fluent.runtime Documentation

Release 0.3

Luke Plant

May 18, 2020

Contents:

These are the docs for `fluent.runtime` 0.3. Check the *Changelog* for significant changes.

CHAPTER 1

Installation

`fluent.runtime` can be installed with pip:

```
$ pip install fluent.runtime
```

Python 2.7 or 3.5+ required. Earlier versions of Python 3 may work but they are not officially supported.

CHAPTER 2

Using fluent.runtime

2.1 Learn the FTL syntax

FTL is a localization file format used for describing translation resources. FTL stands for *Fluent Translation List*.

FTL is designed to be simple to read, but at the same time allows to represent complex concepts from natural languages like gender, plurals, conjugations, and others.

```
hello-user = Hello, { $username }!
```

In order to use fluent.runtime, you will need to create FTL files. Read the Fluent Syntax Guide in order to learn more about the syntax.

2.2 Using FluentLocalization

Once you have some FTL files, you can generate translations using the `fluent.runtime` package. You start with the `FluentLocalization` class:

```
>>> from fluent.runtime import FluentLocalization, FluentResourceLoader
```

The Fluent files of your application are loaded with a `FluentResourceLoader`.

```
>>> loader = FluentResourceLoader("l10n/{locale}")
```

The main entrypoint for your application is a `FluentLocalization`. You pass a list of locales to the constructor - the first being the desired locale, with fallbacks after that - as well as resource IDs and your loader.

```
>>> l10n = FluentLocalization(["de", "en-US"], ["main.ftl"], loader)
>>> val = l10n.format_value("my-first-string")
"Fluent can be easy"
```

This assumes that you have a directory layout like so

and `l10n/de/main.ftl` with:

```
second-string = Eine Übersetzung
```

as well as `l10n/en-US/main.ftl` with:

```
my-first-string = Fluent can be easy
second-string = An original string
```

As you can see, our first example returned the English string, as that's on our fallback list. When retrieving an existing translation, you get the translated results as expected:

```
>>> l10n.format_value("second-string")
"Eine Übersetzung"
```

2.2.1 Python 2

The above examples assume Python 3. Since Fluent uses unicode everywhere internally (and doesn't accept bytestrings), if you are using Python 2 you will need to make adjustments to the above example code. Either add `u` unicode literal markers to strings or add this at the top of the module or the start of your repl session:

```
from __future__ import unicode_literals
```

2.2.2 DemoLocalization

To make the documentation easier to read, we're using a `DemoLocalization`, that just uses a single literal Fluent resource. Find out more about the details in the *Internals of fluent.runtime* section.

```
>>> l10n = DemoLocalization("key = A localization")
>>> pl = DemoLocalization("key = A localization", locale="pl")
```

2.2.3 Numbers

When rendering translations, Fluent passes any numeric arguments (`int`, `float` or `Decimal`) through locale-aware formatting functions:

```
>>> l10n = DemoLocalization(
... "show-total-points = You have { $points } points."
... )
>>> val = l10n.format_value("show-total-points", {'points': 1234567})
>>> val
'You have 1,234,567 points.'
```

You can specify your own formatting options on the arguments passed in by wrapping your numeric arguments with `fluent.runtime.types.fluent_number`:

```
>>> from fluent.runtime.types import fluent_number
>>> points = fluent_number(1234567, useGrouping=False)
>>> l10n.format_value("show-total-points", {'points': 1234567})
'You have 1234567 points.'
```

(continues on next page)

(continued from previous page)

```
>>> amount = fluent_number(1234.56, style="currency", currency="USD")
>>> l10n = DemoLocalization(
... "your-balance = Your balance is { $amount }"
... )
>>> l10n.format_value(balance.value, {'amount': amount})
'Your balance is $1,234.56'
```

The options available are defined in the Fluent spec for NUMBER. Some of these options can also be defined in the FTL files, as described in the Fluent spec, and the options will be merged.

2.2.4 Date and Time

Python `datetime.datetime` and `datetime.date` objects are also passed through locale aware functions:

```
>>> from datetime import date
>>> l10n = DemoLocalization("today-is = Today is { $today }")
>>> val = bundle.format_value("today-is", {"today": date.today() })
>>> val
'Today is Jun 16, 2018'
```

You can explicitly call the `DATETIME` builtin to specify options:

```
>>> l10n = DemoLocalization(
... 'today-is = Today is { DATETIME($today, dateStyle: "short") }'
... )
```

See the DATETIME docs. However, currently the only supported options to `DATETIME` are:

- `timeZone`
- `dateStyle` and `timeStyle` which are proposed additions to the ECMA i18n spec.

To specify options from Python code, use `fluent.runtime.types.fluent_date`:

```
>>> from fluent.runtime.types import fluent_date
>>> today = date.today()
>>> short_today = fluent_date(today, dateStyle='short')
>>> val = l10n.format_value("today-is", {"today": short_today })
>>> val
'Today is 6/17/18'
```

You can also specify timezone for displaying `datetime` objects in two ways:

- Create timezone aware `datetime` objects, and pass these to the `format` call e.g.:

  ```
  >>> import pytz
  >>> from datetime import datetime
  >>> utcnow = datime.utcnow().replace(tzinfo=pytz.utc)
  >>> moscow_timezone = pytz.timezone('Europe/Moscow')
  >>> now_in_moscow = utcnow.astimezone(moscow_timezone)
  ```

- Or, use timezone naive `datetime` objects, or ones with a UTC timezone, and pass the `timeZone` argument to `fluent_date` as a string:

```
>>> utcnow = datetime.utcnow()
>>> utcnow
datetime.datetime(2018, 6, 17, 12, 15, 5, 677597)

>>> l10n = DemoLocalization("now-is = Now is { $now }")
>>> val = bundle.format_pattern("now-is",
...     {"now": fluent_date(utcnow,
...                         timeZone="Europe/Moscow",
...                         dateStyle="medium",
...                         timeStyle="medium")})
>>> val
'Now is Jun 17, 2018, 3:15:05 PM'
```

2.2.5 Custom functions

You can add functions to the ones available to FTL authors by passing a `functions` dictionary to the `FluentLocalization` constructor:

```
>>> import platform
>>> def os_name():
...     """Returns linux/mac/windows/other"""
...     return {'Linux': 'linux',
...             'Darwin': 'mac',
...             'Windows': 'windows'}.get(platform.system(), 'other')

>>> l10n = FluentLocalization(['en-US'], ['os.ftl'], loader, functions={'OS': os_name}
↪)
>>> l10n.format_value('welcome')
Welcome to Linux
```

That's with `l10n/en-US/os.ftl` as:

```
welcome = { OS() ->
   [linux]    Welcome to Linux
   [mac]      Welcome to Mac
   [windows]  Welcome to Windows
  *[other]    Welcome
  }
```

These functions can accept positional and keyword arguments (like the `NUMBER` and `DATETIME` builtins), and in this case must accept the following types of arguments:

- unicode strings (i.e. `unicode` on Python 2, `str` on Python 3)
- `fluent.runtime.types.FluentType` subclasses, namely:
- `FluentNumber` - `int`, `float` or `Decimal` objects passed in externally, or expressed as literals, are wrapped in these. Note that these objects also subclass builtin `int`, `float` or `Decimal`, so can be used as numbers in the normal way.
- `FluentDateType` - `date` or `datetime` objects passed in are wrapped in these. Again, these classes also subclass `date` or `datetime`, and can be used as such.
- `FluentNone` - in error conditions, such as a message referring to an argument that hasn't been passed in, objects of this type are passed in.

Custom functions should not throw errors, but return `FluentNone` instances to indicate an error or missing data. Otherwise they should return unicode strings, or instances of a `FluentType` subclass as above.

2.2.6 Known limitations and bugs

- We do not yet support `NUMBER(..., currencyDisplay="name")` - see this python-babel pull request which needs to be merged and released.
- Most options to `DATETIME` are not yet supported. See the MDN docs for Intl.DateTimeFormat, the ECMA spec for BasicFormatMatcher and the Intl.js polyfill.

Help with the above would be welcome!

CHAPTER 3

Internals of fluent.runtime

The application-facing API for `fluent.runtime` is `FluentLocalization`. This is the binding providing basic functionality for using Fluent in a project. `FluentLocalization` builds on top of `FluentBundle` on top of `FluentResource`.

`FluentLocalization` handles

- Basic binding as an application-level API
- Language fallback
- uses resource loaders like `FluentResourceLoader` to create `FluentResource`

`FluentBundle` handles

- Internationalization with plurals, number formatting, and date formatting
- Aggregating multiple Fluent resources with message and term references
- Functions exposed to Select and Call Expressions

`FluentResource` handles parsing of Fluent syntax.

Determining which language to use, and which languages to fall back to is outside of the scope of the `fluent.runtime` package. A concrete application stack might have functionality for that. Otherwise it needs to be built, Babel has helpers for that. `fluent.runtime` uses Babel internally for the international functionality.

These bindings benefit from being adapted to the stack. Say, a Django project would configure the localization binding through `django.conf.settings`, and load Fluent files from the installed apps.

3.1 Subclassing FluentLocalization

In the *Using fluent.runtime* documentation, we used `DemoLocalization`, which we'll use here to exemplify how to subclass `FluentLocalization` for the needs of specific stacks.

```
from fluent.runtime import FluentLocalization, FluentResource
class DemoLocalization(FluentLocalization):
  def __init__(self, fluent_content, locale='en', functions=None):
    # Call super() with one locale, no resources nor loader
    super(DemoLocalization, self).__init__([locale], [], None, functions=functions)
    self.resource = FluentResource(fluent_content)
```

This set up the custom class, passing `locale` and `functions` to the base implementation. What's left to do is to customize the resource loading.

```
def _bundles(self):
  bundle = self._create_bundle(self.locales)
  bundle.add_resource(self.resource)
  yield bundle
```

That's all that we need for our demo purposes.

3.2 Using FluentBundle

The actual interaction with Fluent content is implemented in `FluentBundle`. Optimizations between the parsed content in `FluentResource` and a representation suitable for the resolving of Patterns is also handled inside `FluentBundle`.

```
>>> from fluent.runtime import FluentBundle, FluentResource
```

You pass a list of locales to the constructor - the first being the desired locale, with fallbacks after that:

```
>>> bundle = FluentBundle(["en-US"])
```

The passed locales are used for internationalization purposes inside Fluent, being plural forms, as well as formatting of values. The locales passed in don't affect the loaded messages, handling multiple localizations and the fallback from one to the other is done in the `FluentLocalization` class.

You must then add messages. These would normally come from a `.ftl` file stored on disk, here we will just add them directly:

```
>>> resource = FluentResource("""
... welcome = Welcome to this great app!
... greet-by-name = Hello, { $name }!
... """)
>>> bundle.add_resource(resource)
```

To generate translations, use the `get_message` method to retrieve a message from the bundle. This returns an object with `value` and `attributes` properties. The `value` can be `None` or an abstract pattern. `attributes` is a dictionary mapping attribute names to abstract patterns. If the the message ID is not found, a `LookupError` is raised. An abstract pattern is an implementation-dependent representation of a Pattern in the Fluent syntax. Then use the `format_pattern` method, passing the message value or one of its attributes and an optional dictionary of substitution parameters. You should only pass patterns to `format_pattern` that you got from that same bundle. As per the Fluent philosophy, the implementation tries hard to recover from any formatting errors and generate the most human readable representation of the value. The `format_pattern` method thereforereturns a tuple containing `(translated string, errors)`, as below.

```
>>> welcome = bundle.get_message('welcome')
>>> translated, errs = bundle.format_pattern(welcome.value)
```

(continues on next page)

(continued from previous page)

```
>>> translated
"Welcome to this great app!"
>>> errs
[]

>>> greet = bundle.get_message('greet-by-name')
>>> translated, errs = bundle.format_pattern(greet.value, {'name': 'Jane'})
>>> translated
'Hello, \u2068Jane\u2069!'

>>> translated, errs = bundle.format_pattern(greet.value, {})
>>> translated
'Hello, \u2068{$name}\u2069!'
>>> errs
[FluentReferenceError('Unknown external: name')]
```

You will notice the extra characters `\u2068` and `\u2069` in the output. These are Unicode bidi isolation characters that help to ensure that the interpolated strings are handled correctly in the situation where the text direction of the substitution might not match the text direction of the localized text. These characters can be disabled if you are sure that is not possible for your app by passing `use_isolating=False` to the `FluentBundle` constructor.

CHAPTER 4

Changelog

4.1 fluent.runtime next

4.2 fluent.runtime 0.3 (October 23, 2019)

- Added `fluent.runtime.FluentResource` and `fluent.runtime.FluentBundle.add_resource`.
- Removed `fluent.runtime.FluentBundle.add_messages`.
- Replaced `bundle.format()` with `bundle.format_pattern(bundle.get_message().value)`.
- Added `fluent.runtime.FluentLocalization` as main entrypoint for applications.

4.3 fluent.runtime 0.2 (September 10, 2019)

- Support for Fluent spec 1.0 (`fluent.syntax` 0.17), including parameterized terms.

4.4 fluent.runtime 0.1 (January 21, 2019)

First release to PyPI of `fluent.runtime`. This release contains a `FluentBundle` implementation that can generate translations from FTL messages. It targets the Fluent 0.7 spec.