or remove a `Group` from a `User` based on the status of the user subscription. You may specify the permissions the `User` is granted by associating them to that Group and running any permission checks as needed. See the Django documenation on "User authentication in Django" for more details. If you do not need to grant a user permissions with a subscription, you may ignore the `Group` model.

A subscription `Plan` contains the following details to dictate how it functions:

- **Plan name**: The name of the subscription plan. This will be displayed to the end user in various views.
- **Plan description**: An optional internal description to help describe or differentiate the plan for the developer. The end user does not see this.
- **Group**: The `Group` model(s) associated to this plan.
- **Tag**: Custom tags associated with this plan. Can be used to organize or categorize related plans.
- **Grade period**: The number of days a subscription will remain active for a user after a plan ends (e.g. due to non-payment).
- **Plan cost**: Describes the pricing details of the plan.

One or more `PlanCost` models may be associated to a `Plan`. This allows you to offer the same plan at difference prices depending on how often the billing occurs. This would commonly be used to offer a discounted price when the user subscribes for a longer period of time (e.g. annually instead of monthly). A `PlanCost` will contain the following details:

- **Recurrence period**: How often the plan is billed per recurrence unit.
- **Recurrence unit**: The unit of measurement for the recurrence period. `one-time`, `second`, `minute`, `hour`, `day`, `week`, `month`, and `year` are supported.
- **Cost**: The amount to charge at each recurrence period.

1.4 Setup a Subscription Plan

Once Django Flexible Subscriptions is setup and running, you will be able to add your first subscription.

Note: You will need an account with staff/admin access to proceed with the following steps. All referenced URLs assume you have added the `django-flexible-subscriptions` URLs at `/subscriptions/`.

---

1. Visit `/subscriptions/dfs/` to access the **Developer Dashboard**.

2. Click the **Subscription plans** link or visit `/subscriptions/dfs/plans/`. Click on the **Create new plan** button.

3. Fill in the plan details and click the **Save** button.

1.5 Understanding a Subscription Plan List

Django Flexible Subscriptions provides basic support to add a "Subscribe" page to your site to allow users to select a subscription plan. The plans listed on this page are controlled by the `PlanList` model. The `PlanList` model includes the following details:

- **Title**: A title to display on the page (may include HTML content).
- **Subttile**: A subtitle to display on the page (may include HTML content).
- **Header**: Content to display before the subscription plans are listed (may include HTML content).
- **Header**: Content to display after the subscription plans are listed (may include HTML content).
- **Active**: Whether this list is active or not.

Note: The first active `PlanList` instance is used to populate the subscribe page. You will need to inactivate or delete older `PlanList` instances if you want a newer one to be used.

Once a `PlanList` is created, you will be able to associate `Plan` instances to specify the following details:

- **HTML content**: How you want the plan details to be presented (may include HTML content).
- **Subscribe button text**: The text to display on the "Subscribe" button at the end of the plan description.

1.6 Creating a Plan List

Once you have created you subscription plan, you can create your `PlanList`.

1. Visit `/subscriptions/dfs/` to access the **Developer Dashboard**.
2. Click the **Plan lists** button or visit `/subscriptions/dfs/plan-lists/`. Click on the **Create a new plan list** button.
3. Fill in the plan list details and click the **Save** button.
4. To add `Plan` instances to your `PlanList` click the **Manage plans** button on the Plan Lists page.
5. Click on the **Add plan** button, fill in the desired details and click the **Save** buton.
6. You can now visit `/subscriptions/subscribe/` to see your plan list.

1.7 Next Steps

If you completed all the steps above, you should now have a working subscription system on your development server. You will likely want to add payment handling and a task runner to automate subscription renewals and expiries. Instructions and examples for this can be found the *Advanced usage* section.

1.8 Considerations for Production

When moving Django Flexible Subscriptions to a production environment, you will probably want to consider the following:

- `django-flexible-subscriptions` comes with its own `styles.css` file - you will need to ensure you run the `collectstatic` management command if you have not overriden it with your own file.
- The `SubscribeView` included with `django-flexible-subscriptions` is intended to be extended to implement payment processing. The base view will automatically approve all payment requests and should be overriden if this is not the desired behaviour.
- `django-flexible-subscriptions` includes management commands to assist with managing subscription renewals and expiries. While these can be ran manually, you should consider implementing some task manager, such as `cron` or `celery`, to run these commands on a regular basis.

CHAPTER 2

Settings

Below is a comprehensive list of all the settings for Django Flexible Subscriptions.

2.1 Admin Settings

These are settings to control aspects of the Django admin support.

2.1.1 DFS_ENABLE_ADMIN

Required: `False`

Default (boolean): `False`

Whether to enable the Django Admin views or not.

2.2 Currency Settings

These are the settings to control aspects of currency repsentation.

2.2.1 DFS_CURRENCY

Required: `False`

Default (string): `en_us`

The currency to use for currency formating. You may either specify a `str` value for the language code you want to use or a `dict` value that declares all the required monetary conventions.

The following `str` values are available:

- `de_de` (Germany, German)

- `en_au` (Australia, English)
- `en_ca` (Canada, English)
- `en_us` (United States of America, English)
- `fa_ir` (Iran, Persian)
- `fr_ca` (Canada, French)
- `fr_ch` (Swiss Confederation, French)
- `fr_fr` (France, French)
- `it_it` (Itality, Italian)
- `pl_pl` (Republic of Poland, Polish)
- `pt_br` (Federative Republic of Brazil, Portuguese)
- `en_in` (India, English)
- `en_ph` (Philippines, English)

Additional values can be added by submitting a pull request with the details added to the `CURRENCY` dictionary in the `subscriptions.currency` module.

To specify a custom format, you can specify the following details in a dictionary:

- `currency_symbol` (`str`): The symbol used for this currency.
- `int_currency_symbol` (`str`): The symbol used for this currency for international formatting.
- `p_cs_precedes` (`bool`): Whether the currency symbol precedes positive values.
- `n_cs_precedes` (`bool`): Whether the currency symbol precedes negative values.
- `p_sep_by_space` (`bool`): Whether the currency symbol is separated from positive values by a space.
- `n_sep_by_space` (`bool`): Whether the currency symbol is separated from negative values by a space.
- `mon_decimal_point` (`str`): The character used for decimal points.
- `mon_thousands_sep` (`str`): The character used for separating groups of numbers.
- `mon_grouping` (`int`): The number of digits per groups.
- `frac_digits` (`int`): The number of digits following the decimal place. Use 0 if this is a non-decimal currency.
- `int_frac_digits` (`str`): The number of digits following the decimal place for international formatting. Use 0 if this is a non-decimal currency.
- `positive_sign` (`str`): The symbol to use for the positive sign.
- `negative_sign` (`str`): The symbol to use for the negative sign.
- `p_sign_posn` (`str`): How the positive sign should be positioned relative to the currency symbol and value (see below).
- `n_sign_posn` (`str`): How the positive sign should be positioned relative to the currency symbol and value (see below).

The sign positions (`p_sign_posn` and `p_sign_posn`) use the following values:

- `0`: Currency and value are surrounded by parentheses.
- `1`: The sign should precede the value and currency symbol.
- `2`: The sign should follow the value and currency symbol.

- `3`: The sign should immediately precede the value.
- `4`: The sign should immediately follow the value.

2.2.2 DFS_CURRENCY_LOCALE

Deprecated - use *DFS_CURRENCY* instead.

2.3 View & Template Settings

These control various aspects of HTML templates and Django views.

2.3.1 DFS_BASE_TEMPLATE

Required: `False`

Default (string): `subscriptions/base.html`

Path to an HTML template that is the 'base' template for the site. This allows you to easily specify the main site design for the provided Django Flexible Subscription views. The template must include a `content` block, which is what all the templates override.

2.3.2 DFS_SUBSCRIBE_VIEW

Required: `False`

Default (string): `subscriptions.views.SubscribeView`

The path to the SubscribeView to use with `django-flexible-subscriptions`. This will generally be set to a class view the inherits from `SubscribeView` to allow customization of payment and subscription processing.

2.4 View & Template Settings

These control various aspects of the management commands.

2.4.1 DFS_MANAGER_CLASS

Required: `False`

Default (string): `subscriptions.management.commands._manager.Manager`

The path to the `Manager` object to use with the management commands. This will generally be set to a class that inherits from the `django-flexible-subscriptions` `Manager` class to allow customization of renewal billings and user notifications.

CHAPTER 3

Advanced usage

3.1 Changing styles and templates

It is possible to override any component of the user interface by overriding the style file or the templates. To override a file, simply create a file with the same path noted in the list below.

It is also possible to setup your `django-flexible-subscriptions` to use a base template already in your project via your settings. See the *settings* section for more details.

- **Core Files and Templates**
  - `static/subscriptions/styles.css` (controls all template styles)
  - `templates/subscriptions/base.html` (base template that all templates inherit from)
  - `templates/subscriptions/subscribe_list.html` (user-facing; list and sign up for subscription plans)
  - `templates/subscriptions/subscribe_preview.html` (user-facing; preview of subscription plan signup)
  - `templates/subscriptions/subscribe_confirmation.html` (user-facing; confirmation of subscription plan signup)
  - `templates/subscriptions/subscribe_thank_you.html` (user-facing; thank you page on successful subscription plan singup)
  - `templates/subscriptions/subscribe_user_list.html` (user-facing; list of a user's subscriptions)
  - `templates/subscriptions/subscribe_cancel.html` (user-facing; confirm cancellation of subscription)
- **Developer-Facing Templates**
  - `templates/subscriptions/base_developer.html` (base template that all developer dashboard templates inherit from)
  - `templates/subscriptions/dashboard.html` (developer-facing; dashboard template)

- `templates/subscriptions/plan_list.html` (developer-facing; list of all subscription plans)
- `templates/subscriptions/plan_create.html` (developer-facing; create subscription plan)
- `templates/subscriptions/plan_update.html` (developer-facing; update subscription plan)
- `templates/subscriptions/plan_delete.html` (developer-facing; delete subscription plan)
- `templates/subscriptions/plan_list_list.html` (developer-facing; list of all plan lists)
- `templates/subscriptions/plan_list_create.html` (developer-facing; create new plan list)
- `templates/subscriptions/plan_list_update.html` (developer-facing; update plan list)
- `templates/subscriptions/plan_list_delete.html` (developer-facing; delete plan list)
- `templates/subscriptions/plan_list_detail_list.html` (developer-facing; list of plan list details)
- `templates/subscriptions/plan_list_detail_create.html` (developer-facing; create new plan list detail)
- `templates/subscriptions/plan_list_detail_update.html` (developer-facing; update plan list detail)
- `templates/subscriptions/plan_list_detail_delete.html` (developer-facing; delete plan list detail)
- `templates/subscriptions/subscription_list.html` (developer-facing; list all user's subscription plans)
- `templates/subscriptions/subscription_create.html` (developer-facing; create new subscription plan for user)
- `templates/subscriptions/subscription_update.html` (developer-facing; update subscription plan for user)
- `templates/subscriptions/subscription_delete.html` (developer-facing; delete subscription plan for user)
- `templates/subscriptions/tag_list.html` (developer-facing; list of tags)
- `templates/subscriptions/tag_create.html` (developer-facing; create new tag)
- `templates/subscriptions/tag_update.html` (developer-facing; update tag)
- `templates/subscriptions/tag_delete.html` (developer-facing; delete tag)
- `templates/subscriptions/transaction_list.html` (developer-facing; list of transactions)
- `templates/subscriptions/tag_detail.html` (developer-facing; details of a single transaction)

3.2 Adding a currency

Currently currencies are controlled by the `CURRENCY` dictionary in the `conf.py` file. New currencies can be added by making a pull request with the desired details. A future update will allow specifying currencies in the settings file.

3.3 Customizing new subscription handling

All subscriptions are handled via the `SubscribeView`. It is expected that most applications will will extend this view to implement some custom handling (e.g. payment processing). To extend this view:

1. Create a new view file (e.g. `/custom/views.py`) and extend the `Subscribe View`

```
# /custom/views.py
from subscriptions import views

class CustomSubscriptionView(views.SubscriptionView):
    pass
```

2. Update your settings file to point to the new view:

```
DFS_SUBSCRIBE_VIEW = custom.views.CustomSubscriptionView
```

From here you can override any attributes or methods to implement custom handling. A list of all attributes and methods can be found in the *package reference*.

3.3.1 Adding payment processing

To implement payment processing, you will likely want to override the `process_payment` method in `SubscribeView` (see *Customizing new subscription handling*. This method is called when a user confirms payment. The request must pass validation of form specified in the `payment_form` attribute (defaults to `PaymentForm`).

You may also need to implement a custom `PaymentForm` if you require different fields or validation than the default provided in `django-flexible-subscriptions`. You can do this by creating a new form and assigning it as value for the `payment_form` attribute of a custom `SubscribeView`:

1. Create a new view file (e.g. `/custom/forms.py`) and create a a Django form or extend the `django-flexible-subscriptions` `PaymentForm`:

```
# /custom/forms.py
from subscriptions.forms import PaymentForm

class CustomPaymentForm(PaymentForm):
    pass
```

2. Update your custom `SubscribeView` to point to your new form:

```
# custom/views.py
from custom.forms import CustomPaymentForm

class CustomSubscriptionView(views.SubscriptionView):
    payment_form = CustomPaymentForm
```

Between the PaymentForm and the SubscribeView you should be able to implement most payment providers. The exact details will depend on the payment provider you implement and is out of the scope of this documentation.

3.4 Subscription renewals and expiries

The management of subscription renewals and expiries must be handled by a task manager. Below will demonstrate this using `cron`, but any application with similar functionality should work.

3.4.1 Extending the subscription manager

First, you will likely need to customize the subscription manager. This is necessary to accomodate payment processing with the subscription renewal process. You can do this by extending the supplied `Manager` class. For example:

1. Create a custom `Manager` class:

```
# custom/manager.py
from subscriptions.management.commands import _manager

CustomManager(_manager.Manager):
    process_payment(self, *args, **kwargs):
        # Implement your payment processing here
```

2. Update your settings to point to your custom manager:

```
...
# settings.py
DFS_MANAGER_CLASS = 'custom.manager.CustomManager'
...
```

3.4.2 Running the subscription manager

Once the subscription manager is setup, you will simply need to call the management command at a regular interval of your choosing. This command can be called via:

```
$ pipenv run python manage.py process_subscriptions
> Processing subscriptions... Complete!
```

If you wanted to renew and expire subscriptions daily, you could use the following `cron` command:

```
#  Minute (0-59)
# |  Hour (0-23)
# | |  Day of Month (1-31)
# | | |  Month (1-12)
# | | | |  Day of week (0-6)
# | | | | |  cron command
# | | | | | |
  0 0 * * * /path/to/pipenv/python manage.py process_subscriptions
```

This could be implemented in other task runners in a similar fashion (e.g. Windows Task Scheduler, Celery).

CHAPTER 4

Contributing

Contributions or forking of the project is always welcome. Below will provide a quick outline of how to get setup and things to be aware of when contributing.

4.1 Reporting issues

If you simply want to report an issue, you can use the GitHub Issue page.

4.2 Setting up the development environment

This package is built using Pipenv, which will take care of both your virtual environment and package management. If needed, you can install `pipenv` through `pip`:

```
$ pip install pipenv
```

To download the repository from GitHub via `git`:

```
$ git clone git://github.com/studybuffalo/django-flexible-subscriptions.git
```

You can then install all the required dependencies by changing to the package directory and installing from `Pipfile.lock`:

```
$ cd django-flexible-subscriptions
$ pipenv install --ignore-pipfile --dev
```

Finally, you will need to build the package:

```
$ pipenv run python setup.py develop
```

You should now have a working environment that you can use to run tests and setup the sandbox demo.

4.3 Testing

All pull requests must have unit tests built and must maintain or increase code coverage. The ultimate goal is to achieve a code coverage of 100%. While this may result in some superfluous tests, it sets a good minimum baseline for test construction.

4.3.1 Testing format

All tests are built with the pytest framework (and pytest-django for Django-specific components). There are no specific requirements on number or scope of tests, but at a bare minimum there should be tests to cover all common use cases. Wherever possible, try to test the smallest component possible.

4.3.2 Running Tests

You can run all tests with the standard `pytest` command:

```
$ pipenv run py.test
```

To check test coverage, you can use the following:

```
$ pipenv run py.test --cov=subscriptions --cov-report=html
```

You may specify the output of the coverage report by changing the `--cov-report` option to `html` or `xml`.

4.3.3 Running Linters

This package makes use of two linters to improve code quality: Pylint and pycodestyle. Any GitHub pull requests must pass all Linter requirements before they will be accepted.

You may run the linters within your IDE/editor or with the following commands:

```
$ pipenv run pylint subscriptions/ sandbox/
$ pipenv run pylint tests/ --min-similarity-lines=12
$ pipenv run pycodestyle --show-source subscriptions/ sandbox/ tests/
```

Of note, tests have relaxed rules for duplicate code warnings. This is to minimize the level of abstraction that occurs within the tests with the intent to improve readability.

4.4 Updating documentation

All documentation is hosted on Read the Docs and is built using Sphinx. All the module content is automatically built from the docstrings and the sphinx-apidoc tool and the sphinxcontrib-napoleon extension.

4.4.1 Docstring Format

The docstrings of this package follow the Google Python Style Guide wherever possible. This ensures proper formatting of the documentation generated automatically by Sphinx. Additional examples can be found on the Sphinx napoleon extension documentation.

4.4.2 Building package reference documentation

The content for the Package reference is built using the `sphinx-apidoc` tool. If any files are added or removed from the `subscriptions` module you will need to rebuild the `subscriptions.rst` file for the changes to populate on Read the Docs. You can do this with the following command:

```
$ pipenv run sphinx-apidoc -fTM -o docs subscriptions subscriptions/migrations␣
↪subscriptions/urls.py subscriptions/apps.py subscriptions/admin.py
```

4.4.3 Linting documentation

If you are having issues with the ReStructuredText (reST) formatting, you can use `rst-lint` to screen for syntax errors. You can run a check on a file with the following:

```
$ pipenv run rst-lint /path/to/file.rst
```

4.5 Distributing package

Django Flexible Subscriptions is designed to be distributed with PyPI. While most contributors will not need to worry about uploading to PyPI, the following instructions list the general process in case anyone wishes to fork the repository or test out the process.

Note: It is recommended you use TestPyPI to test uploading your distribution while you are learning and seeing how things work. The following examples below will use TestPyPI as the upload target.

To generate source archives and built distributions, you can use the following:

```
$ pipenv run python setup.py sdist bdist_wheel
```

To upload the distributions, you can use the following `twine` commands:

```
$ pipenv run twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

You will need to provide a PyPI username and password before the upload will start.

CHAPTER 5

subscriptions package

5.1 Subpackages

5.1.1 subscriptions.templatetags package

Submodules

subscriptions.templatetags.currency_filters module

Template filters for Django Flexible Subscriptions.

`subscriptions.templatetags.currency_filters.`**`currency`**(*value*)
Displays value as a currency based on the provided settings.

5.2 Submodules

5.3 subscriptions.abstract module

Abstract templates for the Djanog Flexible Subscriptions app.

`class` `subscriptions.abstract.`**`CreateView`**(***kwargs*)
Bases: `django.views.generic.edit.CreateView`

Extends CreateView to specify of extensible HTML template

`template_extends`
Path to HTML template that this view extends.

Type str

`get_context_data`(***kwargs*)
Overriding get_context_data to add additional context.

template_extends = 'subscriptions/base.html'

class subscriptions.abstract.**DeleteView**(***kwargs*)
Bases: django.views.generic.edit.DeleteView

Extends DeleteView to specify of extensible HTML template

template_extends
Path to HTML template that this view extends.

Type str

get_context_data(***kwargs*)
Overriding get_context_data to add additional context.

template_extends = 'subscriptions/base.html'

class subscriptions.abstract.**DetailView**(***kwargs*)
Bases: django.views.generic.detail.DetailView

Extends DetailView to specify of extensible HTML template

template_extends
Path to HTML template that this view extends.

Type str

get_context_data(***kwargs*)
Overriding get_context_data to add additional context.

template_extends = 'subscriptions/base.html'

class subscriptions.abstract.**ListView**(***kwargs*)
Bases: django.views.generic.list.ListView

Extends ListView to specify of extensible HTML template

template_extends
Path to HTML template that this view extends.

Type str

get_context_data(**, object_list=None, **kwargs*)
Overriding get_context_data to add additional context.

template_extends = 'subscriptions/base.html'

class subscriptions.abstract.**TemplateView**(***kwargs*)
Bases: django.views.generic.base.TemplateView

Extends TemplateView to specify of extensible HTML template.

template_extends
Path to HTML template that this view extends.

Type str

get_context_data(***kwargs*)
Overriding get_context_data to add additional context.

template_extends = 'subscriptions/base.html'

class subscriptions.abstract.**UpdateView**(***kwargs*)
Bases: django.views.generic.edit.UpdateView

Extends UpdateView to specify of extensible HTML template

template_extends
Path to HTML template that this view extends.

Type str

get_context_data(***kwargs*)
Overriding get_context_data to add additional context.

template_extends = 'subscriptions/base.html'

5.4 subscriptions.conf module

Functions for general package configuration.

subscriptions.conf.**compile_settings**()
Compiles and validates all package settings and defaults.

Provides basic checks to ensure required settings are declared and applies defaults for all missing settings.

Returns All possible Django Flexible Subscriptions settings.

Return type dict

subscriptions.conf.**determine_currency_settings**()
Determines details for Currency handling.

Validates the provided currency locale setting and then returns a Currency object.

Returns a Currency object for the provided setting.

Return type obj

subscriptions.conf.**string_to_module_and_class**(*string*)
Breaks a string to a module and class name component.

subscriptions.conf.**validate_currency_settings**(*currency_locale*)
Validates provided currency settings.

Parameters

currency_locale (str or dict): a currency locale string or a dictionary defining custom currency formating conventions.

Raises

- ImproperlyConfigured – specified string currency_locale not support.
- TypeError – invalid parameter type provided.

5.5 subscriptions.forms module

Forms for Django Flexible Subscriptions.

class subscriptions.forms.**PaymentForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)
Bases: django.forms.forms.Form

Form to collect details required for payment billing.

CC_MONTHS = (('1', '01 - January'), ('2', '02 - February'), ('3', '03 - March'), ('4',

CC_YEARS = [(2020, 2020), (2021, 2021), (2022, 2022), (2023, 2023), (2024, 2024), (202

base_fields = {'address_city': <django.forms.fields.CharField object>, 'address_count

declared_fields = {'address_city': <django.forms.fields.CharField object>, 'address_c

media

class subscriptions.forms.**PlanCostForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Bases: django.forms.models.ModelForm

Form to use with inlineformset_factory and SubscriptionPlanForm.

class Meta

Bases: object

fields = ['recurrence_period', 'recurrence_unit', 'cost']

model

alias of *subscriptions.models.PlanCost*

base_fields = {'cost': <django.forms.fields.DecimalField object>, 'recurrence_period'

declared_fields = {}

media

class subscriptions.forms.**SubscriptionPlanCostForm**(**args*, ***kwargs*)

Bases: django.forms.forms.Form

Form to handle choosing a subscription plan for payment.

base_fields = {'plan_cost': <django.forms.fields.UUIDField object>}

clean_plan_cost()

Validates that UUID is valid and returns model instance.

declared_fields = {'plan_cost': <django.forms.fields.UUIDField object>}

media

class subscriptions.forms.**SubscriptionPlanForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Bases: django.forms.models.ModelForm

Model Form for SubscriptionPlan model.

class Meta

Bases: object

fields = ['plan_name', 'plan_description', 'group', 'tags', 'grace_period']

model
alias of *subscriptions.models.SubscriptionPlan*

base_fields = {'grace_period': <django.forms.fields.IntegerField object>, 'group': <

declared_fields = {}

media

subscriptions.forms.**assemble_cc_years**()
Creates a list of the next 60 years.

5.6 subscriptions.models module

Models for the Flexible Subscriptions app.

class subscriptions.models.**PlanCost**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Cost and frequency of billing for a plan.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

cost
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

display_billing_frequency_text
Generates human-readable billing frequency.

display_recurrent_unit_text
Converts recurrence_unit integer to text.

get_recurrence_unit_display(**, field=<django.db.models.fields.CharField: recurrence_unit>*)

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

next_billing_datetime(*current*)
Calculates next billing date for provided datetime.

Parameters **current** (*datetime*) – The current datetime to compare against.

Returns The next time billing will be due.

Return type datetime

objects = <django.db.models.manager.Manager object>

plan
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

plan_id

recurrence_period
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

recurrence_unit
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

slug
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

subscriptions
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

transactions
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class subscriptions.models.**PlanList**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Model to record details of a display list of SubscriptionPlans.

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

active
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

footer
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

header
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

plan_list_details
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

slug
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

subtitle
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

title
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class subscriptions.models.**PlanListDetail**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Model to add additional details to plans when part of PlanList.

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

html_content
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

order
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

plan
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

plan_id

plan_list
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

plan_list_id

subscribe_button_text
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class subscriptions.models.**PlanTag**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

A tag for a subscription plan.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

plans
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

tag
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class subscriptions.models.**SubscriptionPlan**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Details for a subscription plan.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

costs
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

display_tags()
Displays tags as a string (truncates if more than 3).

grace_period
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

group
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Child.parent is a ForwardManyToOneDescriptor instance.

group_id

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

plan_description
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

plan_list_details
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

plan_name
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

slug
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

tags
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class subscriptions.models.**SubscriptionTransaction**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Details for a subscription plan billing.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

amount
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_transaction
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

get_next_by_date_transaction(***, *field=<django.db.models.fields.DateTimeField: date_transaction>*, *is_next=True*, ***kwargs*)

get_previous_by_date_transaction(***, *field=<django.db.models.fields.DateTimeField: date_transaction>*, *is_next=False*, ***kwargs*)

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

subscription
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

subscription_id

user

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

user_id

class subscriptions.models.**UserSubscription**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

Details of a user's specific subscription.

exception DoesNotExist

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `django.core.exceptions.MultipleObjectsReturned`

active

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

cancelled

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_billing_end

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_billing_last

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_billing_next

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_billing_start

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

subscription
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

subscription_id

user
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

user_id

5.7 subscriptions.management.commands._manager module

Utility/helper functions for Django Flexible Subscriptions.

class subscriptions.management.commands._manager.**Manager**
Bases: object

Manager object to help manage subscriptions & billing.

notify_expired(*subscription*)
Sends notification of expired subscription.

Parameters **subscription** (*obj*) – A UserSubscription instance.

notify_new(*subscription*)
Sends notification of newly active subscription

Parameters **subscription** (*obj*) – A UserSubscription instance.

notify_payment_error(*subscription*)
Sends notification of a payment error

Parameters **subscription** (*obj*) – A UserSubscription instance.

notify_payment_success(*subscription*)
Sends notifiation of a payment success

Parameters **subscription** (*obj*) – A UserSubscription instance.

process_due(*subscription*)
Handles processing of a due subscription.

Parameters **subscription** (*obj*) – A UserSubscription instance.

process_expired(*subscription*)
Handles processing of expired/cancelled subscriptions.

Parameters **subscription** (*obj*) – A UserSubscription instance.

process_new(*subscription*)
Handles processing of a new subscription.

Parameters **subscription** (*obj*) – A UserSubscription instance.

process_payment(**args*, ***kwargs*)
Processes payment and confirms if payment is accepted.

This method needs to be overriden in a project to handle payment processing with the appropriate payment provider.

Can return value that evalutes to `True` to indicate payment success and any value that evalutes to `False` to indicate payment error.

process_subscriptions()
Calls all required subscription processing functions.

static record_transaction(*subscription*, *transaction_date=None*)
Records transaction details in SubscriptionTransaction.

Parameters

- **subscription** (*obj*) – A UserSubscription object.
- **transaction_date** (*obj*) – A DateTime object of when payment occurred (defaults to current datetime if none provided).

Returns The created SubscriptionTransaction instance.

Return type obj

retrieve_transaction_date(*payment*)
Returns the transaction date from provided payment details.

Method should be overriden to accomodate the implemented payment processing if a more accurate datetime is required.

Returns obj: The current datetime.

5.8 subscriptions.views module

Views for the Flexible Subscriptions app.

class subscriptions.views.**DashboardView**(***kwargs*)
Bases: `django.contrib.auth.mixins.PermissionRequiredMixin`, *subscriptions.abstract.TemplateView*

Dashboard view to manage subscription details.

permission_required = 'subscriptions.subscriptions'

raise_exception = True

template_name = 'subscriptions/dashboard.html'

class subscriptions.views.**PlanCreateView**(***kwargs*)
Bases: `django.contrib.auth.mixins.PermissionRequiredMixin`, *subscriptions.abstract.CreateView*

View to create a new subscription plan.

View is extended to handle additional attributes noted below.

success_message
Message to display on successful creation.

Type str

success_url
URL to redirect to on successful creation.

Type str

context_object_name = 'plan'

form_class
alias of *subscriptions.forms.SubscriptionPlanForm*

form_invalid(*form*, *cost_forms*)
Handles re-rendering invalid forms with errors.

Parameters

- **form** (*obj*) – Parent SubscriptionPlanForm instance to return.
- **cost_forms** (*obj*) – PlanCostFormSet instance to return.

Returns Renders original page with form content.

Return type obj

form_valid(*form*, *cost_forms*)
Handles processing of valid forms.

Parameters

- **form** (*obj*) – Parent SubscriptionPlanForm instance to process.
- **cost_forms** (*obj*) – PlanCostFormSet instance to process.

Returns HttpResponseRedirect object to `success_url`.

Return type obj

get(*request*, **args*, ***kwargs*)
Overriding get method to handle inline formset.

model
alias of *subscriptions.models.SubscriptionPlan*

permission_required = 'subscriptions.subscriptions'

post(*request*, **args*, ***kwargs*)
Overriding post method to handle inline formsets.

raise_exception = True

success_message = 'Subscription plan successfully added'

success_url = '/dfs/plans/'

template_name = 'subscriptions/plan_create.html'

class subscriptions.views.**PlanDeleteView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.DeleteView*

View to delete a subscription plan.

View is extended to handle additional attributes noted below.

success_message
Message to display on successful creation.

Type str

success_url
URL to redirect to on successful creation.

Type str

context_object_name = 'plan'

delete(*request*, **args*, ***kwargs*)
Override delete to allow success message to be added.

model
alias of *subscriptions.models.SubscriptionPlan*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'plan_id'

raise_exception = True

success_message = 'Subscription plan successfully deleted'

success_url = '/dfs/plans/'

template_name = 'subscriptions/plan_delete.html'

class subscriptions.views.**PlanListCreateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.contrib.messages.views.SuccessMessageMixin, *subscriptions.abstract.CreateView*

View to create a new plan list.

context_object_name = 'plan_list'

fields = ['title', 'subtitle', 'header', 'footer', 'active']

model
alias of *subscriptions.models.PlanList*

permission_required = 'subscriptions.subscriptions'

raise_exception = True

success_message = 'Plan list successfully added'

success_url = '/dfs/plan-lists/'

template_name = 'subscriptions/plan_list_create.html'

class subscriptions.views.**PlanListDeleteView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.DeleteView*

View to delete a plan list.

View is extended to handle additional attributes noted below.

success_message
Message to display on successful deletion.

Type str

success_url
URL to redirect to on successful deletion.

Type str

context_object_name = 'plan_list'

delete(*request*, **args*, ***kwargs*)
Override delete to allow success message to be added.

model
alias of *subscriptions.models.PlanList*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'plan_list_id'

raise_exception = True

success_message = 'Plan list successfully deleted'

success_url = '/dfs/plan-lists/'

template_name = 'subscriptions/plan_list_delete.html'

class subscriptions.views.**PlanListDetailCreateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.contrib.messages.views.SuccessMessageMixin, *subscriptions.abstract.CreateView*

View to create a new plan list.

fields = ['plan', 'plan_list', 'html_content', 'subscribe_button_text', 'order']

get_context_data(***kwargs*)
Extend context to include the parent PlanList object.

get_success_url()
Return the URL to redirect to after processing a valid form.

model
alias of *subscriptions.models.PlanListDetail*

permission_required = 'subscriptions.subscriptions'

raise_exception = True

success_message = 'Subscription plan successfully added to plan list'

template_name = 'subscriptions/plan_list_detail_create.html'

class subscriptions.views.**PlanListDetailDeleteView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.DeleteView*

View to delete a plan list detail.

View is extended to handle additional attributes noted below.

success_message
Message to display on successful deletion.

Type str

success_url
URL to redirect to on successful deletion.

Type str

context_object_name = 'plan_list_detail'

delete(*request*, **args*, ***kwargs*)
Override delete to allow success message to be added.

get_context_data(***kwargs*)
Extend context to include the parent PlanList object.

get_success_url()

model
alias of *subscriptions.models.PlanListDetail*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'plan_list_detail_id'

raise_exception = True

success_message = 'Subscription plan successfully removed from plan list'

template_name = 'subscriptions/plan_list_detail_delete.html'

class subscriptions.views.**PlanListDetailListView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.DetailView*

List of plan lists.

context_object_name = 'plan_list'

model
alias of *subscriptions.models.PlanList*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'plan_list_id'

raise_exception = True

template_name = 'subscriptions/plan_list_detail_list.html'

class subscriptions.views.**PlanListDetailUpdateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.contrib.messages.views.SuccessMessageMixin, *subscriptions.abstract.UpdateView*

View to update the details of a plan list detail.

fields = ['plan', 'plan_list', 'html_content', 'subscribe_button_text', 'order']

get_context_data(***kwargs*)
Extend context to include the parent PlanList object.

get_success_url()
Return the URL to redirect to after processing a valid form.

model
alias of *subscriptions.models.PlanListDetail*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'plan_list_detail_id'

raise_exception = True

success_message = 'Plan list details successfully updated'

template_name = 'subscriptions/plan_list_detail_update.html'

class subscriptions.views.**PlanListListView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.ListView*

List of plan lists.

context_object_name = 'plan_lists'

model
alias of *subscriptions.models.PlanList*

permission_required = 'subscriptions.subscriptions'

raise_exception = True

template_name = 'subscriptions/plan_list_list.html'

class subscriptions.views.**PlanListUpdateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.contrib.messages.views.SuccessMessageMixin, *subscriptions.abstract.UpdateView*

View to update the details of a plan list.

context_object_name = 'plan_list'

fields = ['title', 'subtitle', 'header', 'footer', 'active']

model
alias of *subscriptions.models.PlanList*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'plan_list_id'

raise_exception = True

success_message = 'Plan list successfully updated'

success_url = '/dfs/plan-lists/'

template_name = 'subscriptions/plan_list_update.html'

class subscriptions.views.**PlanListView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.ListView*

List of all subscription plans.

context_object_name = 'plans'

model
alias of *subscriptions.models.SubscriptionPlan*

permission_required = 'subscriptions.subscriptions'

raise_exception = True

template_name = 'subscriptions/plan_list.html'

class subscriptions.views.**PlanUpdateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.UpdateView*

View to update a subscription plan.

View is extended to handle additional attributes noted below.

success_message
Message to display on successful creation.

Type str

success_url
URL to redirect to on successful creation.

Type str

context_object_name = 'plan'

form_class
alias of *subscriptions.forms.SubscriptionPlanForm*

form_invalid(*form*, *cost_forms*)
Handles re-rendering invalid forms with errors.

Parameters

- **form** (*obj*) – Parent SubscriptionPlanForm instance to return.
- **cost_forms** (*obj*) – PlanCostFormSet instance to return.

Returns Renders original page with form content.

Return type obj

form_valid(*form*, *cost_forms*)
Handles processing of valid forms.

Parameters

- **form** (*obj*) – Parent SubscriptionPlanForm instance to process.
- **cost_forms** (*obj*) – PlanCostFormSet instance to process.

Returns HttpResponseRedirect object to `success_url`.

Return type obj

get(*request*, **args*, ***kwargs*)
Overriding get method to handle inline formset.

model
alias of *subscriptions.models.SubscriptionPlan*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'plan_id'

post(*request*, **args*, ***kwargs*)
Overriding post method to handle inline formsets.

raise_exception = True

success_message = 'Subscription plan successfully updated'

success_url = '/dfs/plans/'

template_name = 'subscriptions/plan_update.html'

class subscriptions.views.**SubscribeCancelView**(***kwargs*)
Bases: `django.contrib.auth.mixins.LoginRequiredMixin`, *subscriptions.abstract.DetailView*

View to handle cancelling of subscription.

View is extended to handle additional attributes noted below.

success_message
Message to display on successful creation.

Type str

success_url
URL to redirect to on successful creation.

Type str

context_object_name = 'subscription'

get_object(*queryset=None*)
Overrides get_object to restrict to logged in user.

get_success_url()
Returns the success URL.

model
alias of *subscriptions.models.UserSubscription*

pk_url_kwarg = 'subscription_id'

post(*request*, **args*, ***kwargs*)
Updates a subscription's details to cancel it.

success_message = 'Subscription successfully cancelled'

success_url = 'dfs_subscribe_user_list'

template_name = 'subscriptions/subscribe_cancel.html'

class subscriptions.views.**SubscribeList**(***kwargs*)
Bases: *subscriptions.abstract.TemplateView*

Detail view of the first active PlanList instance.

View is designed to be the user-facing subscription list and customizable through the PlanList and PlanListDetail models.

context_object_name = 'plan_list'

get(*request*, **args*, ***kwargs*)
Ensures content is available to display, then returns page.

get_context_data(***kwargs*)
Extend context to include the parent PlanList object.

template_name = 'subscriptions/subscribe_list.html'

class subscriptions.views.**SubscribeThankYouView**(***kwargs*)
Bases: django.contrib.auth.mixins.LoginRequiredMixin, *subscriptions.abstract.TemplateView*

A thank you page and summary for a new subscription.

context_object_name = 'transaction'

get_context_data(***kwargs*)
Overriding get_context_data to add additional context.

get_object()
Returns the provided transaction instance.

template_name = 'subscriptions/subscribe_thank_you.html'

class subscriptions.views.**SubscribeUserList**(***kwargs*)
Bases: django.contrib.auth.mixins.LoginRequiredMixin, *subscriptions.abstract.ListView*

List of all a user's subscriptions.

context_object_name = 'subscriptions'

get_queryset()
Overrides get_queryset to restrict list to logged in user.

model
alias of *subscriptions.models.UserSubscription*

template_name = 'subscriptions/subscribe_user_list.html'

class subscriptions.views.**SubscribeView**(***kwargs*)
Bases: django.contrib.auth.mixins.LoginRequiredMixin, *subscriptions.abstract.TemplateView*

View to handle all aspects of the subscribing process.

This view will need to be subclassed and some methods overridden to implement the payment solution.

Additionally, this view is extended from a TemplateView with the additional attributes noted below.

payment_form
Django Form to handle subscription payment.

Type obj

subscription_plan
A SubscriptionPlan instance. Will be set by methods during processing.

Type obj

success_url
URL to redirect to on successful creation.

Type str

template_preview
Path to HTML template for the preview view.

Type str

template_confirmation
Path to HTML template for the confirmation view.

Type str

Notes

View only accessible via POST requests. The request must include an ID to a SubscriptionPlan +/- associated PlanCost instance (if past the preview view).

confirmation = False

get(*request*, **args*, ***kwargs*)
Returns 404 error as this method is not implemented.

get_context_data(***kwargs*)
Overriding get_context_data to add additional context.

get_object()
Gets the subscription plan object.

get_success_url(***kwargs*)
Returns the success URL.

get_template_names()
Returns the proper template name based on payment stage.

hide_form(*form*)
Replaces form widgets with hidden inputs.

Parameters **form** (*obj*) – A form instance.

Returns The modified form instance.

Return type obj

payment_form
alias of *subscriptions.forms.PaymentForm*

post(*request*, **args*, ***kwargs*)
Handles all POST requests to the SubscribeView.

The 'action' POST argument is used to determine which context to render.

Notes

The `action` POST parameter determines what stage to progress view to. `None` directs to preview processing, `confirm` directs to confirmation processing, and `process` directs to payment and subscription processing.

process_payment(**args*, ***kwargs*)
Processes payment and confirms if payment is accepted.

This method needs to be overriden in a project to handle payment processing with the appropriate payment provider.

Can return value that evalutes to `True` to indicate payment success and any value that evalutes to `False` to indicate payment error.

process_subscription(*request*, ***kwargs*)
Moves forward with payment & subscription processing.

If forms are invalid will move back to confirmation page for user to correct errors.

record_transaction(*subscription*, *transaction_date=None*)
Records transaction details in SubscriptionTransaction.

Parameters

- **subscription** (*obj*) – A UserSubscription object.
- **transaction_date** (*obj*) – A DateTime object of when payment occurred (defaults to current datetime if none provided).

Returns The created SubscriptionTransaction instance.

Return type obj

render_confirmation(*request*, ***kwargs*)
Renders a confirmation page before processing payment.

If forms are invalid will return to preview view for user to correct errors.

render_preview(*request*, ***kwargs*)
Renders preview of subscription and collect payment details.

retrieve_transaction_date(*payment*)
Returns the transaction date from provided payment details.

Method should be overriden to accomodate the implemented payment processing if a more accurate datetime is required.

Returns obj: The current datetime.

setup_subscription(*request_user*, *plan_cost*)
Adds subscription to user and adds them to required group.

Parameters

- **request_user** (*obj*) – A Django user instance.
- **plan_cost** (*obj*) – A PlanCost instance.

Returns The newly created UserSubscription instance.

Return type obj

subscription_plan = None

success_url = 'dfs_subscribe_thank_you'

template_confirmation = 'subscriptions/subscribe_confirmation.html'

template_preview = 'subscriptions/subscribe_preview.html'

class subscriptions.views.**SubscriptionCreateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.contrib.messages.views.SuccessMessageMixin, *subscriptions.abstract.CreateView*

View to create a new user subscription.

context_object_name = 'subscription'

fields = ['user', 'subscription', 'date_billing_start', 'date_billing_end']

model
alias of *subscriptions.models.UserSubscription*

permission_required = 'subscriptions.subscriptions'

raise_exception = True

success_message = 'User subscription successfully added'

success_url = '/dfs/subscriptions/'

template_name = 'subscriptions/subscription_create.html'

class subscriptions.views.**SubscriptionDeleteView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.DeleteView*

View to delete a user subscription.

View is extended to handle additional attributes noted below.

success_message
Message to display on successful creation.

Type str

success_url
URL to redirect to on successful creation.

Type str

context_object_name = 'subscription'

delete(*request*, **args*, ***kwargs*)
Override delete to allow success message to be added.

model
alias of *subscriptions.models.UserSubscription*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'subscription_id'

raise_exception = True

success_message = 'User subscription successfully deleted'

success_url = '/dfs/subscriptions/'

template_name = 'subscriptions/subscription_delete.html'

class subscriptions.views.**SubscriptionListView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.ListView*

List of all subscriptions for the users

context_object_name = 'users'

model
alias of django.contrib.auth.models.User

paginate_by = 100

permission_required = 'subscriptions.subscriptions'

queryset

raise_exception = True

template_name = 'subscriptions/subscription_list.html'

class subscriptions.views.**SubscriptionUpdateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.contrib.messages.views.SuccessMessageMixin, *subscriptions.abstract.UpdateView*

View to update the details of a user subscription.

context_object_name = 'subscription'

fields = ['subscription', 'date_billing_start', 'date_billing_end', 'date_billing_last

model
alias of *subscriptions.models.UserSubscription*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'subscription_id'

raise_exception = True

success_message = 'User subscription successfully updated'

success_url = '/dfs/subscriptions/'

template_name = 'subscriptions/subscription_update.html'

class subscriptions.views.**TagCreateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.contrib.messages.views.SuccessMessageMixin, *subscriptions.abstract.CreateView*

View to create a new tag.

context_object_name = 'tag'

fields = ['tag']

model
alias of *subscriptions.models.PlanTag*

permission_required = 'subscriptions.subscriptions'

raise_exception = True

success_message = 'Tag successfully added'

success_url = '/dfs/tags/'

template_name = 'subscriptions/tag_create.html'

class subscriptions.views.**TagDeleteView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.DeleteView*

View to delete a tag.

View is extended to handle additional attributes noted below.

success_message
Message to display on successful deletion.

Type str

success_url
URL to redirect to on successful deletion.

Type str

context_object_name = 'tag'

delete(*request*, **args*, ***kwargs*)
Override delete to allow success message to be added.

model
alias of *subscriptions.models.PlanTag*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'tag_id'

raise_exception = True

success_message = 'Tag successfully deleted'

success_url = '/dfs/tags/'

template_name = 'subscriptions/tag_delete.html'

class subscriptions.views.**TagListView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.ListView*

List of all tags.

context_object_name = 'tags'

model
alias of *subscriptions.models.PlanTag*

permission_required = 'subscriptions.subscriptions'

raise_exception = True

template_name = 'subscriptions/tag_list.html'

class subscriptions.views.**TagUpdateView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.contrib.messages.views.SuccessMessageMixin, *subscriptions.abstract.UpdateView*

View to update the details of a tag.

context_object_name = 'tag'

fields = ['tag']

model
alias of *subscriptions.models.PlanTag*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'tag_id'

raise_exception = True

success_message = 'Tag successfully updated'

success_url = '/dfs/tags/'

template_name = 'subscriptions/tag_update.html'

class subscriptions.views.**TransactionDetailView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.DetailView*

Shows details of a specific subscription payment transaction.

context_object_name = 'transaction'

model
alias of *subscriptions.models.SubscriptionTransaction*

permission_required = 'subscriptions.subscriptions'

pk_url_kwarg = 'transaction_id'

raise_exception = True

template_name = 'subscriptions/transaction_detail.html'

class subscriptions.views.**TransactionListView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, *subscriptions.abstract.ListView*

List of all subscription payment transactions.

context_object_name = 'transactions'

model
alias of *subscriptions.models.SubscriptionTransaction*

paginate_by = 50

```
    permission_required = 'subscriptions.subscriptions'
    raise_exception = True
    template_name = 'subscriptions/transaction_list.html'
```

CHAPTER 6

Changelog

6.1 Version 0 (Beta)

6.1.1 0.15.1 (2020-Aug-10)

Bug Fixes

- Removing `order_by` command from the `SubscriptionListView` to prevent errors with customized user models.

6.1.2 0.15.0 (2020-Jul-22)

Feature Updates

- `DFS_CURRENCY_LOCALE` setting being deprecated in place of `DFS_CURRENCY`. This new setting allows either a language code `str` or a ‘‘dict‘ of currency formatting conventions to be passed. This is then used for subsequent currency formatting operations.
- Adding currency support for India (INR).

6.1.3 0.14.0 (2020-Jun-07)

Feature Updates

- Dropping support for Django 1.11. Various aspects of the package have been updated to leverage Django 2.2 features now (e.g. `path` for URLs).

6.1.4 0.13.0 (2020-May-23)

Feature Updates

- Adding currency support for Brazil (BRL).

6.1.5 0.12.1 (2020-May-07)

Bug Fixes

- Fixing issue with TransactionDetailView and TransactionListView where templates were referencing `SubscriptionTransaction.plan` rather than `SubscriptionTransaction.subscription.plan`.

6.1.6 0.12.0 (2020-Apr-29)

Feature Updates

- Adding currency support for France (EUR).
- Adding currency support for Italy (EUR).
- Adding currency support for Swiss Franc (CHF).

6.1.7 0.11.1 (2020-Apr-15)

Bug Fixes

- Fixed issue where management command files were not included in PyPI release.

6.1.8 0.11.0 (2020-Apr-04)

Feature Updates

- Adding currency support for Poland (PLN).

6.1.9 0.10.0 (2020-Feb-16)

Feature Updates

- Switching `ugettext_lazy` to `gettext_lazy` (this function is being depreciated in Django 4.0).
- Adding a slug field to `SubscriptionPlan`, `PlanCost`, and `PlanList` models. This will make it easier to reference specific subscription details in custom views.

6.1.10 0.9.0 (2020-Jan-15)

Feature Updates

- Adding currency support for (the Islamic Republic of) Iran.

Bug Fixes

- Fixed issues where currency display could not handle non-decimal currencies.

6.1.11 0.8.1 (2019-Dec-25)

Feature Updates

- Removes `django-environ` from development dependencies and switches functionality over to `pathlib`.

Bug Fixes

- Fixing bug with sandbox settings and Django 3.0 involving declaration of languages.
- Fixed issue where the `RecurrenceUnit` of the `PlanCost` model was trying to generate migration due to a change in the default value.

6.1.12 0.8.0 (2019-Dec-15)

Feature Updates

- Removing official support for Django 2.1 (has reach end of life).
- Removing Tox from testing. Too many conflicting issues and CI system can handle this better now.

6.1.13 0.7.0 (2019-Dec-01)

Feature Updates

- Switching `PlanCost` `recurrence_unit` to a CharField to make it more clear what the values represent.
- Adding `PlanCost` as an InlineAdmin field of `SubscriptionPlan`.

6.1.14 0.6.0 (2019-Aug-19)

Feature Updates

- Integrating subscription management utility functions into Django management commands. Documentation has been updated to explain this functionality.

6.1.15 0.5.0 (2019-Aug-18)

Bug Fixes

- Fixed issues where last billing date and end billing date were not diplaying properly when cancelling a subscription.
- Fixing the `SubscribeUserList` view to not show inactive subscriptions.

Feature Updates

- Improving styling for user-facing views and refactoring style sheet.
- Adding support for German (Germany) locale (`de_de`).

6.1.16 0.4.2 (2019-Aug-07)

Bug Fixes

- Resolving issue where subscription form would generate errors on initial display.
- Fixed bug where `PlanList` would display `SubscriptionPlan` instances without associated *PlanCost* instances, resulting in errors on subscription order preview.

Feature Updates

- Streamlining the `PlanList` - `PlanListDetail` - `SubscriptionPlan` relationship to make relationships more apparent and easier to query.
- Added `FactoryBoy` factories to help streamline future test writing.
- Added validation of `PlanCost` `UUID` in the `SubscriptionPlanCostForm` to confirm a valid UUID is provided and return the object immediately.
- Updated `PaymentForm to include validation of credit card numbers and CVV numbers and switched expiry months and years to ``ChoiceField` to ensure valid data collected.

6.1.17 0.4.1 (2019-Aug-05)

Bug Fixes

- Adding `styles.css` to package data.

6.1.18 0.4.0 (2019-Aug-05)

Feature Updates

- Adding responsive styling to all base HTML templates.
- Updating sandbox site to improve demo and testing functions.
- Breaking more template components into snippets and adding base templates to make it easier to override pages.

- Adding pagination to views to better handle long lists.
- Adding support for Django 2.2

6.1.19 0.3.2 (2019-Jul-17)

Bug Fixes

- Bug fixes with settings, sandbox site, and admin pages.

6.1.20 0.3.1 (2019-Jul-02)

Feature Updates

- Adding Australian Dollars to available currencies.

6.1.21 0.3.0 (2019-Jan-30)

Feature Updates

- Creating `PlanList` model to record group of `SubscriptionPlan` models to display on a single page for user selection.
- Creating a view and template to display the the oldest active `PlanList`.

6.1.22 0.2.1 (2018-Dec-29)

Bug Fixes

- Adding missing methods to `SubscribeView` and `Manager` to record payment transactions. Added additional method (`retrieve_transaction_date`) to help with transaction date specification. Reworked method calls around payment processing to streamline passing of arguments between functions to reduce need to override methods.
- Fixing issue in `Manager` class where the future billing date was based off the current datetime, rather than the last billed datetime.
- Adding method to update next billing datetimes for due subscriptions in the `Manager` class.
- Switching the default `success_url` for `SubscribeView` and `CancelView` to the user-specific list of their subscriptions, rather than the subscription CRUD dashboard.

6.1.23 0.2.0 (2018-Dec-28)

Feature Updates

- Switching arguments for the `process_payment` call to keyword arguments (`kwargs`).
- Allow the `SubscriptionView` class to be specified in the settings file to make overriding easier.

Bug Fixes

- Passing the PlanCostForm form into the process_payment call to allow access to the amount to bill.

6.1.24 0.1.1 (2018-Dec-28)

Bug Fixes

- Adding the `snippets` folder to the PyPI package - was not included in previous build.

6.1.25 0.1.0 (2018-Dec-26)

Feature Updates

- Initial package release.
- Allows creation of subscription plans with multiple different costs and billing frequencies.
- Provides interface to manage admin functions either via the Django admin interface or through basic CRUD views.
- Provides user views to add, view, and cancel subscriptions.
- Templates can be customized by either specifying the base HTML template and extending it or overriding templates entirely.
- Template tags available to represent currencies on required locale.
- Manager object available to integrate with a Task Scheduler to manage recurrent billings of subscriptions.
- Sandbox site added to easily test out application functionality.

Python Module Index

S

Index

D

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V