Advanced concepts of Streamlit

Now that you know how a Streamlit app runs and handles data, let's talk about being efficient. Caching allows you to save the output of a function so you can skip over it on rerun. Session State lets you save information for each user that is preserved between reruns. This not only allows you to avoid unnecessary recalculation, but also allows you to create dynamic pages and handle progressive processes.

Caching

Caching allows your app to stay performant even when loading data from the web, manipulating large datasets, or performing expensive computations.

The basic idea behind caching is to store the results of expensive function calls and return the cached result when the same inputs occur again. This avoids repeated execution of a function with the same input values.

To cache a function in Streamlit, you need to apply a caching decorator to it. You have two choices:

1. st.cache_data is the recommended way to cache computations that return data. Use st.cache_data when you use a function that returns a serializable data object (e.g. str, int, float, DataFrame, dict, list). It creates a new copy of the data at each function call, making it safe against mutations and race conditions. The behavior of st.cache_data is what you want in most cases – so if you're unsure, start with st.cache_data and see if it works!
2. st.cache_resource is the recommended way to cache global resources like ML models or database connections. Use st.cache_resource when your function returns unserializable objects that you don’t want to load multiple times. It returns the cached object itself, which is shared across all reruns and sessions without copying or duplication. If you mutate an object that is cached using st.cache_resource, that mutation will exist across all reruns and sessions.

Example:

```
@st.cache_data
def long_running_function(param1, param2):
    return …
```

In the above example, long_running_function is decorated with @st.cache_data. As a result, Streamlit notes the following:

The name of the function ("long_running_function").
The value of the inputs (param1, param2).
The code within the function.

> Before running the code within long_running_function, Streamlit checks its cache for a previously saved result. If it finds a cached result for the given function and input values, it will return that cached result and not rerun the function's code.

> Otherwise, Streamlit executes the function, saves the result in its cache, and proceeds with the script run. During development, the cache updates automatically as the function code changes, ensuring that the latest changes are reflected in the cache.

Streamlit's two caching decorators and their use cases.

1. Use st.cache_data for anything you'd store in a database.
2. Use st.cache_resource for anything you can't store in a database, like a connection to a database or a machine learning model.

Streamlit's two caching decorators and their use cases.

For more information about the Streamlit caching decorators, their configuration parameters, and their limitations, see Caching.

Session State

Session State provides a **dictionary-like** interface where you can save information that is preserved between script reruns. Use **st.session_state with key or attribute** notation to store and recall values. For example, **st.session_state["my_key"] or st.session_state.my_key**. Remember that widgets handle their statefulness all by themselves, so you won't always need to use Session State!

What is a session?

A session is a single instance of viewing an app. **If you view an app from two different tabs in your browser, each tab will have its own session.** So each viewer of an app will have a Session State tied to their specific view. Streamlit maintains this session as the user interacts with the app. If the user refreshes their browser page or reloads the URL to the app, their Session State resets and they begin again with a new session.

Examples of using Session State

Here's a simple app that counts the number of times the page has been run. Every time you click the button, the script will rerun.

```
import streamlit as st

if "counter" not in st.session_state:
    st.session_state.counter = 0

st.session_state.counter += 1

st.header(f"This page has run {st.session_state.counter} times.")
```

st.button("Run it again")

First run: The first time the app runs for each user, Session State is empty. Therefore, a key-value pair is created ("counter":0). As the script continues, the counter is immediately incremented ("counter":1) and the result is displayed: "This page has run 1 times." When the page has fully rendered, the script has finished and the Streamlit server waits for the user to do something. When that user clicks the button, a rerun begins.

Second run: Since "counter" is already a key in Session State, it is not reinitialized. As the script continues, the counter is incremented ("counter":2) and the result is displayed: "This page has run 2 times."

There are a few common scenarios where Session State is helpful. As demonstrated above, Session State is used when you have a progressive process that you want to build upon from one rerun to the next. Session State can also be used to prevent recalculation, similar to caching. However, the differences are important:

Caching associates stored values to specific functions and inputs. Cached values are accessible to all users across all sessions.

Session State associates stored values to keys (strings). Values in session state are only available in the single session where it was saved.

If you have random number generation in your app, you'd likely use Session State. Here's an example where data is generated randomly at the beginning of each session. By saving this random information in Session State, each user gets different random data when they open the app but it won't keep changing on them as they interact with it. If you select different colors with the picker you'll see that the data does not get re-randomized with each rerun. (If you open the app in a new tab to start a new session, you'll see different data!)

```
import streamlit as st
import pandas as pd
import numpy as np

if "df" not in st.session_state:
    st.session_state.df = pd.DataFrame(np.random.randn(20, 2), columns=["x", "y"])

st.header("Choose a datapoint color")
color = st.color_picker("Color", "#FF0000")
st.divider()
st.scatter_chart(st.session_state.df, x="x", y="y", color=color)
```

If you are pulling the same data for all users, you'd likely cache a function that retrieves that data. On the other hand, if you pull data specific to a user, such as querying their personal

information, you may want to save that in Session State. That way, the queried data is only available in that one session.

As mentioned in Basic concepts, Session State is also related to widgets. Widgets are magical and handle statefulness quietly on their own. As an advanced feature however, you can manipulate the value of widgets within your code by assigning keys to them. Any key assigned to a widget becomes a key in Session State tied to the value of the widget. This can be used to manipulate the widget. After you finish understanding the basics of Streamlit, check out our guide on Widget behavior to dig in the details if you're interested.

Connections

As hinted above, you can use @st.cache_resource to cache connections. This is the most general solution which allows you to use almost any connection from any Python library. However, Streamlit also offers a convenient way to handle some of the most popular connections, like SQL! st.connection takes care of the caching for you so you can enjoy fewer lines of code. Getting data from your database can be as easy as:

```
import streamlit as st

conn = st.connection("my_database")
df = conn.query("select * from my_table")
st.dataframe(df)
```

Of course, you may be wondering where your username and password go. Streamlit has a convenient mechanism for Secrets management. For now, let's just see how st.connection works very nicely with secrets. In your local project directory, you can save a .streamlit/secrets.toml file. You save your secrets in the toml file and st.connection just uses them! For example, if you have an app file streamlit_app.py your project directory may look like this:

```
your-LOCAL-repository/
├── .streamlit/
│   └── secrets.toml # Make sure to gitignore this!
└── streamlit_app.py
```

For the above SQL example, your secrets.toml file might look like the following:

```
[connections.my_database]
  type="sql"
  dialect="mysql"
  username="xxx"
  password="xxx"
  host="example.com" # IP or URL
  port=3306 # Port number
```

```
database="mydb" # Database name
```

Since you don't want to commit your secrets.toml file to your repository, you'll need to learn how your host handles secrets when you're ready to publish your app. Each host platform may have a different way for you to pass your secrets. If you use Streamlit Community Cloud for example, each deployed app has a settings menu where you can load your secrets. After you've written an app and are ready to deploy, you can read all about how to Deploy your app on Community Cloud.