### 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/

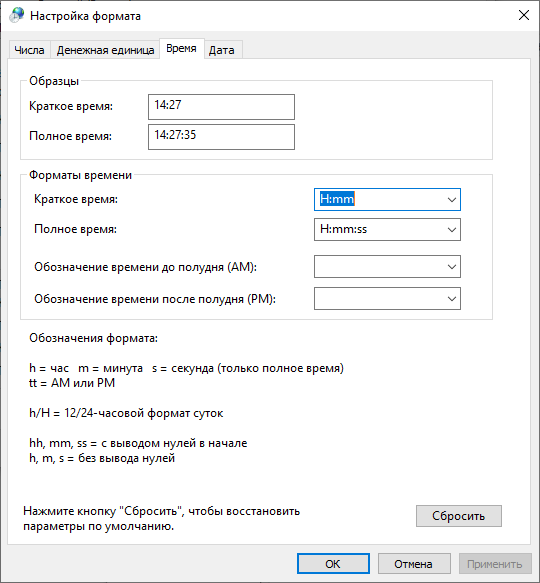
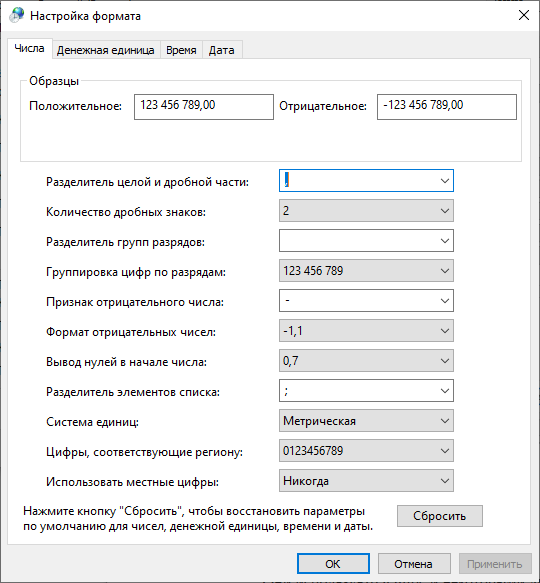
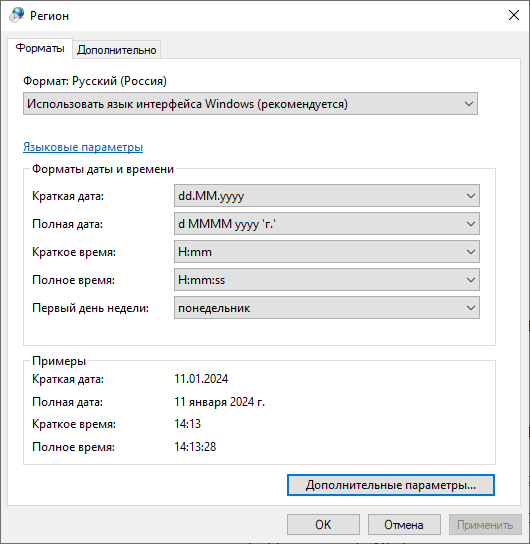
#### Локали.

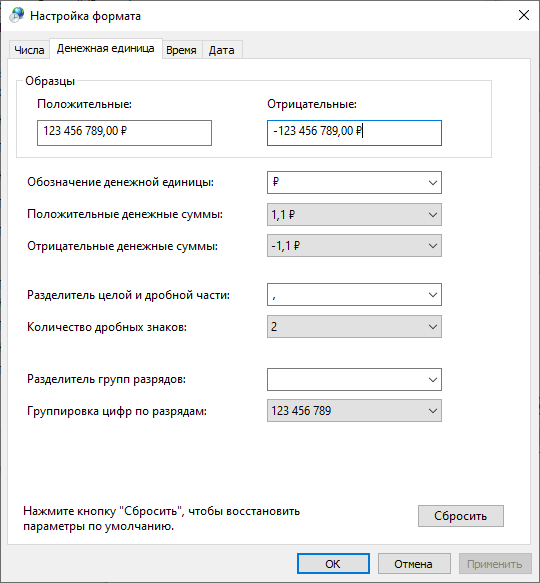
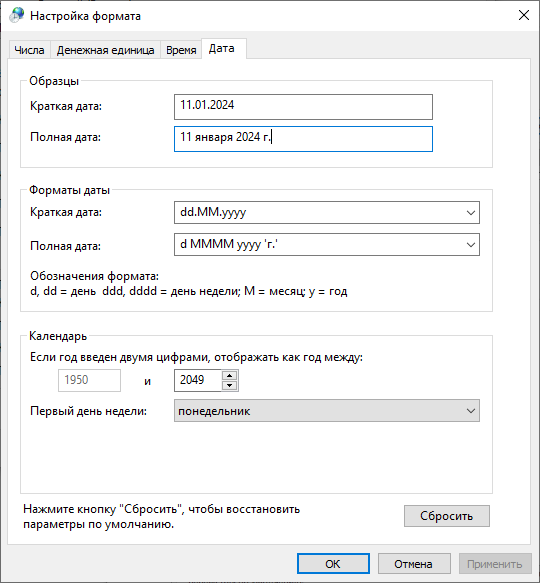
**Локали** — это набор настроек, которые определяют язык, который использует система, а также региональные особенности, такие как денежные знаки, формат чисел, даты и времени и наборы символов.

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:

* + 1. **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.
    2. **ConstI18nMiddleware** - выбирает статически определенную локаль.
    3. **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, не забываем.

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](https://docs.python.org/3/library/gettext.html" \l "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).