**📘 Project Documentation: TaskFlow – Automate Anything, Effortlessly**

**🚀 Project Vision**

**TaskFlow** is an intelligent, AI-powered assistant designed to automate real-world user tasks through natural interactions. Initially focused on healthcare (e.g., medicine ordering via prescriptions), TaskFlow evolved into a multi-domain automation agent capable of handling web interactions, including shopping, booking, research, and data retrieval.

**💡 Core Idea**

The idea is to create an agent that can:

* Understand user inputs (text, voice, or uploads)
* Analyze data (e.g., prescriptions)
* Perform task execution via browser automation
* Support multilingual interactions
* Provide secure handling of sensitive data

This project leverages the **browser-use** GitHub repository as its backbone, where the AI agent uses Playwright and FastAPI for browser automation and backend control.

**🎯 Features**

* 🧠 AI agent capable of automating tasks through browser interaction
* 🌐 Supports multilingual prompts (via argostranslate, Google Translate API, or DeepL)
* 🗣️ Voice-to-command interaction (Planned for full version)
* 🔒 Enhanced security via .env handling and session controls
* 📱 Interactive Gradio/Streamlit-based UI (for visualization and user interaction)

**🧱 File & Codebase Structure**

**main.py**

* Launches the FastAPI app
* Controls server-side logic and initializes routes

**agent/agent.py**

* Core logic of the AI agent
* Accepts task prompts and triggers browser automation

**browser\_controller/playwright\_controller.py**

* Uses Playwright to launch headless Chrome
* Executes automation steps based on user goals

**voice.py (planned)**

* For integrating speech-to-text, using Whisper or other libraries

**translation.py**

* Uses argostranslate and optionally Google/DeepL APIs to support multilingual inputs

**security.py**

* Manages environment variables and credentials
* Future enhancements planned for OAuth/token handling

**config.py**

* Loads environment config (e.g., CHROME\_PATH, API keys)

**.env.example**

* Template for required environment variables
* Supports multilingual toggle and persistent sessions

**🖼️ UI Overview**

* Gradio-based frontend (current setup)
* Plans to migrate/extend to Streamlit for smoother demos (especially on Colab)

**Key UI Screenshots to Include:**

* Agent startup and task submission UI
* Result visualization in browser view (via VNC/noVNC)
* Multilingual command example (e.g., Hindi/Spanish task input)

**🔁 How It Works**

1. **User Input**: Text/Voice/Upload
2. **Agent Parsing**: Understand task using NLP
3. **Translation (if needed)**: Via translation.py
4. **Browser Control**: Executes task via Playwright
5. **Response Handling**: Sends success/failure/output to UI

**🧪 Demo Instructions**

1. Clone the GitHub project: git clone https://github.com/your-repo/browser-use
2. Create .env from .env.example
3. Install dependencies:
4. pip install -r requirements.txt

playwright install

1. Run the app:

python main.py

1. Open the provided port (usually http://127.0.0.1:8000 or Gradio link)

**🛡️ Security & Ethical Concerns**

* Secure credential storage via .env
* No sensitive data stored or logged
* Permissions asked for any automated ordering or login

**📁 browser\_use/ (Main Application Folder)**

Contains the core logic for your automation agent.

**➤ agent/**

Handles the core "agentic" behavior — this is where tasks like processing user input and orchestrating commands happen.

* **What it does:** Think of it as the brain coordinating automation, browser control, and interaction flow.

**➤ browser/**

Contains modules to interact with the browser.

* **What it does:** Uses **Playwright** for controlling browser tasks like clicking, typing, scraping, etc.
* Includes automation actions and Playwright page interaction logic.

**➤ controller/**

Handles execution logic and runtime orchestration.

* **What it does:** Controls when/what/how actions are executed via browser and agent logic.
* May coordinate between input → reasoning → execution.

**➤ dom/**

DOM parsers and HTML element handlers.

* **What it does:** Helps in extracting, interpreting, and interacting with web page content like forms, buttons, etc.

**➤ telemetry/**

Logging and tracking for behavior and performance.

* **What it does:** Tracks user actions and app performance for debugging and monitoring.

**exceptions.py**

Custom exception classes for clean error handling.

**utils.py**

Common utility functions reused across the app.

**logging\_config.py**

Handles how logs are formatted, stored, and displayed.

**\_\_init\_\_.py**

Makes the folder a Python package.

**📁 docs/**

Documentation source (could include markdown or Sphinx docs).

* **What it does:** Should contain user guides, API references, architecture diagrams.

**📁 eval/**

Evaluation scripts — could be for testing agents' capabilities.

**📁 examples/**

Code examples showing how to use the agent or run specific tasks.

**📁 static/**

Static files like images, logos, CSS, JS (if used in a frontend).

**📁 tests/**

Unit and integration tests.

* **What it does:** Ensures code is working properly via test cases.

**🛠️ Root-Level Files**

**.env.example**

Template for environment variables like API keys.

**pyproject.toml / requirements.txt**

Python environment and dependency configuration.

**README.md**

High-level explanation of the project.

**SECURITY.md**

Guidelines and practices for secure handling of data.

**🔐 Multilingual & Security Support**

**Multilingual**

* Use modules like argos-translate, googletrans, or DeepL in the **agent** to translate queries before processing.
* Could also be added as a utility function in utils.py or translation.py (if created).

**Security**

* Sensitive data like login or payment details can be securely stored using dotenv, encrypted storage, or secrets manager.
* Files like SECURITY.md and exceptions.py help define good security practices.