django-productline Documentation

Release 0.3

Hendrik Speidel

May 23, 2017

Contents

> "build feature-oriented product lines for django"

`django-productline` provides a basis and some conventions to develop django web-application product lines. It follows the *feature-oriented software development* (FOSD) methodology.

Products i.e. specific web applications can be generated by selecting a certain set of features from a pool.

There are many definitions of what a feature really is. In the context of `django-productline`, we will use the definition of Apel et al.:

> "A feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option"

More information about FOSD can be found here:

- An overview article on FOSD (pdf)
- The FOSD community portal
- FOSD on Wikipedia
- Podcast with Sven Apel on Software-Engineering Radio: Part1, Part2

I gave a talk on `django-productline` at FOSD Treffen 2013 in Dagstuhl (Slides).

CHAPTER 1

Motivation

With the concept of apps, Django already provides a pretty good modularization mechanism. Integrating an app into a project typically means to change multiple settings and registering the necessary urlpatterns. Some apps provide an API, so to integrate it properly additional code needs to be written.

When developing multiple django projects, you may end up doing this over and over for different projects in slight variations. Particular projects may also need specific additions and changes scattered across multiple locations in the codebase. Therefore, developing and managing multiple projects and copying with their variability can become a rather error-prone and time consuming task.

FOSD allows to encapsulate these additions and changes in features, which form a product line. Specific products can then be composed by assembling some of these feature.

The approach aims at improved reusability, traceability(where is all the code that relates to a specific feature), and automation.

At schnapptack, we use this approach for about a year to build specialized web applications for our clients and have found it to be a real productivity booster. Now, we want to iterate on the tools and scripts we have built internally and develop them in the open from now on. We are currently in the process of cleaning up our codebase and releasing it piece by piece. Also, we are planning to open source some of our core features, so other interested folks may also go product line.

CHAPTER 2

Goal

A typical django web application consists of the following:

- web server configuration of some sort
- a set of external services
- databases
- application code
- templates
- javascript code
- CSS

The goal is to be able to automatically compose entire applications i.e. all required artefacts out of a set of features. In the context of "automatical composition", project-individual integration code must eliminated. Multiple applications can then share common features and differ in others. Generated applications need to be easy to manage over the rest of their product lifecycle(further development, deployment). Also, there needs to be support for managing the products' individual configurations e.g. webserver and database configuration.

CHAPTER 3

Composition Mechanisms

To be able to compose the required artefacts for a product, django-productline makes use of multiple composition mechanisms:

- Python code is composed using featuremonkey
- django's built in composition mechanisms
  - for templates(`django.template.loaders.app_directories.Loader`)
  - and static files (`django.contrib.staticfiles.finders.AppDirectoriesFinder`)
- Templates are refined using django-overextends
- Javascript can be composed using featuremonkey.js
- CSS is composable by simple concatenation.

CHAPTER 4

Getting started

Installation

Here, setting up `django-productline` in ape's container mode is described. This way, you can testdrive or develop multiple product lines isolated from the rest of your system.

If all you need is to deploy a *single* product, you can also use `ape` in standalone mode, as described here.

First, make sure `virtualenv` is installed. On Debian/Ubuntu you can install it like so:

```
$ sudo apt-get install python-virtualenv
```

Then, create a new `ape` container environment and install django-productline and all dependencies:

```
$ wget -O - https://raw.github.com/henzk/django-productline/master/bin/install.py |␣
↪python - webapps
```

For the development version, use:

```
$ wget -O - https://raw.github.com/henzk/django-productline/master/bin/install.py |␣
↪python - --dev webapps
```

This will create a new folder called `webapps` with the following structure:

```
webapps/
    _ape/
        venv/
        activape
```

Note: this is a completely self contained installation. To get rid of everything or to start over, simply delete the `webapps` folder.

Congratulations, the installation is now complete!

SPL (software product line) containers can now be placed into the `webapps` directory. Folder `venv` contains a `virtualenv` that is isolated from the system (created with the `--no-site-packages` option). `ape` and its dependencies have been installed there. If you want to use system packages, either recreate the virtualenv without the `--no-site-packages` option and install `ape` into it or put the system packages back on `sys.path` using softlinks, `.pth` files, or path hacking.

Note: `--no-site-packages` is the default in newer versions of `virtualenv`. To use system packages the flag `--system-site-packages` needs to be specified.

To activate container mode, change into the `webapps` directory and issue the following commands:

```
$ . _ape/activape
```

Under the hood, this takes care of setting some environment variables and activating the virtualenv.

CHAPTER 5

Product Generation

Product Generation Process

Basically, product generation is a two step process (focusing on the Python side of things for now):

Product Context The product context contains the specific configuration of the product e.g. the database configuration and the hostname the product will be serving requests from. The product context is given in JSON format. Features may provide a wishlist of necessary configuration values.

Context Binding the product context is loaded from file and made available as `django_productline.context.PRODUCT_CONTEXT`. The product context is considered to be read only — so it may not be written to.

Feature Equation The list of features selected for the product in the order of composition. The feature equation is given as text file containing one feature per line.

Feature Composition After the product context has been bound, `featuremonkey` is used to compose the selected features. This results in a running django web application where introductions and refinements given by the selected features have been patched in.

The product context

the product context captures environment and configuration settings that are specific to each product, e.g. each product requires a different database configuration.

Use the context only for very specific stuff that NEEDS to be configured on a product basis.

The context is loaded from a file in json format.

Composition of application code

`featuremonkey` is used to compose Python code. It allows introductions of new structures and refinements of existing ones.

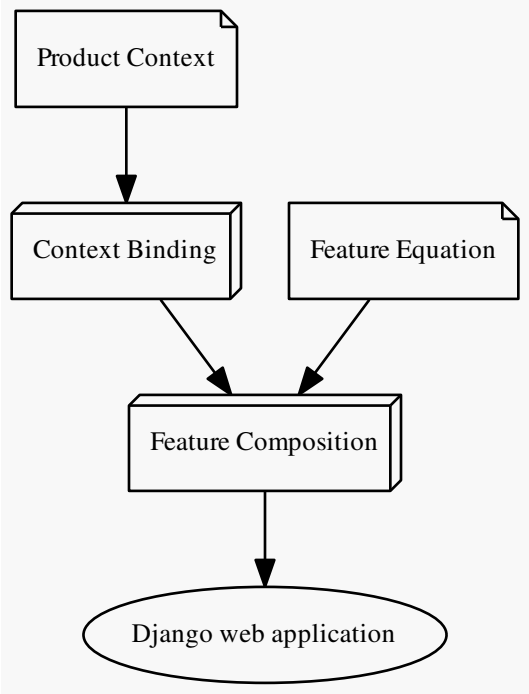


Fig. 5.1: Figure: High-Level overview of the product generation process

For some use cases in the context of `django-productline`, see *Refinements by example*.

Also, see the featuremonkey documentation.

Template composition

You can use django-overextends for feature oriented template development. It is automatically installed as a dependency of `django-productline`.

By default, Django uses the app directories template loader to locate templates. It searches the `templates` folder of each app in the order that the apps are specified in `INSTALLED_APPS`. The loader picks the first template with matching name.

Conceptually, this is a form of feature oriented composer with file level replacements:

- apps represent features
- `INSTALLED_APPS` defines the feature selection and their composition order

On top of that, `django-overextends` provides *overextension* — template block level refinements.

Example

Consider, we have a template called `mytemplate.html` in the template directory of a django app called `myfeature`:

```
myfeature/
    templates/
        mytemplate.html
        ...
    ...
```

Suppose `mytemplate.html` looks like this:

```
<html>
    <head>
        <title>{% block title %}Hello{% endblock %}</title>
    </head>
    <body>
        {% block body %}
        <h1>Hello</h1>
        {% endblock %}
    </body>
</html>
```

Django templates already provide blocks, that are used for template inheritance

`django-overextends` provides template superimposition using the `overextends` tag: To refine `mytemplate.html`, all we need to do is to create another template with that name in a django app that is placed before `myfeature` in `INSTALLED_APPS`:

```
{% overextends "mytemplate.html" %}

{% block title %}Replacement{% endblock %}

{% block body %}
{{ block.super }}
```

```
Refinements are also possible!
{% endblock %}
```

Block tags are used to annotate FST-Nodes. Since blocks can be nested, we can build feature structure trees. Nodes with the same name are superimposed, when the template is rendered. `{{ block.super }}` provides access to the original implementation.

Rendering the above example, would result roughly in the following HTML document:

```
<html>
    <head>
        <title>Replacement</title>
    </head>
    <body>
        <h1>Hello</h1>
        Refinements are also possible!
    </body>
</html>
```

Javascript Composition

If necessary, JavaScript can be composed using `featuremonkey.js`. Essentially, it works the same way as `featuremonkey`.

Have a look at the example product line and feel free to snoop around by viewing the source in your browser.

CSS Composition

feature oriented CSS is easy: concatenation is a pretty good composition mechanism for it.

Task Composition

`django-productline` relies on `ape tasks`. Features may introduce new tasks and refine existing ones by providing a `tasks` module.

Please see the ape tasks documentation for details.

Tasks contributed by `django-productline` are listed in *Available tasks*.

CHAPTER 6

Feature Documentation

django-productline Feature Documentation

`django-productline` should be used as a base feature.

It provides the basis to create feature-oriented django product lines:

- it defines the product generation process
- it provides hooks for other features to refine to add/adapt the functionality

Features

This section documents all the subfeatures django productline provides.

Multilanguage

Multilanguage is a feature which enables django's enhanced language support. For more information read the django docs:

- https://docs.djangoproject.com/en/1.11/topics/i18n/translation/#language-prefix-in-url-patterns
- https://docs.djangoproject.com/en/1.11/topics/i18n/#internationalization-and-localization

django-productline multilanguage

urlpatterns are constructed by refining django_productline.urls.get_urls.

Here, get_urls and get_multilang_urls is called to get the (composed) urlpatterns.

The i18n_patterns must be defined in the root_urlconf, therefore this refinement is necessary. This function is later called in the get_urls() of django-productline.

To make your projects urls multilanguage, you need to modify your url refinements:

```
def refine_get_multilang_urls(original):

    def get_multilang_urls():
        from django.conf.urls import url
        urlpatterns = [
            url(r'foo/$', views.FooView.as_view(), name='foo'),
            url(r'bar/$', views.BarView.as_view(), name='bar')
        ]
        return original() + urlpatterns

    return get_multilang_urls
```

If your projects has e.g. 'en' as default language and you don't want it to appear in the url, then set PREFIX_DEFAULT_LANGUAGE to False.

When refining get_urls using includes like this (in case you use standard django apps for example):

```
urlpatterns = original() + url(r'^', include('app.urls'))
```

and receive errors, you might want to access the url_patterns attribute of the include directly as i18n_patterns() expects a list of url() instances and include returns a Resolver instance.

Like this:

```
def refine_get_multilang_urls(original):

    def get_multilang_urls():
        from django.conf.urls import url, include
        # we need the url_patterns attr as this returns a list of url() instances
        urlpatterns = original() + url(r'^', include('app.urls')).url_patterns
        return urlpatterns

    return get_multilang_urls
```

Multilanguage Admin

This feature enables an optional multi-language admin.

Therefore, it refines the base implementation of the get_admin_urls function in djpladmin.

Obviously, djpladmin needs to be enabled, too and added before this feature.

It simply wraps the original implementation, which simply returns the include of the admin urls, into the i18n_patterns-function.

Refinements by example

This section shows some use cases and patterns to develop features for a django product line.

Refining the django settings

Many features require adaptations to the settings module used by django. `django-productline` always uses `django_productline.settings` as the django settings module. Features can apply refinements to it, to add/adapt settings to support the features functionality. Common cases are the refinement of `INSTALLED_APPS`

to register one or more additional django apps and the refinement of `MIDDLEWARE_CLASSES` to add django middlewares.

As a simple example, we are going to create a feature called `https_only`, that is implemented by integrating and configuring django-secure.

First, we need to create the python package `https_only` by creating a folder with that name that contains an empty `__init__.py`. As the feature needs to refine `django_productline.settings`, we also create a `settings` module within the package:

```
https_only/
    __init__.py
    settings.py
```

Let's use the following settings refinement:

```
#https_only/settings.py

#add djangosecure to the end of the INSTALLED_APPS list
def refine_INSTALLED_APPS(original):
    return original + ['djangosecure']

#add SecurityMiddleware to the end of the MIDDLEWARE_CLASSES list
def refine_MIDDLEWARE_CLASSES(original):
    return original + ['djangosecure.middleware.SecurityMiddleware']

#introduce some new settings into django_productline.settings
introduce_SESSION_COOKIE_SECURE = True
introduce_SESSION_COOKIE_HTTPONLY = True
introduce_SECURE_SSL_REDIRECT = True
```

This adds `djangosecure` to the list of installed apps and adds the middleware it depends on. Also, it introduces some security related settings.

> **Warning:** Before using this in production, please consult the django-secure documentation.

Now, we need to make sure the settings refinement is applied, when feature `https_only` is bound:

To use `https_only` as a feature, we need to add a module called `feature` to it. Let's create `feature.py` with the following content:

```
#https_only/feature.py

def select(composer):
    '''bind feature https_only'''
    #import settings refinement (https_only.settings)
    from . import settings
    #import project settings
    import django_productline.settings
    #apply the refinement to the project settings
    composer.compose(settings, django_productline.settings)
```

This applies our settings refinement, when the feature is bound. We can now add the functionality to products by selecting the `https_only` feature.

Since this feature refines INSTALLED_APPS and MIDDLEWARE_CLASSES, the composition order needs to be chosen carefully as the web application's behaviour is dependent on the order of their entries.

Registering urlpatterns

`django_productline.urls` exports the function `get_urls`. It is called through `django_productline.root_urlconf` which is registered as django's `ROOT_URLCONF`.

To introduce new urlpatterns, features may refine `get_urls`. The convention is to specify the refinement in a module called `urls` within the feature module.

Example:

```
#in myfeature.feature.select()
from . import urls #import the refinement definition
import featuredjango.urls #import the base module
#apply the refinement to the base module
featuremonkey.compose(urls, featuredjango.urls)

#in myfeature.urls
def refine_get_urls(original):
    def get_urls():
        '''
        introduce myfeature`s views
        '''
        from django.urls import patterns
        return original() + patterns('',
            (r'^foo/$', 'myfeature.views.foo'),
            (r'^bar/(\d{4})/$', 'myfeature.views.bar'),
        )

    return get_urls
```

Django Model composition

Django already provides an excellent database modularisation mechanism using apps. An app may contain multiple models, i.e. ORM-managed database tables. However, there is no easy way to introduce fields into existing models.

`featuremonkey` introductions will not work because of the custom metaclass used by django models, that takes care of additional book-keeping during construction of the model class. As introductions are applied after the class has been constructed, the book-keeping code is not executed in this case.

Fortunately, django provides a mechanism that takes care of the book-keeping even for attributes that are added after the class is constructed: model fields and managers provide a `contribute_to_class` method.

To make use of that, `django_productline` extends the composer to also support another operation called `contribute`. It can be used just like `introduce` except that it does not support callable introductions. Under the hood, it calls `contribute_to_class` instead of `setattr` which enables the introduction of fields to models.

Field Introduction Example

Let's look at an example. Suppose you are working on a todo list application. Then, some clients want an additional description field for their todo items, but others don't. So, you decide to create a feature to add that field conditionally.

Suppose the todo item model lives in `todo.models` and looks like this:

```
# todo/models.py
from django.db import models

class TodoItem(models.Model):
```

```
    name=models.CharField(max_length=100)
    done=models.BooleanField()
```

Now, let's create a feature called `todo_description`:

```
todo_description/
    __init__.py
    feature.py
    todo_models.py
```

Let's write a refinement for the `todo.models` module and place it in `todo_models.py`:

```
#todo_description/todo_models.py
#refines todo/models.py
from django.db import models

class child_TodoItem(object):
    contribute_description = models.TextField
```

Please note, that we are using `contribute` instead of `introduce` to let django do its model magic.

Next, let's apply the refinement in the feature binding function:

```
#todo_description/feature.py

def select(composer):
    compose_later(
        'todo_description.todo_models',
        'todo.models'
    )
```

That was it. The description field can now be added by selecting feature `todo_description`. Obviously, since there is a database involved, the schema needs to be created or modified if it exists already.

If the database table for todo items does not exist already, the field is automatically created in the database upon `syncdb`

If the table exists already, because the product has been run before selecting feature `todo_description`, we can use south to do a `schemamigration`:

```
$ ape manage schemamigration todo --auto
$ ape manage migrate todo
```

To compose models, we need to use `compose_later` as importing `django.db.models` starts up all the django initialization machinery as a side effect. At this point, this could result in references to partially composed objects and hard to debug problems.

To prevent you from importing these parts of django by accident, `django_productline` uses import guards for specific modules during composition. After all features are bound, those guards are dropped again and importing the modules is possible again.

The guarded packages/modules currently are:

- `django.conf`
- `django.db`

Adding WSGI Middleware

If you are using special WSGI-Middleware with your django project and would like to continue to do so using `django-productline`, you can directly refine `django.core.wsgi` to achieve that. So if your feature is called `mywsgifeature`, you can do it as presented in the following example:

First, create a module called `wsgi` in `mywsgifeature` containing and define a refinement for `get_wsgi_application`:

```
#mywsgifeature/wsgi.py

def refine_get_wsgi_application(original):
    def get_wsgi_application():
        application = original()
        from dozer import Dozer
        return Dozer(application)
    return get_wsgi_application
```

This refinement will add the Dozer WSGI middleware that can be used to track down memory leaks.

To use this for all products that contain `mywsgifeature`, we need to apply the refinement in `mywsgifeature.feature.select`:

```
#mywsgifeature/feature.py

def select(composer):
    from . import wsgi #import our refinement
    import django.core.wsgi #import base module
    #apply refinement
    composer.compose(wsgi, django.core.wsgi)

    #apply other necessary refinements of mywsgifeature
```

Note: If multiple features of your product line add WSGI middlewares to your application, the order in which the middlewares are applied is defined by the composition order of the selected features.

Available tasks

(All of these commands are issued by entering `ape` + commandname)

Product-Lifecycle tasks

`install_container <containername>` install a container into the development environment.

`select_features` selects and activates the features that are listed in the product equation if run. This needs to be called on every first startup of the environment.

`deploy` deploy the selected application to the server

From this point forward you can use the `ape manage` commands which are similar to the `python manage.py` commands from pythons virtualenv.

Container selection tasks

cd <target directory> change into target directory

zap This changes the focus on the previously installed container. The first argument is the name of the container itself, the second one is the context in which the container is setup. In detail this changes some things in the product equation, e.g. to provide different setups for productive or development setups. Usually these products are `website_dev` respectively `website_prod`. They can be looked up by taking the directory names from `/dev`

zap <containername>:<product> alias for "teleport". Use this the following way: `ape zap <containername>:<product>` like: `ape zap slimcontent:sample_dev` or similar

switch <target> switch the context to the specified target.

teleport <dir target> change the directory and switch to the target inside this directory

Further available ape commands

dev() starts up the development server. This is equal to `ape manage runserver`. Runserver optionally accepts an IP- Adress as an argument to run the dev server on a custom IP- Adress. If the server is started under `0.0.0.0:<PORT>` it exposes `<PORT>` to the LAN under `<IP- Adress of devmachine>:<PORT>`. This is useful for sharing development states amongst diferrent machines e.g. for mobile development similar tasks.

help(task) prints details about available tasks

info() prints information about the development environment

manage <...args> calls django-specific management tasks. This is equal to django's default `python manage.py` - command.

prepare prepares a product for deployment. This is a combo command that runs the following three comands prefixed with `prepare_` in order of appearance here. Under the hood this runs tasks such as:

- setting up the database and database schema
- generating the webserver configuration
- basically everything that's necessary for the server to run your app

must be executed every time after feature selection and/or changes of the product context

prepare_db Creates the database, prepares it for sync. By default this does nothing but can be refined by certain features to accomplish specific database creation tasks

prepare_db_schema This is a combo command that runs `syncdb` and applies database migrations afterwards

prepare_fs Prepares the filesystem for deployment. If you use the base implementation this creates the data dir.

requires_product_environment Task decorator that checks if the product environment of django-productline is activated which is necessary for the environment to run. Specifically it checks whether: - context is bound - features have been composed

manage Calls fundamental Django management tasks

deploy the base implementation delegates to the `dev` task.Features may refine this to add support for mod_wsgi,uwsgi,gunicorn,...

Required context data

`django_productline` requires the following keys in the product context:

DATABASES database configuration in the form required by django.conf.settings.DATABASES.

SITE_ID `site_id` to be used by django. See django.conf.settings.SITE_ID

DATA_DIR absolute path to directory where application data will be stored. This directory needs to be writable by the application user. Data is placed in the following subfolders:

- `uploads/` — content uploaded by users is stored here (see django.conf.settings.MEDIA_ROOT)
- `generated_static/` — static files created by `manage collectstatic` are placed here (see django.conf.settings.STATIC_ROOT)

SECRET_KEY see django.conf.settings.SECRET_KEY

CHAPTER 7

Feature Model

django-productline feature model

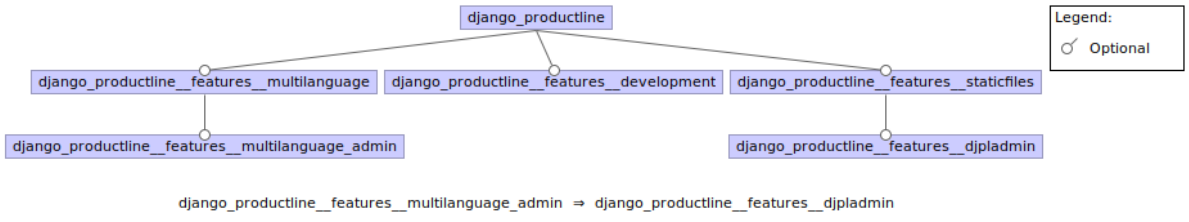
The following shows the feature model of `django-productline`. This was created using the Eclipse plugin FeatureIDE.

You can install it from here:

http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

Or you can use the Eclipse marketplace.

django-productline feature model



You can download the model from here: `model.xml`

To setup this feature model, create a new FeatureIDE project and override the model.xml.

CHAPTER 8

Tutorial

Tutorial

A tutorial for Django- Productline can be found here:

Django- Productline Tutorial

CHAPTER 9

Changelog

Changelog

HEAD

0.3

- added feature `development`
- feature `staticfiles` now serves media files when using devserver
- product context: better exception handling
- improved docs
- multitude of small improvements by tonimichel

0.2

- first release on PYPI
- django model composition
- features `staticfiles` and `admin`
- initial docs

0.1

- initial version

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

D