PictoPy

Project Description

This project leverages edge technologies and libraries to provide an intuitive, visually appealing user interface, efficient backend logic, and a robust system for smart tagging of photos based on objects or scenes. The goal is to create a tool that not only enhances user experience but also ensures privacy by processing images locally, without the need for remote servers.

Technical Stack and Tools

- **Why not Tkinter ?**
    - **Performance Concerns:** Tkinter's performance may not be optimal for complex applications due to its interface with the underlying Tcl/Tk toolkit.
    - **Debugging Difficulties**: Debugging Tkinter applications can be challenging because Tkinter widgets at their core aren't Python objects but wrappers around tk widgets, leading to cryptic error messages and making it harder to pinpoint the source of issues.
    - **Aesthetics**: Some users find Tkinter's default look and feel to be outdated or unattractive compared to more modern GUI frameworks, which can affect the user experience and the overall appearance of applications.
    - **Not Designed for Production:** Tkinter was not specifically designed for use in production environments, but rather for prototyping and educational purposes.

- Qt Designer
    - **Why Qt Designer?** Qt Designer is a visual UI creator that simplifies designing, creating, and maintaining UI elements. It allows for a more intuitive approach to UI design, reducing the need for extensive coding and manual adjustments. This tool is particularly beneficial for our project as it will enable us to focus more on the application logic rather than the intricacies of UI design.
    - **Advantages**
        - **Efficiency:** By using Qt Designer, we can significantly reduce the time spent on UI development.
        - **Cross-Platform Compatibility:** Qt Designer supports a wide range of platforms, ensuring that our application will look and function consistently across different operating systems.
        - **Integration with PyQt:** Qt Designer's .ui files can be easily converted to Python scripts using PyQt, facilitating a seamless transition from design to implementation.

- PyQt
  - **Why PyQt?** PyQt is a set of Python bindings for Qt libraries used to create cross-platform applications. It allows us to convert the .ui files created with Qt Designer into Python scripts, enabling us to define actions and functionality and bind them to UI elements. Also, it was the only option listed other than Tkinter.

- Backend
  - SQLite
    - A lightweight, serverless, and self-contained database engine, makes it simple and efficient for various applications.
    - Already mentioned in required skills, so the backend has probably already been finalized.

- Object detection
  - Yolo

    (been demonstrated in the test branch already)
    - **Speed and Accuracy:** YOLO is known for its high speed and accuracy, making it ideal for real-time object detection tasks such as video surveillance or self-driving cars.
    - **Single-shot Detection:** It processes an entire image in a single pass, which is computationally efficient and allows for real-time detection.
    - **Limitations**: YOLO can struggle with detecting very small objects, especially if they are close to other objects, and may not perform as well in crowded scenes or when objects are far away from the camera. **RetinaNet** might be considered as an alternative if requirements aren't met.

- Metadata extraction
  - Pillow
    - **Support for Multiple Image Formats:** Pillow supports a wide range of image file formats, makes it versatile for handling various types of images.
    - **Ease of Use:** High level interface for image processing tasks, including metadata extraction, making it accessible for developers with varying levels of expertise.
    - **Python Integration:** Being a Python library, Pillow integrates seamlessly with Python applications, enabling developers to incorporate image metadata extraction into their Python projects easily.

      Here's a write up showcasing metadata extraction.

Distribution

- Pyinstaller
  (Or other tools mentioned in the same doc, if pyinstaller fails to serve the purpose)
  - A program that converts (packages) Python programs into stand-alone executables, for Windows, Linux, Mac and Irix.
  - Cross-platform distribution can be managed in this manner.

- Flatpak (Specifically for Linux Distributions)
  - **Why Flatpak?** For Linux distributions, distributing the application as a Flatpak package is a practical choice. Flatpak provides a sandboxed environment for applications, ensuring security and isolation, and simplifies distribution across different Linux distributions.
  - **Advantages**
    - **Cross-Distribution Compatibility:** Flatpak packages work across all major Linux distributions, simplifying distribution and installation for users.
    - **Security and Isolation:** Flatpak's sandboxed environment enhances security by isolating applications from the rest of the system.
  - **Why not alternatives ?**
    - **Security:**
      Unlike Snaps, Flatpak doesn't rely on systemd making it more portable and less vulnerable.
      Mandatory sandboxing ensures applications run in isolation, enhancing security.
    - **Efficiency:**
      Flatpak shares runtimes across applications, minimizing wasted disk space and resources compared to AppImages.
      Here's a brief write up comparing them (not so comprehensive though)

- **Docker**
  - While not ideal for end-users, we can keep it as an option, it'll be helpful in the dev phase.

- **VMs (QEMU)**
  - **Why VMs?** To ensure compatibility with various platforms, including Windows, macOS, Linux, and other Unix-like operating systems.

Implementation for UI Conversion and Functionality Binding

The UI conversion and functionality binding using PyQt will follow a structured approach that leverages the capabilities of PyQt and Qt Designer. This process will involve converting the .ui files created with Qt Designer into Python scripts, defining actions and functionality, and binding these with the UI elements.

Here's a write up demonstrating the possible implementation.

1. Converting .ui Files to Python Scripts The first step is to convert the .ui files generated by Qt Designer into Python scripts. This can be achieved using the `pyuic` tool that comes with PyQt. The command to convert a .ui file to a .py file is as follows:

   ```
   python -m PyQt5.uic.pyuic -x input.ui -o ui_form.py
   ```

   This command takes the input.ui file and converts it into a Python script named output.py. This step can be automated using docker compose.

2. Defining Actions and Functionality After converting the .ui files to Python scripts, the next step is to define the actions and functionality that will be bound to the UI elements. This involves creating a separate Python file (e.g., actions.py) where all the functions related to the application's logic are defined. These functions will be responsible for handling user interactions, processing data, and updating the UI accordingly.
3. Binding Functions with UI Elements Once the actions and functionality are defined, they need to be bound with the corresponding UI elements. This is done by importing the generated Python script (from Step 1) and the actions.py file into the main application script. Then, using the Ui_Form class generated by the pyuic tool, the functions can be bound to the UI elements.

For example:

```
from PyQt5 import QtWidgets
from ui_form import Ui_Form
from actions import your_function

class MainWindow(QtWidgets.QMainWindow, Ui_Form):
  def __init__(self, parent=None):
    super(MainWindow, self).__init__(parent)
    self.setupUi(self)
    self.yourButton.clicked.connect(your_function)
```

In this example, yourButton is the name of the button in the UI, and your_function is the function defined in actions.py that should be executed when the button is clicked.

Rough timeline

This is just a general outline, and specific details will be finalized with mentor:

- **Community Bonding Period (May):**
    - Familiarize with the codebase and development process.
    - Start discussions with the mentor on the project approach.
- **Development Phase (June - July):**
    - Focus on core development tasks outlined in the proposal.
    - Regularly communicate progress with the mentor and seek feedback.
    - Participate in code reviews and integrate contributions.
- **Documentation and Testing Phase (August):**
    - Write comprehensive user documentation for the completed features.
    - Thoroughly test implemented functionalities.
    - Fix any bugs or issues identified during testing.