# django-ftl Documentation

***Release 0.12.1***

**Luke Plant**

**Nov 12, 2020**

# Contents

Contents:

CHAPTER 1

# Installation

At the command line:

```
$ pip install django-ftl
```

You can also install from latest master on GitHub.

CHAPTER 2

# Usage

Using Fluent in a Django project requires understanding a number of concepts and APIs, in addition to understanding the Fluent syntax. This guide outlines the main things you need.

## 2.1 Terminology

Internationalization and localization (i18n and l10n) tools usually distinguish between 'languages' and 'locales'. 'Locale' is a broader term than includes other cultural/regional differences, such as how numbers and dates are represented.

Since they go together, Fluent not only addresses language translation, it also integrates locale support. If a message contains a number substitution, when different locales are active the number formatting will match the language automatically. For this reason the django-ftl docs generally do not make a big distinction between these terms, but tend to use 'locale' (which includes language).

Django's i18n docs distinguish between 'locale name' (which look like `it`, `en_US` etc) and 'language code' (which look like `it`, `en-us`). In reality there is a lot of overlap between these. Most modern systems (e.g. unicode CLDR) use BCP 47 language tags, which are the same as 'language codes'. They in fact represent locales as well as languages, and have a mechanism for incorporating more specific locale information.

Fluent and django-ftl use BCP 47 language tags in all their APIs (more information below).

## 2.2 FTL files and layout

Fluent translations are placed in Fluent Translation List files, with the suffix `.ftl`. For them to be found by django-ftl, you need to use the following conventions, which align with the conventions used across other tools that use Fluent (such as Pontoon).

For the sake of this guide, we will assume you are writing a Django app (reusable or non-reusable) called `myapp` - that is, it forms a Python top-level module/package called `myapp`. You will need to replace `myapp` with the actual name of your app.

You will need your directory layout to match the following example:

```
myapp/
  __init__.py
  ftl_bundles.py
  locales/
    en/
      myapp/
          main.ftl
    de/
      myapp/
          main.ftl
```

That is:

- Within your `myapp` package directory, create a `locales` directory. In a typical Django app, this `locales` directory exists at the same level as your app-specific `templates`, `templatetags`, `static` etc. directories.
- For each locale you support, within that folder create a directory with the locale name. The example above shows English and German. Locale names should be in BCP 47 format.
- It is recommended that you follow the capitalization convention in BCP 47, which is:
  - Lower case for the language code
  - Title case for script code
  - Upper case for region code

  e.g. en, en-GB, en-US, de-DE, zh-Hans-CN

  django-ftl does not enforce this convention - it will find locale files if different capitalization is used. However, if multiple directories exist for the same locale, differing only by case (e.g. `EN-US` and `en-US`), and their contents are not the same, then your FTL files will probably not be found correctly.

  Finally, django-ftl will also find the FTL files if you name the directories in Unix "locale name" convention with underscores e.g. `en_GB`, although for the sake of consistency and other tools this is also not recommended.
- Within each specific locale directory, create another directory with the name of your app. This is necessary to give a separate namespace for your FTL files, so that they don't clash with the FTL files that might be provided by other Django apps. By doing it this way, you can reference FTL files from other apps in your app — this is very similar to how templates and static files work in Django.
- Within that `myapp` directory, you can add any number of further sub-directories, and can split your FTL files up into as many files as you want. For the remainder of this guide we will assume a single `myapp/main.ftl` file for each locale.

The contents of these files must be valid Fluent syntax. For the sake of this guide, we will assume `myapp` has an 'events' page which greets the user, and informs them how many new events have happened on the site since their last visit. It might have an English `myapp/main.ftl` file that looks like this:

```
events-title = MyApp Events!

events-greeting = Hello, { $username }

events-new-events-info = { $count ->
    [0]     There have been no new events since your last event.
    [1]     There has been one new event since your last visit.
   *[other] There have been { $count } new events since your last visit.
 }
```

In this `.ftl` file, `events-title`, `events-greeting` and `events-new-events-info` are Fluent message IDs. Note that we have used `events-` as an adhoc namespace for this 'events' page, to avoid name clashes with other

messages from our app. It's recommended to use a prefix like this for different pages or components in your app.

## 2.3 Bundles

To use `.ftl` files with django-ftl, you must first define a *Bundle*. They represent a collection of `.ftl` files that you want to use, and are responsible for finding and loading these files. The definition of a `Bundle` can go anywhere in your project, but we recommend the convention of creating a `ftl_bundles.py` file inside your Python `myapp` package, i.e. a `myapp.ftl_bundles` module.

Our `ftl_bundles.py` file will look like this:

```
from django_ftl.bundles import Bundle

main = Bundle(['myapp/main.ftl'])
```

`Bundle` takes a single required positional argument which is a list of FTL files. See *Bundle* API docs for other arguments.

## 2.4 Activating a locale/language

The most direct way to activate a specific language/locale is to use *django_ftl.activate()*:

```
from django_ftl import activate

activate("en-US")
```

The argument can be any BCP 47 language tag, or a "language priority list" (a prioritized, comma separated list of language tags). For example:

```
"en-US, en, fr"
```

It is recommended that the value passed in should be validated by your own code. Normally it will come from a list of options that you have given to a user (see *Setting the user language preference* below).

As soon as you activate a language, all `Bundle` objects will switch to using that language, for the current thread only. (Before activating, they will use your `LANGUAGE_CODE` setting as a default if `require_activate=False`, and this is also used as a fallback in the case of missing FTL files or messages).

Please note that `activate` is stateful, meaning it is essentially a global (thread local) variable that is preserved between requests. This introduces the possibility that one user's request changes the behavior of subsequent requests made by a completely different user. This problem can also affect test isolation in automated tests. The best way to avoid these problems is to use *django_ftl.override()* instead:

```
from django_ftl import override

with override("en-US"):
    pass  # Code that uses this language
```

Alternatively, ensure that *django_ftl.deactivate()* is called at the end of a request.

### 2.4.1 Using middleware

The way you choose to activate a given language will depend on your exact setup.

`django-ftl` comes with a few middleware that may help you automatically activate a locale for every request.

If you were using Django's built-in i18n solution previously, or are still using it for some parts of your app, you may also be using django.middleware.locale.LocaleMiddleware. If that is the case, and if you want to continue using `LocaleMiddleware`, the easiest solution is to add `"django_ftl.middleware.activate_from_request_language_code"` after it in your `MIDDLEWARE` setting:

```
MIDDLEWARE = [
      ...
      "django.middleware.locale.LocaleMiddleware",
      "django_ftl.middleware.activate_from_request_language_code"
      ...
]
```

This is a very simple middleware that simply looks at `request.LANGUAGE_CODE` (which has been set by `django.middleware.locale.LocaleMiddleware`) and activates that language for django-ftl.

Instead of these two, you could also use `"django_ftl.middleware.activate_from_request_session"` by adding it to your `MIDDLEWARE` (somewhere after the session middleware). This middleware looks for a language set in `request.session`, as set by the `set_language` view that Django provides (see set_language docs), and uses this value, falling back to `settings.LANGUAGE_CODE` if it is not found. It also sets `request.LANGUAGE_CODE` to the same value, similar to how `django.middleware.locale.LocaleMiddleware` behaves.

Both of these provided middleware use `override` to set the locale, not `activate`, as per the advice above, for better request and test isolation.

You are not limited to these middleware, or to using Django's `set_language` view — these are provided as shortcuts and examples. In some cases it will be best to write your own, using the middleware source code as a starting point.

### 2.4.2 Outside of the request-response cycle

If you need to generate localized text from code running outside of the request-response cycle (e.g. cron jobs or asynchronous tasks), you will not be able to use middleware, and will need some other way to determine the locale to use. This might involve:

- a field on a model (e.g. `User` class) to store the locale preference.
- for asynchronous tasks such as Celery, you could pass the locale as an argument. For Celery, signals such as task-prerun might be useful.

Once you have determined the locale you need, use *`django_ftl.activate()`* or *`django_ftl.override()`* to activate it.

## 2.5 Using bundles from Python

After you have activated a locale, to obtain a translation you call the `Bundle` *`format()`* method, passing in a message ID and an optional dictionary of arguments:

```
>>> from myapp.ftl_bundles import main as ftl_bundle
>>> ftl_bundle.format('events-title')
'MyApp Events!'

>>> ftl_bundle.format('events-greeting', {'username': 'boaty mcboatface'})
'Hello, \u2068boaty mcboatface\u2069'
```

The \u2068 and \u2069 characters are unicode bidi isolation characters that are inserted by Fluent to ensure that the layout of text behaves correctly in case substitutions are in a different script to the surrounding text.

That's it for the basic case. See *format()* for further info about passing numbers and datetimes, and about how errors are handled.

### 2.5.1 Lazy translations

Sometimes you need to translate a string lazily. This happens when you have a string that is defined at module load time (see the Django lazy translation docs for more info). For this situation, you can use *format_lazy()* instead of format. It takes the same parameters, but doesn't generate the translation until the value is used in a string context, such as in template rendering.

For example, the verbose_name and help_text attributes of a model field could be done this way:

```
from django.db import models
from myapp.ftl_bundles import main as ftl_bundle

class Kitten(models.Model):
    name = models.CharField(
        ftl_bundle.format_lazy('kitten-name'),
        help_text=ftl_bundle.format_lazy('kitten-name.help-text'))
```

```
# kittens.ftl

kitten-name = name
    .help-text = Use most recent name if there have been are multiple.
```

Note that here we have used attributes to combine the two related pieces of text into a single message

If you do not use format_lazy, then the verbose_name and help_text attributes will end up always having the text translated into the default language.

As a more effective way to prevent this from happening, you can also pass require_activate=True parameter to *Bundle*. As long as there is no activate call at module level in your project, this will cause the Bundle to raise an exception if you attempt to use the format method at module level.

---

**Note:** If you pass require_activate=True, you may have trouble with some features like Django migrations which will attempt to serialize model and field definitions, which forces lazy strings to be evaluated.

You can work around this problem by putting the following code in your ftl_bundles.py files:

```
import sys
import os.path
from django_ftl import activate

if any(os.path.split(arg)[-1] == 'manage.py' for arg in sys.argv) and 'makemigrations
↪' in sys.argv:
    activate('en')
```

---

### 2.5.2 Aliases

If you are using the format and format_lazy functions a lot, you can save on typing by defining some appropriate aliases for your bundle methods at the top of a module - for example:

```
from myapp.ftl_bundles import main as ftl_bundle

ftl = ftl_bundle.format
ftl_lazy = ftl_bundle.format_lazy
```

Then use `ftl` and `ftl_lazy` just as you would use `ftl_bundle.format` and `ftl_bundle.format_lazy`.

## 2.6 Using bundles from Django templates

To use django-ftl template tags in a project, `django_ftl` must be added to your `INSTALLED_APPS` like this:

```
INSTALLED_APPS = (
    ...
    'django_ftl.apps.DjangoFtlConfig',
    ...
)
```

Put `{% load ftl %}` at the top of your template to load the template tag library. It provides 3 template tags, at least one of which you will need:

### 2.6.1 ftlconf

This is used to set up the configuration needed by `ftlmsg`, namely the `bundle` to be used and the rendering `mode`. It should be used once near the top of a template (before any translations are needed), and should be used in the situation where most of the template will use the same bundle. For setting the configuration for just part of a template, use `withftl`.

`bundle` is either a bundle object (passed in via the template context), or a string that is a dotted path to a bundle.

`mode` is currently limited to a single string value `'server'`. In the future further options will be added (to enable support for client-side rendering/Pontoon), so it is recommended to use a context processor to add this value into template context, so that this single context processor can be changed in future to make use of those features.

Example:

```
{% load ftl %}
{% ftlconf mode='server' bundle='myapp.ftl_bundles.main' %}
```

Example where we use a context processor to set the mode, and pass in the bundle object from the view:

```
# In settings.py

TEMPLATE = [
   {
     'BACKEND': 'django.template.backends.django.DjangoTemplates',
     'OPTIONS': {
         'context_processors': [
            # ...
            'myapp.context_processors.ftl_mode',
         ]
     },
   }
]
```

```
# myapp/context_processors.py

def ftl_mode(request):
    return {'ftl_mode': 'server'}
```

```
# myapp.views

from myapp.ftl_bundles import main as main_bundle

def my_view(request):
    # ...
    return render(request, 'myapp/mypage.html',
                  {'ftl_bundle': main_bundle})
```

```
{# myapp/events.html #}

{% load ftl %}
{% ftlconf mode=ftl_mode bundle=ftl_bundle %}
```

`mode` and `bundle` are both optional and can be set on different calls to `ftlconf`, but both must be set before using `ftlmsg`.

### 2.6.2 `withftl`

`withftl` is similar to `ftlconf` in that its purpose is to set configuration data for generating messages. It differs in that:

1. It sets the data only for the contained template nodes, up to a closing `endwithftl` node, which is required.
2. It also takes a `language` parameter that can be used to override the language, in addition to the `mode` and `bundle` parameters that `ftlconf` take. This should be a string in BCP 47 format.

Multiple nested `withftl` tags can be used, and they can be nested into a template that has `ftlconf` at the top, and their scope will be limited to the contained template nodes as you would expect.

Example:

```
{% load ftl %}
{% ftlconf mode='server' %}

{% withftl bundle='myapp.ftl_bundles.main' %}
   {% ftlmsg 'events-title' %}
{% endwithftl %}

{% withftl bundle='myapp.ftl_bundles.other' language='fr' %}
   {% ftlmsg 'other-message' %}
{% endwithftl %}
```

As with `ftlconf`, the parameters do not have to be just literal strings, they can refer to values in the context as most template tags can. You must supply one or more of `mode`, `bundle` or `language`.

### 2.6.3 `ftlmsg`

Finally, to actually render a message, you need to use `ftlmsg`. It takes one required parameter, the message ID, and any number of keyword arguments, which correspond to the parameters you would pass in the arguments dictionary when calling *`format()`* in Python code.

Example:

```
{% load ftl %}
{% ftlconf mode='server' bundle='myapp.ftl_bundles.main' %}

<body>
   <h1>{% ftlmsg 'events-title' %}</h1>

   <p>{% ftlmsg 'events-greeting' username=request.user.username %}</p>
</body>
```

### 2.6.4 Alternative configuration

In some cases, use of `ftlconf` or `withftl` in templates can be tedious and you may want to specify configuration of mode/bundle using a more global method.

An alternative is to set some configuration variables in the template context. You can do this using some manual method, or using a context processor. The variables you need to set are given by the constants below:

- `django_ftl.templatetags.ftl.MODE_VAR_NAME` for mode.
- `django_ftl.templatetags.ftl.BUNDLE_VAR_NAME` for the bundle.

For example, the following is a context processor that will return the required configuration for the `ftlmsg` template tag.

```
import django_ftl.templatetags.ftl

from my_app.ftl_bundles import main

def ftl(request):
    return {
        django_ftl.templatetags.ftl.MODE_VAR_NAME: 'server',
        django_ftl.templatetags.ftl.BUNDLE_VAR_NAME: main,
    }
```

This could be configured to be used always via your `TEMPLATES` context_processors setting, or invoked manually and merged into a context dictionary.

## 2.7 HTML escaping

django-ftl plugs in to `fluent_compiler`'s escaping mechanism and provides an escaper out of the box that allows you to handle HTML embedded in your messages. To use it, give your message IDs the suffix `-html`. For example:

```
welcome-message-html = Welcome { $name }, you look <i>wonderful</i> today.
```

In this example, `$name` will have HTML escaping applied as you expect and need, while the `<i>wonderful</i>` markup will be left as it is. The whole message will be returned as a Django `SafeText` instance so that further HTML escaping will not be applied.

It is recommended not to use `-html` unless you need it, because that will limit the use of a message to HTML contexts, and it also requires translators to write correct HTML (for example, with ampersands written as `&amp;`).

Note that there are rules regarding how messages with different escapers can be used. For example:

```
-brand = Ali & Alisha's ice cream

-brand-html = Ali &amp; Alisha's <b>cool</b> ice cream
```

The `-brand` term can be used from any other message, and from a `...-html` message it will be correctly escaped. The `-brand-html` term, however, can only be used from other `...-html` messages.

## 2.8 Template considerations

A very common mistake in i18n is forgetting to set the `lang` tag on HTML content. In the normal case, each base template that contains an `<html>` tag needs to be modified to add the `lang` attribute - assuming you've used middleware as described above this could be as simple as:

```
<html lang="{{ request.LANGUAGE_CODE }}">
```

See w3c docs on the lang attribute for more information.

## 2.9 Setting the user language preference

How you want to set and store the user's language preference will depend on your application. For example, you can set it in a cookie, in the session, or store it as a user preference.

Django has a built-in `set_language` view that you can use with django-ftl - see the set_language docs. (It is designed to work with Django's built-in i18n solution but works just as well with django-ftl). It saves a user's language preference into the session (or a cookie if you are not using sessions), which you can then use later in a middleware or view, for example.

## 2.10 Auto-reloading

By default, django-ftl loads and caches all FTL files on first usage. In development, this can be annoying as changes are not reflected unless you restart the development server. To solve this, django-ftl comes with an auto-reloading mechanism for development mode. To use it, you must install pyinotify:

```
$ pip install pyinotify
```

By default, if you have `DEBUG = True` in your settings (which is normally the case for development mode), the reloader will be used and any changes to FTL files references from bundles will be detected and picked up immediately.

You can also control this manually with your `FTL` settings in `settings.py`:

```
FTL = {
    'AUTO_RELOAD_BUNDLES': True
}
```

Also, you can configure this behavior via the *Bundle* constructor.

CHAPTER 3

# API documentation

## 3.1 Activating/deactivating locales

`django_ftl.`**`activate`**(*locale_code*)

Activate a locale given by a BCP47 locale code (e.g. "en-US"). All *`Bundle`* objects will be switched to look for translation files with that locale.

This uses a thread local variable internally to store the current locale.

`django_ftl.`**`deactivate`**()

De-activate the currently activated locale. All *`Bundle`* objects will fallback to the default locale if you try to generate messages with them (or throw exceptions, depending on the value of `require_activate`), until you activate another language.

`django_ftl.`**`override`**(*locale_code*)

A Python context manager that uses *`activate()`* to set a locale on entry, and then re-activates the previous locale on exit. It can also be used a function decorator.

## 3.2 Bundles

**`class`** `django_ftl.bundles.`**`Bundle`**(*files*, *default_locale=None*, *require_activate=False*, *use_isolating=True*)

Create a bundle from a list of files.

**Parameters**

- **`list(str)`** (*`files`*) – Files are specified as relative paths that start from a specific locale directory.

  For example, if you are writing `myapp`, and you have `myapp/locales/en/myapp/main.ftl` for English and `myapp/locales/de/myapp/main.ftl` for German, then you would pass `["myapp/main.ftl"]` which will refer to either of these files depending on the active language.

If multiple paths are given, they will be added in order. This means that if later files contain the same message IDs as earlier files, the later definitions will shadow and take precedence over earlier ones.

- **default_locale** (*str*) – You may pass keyword argument `default_locale` (as a BCP47 string e.g. "en-US"), which will be used as a fallback if an unavailable locale is activated, or if a message ID is not found in the current locale. By default, your `LANGUAGE_CODE` setting will be used (see settings) if nothing is passed.
- **require_activate** (*bool*) – By default the `default_locale` will be used as a fallback if no language has been activated. By passing `require_activate=True`, *`format()`* will raise an exception if you attempt to use it without first activating a language. This can be helpful to ensure that all code paths that use Bundles are setting a language first, and especially for ensuring that all module level uses of a `Bundle` use *`format_lazy()`* instead of *`format()`*.
- **use_isolating** (*bool*) – Controls whether substitutions in messages should be surrounded with bidi isolation characters. Defaults to `True`. Pass `False` to disable this (if, for example, all your text and substitutions are in scripts that go in the same direction).
- **auto_reload** (*bool*) – Controls whether the Bundle will attempt to detect changes in `FTL` files and reload itself. If nothing is passed, automatic behavior will be used, which is:
  - `settings.AUTO_RELOAD_BUNDLES` if it is set, otherwise:
    * `True` if `settings.DEBUG == True` and pyinotify is installed
    * `False` otherwise.

**format**(*message_id*, *args=None*)

Generate a translation of the message specified by the message ID, in the currently activated locale.

`args` is an optional dictionary of parameters for the message. These will normally be:

- strings
- integers or floating point numbers (which will be formatted according to locale rules)
- datetime objects (which will be formatted according to locale rules)

To specify or partially specify your own formatting choices for numbers and datetime objects, see the `fluent_compiler` docs for fluent_compiler.types.fluent_number and fluent_compiler.types.fluent_datetime.

The arguments passed in may also be strings or numbers that are used to select variants.

**format_lazy**(*message_id*, *args=None*)

Same as *`format()`*, but returns an object that delays translation until the object is used in a string context.

This is important when defining strings at module level which should be translated later, when the required locale is known.

### 3.2.1 Error handling in Bundle

Fluent's philosophy is that in general, when generating translations, something is usually better than nothing, and therefore it attempts to recover as much as possible from error conditions. For example, if there are syntax errors in `.ftl` files, it will try to find as many correct messages as possible and pass over the incorrect ones. Or, if a message is formatted but it is missing an argument, the string `'???'` will be used rather than turning the whole message into an error of some kind. At the same time, these errors should be reported somehow.

django-ftl in general follows the same principle. This means that things like missing `.ftl` files are tolerated, and most `Bundle` methods rarely throw exceptions.

Instead, when errors occur they are collected and then logged. Errors found in `.ftl` message files, or generated at runtime due to bad arguments, for example, will be logged at `ERROR` level using the stdlib logging framework, to the `django_ftl.message_errors` logger. Ensure that these errors are visible in your logs, and this should make these problems more visible to you.

If a message is missing entirely, for instance, you will get `'???'` returned from `Bundle.format` rather than an exception (but the error will be logged). If the message is missing from the requested locale, but available in the default locale, the default will be used (but you will still get an error logged). Therefore, you don't need to add `try` / `except` around calls to `Bundle.format` to provide a fallback, because that is done for you.

There are some places where django-ftl does throw exceptions, however. These include:

- `Bundle.format`: If any of the bundle's specified `.ftl` are missing from the default locale, a `django_ftl.bundles.FileNotFoundError` exception will be raised. It is assumed that such a problem with the default locale is a result of a typo, rather than just a locale than has not been fully translated yet, and so the developer is warned early. An empty `.ftl` file at the correct path is sufficient to silence this error.
- `Bundle.format`: If `require_activate` is True, this method will raise a `django_ftl.bundles.NoLocaleSet` exception if you attempt to use it before calling `activate`. This is a deliberate feature to help flush out cases where you are using *`Bundle.format()`* before setting a locale, instead of *`Bundle.format_lazy()`*.

These are deliberately intended to cause crashes, because you have a developer error that should cause failure as early and as loudly as possible.

CHAPTER 4

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 4.1 Types of Contributions

### 4.1.1 Report Bugs

Report bugs at https://github.com/django-ftl/django-ftl/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

### 4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

### 4.1.4 Write Documentation

django-ftl could always use more documentation, whether as part of the official django-ftl docs, in docstrings, or even on the web in blog posts, articles, and such.

### 4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/django-ftl/django-ftl/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 4.2 Get Started!

Ready to contribute? Here's how to set up `django-ftl` for local development.

1. Fork the `django-ftl` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-ftl.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-ftl
$ cd django-ftl/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
    $ pip install -r requirements_test.txt
    $ ./runtests.py
    $ flake8 src tests

To run tests against all supported versions::

    $ pip install tox
    $ tox

You can also run tests with py.test (although it is much slower for some reason)::

    $ pip install pytest
    $ py.test
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

## 4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7 and 3.6. Check https://travis-ci.org/django-ftl/django-ftl/pull_requests and make sure that the tests pass for all supported Python versions.

## 4.4 Tips

To run a specific test or subset of tests, use dotted paths to module/class/method e.g.:

```
$ ./runtests.py tests.test_bundle.TestBundles.test_fallback
```

CHAPTER 5

# Credits

## 5.1 Development Lead

- Luke Plant <L.Plant.98@cantab.net>

## 5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

# History

## 6.1 0.12.1 (2020-05-09)

- Fixed broken (and undocumented) `check_all` method.

## 6.2 0.12 (2020-04-02)

- Switch to the new APIs available in `fluent_compiler` 0.2.
- Performance improvements - large reduction in the percentage overhead introduced by django-ftl (compared to raw `fluent_compiler` performance).
- Undocumented `MessageFinderBase` class has changed slightly: its `load` method now returns a `fluent_compiler.resource.FtlResource` object instead of a string. If you used a custom `finder` for `Bundle` you may need to update it for this change.

## 6.3 0.11 (2020-03-24)

- Switched to using `fluent_compiler` as backend instead of experimental branch in `fluent.runtime`. This means **import changes are required**:
  - `fluent_number` and `fluent_date`, if you are using them, should be imported from `fluent_compiler.types` instead of `fluent.runtime.types`
- Added `Bundle.check_all` method.
- Django 3.0 support
- Dropped support for Python 3.4 (it may work, but recent versions of lxml do not install on it, which made running tests harder).

## 6.4 0.10 (2019-05-23)

- Upgraded to more recent version of fluent.runtime (0.1 with modifications)
- Fixed `use_isolating` behavior (BDI characters are now inserted for HTML messages)
- Thread-safety fixes for loading bundles.
- Corrected order of using 'locales' directories found via `INSTALLED_APPS` to be consistent with normal Django convention.

## 6.5 0.9.1 (2019-03-02)

- Changed development autoreload mechanism to not interfere with Django's development server autoreload.
- Bug fix for case when invalid mode is specified in template tag.
- Various fixes and improvements to middlewares (plus tests)
- Thread-safe Bundle
- Method for configuring `ftlmsg` via context processor.

## 6.6 0.9 (2018-09-10)

- Working version
- Depends on our version of python-fluent

## 6.7 0.0.1 (2018-05-19)

- First release on PyPI - empty placeholder package

# Index