i18n with Python TS

I want to build a website and incorporate localization from the beginning and maintain it through subsequent development. Are there any free, open source tools or libraries that I can use. My backend stack is python, but my front end stack is probably typescript.

Yes, you can build a well-localized website from the start using free, open-source tools that work seamlessly with your **Python backend** and **TypeScript frontend**. Here's a breakdown of tools by layer:

Frontend (TypeScript/JavaScript)

1. i18next

- **What it is**: One of the most widely used internationalization (i18n) frameworks for JavaScript and TypeScript apps.
- **Strengths**:
  - Works with React, Vue, Svelte, or plain JS/TS
  - Supports dynamic loading, pluralization, interpolation, nested translations
  - JSON-based translation files
- **Tooling**:
  - Use with **`` `react-i18next` ``**, **`` `svelte-i18next` ``**, etc.

  - Optional CLI (`i18next-scanner`) to extract strings from code.

2. FormatJS / react-intl

- **What it is**: Framework from the same team behind Intl Polyfill. Ideal for React apps.
- **Strengths**:
  - Native integration with `Intl` APIs
  - Good for message formatting, dates, numbers, and currency
  - Supports rich formatting directly in JSX
- **Use case**: Best for React apps that need complex formatting.

Backend (Python)

1. Babel (Python Babel)

- **What it is**: Python library for internationalizing and localizing Python applications.
- **Strengths**:
  - Extracts strings from Python code and templates
  - Works with `gettext`/`.po` and `.mo` files
  - Supports date/time/number formatting for many locales
- **Integrations**: Works well with Flask, Django, and CLI apps.

2. Flask-Babel / Django Translation Framework

- If using **Flask**:
  - `Flask-Babel` or `Flask-BabelEx` can hook into the Babel library.

- If using **Django**:
  - Built-in translation framework using `` `gettext` ``.

Cross-stack Format Suggestion

- Use **i18next** on the frontend with `` `.json` `` files for each locale.
- Use **Babel + Flask-Babel** on the backend with `` `.po` `` files.
- Optionally centralize translation strings in a shared format (e.g., `` `.po` `` → `` `.json` ``) using conversion tools if needed.

Optional Helpers & Dev Tools

Tool	Purpose	Notes
`` `i18next-scanner` ``	Extracts translatable strings from code	Helpful for TypeScript/React
`` `po2json` ``	Converts `` `.po` `` files to `` `.json` ``	Enables syncing backend and frontend translations
`` `Crowdin CLI` `` / `` `Weblate` ``	Translation management	Optional if you want contributors/translators to work via GUI

Workflow Recommendation

1. **Designate translatable strings** early using keys or gettext-style wrappers.
2. **Organize translation files per locale**, e.g.:
    - `` `en.json` ``, `` `fr.json` `` for frontend
    - `` `en.po` ``, `` `fr.po` `` for backend
3. **Automate extraction and syncing** using `` `i18next-scanner` `` and `` `pybabel extract` ``.
4. **Use context-based keys** (e.g. `` `"homepage.welcome"` ``) to keep translations clean.
5. **Set up dynamic locale switching** in frontend and API support in backend (e.g., via `` `Accept-Language` `` header or user settings).

Would you like a sample project layout or code snippet showing how this would work in a Flask + React setup?

Is there any way I can leverage llms to generate initial translations?

Yes, **LLMs are a powerful and cost-effective way to bootstrap initial translations** — especially when you're just starting with localization. You can use them in a few different ways depending on your workflow and how much control you want.

Approach 1: One-Time Bulk Translation of Resource Files

If you already have a source file like `en.json` or `en.po`, you can use an LLM to generate translated versions in other languages.

Example: JSON (i18next-style)

Input (English):

```
{
  "homepage": {
    "welcome": "Welcome to our site!",
    "tagline": "We make localization easy."
  }
}
```

Use LLM (e.g., GPT-4) prompt:

> *“Translate this JSON file into French. Keep the same keys and structure. Output only valid JSON.”*

Output (French):

```
{
  "homepage": {
    "welcome": "Bienvenue sur notre site !",
    "tagline": "Nous facilitons la localisation."
  }
}
```

You can do this in ChatGPT, or programmatically via API (e.g., OpenAI or Ollama if local), feeding each string or block of keys in sequence.

Approach 2: Automated Script with LLM API (for bulk jobs)

Use Python to send each key or string to an LLM and generate the localized file.

Sketch: Python script using OpenAI

```python
import openai
import json

openai.api_key = "your-api-key"

def translate_text(text, target_lang="fr"):
    prompt = f"Translate to {target_lang}:\n'{text}'"
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}]
    )
    return response['choices'][0]['message']['content'].strip()

with open('en.json') as f:
    en_data = json.load(f)

fr_data = {}
for k, v in en_data["homepage"].items():
    fr_data[k] = translate_text(v, target_lang="French")

# Save as fr.json
with open('fr.json', 'w', encoding='utf-8') as f:
    json.dump({"homepage": fr_data}, f, ensure_ascii=False, indent=2)
```

This can be expanded to handle nested keys and other formats like `.po`.

Approach 3: Human-in-the-loop Translation

Use an LLM to generate translations, but flag or annotate them for review. Example use cases:

- Let the LLM translate and automatically mark strings it's uncertain about (e.g., with confidence scores or special tokens).
- Use a GUI tool like **Weblate** and pre-fill translations with LLM output for human editing.

Approach 4: In-App Dynamic Translation (Less common, but possible)

If your content is very dynamic and you don't want to manage translation files, you could:

- Use an LLM (local or cloud) as a **real-time translation layer**.
- Store original strings and call the LLM as needed with caching for performance.

Not recommended for SEO-sensitive pages or legal content without QA, but viable for:

- Chatbots
- Internal tools
- MVPs with low-latency tolerance

LLM Tools for Localization

Tool	Type	Notes
BabelFish / OpenAI API	Scriptable API-based translation	Works well for JSON/PO
ChatGPT with file upload	Manual batch processing	Good for small files
Poedit Pro (commercial, but has AI plugin)	GUI-based `.po` editor with GPT integration	Best for Django/Flask `.po`
Locofy, Tolgee, Phrase (w/ GPT plugins)	Web-based TMS with AI assist	Mixed open/commercial; review required

---

Would you like a **ready-made Python script** to translate an entire `.json` or `.po` file using an LLM API like OpenAI, Mistral, or even a local Ollama model?

Yes

Here’s a **ready-made Python script** to translate an entire `en.json` file using the **OpenAI API**, with support for nested keys (common in i18next-style JSON). I’ll also note how to adapt it for local models (e.g., via **Ollama** or **LM Studio**) at the end.

Prerequisites

1. Install dependencies:

```
pip install openai
```

2. Set your OpenAI API key:

```
export OPENAI_API_KEY=your-api-key
```

Script: Translate JSON Localization File Using OpenAI

```
import json
import os
import openai

# Configuration
SOURCE_LANG = "English"
TARGET_LANG = "French"
INPUT_FILE = "en.json"
OUTPUT_FILE = "fr.json"
MODEL = "gpt-4"

# Setup API
openai.api_key = os.getenv("OPENAI_API_KEY")

# Helper to flatten nested JSON (dot notation)
def flatten_json(y, prefix=''):
    out = {}
    for k, v in y.items():
        full_key = f"{prefix}.{k}" if prefix else k
        if isinstance(v, dict):
            out.update(flatten_json(v, full_key))
        else:
            out[full_key] = v
    return out

# Helper to unflatten dot-notation JSON
def unflatten_json(flat):
    result = {}
    for compound_key, value in flat.items():
        keys = compound_key.split(".")
        d = result
        for key in keys[:-1]:
            d = d.setdefault(key, {})
        d[keys[-1]] = value
    return result

# Translation using OpenAI
def translate_text(text):
    prompt = f"Translate the following from {SOURCE_LANG} to {TARGET_LANG}:\n\"{text}\""
    try:
        response = openai.ChatCompletion.create(
            model=MODEL,
            messages=[{"role": "user", "content": prompt}],
            temperature=0.2
        )
        return response['choices'][0]['message']['content'].strip().strip('"')
    except Exception as e:
        print(f"Error translating '{text}': {e}")
        return text  # fallback to original

# Main process
with open(INPUT_FILE, 'r', encoding='utf-8') as f:
    source_data = json.load(f)

flat = flatten_json(source_data)
translated = {k: translate_text(v) for k, v in flat.items()}
result = unflatten_json(translated)

with open(OUTPUT_FILE, 'w', encoding='utf-8') as f:
    json.dump(result, f, ensure_ascii=False, indent=2)
```

```
print(f"Translation complete. Output saved to {OUTPUT_FILE}")
```

Sample Input: `en.json`

```
{
  "home": {
    "welcome": "Welcome to our site!",
    "tagline": "We make localization easy."
  },
  "footer": {
    "contact": "Contact us"
  }
}
```

Output: `fr.json` (example result)

```
{
  "home": {
    "welcome": "Bienvenue sur notre site !",
    "tagline": "Nous facilitons la localisation."
  },
  "footer": {
    "contact": "Contactez-nous"
  }
}
```

Adapting for Ollama or Local Models

If you are using a local model via a REST API (e.g., Ollama or LM Studio), change the `translate_text` function to call your local endpoint:

```python
import requests

def translate_text(text):
    prompt = f"Translate the following from {SOURCE_LANG} to {TARGET_LANG}:\n\"{text}\""
    response = requests.post("http://localhost:11434/api/generate", json={
        "model": "mistral",
        "prompt": prompt,
        "stream": False
    })
    return response.json().get("response", "").strip().strip('"')
```

Would you like a `.po` version of this script for Python/Flask/Django backends?