Strategy for Writing Modular Streamlit Scripts

1. Project Structure

Organize your Streamlit project with a clear directory structure to separate presentation logic (Streamlit UI) from other logic (data processing, utilities, etc.). A sample structure might look like this:

my\_streamlit\_app/

├── app.py # Main Streamlit entry point

├── pages/ # Multi-page app scripts

│ ├── page1.py

│ ├── page2.py

├── components/ # Reusable Streamlit UI components

│ ├── ui\_elements.py

│ ├── charts.py

├── logic/ # Business logic and data processing

│ ├── data\_processing.py

│ ├── calculations.py

├── utils/ # Utility functions (e.g., helpers, configs)

│ ├── config.py

│ ├── helpers.py

├── data/ # Static data files or cached data

│ ├── sample\_data.csv

├── tests/ # Unit tests for logic and utilities

│ ├── test\_data\_processing.py

├── requirements.txt # Dependencies

├── README.md # Documentation

* app.py: The main entry point for the Streamlit app, handling top-level navigation and minimal UI logic.
* pages/: Contains scripts for individual pages in a multi-page Streamlit app.
* components/: Houses reusable UI components (e.g., custom buttons, charts, or forms).
* logic/: Contains business logic, data processing, or API calls, separate from Streamlit code.
* utils/: General-purpose utilities like configuration management or helper functions.
* data/: Static data or cached files.
* tests/: Unit tests to ensure logic and utilities work as expected.

This structure keeps files small, focused, and easy to navigate.

2. Separation of Concerns

Separate the presentation layer (Streamlit UI) from the business logic and data processing to avoid tight coupling and improve testability.

* Presentation Layer (Streamlit):
  + Use app.py and pages/\*.py for Streamlit-specific code (e.g., st.write, st.button, st.plotly\_chart).
  + Keep these files focused on rendering the UI and capturing user input.
  + Avoid complex computations or data processing in these files.
  + Example (pages/page1.py):

python

import streamlit as st

from components.ui\_elements import render\_header, render\_data\_table

from logic.data\_processing import load\_data, filter\_data

def main():

render\_header("Page 1: Data Explorer")

data = load\_data("data/sample\_data.csv")

filtered\_data = filter\_data(data, st.session\_state.get("filters", {}))

render\_data\_table(filtered\_data)

if \_\_name\_\_ == "\_\_main\_\_":

main()

* Business Logic and Data Processing:
  + Place data processing, calculations, or API calls in the logic/ directory.
  + These modules should be independent of Streamlit and return processed data or results.
  + Example (logic/data\_processing.py):

python

import pandas as pd

def load\_data(file\_path):

return pd.read\_csv(file\_path)

def filter\_data(data, filters):

filtered = data.copy()

for key, value in filters.items():

if value:

filtered = filtered[filtered[key] == value]

return filtered

* Reusable UI Components:
  + Store reusable Streamlit components (e.g., custom charts, forms) in components/.
  + These should focus on rendering UI elements and accept data as input.
  + Example (components/ui\_elements.py):

python

import streamlit as st

def render\_header(title):

st.title(title)

st.markdown("---")

def render\_data\_table(data):

st.dataframe(data, use\_container\_width=True)

* Utilities:
  + Store helper functions, configurations, or logging in utils/.
  + Example (utils/config.py):

python

DATA\_PATH = "data/sample\_data.csv"

APP\_TITLE = "My Streamlit App"

This separation ensures that:

* UI code is clean and focused on presentation.
* Business logic is reusable, testable, and independent of Streamlit.
* Changes in one layer (e.g., UI) don’t break the other (e.g., logic).

3. Keep Files Small and Focused

* Single Responsibility Principle: Each file should have a single purpose (e.g., one page, one set of UI components, one data processing task).
* Limit File Size: Aim for files under 200-300 lines. If a file grows larger, refactor it into smaller modules.
* Modular Functions: Break down complex functions into smaller ones. For example, instead of a single process\_data function, split it into load\_data, clean\_data, and aggregate\_data.
* Avoid Global State: Use st.session\_state for state management in Streamlit, but avoid global variables in logic/ or utils/ to prevent interference.

4. Debugging and Error Handling

* Modular Testing:
  + Write unit tests for logic/ and utils/ modules using a framework like pytest.
  + Example (tests/test\_data\_processing.py):

python

import pytest

from logic.data\_processing import filter\_data

def test\_filter\_data():

data = pd.DataFrame({"A": [1, 2, 3], "B": ["x", "y", "x"]})

filters = {"B": "x"}

result = filter\_data(data, filters)

assert len(result) == 2

assert all(result["B"] == "x")

* + Since logic/ is Streamlit-independent, you can test it without running the app.
* Error Handling:
  + Add try-except blocks in logic/ to handle data or API errors gracefully.
  + Display user-friendly error messages in the UI using st.error.
  + Example (logic/data\_processing.py):

python

def load\_data(file\_path):

try:

return pd.read\_csv(file\_path)

except FileNotFoundError:

raise ValueError(f"Data file not found: {file\_path}")

In pages/page1.py:

python

try:

data = load\_data("data/sample\_data.csv")

except ValueError as e:

st.error(str(e))

return

* Logging:
  + Use Python’s logging module in utils/ to log errors or debug information.
  + Example (utils/helpers.py):

python

import logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

def log\_action(action):

logger.info(f"Performed action: {action}")

* Streamlit Debugging:
  + Use st.write or st.json to inspect data during development.
  + Enable Streamlit’s --logger.level=debug flag when running the app to get detailed logs:

bash

streamlit run app.py --logger.level=debug

5. Preventing Code Interference

* Avoid Circular Imports:
  + Ensure modules in logic/ and utils/ don’t import from pages/ or components/.
  + Use explicit imports (e.g., from logic.data\_processing import load\_data) instead of relative imports to reduce confusion.
* Namespace Isolation:
  + Use unique function and variable names to avoid conflicts.
  + If using multiple pages, ensure st.session\_state keys are page-specific (e.g., page1\_filters vs. page2\_filters).
* Dependency Management:
  + List all dependencies in requirements.txt to ensure consistent environments.
  + Example:
  + streamlit==1.38.0
  + pandas==2.2.2

pytest==8.3.2

* Version Control:
  + Use Git to track changes and avoid accidental overwrites.
  + Commit small, logical changes to make debugging easier.

6. Streamlit-Specific Best Practices

* Multi-Page Apps:
  + Use Streamlit’s multi-page app feature by placing page scripts in pages/.
  + Configure navigation in app.py:

python

import streamlit as st

st.set\_page\_config(page\_title="My App", layout="wide")

st.sidebar.title("Navigation")

page = st.sidebar.selectbox("Choose a page", ["Home", "Page 1", "Page 2"])

if page == "Home":

st.write("Welcome to the app!")

elif page == "Page 1":

from pages.page1 import main

main()

elif page == "Page 2":

from pages.page2 import main

main()

* Session State for Interactivity:
  + Use st.session\_state to store user inputs or app state, but keep it minimal to avoid complexity.
  + Example:

python

if "filters" not in st.session\_state:

st.session\_state.filters = {}

* Caching for Performance:
  + Use @st.cache\_data or @st.cache\_resource for expensive operations in logic/.
  + Example (logic/data\_processing.py):

python

import streamlit as st

@st.cache\_data

def load\_data(file\_path):

return pd.read\_csv(file\_path)

* Responsive UI:
  + Use st.columns and st.container for layout control.
  + Test the app in different screen sizes to ensure responsiveness.

7. Development Workflow

* Iterative Development:
  + Start with a minimal app.py and one page.
  + Gradually add logic/, components/, and utils/ as needed.
* Testing:
  + Test logic/ and utils/ independently using pytest.
  + Test the UI by running the app locally: streamlit run app.py.
* Documentation:
  + Add docstrings to functions in logic/ and utils/.
  + Update README.md with setup instructions and project overview.
* Code Reviews:
  + If working in a team, use pull requests to review code and catch potential interference.

8. Example End-to-End

Here’s a minimal example combining the above principles:

app.py:

python

import streamlit as st

from pages.page1 import main as page1\_main

st.set\_page\_config(page\_title="My App")

st.sidebar.title("Navigation")

page = st.sidebar.selectbox("Choose a page", ["Page 1"])

if page == "Page 1":

page1\_main()

pages/page1.py:

python

import streamlit as st

from components.ui\_elements import render\_header, render\_data\_table

from logic.data\_processing import load\_data, filter\_data

def main():

render\_header("Data Explorer")

try:

data = load\_data("data/sample\_data.csv")

filter\_key = st.selectbox("Filter by", ["A", "B"])

filter\_value = st.text\_input("Filter value")

st.session\_state.filters = {filter\_key: filter\_value}

filtered\_data = filter\_data(data, st.session\_state.filters)

render\_data\_table(filtered\_data)

except ValueError as e:

st.error(str(e))

if \_\_name\_\_ == "\_\_main\_\_":

main()

components/ui\_elements.py:

python

import streamlit as st

def render\_header(title):

st.title(title)

def render\_data\_table(data):

st.dataframe(data)

logic/data\_processing.py:

python

import streamlit as st

import pandas as pd

@st.cache\_data

def load\_data(file\_path):

try:

return pd.read\_csv(file\_path)

except FileNotFoundError:

raise ValueError(f"Data file not found: {file\_path}")

def filter\_data(data, filters):

filtered = data.copy()

for key, value in filters.items():

if value:

filtered = filtered[filtered[key] == value]

return filtered

tests/test\_data\_processing.py:

python

import pytest

import pandas as pd

from logic.data\_processing import filter\_data

def test\_filter\_data():

data = pd.DataFrame({"A": [1, 2, 3], "B": ["x", "y", "x"]})

filters = {"B": "x"}

result = filter\_data(data, filters)

assert len(result) == 2

assert all(result["B"] == "x")

9. Additional Tips

* Linters and Formatters: Use tools like flake8 or black to enforce code style and catch errors early.
* Virtual Environments: Use venv or poetry to isolate dependencies.
* Profiling: Use Python’s cProfile or Streamlit’s --profile flag to identify performance bottlenecks.
* Version Compatibility: Regularly test with the latest Streamlit version, as updates may introduce breaking changes.

This strategy ensures your Streamlit app is modular, debuggable, and maintainable, with clear separation between presentation and logic, minimizing interference and maximizing reusability. Let me know if you need help with a specific part of this setup!