### Интернационализация и Локализация.

**Интернационализация** (англ. internationalization или сокращенно i18n (i 18 букв n )) – 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 локалей (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 локали 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 локалями — 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, helps 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 так же сам перевод. А локализация — это приведение всех элементов, такие как тексты, картинки, шрифты, к национальным. И эти понятия очень тесно связаны и иногда грань между ними стёрта.

Подробнее можно ознакомиться здесь:

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

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

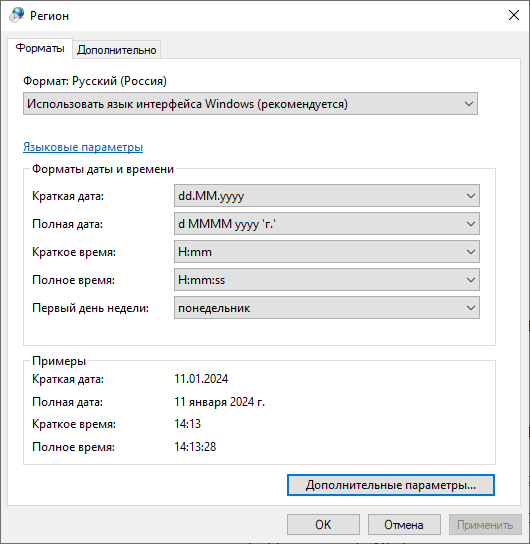
### Локали.

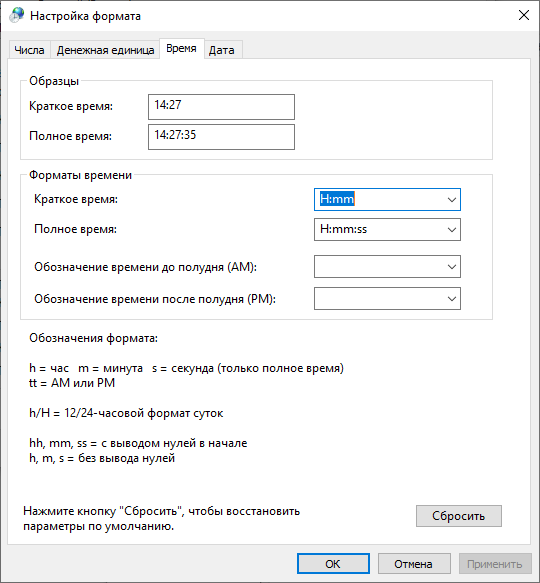
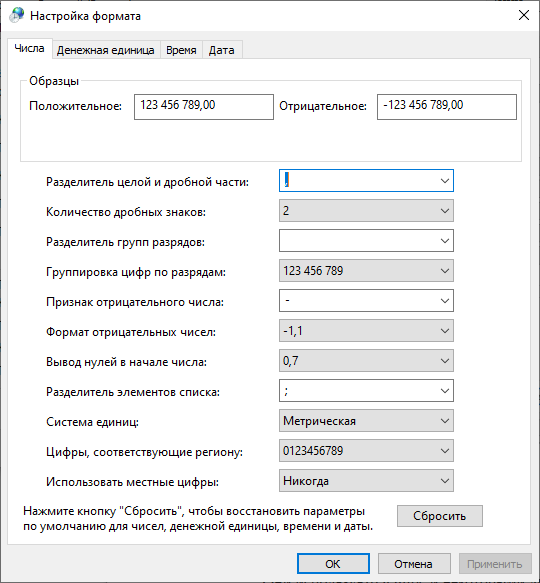
Локали — это набор настроек, которые определяют язык, который использует система, а также региональные особенности, такие как денежные знаки, формат чисел, даты и времени и наборы символов.

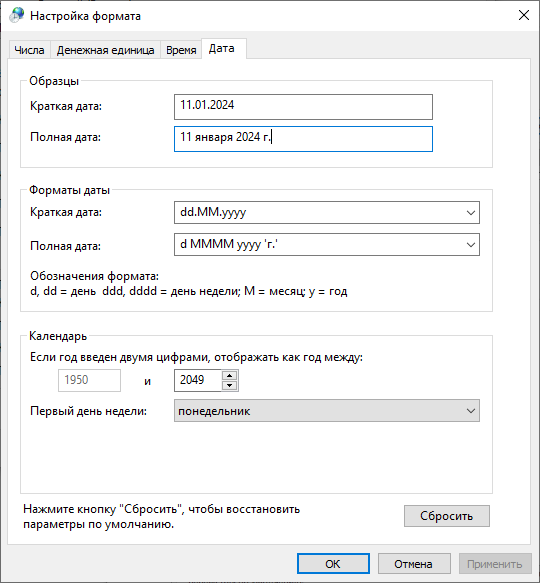
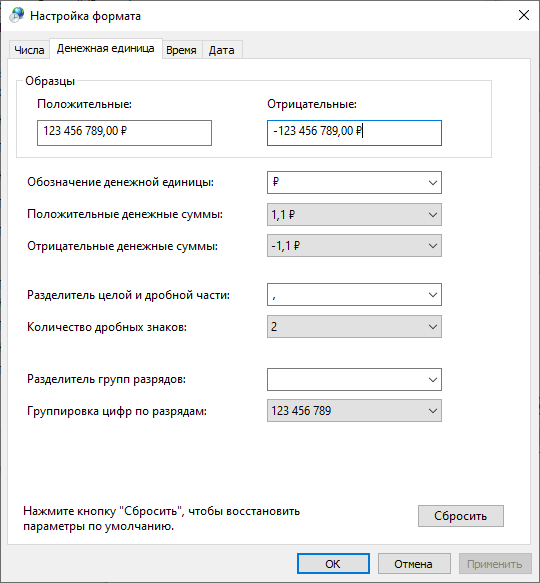
Если у вас Windows, то выполните команду

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/ 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:

apt.mo

debconf.mo

dpkg.mo

mc.mo

xdg-user-dirs.mo

В дальнейшем имя приложения для перевода обозначается в терминологии инструментов перевода как **домен**.

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 == \_\_("My menu entry"))  
 …

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.

…  
locales  
├── bot.pot  
├── en  
│ └── LC\_MESSAGES  
│ └── bot.po  
├── ru  
│ └── LC\_MESSAGES  
│ └── bot.po  
…

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()

### Создание переводов и работа с Babel

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

…  
locales  
├── bot.pot  
├── en  
│ └── LC\_MESSAGES  
│ └── bot.po  
├── ru  
│ └── LC\_MESSAGES  
│ └── 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 PROJECT.  
# Copyright (C) 2024 ORGANIZATION  
# This file is distributed under the same license as the PROJECT project.  
# FIRST AUTHOR <EMAIL@ADDRESS>, 2024.  
#  
#, fuzzy  
msgid ""  
msgstr ""  
"Project-Id-Version: PROJECT VERSION\n"  
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"  
"POT-Creation-Date: 2024-01-10 18:36+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, копирайты и электронный адрес для связи с переводчиком.

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 messages -l en

Где,

-i locales/messages.pot - путь к нашему шаблону .pot  
  
-d locales - наш каталог переводов   
  
-D messages - наш домен переводов   
  
-l en — код языка.

Будет создан файл перевода messages.po в папке locales/en/LC\_MESSAGES/. И вот 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 формула по которой определяется для конкретного языка форма слова во множественном числе.

Например нам нужно добиться правильного перевода и подстановки формы слов после числа:

You have 1 mrssage, you have 2 messages, в английском языке.

Но в русском будет так: У вас 1 сообщение, у вас 2 сообщения, у вас 5 сообщений

При генерации Babel по коду языка сгенерировал формулу определения форм слова.

В английской версии у нас:

"Language: en\n"  
"Plural-Forms: nplurals=2; plural=(n != 1);\n"

А в русской:

"Language: ru\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"

То есть это обычное тернарное выражение с булевыми операциями, что

В английском

* две формы слова (нумерация с 0).
* если цифра 1, то форма слова не является множественным числом.
* в остальных случаях это множественное число.

В русском:

* три формы слова (нумерация с 0)
* первая форма - заканчивается число на 1 (за исключением остатка от деления на 100 равного 11), то есть 1 сообщение, 101 сообщение, но 111 сообщений.
* Вторая форма от 2 до 4х 4 сообщения и 44 сообщения
* все остальное третья форма, 5, 56, 120, 129, 1007сообщений а так же сюда попадают и 11, 211 и т.д сообщений.

Нам останется только в переводе указать какая форма 0, 1 или 2.

Этот файл заполняем переведенными строками на нужный язык. Для удобства используем программу PoEdit https://poedit.net/

Создаем еще один файл для русского языка

pybabel init -i locales/messages.pot -d locales -D messages -l ru

И также переводим его.

Особенность работы с gettext и Babel заключается в том, что все файлы переводов должны быть предварительно скомпилированы. Компилируем переводы командой

pybabel compile -d locales -D messages

И получаем по этому же пути файлы .mo

При обновлении кода и переводов вам потребуется обновить переводы — файлы .po и .mo

Обновим шабон переводов .pot

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

А затем обновим .po -файлы командой update (которая сохранит предыдущие изменения и добавит новое)

pybabel update -d locales -D messages -i locales/messages.pot

Обновим свои переводы вручную или с помощью PoEdit.

И снова скомпилируем переводы в .mo файлы.

pybabel compile -d locales -D messages