Интернационализация и Локализация

в фреймворке Aiogram.

Оглавление

Интернационализация и Локализация.

Интернационализация (англ. internationalization или сокращенно i18n) – это процесс проектирования и разработки, который позволяет легко перевести продукт, приложение или документ на другие языки и регионы без необходимости внесения технических изменений. По сути, речь идет о том, чтобы продукт или услуга разрабатывались таким образом, чтобы не отдавать предпочтение одной культуре или языку по сравнению с другой, и подготовить их для мировой аудитории.

Локализация (англ. localization или l10n) – это процесс перевода и культурной адаптации продукта к особенностям определенной страны, региона.

В принципе можно пойти в лоб и создать по одному приложению на каждый регион/страну. Такой вариант очень дорого и неудобно поддерживать, ведь работать нужно будет с каждым продуктом отдельно. Далее приходит на ум более удобный вариант — создать приложение, которое включает в своём коде локализацию для всех необходимых вам регионов/стран. Вариант уже лучше, но и его достаточно сложно будет поддерживать + велика вероятность, что данные локализации будут вплотную пересекаться с основным кодом приложения. Далее мы приходим к мысли, что было бы удобно создать продукт, в котором региональные и культурные особенности (текст, картинки, форматы даты, времени и т.п.) будут вынесены в отдельные блоки (никакого хардкода в переводимых местах), которые будут подгружаться при использовании того или иного региона/страны. Данный набор ресурсов называют "локалью" (locale).

Реализацию интернационализации обычно начинают на ранних этапах проекта, чтобы подготовить ваш продукт к будущей локализации. Во время процесса интернационализации определяют, что будет изменяться для будущих локален (например текст, изображения и т.п.) и выносят эти данные во внешние файлы. Также во время интернационализации (и при локализации тоже) нужно добавить возможность изменять календари, форматы даты, времени, цифр, денежных символов и в целом символов, специфичных для определенных языков и многое другое. Как итог, в идеальном варианте, добавление новой окали не должно требовать от нас изменения исходного кода продукта.

Ни конечно очень тесно связана с этим процессом локализация. На этой стадии участники разработки продукта работают с локациями — внешними ресурсами (файлами), которые подгружаются приложением для загрузки локализации для вашей страны/региона. Основные зоны локализации, то есть адаптации к местным нормам и традициям:

- текст и связанные с ним функции (например сортировка, поиск, поддержка спец. символов и т.п.)
- документация (мануалы, гайды, FAQ, разделы справки и т.п.)
- форматы даты и времени (Месяц/Дата/Год (США) или День Месяц Год (Россия))
- формат чисел (разделитель десятичных знаков точка или запятая)
- формат денежных величин
- поддержка различных календарей (например, неделя начинается с понедельника (Европа) или с воскресения (США), праздники по лунному календарю (Китай))
- изображения (картинки, иконки)
- звук (в частности, озвучка, если таковая имеется)
- реклама (текстовая, аудио, видео)
- и т.д.

Если говорить очень по-простому, то **интернационализация — это проектирование и написание кода**, пригодного для перевода на разные языки.

А **локализация — это перевод и культурная адаптация** всех элементов, такие как тексты, картинки, шрифты.

Подробнее можно ознакомиться здесь:

http://www.motaword.com/ru/blog/localization-vs-internationalization

https://habr.com/ru/articles/532836/

https://phrase.com/blog/posts/translation-technology/

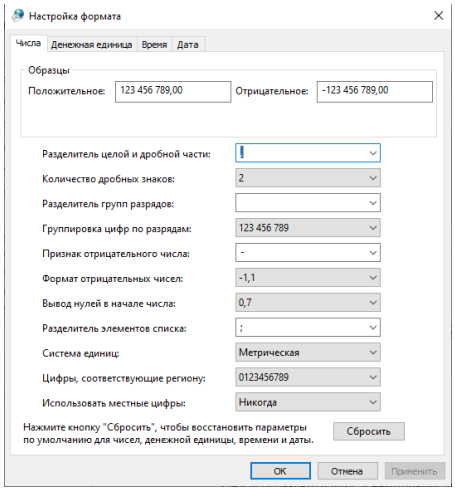
Локали.

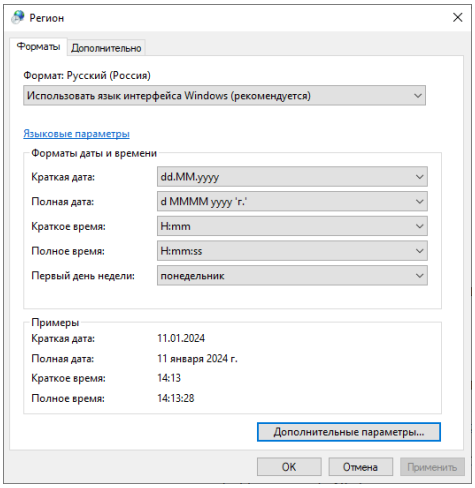
Локали — это набор настроек, которые определяют язык, который использует система, а также региональные особенности, такие как денежные знаки, формат чисел, даты и времени и наборы символов.

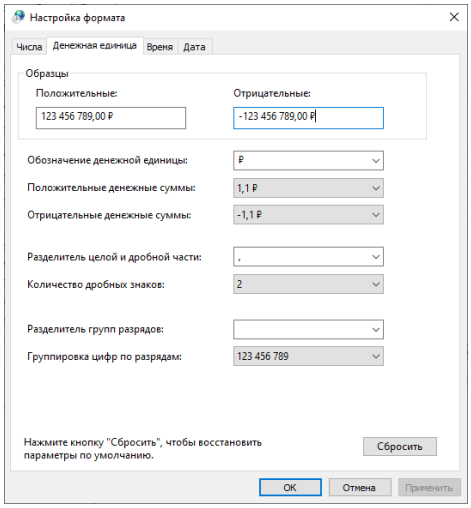
Если у вас Windows, то выполните команду

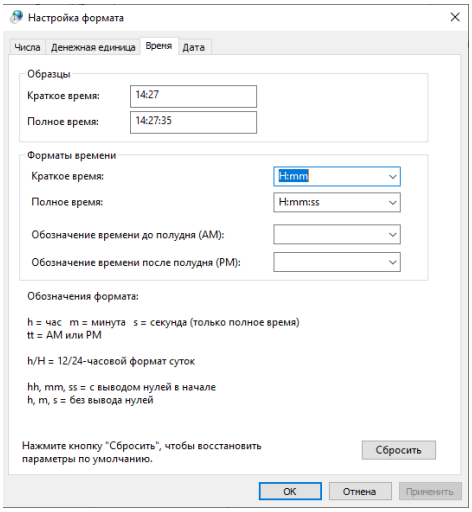
```
intl.cpl
```

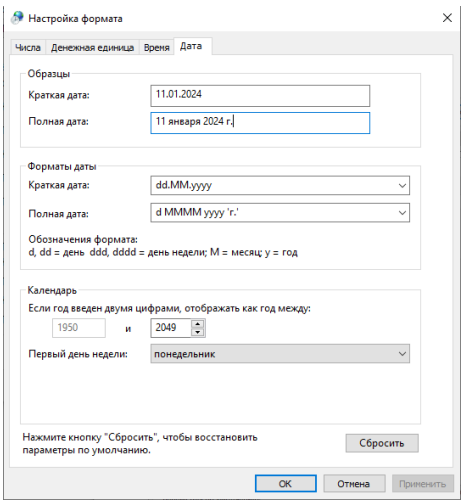
и вы увидите региональные настройки и в дополнительных параметрах форматы чисел, денег, времени, дат.











Это все требует не просто перевода, а подлежит приведению в соответствии с особенностями региона.

Поскольку нам интересен запуск на сервере, то мы будем рассматривать все на примере ОС Linux. В Linux эти настройки хранятся в переменных среды. Выполним команду:

```
locale
```

И получим список переменных среды, в которых хранятся все региональные настройки:

```
LANG=en_US.UTF-8
LANGUAGE=
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

А сами переводы для программ лежат по пути /usr/share/locale/ (или /usr/local/share/locale или в директории приложение/locale) в папках с именем языка и подпапкой соответствующей переменной среды для которой перевод. Например, для русского языка это ru/LC_MESSAGES

И в этой папке скомпилированные переводы для приложений по имени каждого приложения, например:

```
apt.mo
debconf.mo
dpkg.mo
mc.mo
xdg-user-dirs.mo
```

А само имя нашего приложения в дальнейшем для перевода обозначается в терминологии инструментов перевода как **домен**. Если мы создаем приложение на Python как пакет, то доменом у нас будет название приложения, совпадающее с папкой пакета.

Для нашего приложения нам нет необходимости переводить большой объем разных данных и устанавливать в систему, а будем переводить только строки, которые выводятся пользователю LC_MESSAGES.

При запуске любого приложения (не важно на каком языке это все написано, python или C) инструменты перевода производят поиск файлов перевода для подстановки переведенных строк в папке, указанной в настройках и файлы с именем домена. Для нашего приложения это будет папка с именем locales в корне проекта. Далее это мы подробнее разберем на практике.

Интернационализация в Aiogram утилитами GNU gettext.

В Aiogram предусмотрена возможность использования интернационализации (https://docs.aiogram.dev/en/latest/utils/i18n.html) с помощью библиотеки GNU gettext (https://docs.python.org/3/library/gettext.html) и инструментов Babel (https://babel.pocoo.org/en/latest/).

Установка пакета для переводов происходит через дополнительную зависимость командой

```
pip install aiogram[i18n]
```

или непосредственно устанавливаем сам Babel

```
pip install Babel
```

Первый и важный шаг для работы по интернационализации — нам нужно подготовить наш код таким образом, чтобы он смог использовать файлы перевода и загружать необходимые фразы из нужной локали. Для этого все переводимые строки необходимо обернуть функцией gettext. Функцию подстановки перевода из gettext принято обозначать _ - одинарное нижнее подчеркивание, а вызов этой функции _().

```
from aiogram import html
from aiogram.utils.i18n import gettext as _  # импортируем модуль gettext из aiogram utils как _
```

Обертываем все строки, которые нуждаются в переводе функцией gettext.

Было:

```
async def my_handler(message: Message) -> None:
    await message.answer(f"Hello, {html.quote(message.from_user.full_name}!"))
```

Стало:

```
async def my_handler(message: Message) -> None:
    await message.answer(_("Hello, {name}!").format(name=html.quote(message.from_user.full_name)))
```

Внимание. Gettext не может использовать f-строки. Поскольку при использовании f-строк нельзя сначала создать шаблон, а затем его использовать. Это происходит из- за того, что f-строка сразу выполняется и в нее подставляются значения переменных, которые должны быть определены ранее. А у нас сначала должен произойти перевод с подстановкой в шаблон строки. Поэтому нужно использовать метод строк format().

Более того, когда нам необходимо использовать перевод в фильтрах ключевых слов или магических фильтрах, то нужно будет использовать ленивые вызовы gettext - **lazy_gettext**, которые будут обозначены __ - двойное подчеркивание, а вызов этой функции __().

```
from aiogram import F
from aiogram.utils.i18n import lazy_gettext as __  # Импортируем функцию ленивого вызова gettext как _ _

@router.message(F.text == __("Start"))
        …
```

В документации особо обращено внимание на то, что ленивые вызовы lazy gettext всегда следует использовать, если текущий язык в данный момент неизвестен.

Также важно, что lazy gettext нельзя использовать в качестве значения для методов API или любого объекта Telegram (например, для aiogram.types.inline_keyboard_button.InlineKeyboardButton и т. д.).

Конфигурация движка перевода

Сначала в коде проекта мы создаем объект класса I18n, чтобы было понятно, какой язык будет использоваться:

```
i18n = I18n(path="locales", default_locale="en", domain="my-super-bot")
```

где path= путь к папкам с локалями, в данном случае путь будет сформирован будет так: locales/{language}/LC_MESSAGES/messages.po, и мы указываем верхний уровень locales, исходя из нашей структуры.

```
        …
locales
├── messages.pot
├── en
│   └── LC_MESSAGES
│       └── my-super-bot.mo
├── ru
│   └── LC_MESSAGES
│       └── my-super-bot.mo
…
```

default_locale= локаль по умолчанию,

domain= домен - это название домена переводов в gettext, по сути это название приложения, для которого будет создана локаль (используется чаще название того приложения, что мы переводим).

Движок перевода - это middleware для I18n. И теперь нам необходимо выбрать движок перевода, основанный на 3 встроенных в aiogram классах middleware из aiogram.utils.i18n.middleware:

SimpleI18nMiddleware - выбирает код языка из объекта User, полученного в событии. Однако не все клиенты Telegram отдают это значение. Очень часто объект language_code не заполнен и является пустой строкой.

ConstI18nMiddleware - выбирает статически определенную локаль.

FSMI18nMiddleware - хранит локаль в хранилище FSM.

Так же есть 4 вариант:

I18nMiddleware - это базовый абстрактный класс для наследования и создания собственного обработчика.

Наш код будет выглядеть примерно так:

```
from aiogram import Bot, Dispatcher, F
from aiogram.types import Message

from aiogram.utils.i18n import gettext as _
from aiogram.utils.i18n import lazy_gettext as __
from aiogram.utils.i18n import I18n, ConstI18nMiddleware
TOKEN = "token"
dp = Dispatcher()
@dp.message(F.text == __('Test'))
async def test1(message: Message) -> None:
    await message.answer(_("Hello, {name}!").format(name=html.quote(message.from_user.full_name)))

def main() -> None:
    bot = Bot(TOKEN, parse_mode="HTML")
    i18n = I18n(path=”locales”, default_locale="en", domain=”my-super-bot”)
    dp.message.outer_middleware(ConstI18nMiddleware(locale='en', i18n=i18n))
    dp.run_polling(bot)

if __name__ == "__main__":
    main()
```

Локализация Aiogram, создание переводов и работа с Babel.

Шаблоны переводов

Переходим ко второму шагу к локализации. Теперь нам необходимо создать сами переводы, основываясь на переменных, которые уже есть в нашем коде и создать папки по такой структуре.

```
…
locales
├── messages.pot
├── en
│   └── LC_MESSAGES
│       └── my-super-bot.po
├── ru
│   └── LC_MESSAGES
│       └── my-super-bot.po
…
```

Предварительно нужно только создать папку locales в корне проекта.

Остальная структура создается автоматически с помощью ранее установленного пакета утилит Babel.

Создаем основу — шаблон переводов. Запускаем в корне проекта из командной строки команду:

```
pybabel extract --input-dirs=. -o locales/messages.pot
```

Утилита проходит по нашим файлам и извлекает все строковые переменные, обернутые функциями _() и __(), в файлик messages.pot.

У нас получится такой файл — это шаблон переводов, на основании которого генерируются переводы:

```
# Translations template for Bot Super Project.
# Copyright (C) 2024 John Doe
# This file is distributed under the same license as the Bot Super Project
# project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2024.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: Bot Super Project 0.1\n"
"Report-Msgid-Bugs-To: john@doe-email.com\n"
"POT-Creation-Date: 2024-01-12 16:11+0500\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 2.13.1\n"

#: lesson1.py:13
msgid "Hello, {name}!"
msgstr ""
```

Обратите внимание, что при работе с gettext и Babel все комментари являются значимыми, то есть их нельзя удалять.

Файлик мы заполним своими данными в шапке — это название проекта, версия, копирайты и электронный адрес для связи в случае багов. По идее это можно сразу сделать из командной строке при формировании шаблона переводов:

```
 pybabel extract -o locales/messages.pot --copyright-holder="John Doe" --project="Bot Super Project" --version=0.1
--msgid-bugs-address=john@doe-email.com --input-dirs=.
```

Шаблон генерируется каждый раз после исправления или доработки кода, поэтому мы не храним его в репозитории исходников git.

На основании шаблона будут создаваться файлы переводов на нужные нам языки.

Давайте создадим файл перевода на английский язык. Выполним в командной строке:

```
pybabel init -i locales/messages.pot -d locales -D my-super-bot -l en
```

А затем на русский:

```
pybabel init -i locales/messages.pot -d locales -D my-super-bot -l ru
```

Где,

```
-i locales/messages.pot - путь к нашему шаблону .pot
-d locales - наш каталог переводов
-D my-super-bot -  наш домен переводов
-l en – код языка.
```

Будет создан файл перевода my-super-bot.po в папке locales/en/LC_MESSAGES/ и locales/ru/LC_MESSAGES/.

Файлы переводов .po

Файлы в формате .po предназначены для переводчиков. И храним мы их в репозитории в development ветке. Они нам нужны на случай изменения или добавления строк в проекте (об этом чуть позже).

Сначала откроем созданные файлы и отредактируем их.

Нас интересуют строки вида

```
#: lesson1.py:13
msgid "Hello, {name}!"
msgstr ""
```

В комментарии указан файл, откуда взялась текстовая строка и номер строки в этом файле. Затем идентификатор msgid и перевод msgstr, который будет подставлен пользователю с выбранным языком. Заполняем перевод.

```
#: lesson1.py:13
msgid "Hello, {name}!"
msgstr "Привет, {name}!"
```

Теперь пользователь у которого язык английский, получит английское сообщение , а русский — русское. Естественно какой у пользователя язык, мы должны считать через наш middleware i18n.

Затем компилируем переводы в файлы .mo и готово:

```
pybabel compile -d locales -D my-super-bot
```

Внесение изменений в файлы переводов .po

Разберем еще один момент, связанный с изменениями переводов.

В какой-то момент мы решили изменить логику бота. И изменили код программы, изменив старые строки и добавив новые. Естественно мы вносим изменения в код в парадигме интернационализации.

```
from aiogram import Bot, Dispatcher, F, html
from aiogram.types import Message
from aiogram.utils.i18n import gettext as _


from aiogram.utils.i18n import lazy_gettext as __
from aiogram.utils.i18n import I18n, ConstI18nMiddleware

TOKEN = "token"
dp = Dispatcher()



@dp.message(F.text == __('start'))
async def handler_1(message: Message) -> None:
    await message.answer(_("Welcome, {name}!").format(name=html.quote(message.from_user.full_name)))
    await message.answer(_("How many coins do you have? Input number, please:"))

@dp.message(F.text)
async def handler_2(message: Message) -> None:
    await message.answer(_("You have {} coins!").format(message.text))



def main() -> None:
    bot = Bot(TOKEN, parse_mode="HTML")
    i18n = I18n(path="locales", default_locale="en", domain="my-super-bot")
```

```
    dp.message.outer_middleware(ConstI18nMiddleware(locale='en', i18n=i18n))
    dp.run_polling(bot)


if __name__ == "__main__":
    main()
```

Мы добавили вопрос к пользователю и переделали приветственное сообщение.

Теперь нам снова нужно извлечь строки. Формируем .pot файл. Для удобства в версию добавляем минорный релиз.

```
pybabel extract -o locales/messages.pot --copyright-holder="John Doe" --project="Bot Super Project" –version=0.1.1
--msgid-bugs-address=john@doe-email.com –input-dirs=.
```

И получаем новый шаблон:

```
# Translations template for Bot Super Project.
# Copyright (C) 2024 John Doe
# This file is distributed under the same license as the Bot Super Project
# project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2024.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: Bot Super Project 0.1.1\n"
"Report-Msgid-Bugs-To: john@doe-email.com\n"
"POT-Creation-Date: 2024-01-12 17:25+0500\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 2.13.1\n"

#: lesson1.py:15
msgid "Welcome, {name}!"
msgstr ""

#: lesson1.py:16
msgid "How many coins do you have? Input number, please:"
msgstr ""

#: lesson1.py:20
msgid "You have {} coins!"
msgstr ""
```

Обновляем файлы переводов командой update.

```
pybabel update -i locales/messages.pot -d locales -D my-super-bot -l ru
pybabel update -i locales/messages.pot -d locales -D my-super-bot -l en
```

И мы видим следующую картину.

```
# Russian translations for Bot Super Project.
# Copyright (C) 2024 John Doe
# This file is distributed under the same license as the Bot Super Project
# project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2024.
#
msgid ""
msgstr ""
"Project-Id-Version: Bot Super Project 0.1\n"
"Report-Msgid-Bugs-To: john@doe-email.com\n"
"POT-Creation-Date: 2024-01-12 17:28+0500\n"
"PO-Revision-Date: 2024-01-12 16:16+0500\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language: ru\n"
"Language-Team: ru <LL@li.org>\n"
"Plural-Forms: nplurals=3; plural=(n%10==1 && n%100!=11 ? 0 : n%10>=2 && "
"n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2);\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 2.13.1\n"
```

```
#: lesson1.py:15
#, fuzzy
msgid "Welcome, {name}!"
msgstr "Привет, {name}!"

#: lesson1.py:16
msgid "How many coins do you have? Input number, please:"
msgstr ""

#: lesson1.py:20
msgid "You have {} coins!"
msgstr ""
```

Прежний перевод сохранился, но при этом у нас строка была изменена с Hello на Welcome.

Babel увидел это, сохранил нам строку но пометил перевод коментарием #, fuzzy что обозначает нечеткий перевод. Если скомпилировать сразу, то эта строка не будет переводиться.

Нам нужно поправить текст и убрать эту метку fuzzy/

```
#: lesson1.py:15
msgid "Welcome, {name}!"
msgstr "Добро пожаловать, {name}!"

#: lesson1.py:16
msgid "How many coins do you have? Input number, please:"
msgstr "Сколько у тебя монет? Введи число, пожайлуйста:"

#: lesson1.py:20
msgid "You have {} coins!"
msgstr "У тебя {} монет!"
```

То же самое делаем со вторым языком, и снова компилируем переводы.

В результате у нас все хорошо кроме такого момента.

Если мы введем 1, то бот ответит «У тебя 1 монет!» или «You have 1 coins!», что с точки зрения языка — неверно.

1 монета 2, 3 или 4 монет, 11 монет, А если слово сообщения, то 1 сообщение, 2 сообщения, 10 сообщений. И в английском у нас тоже проблема со множественными числами — 1 coins.

Множественные формы

Давайте победим и эту историю.

Помните, я говорил о значащих комментариях в файлах. В частности в файле переводов .po для каждого языка формируется формула, которая определяет количество множественных форм и правила их формирования. Тут будет вся магия работы с переводами. Она содержится в строчках:

```
"Plural-Forms: nplurals=3; plural=(n%10==1 && n%100!=11 ? 0 : n%10>=2 && "
"n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2);\n"
```

Это та формула по которой определяется для конкретного языка форма слова во множественном числе.

Для начала нам нужно вернуться к интернационализации нашего кода. Функция gettext не умеет работать со множественными формами. Для этого существует ngettext из стандартной библиотеки python https://docs.python.org/3/library/gettext.html#gettext.ngettext, но для удобства в Aiogram это уже все спрятано в функции gettext из aiogram.utils.i18n. Для одного языка (в нашем случае идентификаторы на английском) мы передаем фразу в единственном , затем во множественном числе, и указываем количество множественных форм 2 для английского.

Изменим наш код.

```
import os

from aiogram import Bot, Dispatcher, F, html
from aiogram.types import Message
from aiogram.utils.i18n import gettext as _
```

```
from aiogram.utils.i18n import lazy_gettext as __
from aiogram.utils.i18n import I18n, ConstI18nMiddleware

TOKEN = "A:1234567890"
dp = Dispatcher()


@dp.message(F.text == __("Start"))
async def handler_1(message: Message) -> None:
    await message.answer(_("Welcome, {name}!").format(name=html.quote(message.from_user.full_name)))
    await message.answer(_("How many coins do you have? Input number, please:"))


@dp.message(F.text)
async def handler_2(message: Message) -> None:
    try:
        n = int(message.text)
        await message.answer(_("You have {} coin!", "You have {} coins!", n).format(n))
    except:
        await message.answer(_("Please, enter a number"))


def main() -> None:
    bot = Bot(TOKEN, parse_mode="HTML")
    i18n = I18n(path="locales", default_locale="ru", domain="my-super-bot")
    dp.message.outer_middleware(ConstI18nMiddleware(locale='ru', i18n=i18n))
    dp.run_polling(bot)


if __name__ == "__main__":
    main()
```

И теперь извлечение нужно произвести с опцией `-k _:1,1t -k _:1,2` и -k __ для lazy gettext (два подчеркивания)

```
pybabel extract -o locales/messages.pot -k _:1,1t -k _:1,2 -k __ --copyright-holder="John Doe" --project="Bot
Super Project" --version=0.1.1 --msgid-bugs-address=john@doe-email.com --input-dirs=.
```

Babel может неадекватно извлекать строки, поэтому можно воспользоваться командой xgettext из пакета утилит GNU gettext.

```
xgettext -L Python --keyword=_:1,2 --keyword=__ -d my-super-bot
```

Заглянем в наш шаблон .pot, и увидим, что теперь перевод имеет строку для перевода единственного и множественного числа:

```
…
#: lesson1.py:19
msgid "You have {} coin!"
msgid_plural "You have {} coins!"
msgstr[0] ""
msgstr[1] ""
…
```

Обновим перевод:

```
pybabel update -i locales/messages.pot -d locales -D my-super-bot -l ru
```

При генерации Babel по коду языка сгенерировал в файле .po для каждого языка свою формулу определения форм слова. При этом в коде выше в функции handler_2 мы указали всего две формы для английского языка, а Babel сгенерировал так, как нужно.

В английской версии у нас две формы единственное и множественное число:

```
…

"Plural-Forms: nplurals=2; plural=(n != 1);\n"
…
#: lesson1.py:19
msgid "You have {} coin!"
msgid_plural "You have {} coins!"
msgstr[0] ""
msgstr[1] ""
```

И ниже он пометил старые строки удаленными (У меня не было перевода и Babael посчитал их не нужными)

```
#~ msgid "You have {} coins!"
#~ msgstr ""
…
```

А в русском языке три формы. Единственное, малое множественное и множественное:

```
…
"Plural-Forms: nplurals=3; plural=(n%10==1 && n%100!=11 ? 0 : n%10>=2 && "
"n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2);\n"
…
#: lesson1.py:19
#, fuzzy
msgid "You have {} coin!"
msgid_plural "You have {} coins!"
msgstr[0] "У Вас {} монет!"
msgstr[1] ""
msgstr[2] ""
…
```

Здесь Babel сохранил наш старый перевод и пометил как не точный fuzzy. Чтоб мы доделали перевод.

Формула для вычисления множественных форм - это обычное тернарное булево выражение на СИ-подобном языке. И именно для ее работы мы компилируем переводы.

Итак, в английском у нас две формы слова nplurals=2. А plural=(n != 1);\n" означает:

- если цифра равна 1, то форма слова не является множественным числом.
- в остальных случаях это множественное число.

В русском три формы слова nplurals=3. Формула plural=(n%10==1 && n%100!=11 ? 0 : n%10>=2 && "
"n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2);\n" означает:

- первая форма - это и единственное и множественное число для чисел заканчивающихся на 1 (за исключением заканчивающихся на 11), то есть 1 монета, 101 монета, но 111 монет.
- Вторая форма для чисел, заканчивающихся на 2, 3 и 4. Например, 3 монеты и 44 монеты.
- все остальное третья форма, 5, 56, 120, 129, 1007сообщений а так же сюда попадают и 11, 211 и т.д сообщений.

А выбор перевода это просто взятие k-го элемента массива, где k вычислено по этой формуле.

Переводим недостающие элементы, не забываем удалить строки, помеченные для удаления, и метки неточного перевода fuzzy.

Компиляция переводов, файлы формата mo.

Особенность работы с gettext и Babel заключается в том, что все файлы переводов должны быть предварительно скомпилированы, поскольку переводы вибираются по формулам.

Компилируем переводы командой:

```
pybabel compile -d locales -D messages
```

И получаем в нашей локали файлы .mo, радом с файлами .po.

Файлы mo храним в репозитории в ветке production, они нам нужны для работы готовой программы (в отличии от файлов .po, которые напомню, для разработки)

Автоматические переводы

Для переводчиков существую целые платформы онлайн и оффлайн программы для переводов программного обеспечения, как платные, так и бесплатные. Основные удобства заключаются в реализации базовых вещей для перевода:

- Организация памяти переводов (Translation memory),

- Автоматизация и черновые машинные переводы,
- Работа в сообществе и соавторстве,
- Поддержка контекстов переводов (Например, слово Save в меню — сохранить, Save в играх — спасти).

Примеры инструментов:

https://crowdin.com/

https://www.transifex.com/open-source/

https://pofile.net/ - PoEdit, бесплатный онлайн инструмент для работы с .po файлами. (будьте аккуратнее — оно исправляет plural forms по очень странным правилам)

https://poedit.net/ - PoEditor, оффлайн программа для удобного пакетного перевода .po файлов (в версии про умеет использовать нейросеть DeepL).

https://weblate.org/ru/ - инструмент он-лайн переводов, который также можно развернуть из Docker на своем сервере.

https://omegat.org/ - OmegaT проект организации памяти переводов (Translation memory)

Пример использования OmegaT: https://blog.wtigga.com/tag/omegat/

https://www.deepl.com/translator

Ну и еще их куча как пример поиска в интернете:

https://www.g2.com/categories/translation-management/free

https://аналог-программы.рф/app/poedit

https://suse.me/apps/poedit/

Финальный результат.

```
from aiogram import Bot, Dispatcher, F, html
from aiogram.types import Message
from aiogram.utils.i18n import gettext as _


from aiogram.utils.i18n import lazy_gettext as __
from aiogram.utils.i18n import I18n, ConstI18nMiddleware

TOKEN = "token"
dp = Dispatcher()


@dp.message(F.text == __("Start"))
async def handler_1(message: Message) -> None:
    await message.answer(_("Welcome, {name}!").format(name=html.quote(message.from_user.full_name)))
    await message.answer(_("How many coins do you have? Input number, please:"))


@dp.message(F.text)
async def handler_2(message: Message) -> None:

    try:
        n = int(message.text)
        await message.answer(_("You have {} coin!", "You have {} coins!", n).format(n))
    except ValueError:
        await message.answer(_("Please, enter a number"))


def main() -> None:
    bot = Bot(TOKEN, parse_mode="HTML")
    i18n = I18n(path="locales", default_locale="en", domain="my-super-bot")
    dp.message.outer_middleware(ConstI18nMiddleware(locale='ru', i18n=i18n))
    dp.run_polling(bot)
```

Запускаем код, указываем константный русский язык в строке

```
dp.message.outer_middleware(ConstI18nMiddleware(locale='ru', i18n=i18n))
```

Меняем значение на en и снова запускаем и тестируем.

```
dp.message.outer_middleware(ConstI18nMiddleware(locale='en', i18n=i18n))
```

Для динамического переключения языков, нам нужно хранить язык в базе данных и реализовать свой класс middleware на базе I18nMiddleware из aiogram.utils.i18n.middleware. Это мы сделаем чуть позже. А пока разберемся с еще одним инструментом для локализации и интернационализации на базе проекта Fluent от Mozilla.

Локализация и интернационализация на базе проект Fluent от Mozilla.

Fluent — это семейство спецификаций, реализаций и практик локализации, разработанных компанией Mozilla, при разработке своего браузера Firefox и сопутствующих продуктов. С проектом Fluent вы можете ознакомиться здесь: https://projectfluent.org/ и реализацией под Python: https://github.com/projectfluent/python-fluent

Одной из постоянных задач глобальной разработки программного обеспечения является сокращение технического долга и устаревшего кода. Расстановка приоритетов имеет место, когда становится ясно, что организации необходимо заменить устаревший код чем-то более эффективным и современным. Очень часто устаревший код, влияющий на интернационализацию (i18n) и локализацию (l10n), оказывается одной из последних областей кодовой базы, которым уделяется приоритетное внимание.

Согласно традиционному процессу локализации программного обеспечения, локализованный продукт создается в результате объединения статических языковых ресурсов в исполняемый файл, который затем распространяется среди пользователей. Любое обновление этих языковых ресурсов требует создания нового исполняемого файла и передачи его пользователям по цепочке распространения. По этой причине большинство компаний-разработчиков программного обеспечения предпочитают откладывать обновления локализации с момента их доступности до момента, когда их можно будет объединить с другими улучшениями программного обеспечения.

С помощью Fluent этот процесс можно разделить, что позволяет выпускать обновления локализации независимо от более широкого графика выпуска. Языковые ресурсы не являются частью программного пакета, а доставляются пользователям посредством безопасных вызовов API при запуске программного обеспечения. Более того, эти вызовы API позволяют доставлять обновления локализации без вмешательства пользователя — нет необходимости вручную инициировать обновление или даже перезапускать программное обеспечение.

Еще одна проблема, которую решает проект, в локализации программного обеспечения доминирует устаревшая парадигма: перевод представляет собой всего лишь словарь строк, которые взаимно однозначно сопоставляются с английской (en-US) копией. Эта парадигма несправедлива и ограничивает языки с более сложной грамматикой, чем английская. Для любой грамматической функции, не поддерживаемой английским языком, в исходный код должен быть добавлен специальный случай, что приводит к пробросу этой логики во все переводы. Более того, создание хороших пользовательских интерфейсов, которые зависят от множества внешних аргументов, сложно и требует от разработчика понимания грамматики языков, на которые нацелен продукт.

Fluent поддерживает ассиметричную локализацию. Асимметричная локализация не ограничивается множественным числом. Свободный перевод может варьироваться в зависимости от пола, грамматического падежа, операционной системы и многих других переменных. Все это происходит изолированно; тот факт, что один язык имеет преимущества более продвинутой логики, не требует какой-либо другой локализации для его применения. Каждая локализация контролирует сложность перевода. То есть Fluent дает переводчикам возможность создавать грамматически правильные переводы и использовать выразительную силу своего языка.

Кроме этого, более гибок процесс отображения непереведенных элементов. Он не привязан жестко к английскому варианту, а выбирается из цепочки резервных локалей, которые понимает конкретный не англоговорящий пользователь.

Обзорную статью можно посмотреть здесь: https://multilingual.com/issues/sept-oct-2019/fluent-firefoxs-new-localization-system/

Хорошие практики и описание почему при переводе надо отказаться от принципа DRY (Don't Repeat Yourself) в пользу принципа WET (Write Everything Twice), вы можете прочитать здесь: https://github.com/projectfluent/fluent/wiki/Good-Practices-for-Developers

Вообще интернационализация и локализация заставляют применять иные и очень разнообразные подходы к разработке программного обеспечения. Приходится совмещать несовместимые вещи, такие как избежание дублирования кода с многократным повторением кода и текстов, адаптированных под национальные особенности.

Интернационализация в Aiogram с использованием Fluent.

Для хорошей поддержки интернационализации и локализации разработчики aiogram создали отдельный пакет aiogram_i18n, в который добавили поддержку движка локализации Fluent для Python https://github.com/projectfluent/python-fluent.

Установим его с помощью команды:

```
pip install aiogram_i18n
```

Также нам потребуются FluentCompileCore и FluentRuntimeCore

```
pip install fluent_compiler
pip install fluent.runtime
```

В предыдущем разделе мы рассказывали о том, что подход к созданию файлов перевода у Fluent в корне отличается от привычного. По сути теперь интернационализация полностью в ваших руках, а локализацию выполняет ядро Python Fluent. У нас теперь нет шаблонов перевода и мы не извлекаем строки из исходного кода нашего проекта. Отказ от использования строк английского текста как ключей, накладывает некоторые ограничения в угоду гибкости самих переводов. Теперь нам нужно самим проектировать ПО так, чтобы перевод был возможен, и по началу будет много ручной работы. Проект Fluent (2017 год) по меркам gettext (1995 год) молодой и инструменты автоматизации, которые я нашел, созданы в основном для JavaScript (в силу специфики разработки под локализацию браузера). Поэтому пока будем создавать файлы перевода вручную.

Создадим код нашего нового проекта:

```
import asyncio
from logging import basicConfig, INFO
from typing import Any

from aiogram import Router, Dispatcher, F, Bot
from aiogram.enums import ParseMode
from aiogram.filters import CommandStart
from aiogram.types import Message

from aiogram_i18n import I18nContext, LazyProxy, I18nMiddleware
from aiogram_i18n.cores.fluent_runtime_core import FluentRuntimeCore
from aiogram_i18n.types import (
    ReplyKeyboardMarkup, KeyboardButton
    # you should import mutable objects from here if you want to use LazyProxy in them
)

router = Router(name=__name__)
rkb = ReplyKeyboardMarkup(
    keyboard=[
        [KeyboardButton(text=LazyProxy("help"))]  # or L.help()
    ], resize_keyboard=True)


@router.message(CommandStart())
async def cmd_start(message: Message, i18n: I18nContext) -> Any:
    name = message.from_user.mention_html()
    return message.reply(
        text=i18n.get("hello", user=name),  # or i18n.hello(user=name)
        reply_markup=rkb)


@router.message(F.text == LazyProxy("help"))
async def cmd_help(message: Message) -> Any:
    return message.reply(text="-- " + message.text + " --")


async def main() -> None:
    basicConfig(level=INFO)
    bot = Bot("TOKEN", parse_mode=ParseMode.HTML)
    i18n_middleware = I18nMiddleware(
        core=FluentRuntimeCore(
            path="locales/{locale}/LC_MESSAGES",
        ),
        default_locale="ru")

    dp = Dispatcher()
    dp.include_router(router)
```

```
    i18n_middleware.setup(dispatcher=dp)

    await dp.start_polling(bot)


if __name__ == "__main__":
    asyncio.run(main())
```

Импортируем следующие объекты:

```
from aiogram_i18n import I18nContext, LazyProxy, I18nMiddleware
from aiogram_i18n.cores.fluent_runtime_core import FluentRuntimeCore
from aiogram_i18n.types import (
    ReplyKeyboardMarkup, KeyboardButton
    # you should import mutable objects from here if you want to use LazyProxy in them
)
```

Нам понадобится сам движок FluentRuntimeCore, а также контекст I18nContext, и один из вариантов middleware (для примера я взял I18nMiddleware). Также нужные нам LazyProxy и изменяемые объекты Aiogram, такие как клавиатура, которые нужно брать именно из aiogram_i18n.types. Эти объекты нам нужны, когда работа с объектом происходит, но язык еще не известен, так как код выполняется еще за пределами роутеров.

Создадим объект нашей middleware

```
…
i18n_middleware = I18nMiddleware(
        core=FluentRuntimeCore(
            path="locales/{locale}/LC_MESSAGES", # путь к папке локалей
            ),
        default_locale="ru") # язык интерфейса. Переключать научимся позже.
…
```

И зарегистрируем его через встроенный метод setup (в этом методе реализована регистрация компонентов в нужном порядке)

```
…
i18n_middleware.setup(dispatcher=dp)
…
```

Создадим файл переводов в формате FTL (Fluent Translation List).

Файл с английским переводом my-super-bot.ftl положим в папку locales/en/LC_MESSAGES:

```
hello = Hello, <b>{ $user }</b>!
cur-lang = Your current language: <i>{ $language }</i>
help = Help
```

Файл с русским переводом my-super-bot.ftl положим в папку locales/ru/LC_MESSAGES:

```
hello = Привет, <b>{ $user }</b>!
cur-lang = Текущий язык : <i>{ $language }</i>
help = Помощь
```

Запустим и проверим работу на русском языке. Затем изменим язык в middleware и проверим на английском:

```
i18n_middleware = I18nMiddleware(
        core=FluentRuntimeCore(
            path="locales/{locale}/LC_MESSAGES", # путь к папке локалей
            ),
        default_locale="en")
```

Локализация Aiogram с помощью проекта Fluent.

Синтаксис Fluent.

Официальная справка по синтаксису: https://projectfluent.org/fluent/guide/

FTL спроектирован так, чтобы его было легко читать, но в то же время он позволял представлять сложные понятия из естественных языков, такие как род, множественное число, спряжения и другие.

Во Fluent основная единица перевода называется сообщением. Сообщения — это контейнеры для информации. Вы используете сообщения для идентификации, хранения и вызова информации о переводе, которая будет использоваться в продукте. Базовый синтаксис это — идентификатор = перевод. В синтаксисе идентификаторов используется kebab-case.

```
# Это комментарий
# Ниже идет сообщение
Hello = Привет

# Это пример многострочного сообщения, отступы обязательны. Длинные идентификаторы пишутся через дефис. Можно
включать другие сообщения
long-message = { Hello }, это
    многострочное сообщение.
    Продолжение начинается с отступа.
    Отступ может быть только пробелом.
    Табуляция не считается отступом.
```

Синтаксис Fluent поддерживает ссылки на другие сообщения, переменные, селекторы, и даже функции.

Давайте посмотрим их использование на примере нашего кода про монеты.

Добавим наш хэндлер про монеты. При вводе числа бот будет писать сколько у вас монет.

```
@router.message(F.text)
async def handler_2(message: Message, i18n: I18nContext) -> None:
    try:
        name = message.from_user.mention_html()
        n = int(message.text)
        await message.answer(text=i18n.get("you-have-coin", value=n, user=name))
    except ValueError as e:
        logging.log(INFO, e)
        await message.answer(text=i18n.get("enter-a-number"))
```

Нам потребуется имя пользователя и количество монет, чтобы передать эти данные в перевод. В переменную value попадет n, а в переменную user отправим name. Переменные отправляются как именованные аргументы (kwargs).

В нашем файле перевода добавим сообщение с использованием переменных и селекторов, которые помогут выбрать форму сообщения в зависимости от количества монет. И для удобства и красоты добавим селектор [0], который будет выводить сообщение что монет нет, без указания самого числа, если монет 0. Будьте внимательны при переносе строк.

```
hello = Привет, <b>{ $user }</b>!
cur-lang = Текущий язык : <i>{ $language }</i>
help = Помощь
you-have-coin = У пользователя { $user } { $value ->
        [0] совсем нет монет
        [one] имеется одна монета
        [few] { $value } монеты
        *[many] есть { $value } монет
    }!
enter-a-number = введите число
```

И так, у нас есть сообщение, которое формирует строку перевода, подставляет из { $user } имя пользователя с помощью переменной user, равной name, а из { $value } берется наша переменна n. В с помощью селекторов происходит выбор множественной формы: [one] при n равному 1, [few] при n от 2 до 4x ,

и [many] в остальных случаях. Кроме того, мы добавили (просто для примера) собственный селектор [0], для случая n, равного 0. Звездочкой отмечается вариант селектора по-умолчанию, если Fluent не смог применить ни один селектор.

Обратите внимание, что нам удалось всю сложную логику уместить в одном сообщении.

Теперь сделаем то же самое для английского.

```
hello = Hello, <b>{ $user }</b>!
cur-lang = Your current language: <i>{ $language }</i>
help = Help
you-have-coin = The user { $user } { $value ->
        [zero] hasn't got nothing
        [one] have one coin
        *[many] has { $value } coins
    }!
enter-a-number = Input a number, please.
```

В английском у нас нет малой множественной формы, поэтому мы делаем свою логику. При этом не надо ничего менять в коде проекта. При вхождении числа 2 выберется вариант по-умолчанию, отмеченный звездочкой.

Селекторы обрабатываются неявно встроенными функциями внутри Fluent. Но если нужны еще более сложные вещи, то можно придумать свою функцию внутри перевода и применять ее в конкретном переводе. Об этом можно прочитать в документации: https://projectfluent.org/fluent/guide/functions.html.

FTL имеет синтаксис выражения выбора, который позволяет определять несколько вариантов перевода и выбирать между ними на основе значения селектора. Индикатор * обозначает вариант по умолчанию. Требуется вариант по умолчанию.

Еще пару слов о селекторах. Селектор может быть строкой, и в этом случае он будет сравниваться непосредственно с ключами вариантов, определенных в выражении выбора. Для селекторов, которые являются числами, ключи вариантов либо точно соответствуют числу, либо соответствуют категории множественного числа по справочнику проекта CLDR (https://cldr.unicode.org/) для числа. Возможные категории множественных чисел: [zero], [one], [two], [few], [many], [other].

Если перевод требует, чтобы число было отформатировано не по умолчанию, селектор должен использовать те же параметры форматирования. Отформатированное число затем будет использоваться для выбора правильной категории множественного числа CLDR, которая для некоторых языков может отличаться от категории неформатированного числа:

```
 your-score =
    { NUMBER($score, minimumFractionDigits: 1) ->
        [0.0]   You scored zero points. What happened?
       *[other] You scored { NUMBER($score, minimumFractionDigits: 1) } points.
    }
```

Замечу, что здесь NUMBER – это строенная функция, которая явно вызывается. Подробнее в документации: https://projectfluent.org/fluent/functions.html#built-in-functions

Еще один пример использования селекторов — склонение имен и выражения для разных родов существительных.

Например у нас уже есть в базе данных пол пользователя. И мы хотим вывести строку Вася ответил(а) на ваше сообщение.

```
mention = {$mention-user} ответил(а) на ваше сообщение.
```

Преобразуем это в человеческий вид:

```
mention = {$mention-user} {$user-gender ->
        *[male]  ответил
        [female] ответила
        [other] ответил(а)
        }  на ваше сообщение.
```

Мы добавили третью опцию [other], которая в непонятных случаях будет выдавать обезличенную строку. И в хэндлер нам нужно лишь дополнительно передать селектор в виде пола (естественно из базы нужно его извлечь в виде male, female, other, так как будет произведено сравнение строк).

```
gender = database.get_data(gender_data)
await message.answer(text=i18n.get("mention", mention-user=gender))
```

Еще интересный кейс, это передача параметризованных терминов, что очень важно для флективных языков.

Термины, это отдельные виды сообщений, помеченных тире. Значения терминов следуют тем же правилам, что и значения сообщений. Они могут быть и простым текстом, и включать в себя другие выражения, включая переменные. Однако, хотя сообщения получают данные для переменных непосредственно из приложения, термины получают такие данные из сообщений, в которых они используются. Такие ссылки принимают форму

-term(…), где переменные, доступные внутри термина, определены в скобках, например -term(param: «value»).

Передавая переменные в термин, вы можете определить выражение с несколькими вариантами одного и того же значения термина.

```
-brand-name =
    { $case ->
        *[nominative] Aiogram
        [locative] Aiogram’a
    }
# "About Aiogram."
about = Информация об { -brand-name(case: "locative") }.
```

Этот шаблон может быть полезен для определения аспектов термина, которые могут быть связаны с грамматической или стилистической особенностью языка. Во многих флективных языках (немецком, финском, венгерском, и славянских языках), предлог о (об) определяет падеж дополнения. Это может быть винительный падеж (немецкий, русский), абляционный падеж (латинский) или локатив (славянские языки). Грамматические падежи могут быть определены как варианты одной и той же темы и упоминаться посредством параметризации из других сообщений. Вот что происходит с сообщением about из примера выше.

Если в термин не переданы никакие параметры или если на термин ссылаются без скобок, будет использован вариант по умолчанию.

```
# "Aiogram has been successfully updated."
update-successful = { -brand-name } был успешно обновлен.
```

При этом, имейте в виду, что с простыми сообщениями это не работает, и нужно использовать селекторы.

Что касается правильного формирования дат, синтаксис Fluent поддерживает библиотеки форматов вышеупомянутой CLDR, достаточно записать строку с нужными параметрами форматирования и передать в неё время:

```
# Это в нашем ftl файле
order-time = Время заказа: { DATETIME($date, month: "long", year: "numeric", day: "numeric", weekday: "long") }
```

В качестве переменной мы должны передать время в формате Unix Time

unixdate = 556593884000

```
await message.answer(text=i18n.get("order-time", date=unixdate))
```

На выходе мы получим: «Время заказа: вторник, 30 апреля 2019 г.» - для русского языка, и «Время заказа: Tuesday, April 30, 2019» для английского языка.

Документация, перевод документации и примеры использования по ссылкам ниже:

https://projectfluent.org/fluent/index.html

https://blog.wtigga.com/fluent-syntax/

https://projectfluent.org/play/

https://blog.wtigga.com/fluent-practice/

Не смотря на то, что проект Fluent направлен на работу с фронтэндом и переводом UI браузера, мы смело можем его использовать в своих проектах на Python.

Динамическое определение и переключение языка в Aiogram

Теперь мы знаем, как проводится интернационализация и локализация продукта. Но нам бы хотелось это все внедрить в интерфейс нашего приложения на фреймворке Aiogram, чтобы повысить качество UX (user experience).

Пример будет реализован с помощью Fluent, поскольку gettext уже изучен и по факту является промышленным стандартом. А нам хочется попробовать что-то новое.

Спроектируем взаимодействие с пользователем следующим образом.

1. Если пользователь впервые начал взаимодействовать с ботом, то мы не знаем на каком языке он говорит. Попытаемся получить язык пользователя из апдейта. Но эта опция зависит от клиента Telegram, которым пользуется пользователь, и часто это поле не заполнено. В таком случае отдадим какой-то язык по умолчанию, например, английский.

2. Если пользователь выбрал в меню бота или отправил команду выбора языка, то сохраним язык в базе данных и переключим язык. В дальнейшем, при взаимодействии, будем использовать язык из базы данных.

3. Нам нужно сделать переводы для всех элементов интерфейса, включая сообщения, клавиатуры, меню, алерты.

4. В каких-то случаях при отправке картинок или документов — локализовать картинки или документы.

В учебных целях мы реализуем базовые вещи из первых трех пунктов.

Итак, наш бот будет стартовать по команде **/start**, выводить текст помощи по команде **/help** или после получения слова **помощь** или **help**, в зависимости от текущего языка. Отдавать клавиатуру на нужном языке. И по команде **/languge_en** и **/languge_ru** переключаться на соответствующий язык. Также по команде **/photo** отправлять картинку, переведенную на текущий язык.

Нам потребуется база данных, для хранения языка пользователя. В коде будет минимальный пример, просто «чтоб работало». Ну и как ранее говорилось, будет много дублирования кода (здравствуй WET, прощай DRY) и сообщений и много контекста.

Структура необходимых компонентов будет примерно такой:

```
…
├──database
│       bot.db
│       database.py
├──locales
│   ├──en
│   │   ├──LC_MESSAGES
│   │   │       messages.ftl
│   │   └──static
│   │           bayan_en.jpg
│   └──ru
│       ├──LC_MESSAGES
│       │       messages.ftl
│       └──static
│               bayan_ru.jpg
└──middlewares
        db_middleware.py
        i18n_middleware.py
…
```

База данных, интерфейс к ней, папка с переводами и локализованными картинками, и два внешних middleware для работы с базой данных и переводами.

База данных и интерфейс к ней вы уже научились делать. Интерфейс БД должен уметь добавлять пользователя и получать его язык. Этого достаточно для нашего примера.

Самое главное, это то, что классы движка перевода, реализованные в Aiogram, не позволяют из коробки реализовать задуманный нами функционал. В главе Конфигурация движка перевода мы уже приводили описание этих классов. Напомню :

SimpleI18nMiddleware - выбирает код языка из объекта User, полученного в событии. Однако не все клиенты Telegram отдают это значение. Очень часто объект language_code не заполнен и является пустой строкой.

ConstI18nMiddleware - выбирает статически определенную локаль.

FSMI18nMiddleware - хранит локаль в хранилище FSM. Но у нас ее там пока нет.

И у нас есть то, что нужно. **I18nMiddleware** - это базовый абстрактный класс для наследования и создания собственного обработчика.

Создадим в нашем приложении объект этого класса, и передадим туда наш кастомный менеджер, который реализуем ниже.

```
async def main() -> None:

…
    dp = Dispatcher()
    dp.include_router(router)

    i18n = I18nMiddleware(
        core=FluentRuntimeCore(
            path="locales/{locale}/LC_MESSAGES"
        ),
        # передаем наш кастомный менеджер языка из middlewares/i18n_middleware.py
        manager=i18n_middleware.UserManager(),
        default_locale="en")
…
```

Начнем с реализации своего менеджера.

Создадим файл middlewares / i18n_middleware.py.

```
from aiogram_i18n.managers import BaseManager
from aiogram.types.user import User
from database.database import Database


class UserManager(BaseManager):
    """Собственная реализация middleware для интернационализации
    на базе класса BaseManager из библиотеки aiogram_i18n.
    BaseManager имеет абстрактные методы set_locale и get_locale, которые
    нам нужно реализовать. Кроме того, при инициализации объекта класса,
    выполняются LocaleSetter и LocaleGetter (см. реализацию BaseManager).

    В случае использования gettext необходимо проверить реализацию
    класса, так как не gettext не тестировалось
    """
    async def get_locale(self, event_from_user: User, db: Database = None) -> str:
        default = event_from_user.language_code or self.default_locale
        if db:
            user_lang = db.get_lang(event_from_user.id)
            if user_lang:
                return user_lang
        return default


    async def set_locale(self, locale: str, event_from_user: User, db: Database = None) -> None:
        if db:
            db.set_lang(event_from_user.id, locale)
```

По сути мы просто реализовали два метода:

get_locale – геттер, который сначала проверяет есть ли в базе данных у пользователя какой-то язык. Если в базе ничего нет, то пытается получить язык из клиента, и если и его нет - просто отдает дефолтную локаль.

set_locale – сеттер, который просто записывает язык в базу данных, а если базы нет, то ничего не делает.

Естественно, эту логику работы с языком каждый придумывает себе сам под свои задачи и особенности работы и используемые инструменты (кэш, хранилище и т.п.).

Регистрируем middleware. Сначала для базы данных, а затем i18n. Не забываем, что у i18n есть метод .setup(), который правильно регистрирует этот middleware.

```
 # Регистрация мидлварей. Сначала регистрируется база данных, так как там хранится язык.
    dp.update.outer_middleware.register(db_middleware.DBMiddleware())
    i18n.setup(dispatcher=dp)
```

Сначала пропишем наши хэндлеры. А уже в конце займемся переводами. Импорты:

```
from aiogram_i18n import I18nContext, LazyProxy, I18nMiddleware
from aiogram_i18n.cores.fluent_runtime_core import FluentRuntimeCore
from aiogram_i18n.types import (
    ReplyKeyboardMarkup, KeyboardButton, ReplyKeyboardRemove
    # you should import mutable objects from here if you want to use LazyProxy in them
)
```

Первый хэндлер обрабатывает команду /start и сохраняет пользователя в БД. Язык нам не известен, поэтому его мы не сохраняем.

```
@router.message(CommandStart())
async def process_start_command(message: Message, i18n: I18nContext, db: Database):
    if not db.get_user(message.from_user.id):
        db.add_user(message.from_user.id, message.from_user.username)
    name = message.from_user.full_name

    await message.answer(text=i18n.hello(user=name, language=i18n.locale),# text=i18n.get("hello", user=name))
                         reply_markup=rkb
                         )
```

Следующий хэндлер обрабатывает команду /help и слова help, Help, помощь, Помощь,то есть введенные на родном языке пользователя. Поскольку на момент попадания в фильтрацию объект i18n middleware не вызывается, язык мы не можем получить. Поэтому используем ленивую подстановку текстов LazyProxy. Мутабельные объекты, например клавиатуры, для LazyProxy экспортируем не из основной библиотеки aiogram, а из aiogram_i18n.types.

```
@router.message(Command("help"))
@router.message(F.text == LazyProxy("help", case="capital"))
@router.message(F.text == LazyProxy("help", case="lower"))
async def cmd_help(message: Message, i18n: I18nContext) -> Any:
    return message.reply(text=i18n.get("help-message"))
```

Создадим хэндлер для команды обработки смены языка.

```
async def switch_language(message: Message, i18n: I18nContext, locale_code: str):
    await i18n.set_locale(locale_code)
    await message.answer(i18n.get("lang-is-switched"), reply_markup=rkb)


@router.message(Command("language_en"))
async def switch_to_en(message: Message, i18n: I18nContext) -> None:
    await switch_language(message, i18n,"en")


@router.message(Command("language_ru"))
async def switch_to_en(message: Message, i18n: I18nContext) -> None:
    await switch_language(message, i18n,"ru")
```

Мы видим дублирование кода, но это неизбежно. Часть было вынесено в функцию switch_language().

Далее отправка изображения. Изображения лежат в locale/имя_локали/static/имя_картинки_локаль.jpg.

```
@router.message(Command("photo"))
@router.message(F.text == LazyProxy("photo"))
async def sent_photo(message: Message, i18n: I18nContext) -> None:
    locale_code = i18n.locale
    path_to_photo = f"locales/{locale_code}/static/bayan_{locale_code}.jpg"
    await message.answer_photo(photo=FSInputFile(path_to_photo))
```

Этот хэндлер отвечает за обработку остальных сообщений. При этом он после ответ выдает дату сообщения в формате, специфичном для локали пользователя. То есть «День Месяц Год» или «Month Day, Year».

```
@router.message()
async def handler_common(message: Message, i18n: I18nContext) -> None:
    await message.answer(text=i18n.get("i-dont-know"))
    await message.answer(text=i18n.get("show-date", date_=message.date))
```

Ну и клавиатура, которую мы импортировали from aiogram_i18n.types import (ReplyKeyboardMarkup, KeyboardButton, ReplyKeyboardRemove)

```
# Это тестовая клавиатура
rkb = ReplyKeyboardMarkup(
    keyboard=[
        [KeyboardButton(text=LazyProxy("help", case="capital"))]  # or L.help()
    ], resize_keyboard=True
)
```

Текст клавиатуры будет также лениво переведен в момент отправки сообщения, когда уже язык будет известен.

Осталось сделать саму локализацию. Складываем картинки в папки локалей. И создаем файлы переводов в формате .ftl в соответствующих папках. Логика работы описана в комментариях в каждом файле.

Английский перевод:

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
# Это был пример лицензии

### Файл примера перевода на английский язык
### Логика перевода изменится, не затрагивая код и другие переводы
### С тройного шарпа начинается комментарий уровня файла

## Это комментарий уровня группировки блоков в тексте. См. документацию.
## Hello section

# Это пример термина. Термин начинается с дефиса.
# Посмотрите как это работало в русском переводе. Здесь же мы изменим логику.
# Падежи нам не нужны, но может потребоваться притяжательная форма
-telegram = { $case ->
      *[common] Telegram
      [possessive] Telegram's
    }

# { $user } - user name, { $language } - language code.
# Это было описание переменных, которые попадают сюда из основного кода приложения.
# Термин мы берем из этого же файла перевода,
# и вставляем с параметром нужного контекста использования (в нашем случае падежа).
hello = Hi, <b>{ $user }</b>!
    { $language ->
    [None] In your { -telegram(case: "common") } client a language isn't set.
    Therefore, everything will be displayed in default language.
    *[any] Your Telegram client is set to { $language }. Therefore, everything will be displayed in this language.
    }

help = { $case ->
    *[capital] Help
    [lower] help
    }
help-message = <b>Welcome to the bot.</b>
    Our bot can't do anything useful, but it can switch languages with dexterity.
    The following commands are available in the bot:
    /start to start working with the bot.
    /help or just send the word <b><i>help</i></b> to show this message.
    /language_en { switch-to-en }
    /language_ru { switch-to-ru }
    /photo or just send the word <b><i>photo</i></b> to send photo to you.

    You can also write to the bot


# { $language } - language code.
# The current language is { $language }.
cur-lang = The current language is: <i>{ $language }</i>

## Switch language section
```

```
# Название языка мы отображаем на родном языке, чтоб человек увидел знакомые буквы и поонял, что не все потеряно.
en-lang = English
ru-lang = Русский
switch-to-en = Switch the interface to { en-lang }.
switch-to-ru = Switch the interface to { ru-lang }.
lang-is-switched = Display language is { en-lang }.

photo = photo

i-dont-know = I'm so stupid bot. Make me clever.
show-date = But look! Pretty date on English: { DATETIME($date_, month: "long", year: "numeric", day: "numeric",
weekday: "long") }
```

Русский перевод:

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
# Это был пример лицензии

### Файл примера перевода на русский язык
### Важно. Не забудь полить помидоры...
### С тройного шарпа начинается комментарий уровня файла

## Это комментарий уровня группировки блоков в тексте. См. документацию.
## Hello section

# Это пример термина. Термин начинается с дефиса.
# Термины можно передавать внутри сообщений, указывая переменные для параметризации в скобках.
# То есть это как атрибуты, но мы их задаем в тексте переводов, а не получаем извне.
# Мы будем издеваться над языком, чтобы увидеть как и что работает
-telegram = {$case ->
    *[nominative] Телеграм {"{"}Telegram{"}"}
    [genitive] Телеграма ({"{"}Telegram'а{"}"})
    [dative] Телеграму ({"{"}Telegram'у{"}"})
    [accusative] Телеграм ({"{"}Telegram{"}"})
    [instrumental] Телеграмом ({"{"}Telegram'ом{"}"})
    [prepositional] Телеграме ({"{"}Telegram'е{"}"})
    }
# {"}"}  это пример экранированного символа.
# Падежи
# nominative - именительный
# genitive - родительный
# dative - дательный
# accusative - винительный
# instrumental творительный.
# prepositional - предложный

# { $user } - user name, { $language } - language code.
# Это было описание переменных, которые попадают сюда из основного кода приложения.
# Термин мы берем из этого же файла перевода,
# и вставляем с параметром нужного контекста использования (в нашем случае падежа).
hello = Привет, <b>{ $user }</b>!
    У тебя в клиенте { -telegram(case: "nominative") } { $language ->
    [None] не указан язык, поэтому все будет отображается на языке по-умолчанию.
    *[any] указан язык { $language }, поэтому все будет отображается на этом языке.
    }

# а так мы вставляем символы unicode по номеру \uHHHH. Например,
# tears-of-joy1 = {"\U01F602"}
# tears-of-joy2 = 😂

help = { $case ->
    *[capital] Помощь
    [lower] помощь
    }

help-message = <b>Добро пожаловать в бота.</b>
    Наш бот не умеет ничего полезного, однако с ловкостью может переключать язык.
    В боте доступны следующие команды:
    /start чтобы начать работать с ботом
    /help или просто отправьте слово <b><i>помощь</i></b>, чтобы показать это сообщение
    /language_en { switch-to-en }
    /language_ru { switch-to-ru }
    /photo или просто отправьте слово <b><i>фото</i></b>, чтобы прислать вам картинку


# Это комментарий подсказка для переводчиков (чтобы не искать что значат эти переменные в коде,
# который не факт ,что они получат, а если и получат, то не поймут:

# { $language } - language code.
# The current language is { $language }.
cur-lang = Текущий язык: <i>{ $language }</i>

## Switch language section
```

```
en-lang = English
ru-lang = Русский
switch-to-en = Переключить интерфейс на { en-lang } язык.
# В фигурных скобках пример интерполяции одного сообщения в другом.
switch-to-ru = Переключить интерфейс на { ru-lang } язык.
lang-is-switched = Язык переключен на { ru-lang }.

photo = фото


i-dont-know = Я тупой бот. Сделай меня умным.
show-date = Но посмотри! Красивая дата по правилам Русского языка: { DATETIME($date_, month: "long", year:
"numeric", day: "numeric", weekday: "long") }
```

Запускаем, тестируем.

Исправляем ошибки.

aiogram_i18n.exceptions.NoTranslateFileExistsError: files with extension (.ftl)in folder (locales/ru/LC_MESSAGES) not found — возникает когда файл перевода не найден по указанному нами пути.

KeyNotFoundError: Key 'enter-a-number' not found — ошибка возникает, когда в коде есть ключ, а в переводе его нет. Например, вызываем i18n.get(“help”) , а такой строчки нет в файле перевода соответствующего языка.

fluent.runtime.errors.FluentReferenceError: Unknown external: user — такая ошибка возникает, когда вы забываете передать в вызове функции основного кода нужный аргумент для ключа. Например, в нашем переводе есть такое сообщение:

```
hello = Привет, <b>{ $user }</b>!
    У тебя в клиенте { -telegram(case: "nominative") } { $language ->
    [None] не указан язык, поэтому все будет отображается на языке по-умолчанию.
    *[any] указан язык { $language }, поэтому все будет отображается на этом языке.
    }
```

Здесь { -telegram }– это термин. И он управляется конструкцией (case: "nominative") только внутри языкового файла. А вот дальше используются аргументы { $user } и { $language }, которые нужно передать из основного кода. Мы их передаем как именованные аргументы:

```
await message.answer(text=i18n.get(“hello”, user=name, language=i18n.locale))
```

или еще возможен такой способ:

```
await message.answer(text=i18n.hello(user=name, language=i18n.locale))
```

Как вы видите, наш код стал достаточно сложным. Но появилась гибкость в отображении контента с учетом локальных особенностей пользователей разных регионов.

Старайтесь продумывать архитектуру, не бойтесь экспериментировать.