Web App Development with Streamlit

Intro

```
import streamlit as st
st.write("Hello")
```

```
streamlit --help
streamlit run <script.py> [--script args]
streamlit cache clear
streamlit docs
streamlit --version
streamlit run --help
streamlit config show
streamlit hello
```

You may configure these options using one of the four following methods:

1. Using a global config file at .streamlit/config.toml
2. Using a config file for each project in your project's directory: C:/Users/.../.streamlit/config.toml.
3. Using STREAMLIT_* environment variables
4. Using flags in the command line when running your script: streamlit run <script.py> --server.port 80

Writing and Testing Code

```
def calculate_sum(n1, n2):
    return n1 + n2
st.title("Add numbers")
n1 = st.number_input("First Number")
n2 = st.number_input("Second Number")
if st.button("Calculate"):
    st.write("Summation is: " + str(calculate_sum(n1, n2)))
```

```
# unit_test.py
from main import calculate_sum
from selenium import webdriver
from selenium.webdriver.chrome.options import Options import time
def test_user_interface():
    # Path to chromedriver. Can end with .sh if on (Li/U)nix environments
    driver_path = r"----------\chromedriver.exe"
    options = Options()
    options.add_argument("--headless") # To not open a real chrome window
    with webdriver.Chrome(driver_path, chrome_options=options) as driver:
```

```
        url = "http://127.0.0.1:8501"
        driver.get(url)
        time.sleep(5)
        html = driver.page_source

    assert "Add numbers" in html
    assert "First Number" in html
    assert "Second Number" in html

def test_logic():
    assert calculate_sum(1, 1) == 2
    assert calculate_sum(1, -1) == 0
    assert calculate_sum(1, 9) == 10

if __name__ == "__main__":
    test_logic()
    test_user_interface()
```

To test rendering, we first need the application to be running, so it can be accessed by selenium's driver, which is Chrome in this example. To automate launching the application before running the tests, we might need to use an advanced IDE. PyCharm can do the required job for us.

Having a Streamlit web application starts from executing the binary, whether streamlit.exe on Windows or stream-lit.sh on MacOS or Linux, against the target document using the default Python interpreter. This will initialize the application configuration such as secrets, settings, themes, and, most importantly, the Delta Generator (DG for short) which acts as the middleman between the Python script and the ReactJS web application served by Streamlit.

Streamlit uses ReactJS's virtual DOM to insert elements and manage their state. Knowing this fact and Streamlit's source code, we can conclude that Streamlit uses built-in ReactJS components grouped to make a fully fledged JavaScript web application with Python! In addition to that, we can leverage Streamlit's generic treatment of components to build custom and complex ones that are not provided out of the box in later chapters.

Streamlit basics

User Input Forms

Creating forms in Streamlit can be as simple as bundling several text, number, and other input widgets together on a page coupled with a button that triggers an action such as recording the entries on a database or saving it to the session state. The caveat with this approach is that each time the user interacts with any of the widgets, Streamlit will automatically rerun the entire script from top to bottom. While this approach provides the developer with a logical and seamless flow to their program, it may sometimes be useful to bundle input widgets together and have Streamlit rerun the script only when prompted by the user. This can be done using the st.form command.

```
with st.form('feedback_form'):
    st.header('Feedback form')
    # Creating columns to organize the form
```

```
    col1, col2 = st.columns(2)
    with col1:
        name = st.text_input('Please enter your name')
        rating = st.slider('Please rate this app',0,10,5)
    with col2:
        dob = st.date_input('Please enter your date of birth')
        recommend = st.radio('Would you recommend this app to
        others?',('Yes','No'))
    submit_button = st.form_submit_button('Submit')
if submit_button:
    st.write('**Name:**', name, '**Date of birth:**', dob, '**Rating:**',
rating,'**Would recommend?:**', recommend)
```

A Streamlit form can be called within a with statement and using the st.form command.

Conditional flow

It may be necessary to introduce conditional flow to your Streamlit applications whereby certain actions will be dependent on prior actions or on the state of widgets. This is especially useful for prompting the user to correctly fill in forms and/or use the application correctly. Without implementing conditional flow, your application will be prone to throwing errors should the user make incorrect use of it.

```
def display_name(name):
    st.info(f'**Name:** {name}')
name = st.text_input('Please enter your name')
if not name:
    st.error('No name entered')
if name:
    display_name(name)
```

```
def display_name(name):
    st.info(f'**Name:** {name}')
name = st.text_input('Please enter your name')
if not name:
    st.error('No name entered')
    st.stop()
display_name(name)
```

The only difference here is that we are utilizing the st.stop command to stop the execution of the script if the name field is empty. And if it is not empty, we are proceeding with the rest of the script that displays the entered name. The benefit of this method is that it eliminates the need for an additional if statement and simplifies your script; otherwise, in terms of utility, both examples are identical.

Conditional flow programming can be applied to both trivial and nontrivial applications. Indeed, this technique can be scaled and implemented with nested if statements, while loops, and other methods if required.

Managing and Debugging Errors

If you are running Streamlit in development mode and have configured showErrorDetails = True. Then Streamlit will natively display runtime exceptions on the web page similar to how any other IDE would display such messages on the console. This is far from ideal as the user will not be able to make much sense of it and will be left confused. More importantly, leaving exceptions mismanaged can trigger a chain of fatal errors in subsequent parts of your code that can have a cascading effect on other systems. In addition, Streamlit will display the errorsome segment of your code that is causing the exception to the user, which may infringe on intellectual property rights, should your source code be subject to such restrictions.

```
# No try and except
col1, col2 = st.columns(2)
with col1:
    number_1 = st.number_input('Please enter the first
number',value=0,step=1)
with col2:
    number_2 = st.number_input('Please enter the second
number',value=0,step=1)
st.info(f'**{number_1}/{number_2}=** {number_1/number_2}')
```

```
# With try and except
col1, col2 = st.columns(2)
with col1:
    number_1 = st.number_input('Please enter the first
number',value=0,step=1)
with col2:
    number_2 = st.number_input('Please enter the second
number',value=0,step=1)
try:
    st.info(f'**{number_1}/{number_2}=** {number_1/number_2}')
except ZeroDivisionError:
    st.error('Cannot divide by zero')
```

Another way of managing exceptions is to use a general except statement, and within it displaying a curated message of the actual error occurring. This can be particularly useful for debugging the script for the developer themselves while not compromising on the user experience.

```
col1, col2 = st.columns(2)
with col1:
    number_1 = st.number_input('Please enter the first
number',value=0,step=1)
with col2:
    number_2 = st.number_input('Please enter the second
number',value=0,step=1)
try:
    st.info(f'**{number_1}/{number_2}=** {number_1/number_2}')
```

```
except Exception as e:
    st.error(f'Error: {e}')
```

Mutating dataframes

As a developer you will often need to mutate dataframes in response to a user input or need. Here: a non-exhaustive list of some of the most commonly used methods of mutating Pandas dataframes.

Filter

We can simply specify a condition for a column numerical and/or string, that is, df[df['Column 1'] > -1], and filter the rows based on that.

```
import pandas as pd
import numpy as np
np.random.seed(0)
df = pd.DataFrame(
    np.random.randn(4, 3),
    columns=('Column 1','Column 2','Column 3')
                  )
st.subheader('Original dataframe')
st.write(df)

df = df[df['Column 1'] > -1]
st.subheader('Mutated dataframe')
st.write(df)
```

Select

Dataframe columns can be selected by using the method shown below. We can specify which columns to keep by their names, that is, df[['Column 1', 'Column 2']], and remove other columns. Alternatively, the exact same function can be achieved by using the drop command, that is, df.drop(columns=['Column 3']).

```
import pandas as pd import numpy as np
np.random.seed(0)
df = pd.DataFrame(
    np.random.randn(4, 3),
    columns=('Column 1','Column 2','Column 3')
                  )
st.subheader('Original dataframe')
st.write(df)
df = df[['Column 1', 'Column 2']]
st.subheader('Mutated dataframe')
st.write(df)
```

Arrange

Dataframe columns can be arranged and sorted in ascending and/or descending order based on the numerical or nominal value of a specified column by using the method shown in Listing 2-9. We can specify which column to sort and in which order, that is, df.sort_values(by='Column 1',ascending=True), as shown in below. Once the column has been sorted, the index will be modified to reflect the new order. If necessary, you may reset the index by using the df.reset_index(drop=True) command to restart the index from zero.

```
import pandas as pd import numpy as np
np.random.seed(0)
df = pd.DataFrame(
    np.random.randn(4, 3),
    columns=('Column 1','Column 2','Column 3')
                  )
st.subheader('Original dataframe')
st.write(df)
df = df.sort_values(by='Column 1',ascending=True)
st.subheader('Mutated dataframe')
st.write(df)
```

Mutate

Dataframe columns can be mutated by assigning new columns based on the value of another column by using the method shown below. We can specify a simple lambda function to apply to the values of an existing column, that is, Column_4 = lambda x: df['Column 1]*2, to compute the output.

```
import pandas as pd import numpy as np
np.random.seed(0)
df = pd.DataFrame(
    np.random.randn(4, 3),
    columns=('Column 1','Column 2','Column 3')
                  )
st.subheader('Original dataframe')
st.write(df)
df = df.assign(Column_4 = lambda x: df['Column 1']*2)
st.subheader('Mutated dataframe')
st.write(df)
```

Group By

Sometimes, it may be necessary to group or aggregate the values in a column or several columns in a dataframe. This can be done in Pandas using the method shown below. We can specify which column or columns to group by with the df.groupby(['Column 1', 'Column 2']) command. This will reindex the dataframe and group the relevant rows together.

```
import pandas as pd import numpy as np
df = pd.DataFrame(
    np.random.randn(12, 3),
```

```
    columns=('Score 1','Score 2','Score 3')
)
df['Name'] = pd.DataFrame(['John','Alex','Jessica','John','Alex', 'John',
'Jessica','John','Alex','Alex','Jessica','Jessica'])
df['Category'] =
pd.DataFrame(['B','A','D','C','C','A','B','C','B','A','A','D'])
st.subheader('Original dataframe')
st.write(df)
df = df.groupby(['Name','Category']).first()
st.subheader('Mutated dataframe')
st.write(df)
```

Merge

Multiple dataframes can be merged together with Pandas utilizing a common column as a reference using the method shown below. We can specify which column to merge on and whether the merge should be a union or intersection of both dataframes with the df1.merge(df2,how ='inner',on='Name') command. This will create a combined dataframe.

```
import pandas as pd
df1 = pd.DataFrame(data={'Name':['Jessica','John','Alex'], 'Score 1':
[77,56,87]})
df2 = pd.DataFrame(data={'Name':['Jessica','John','Alex'], 'Score 2':
[76,97,82]})
st.subheader('Original dataframes')
st.write(df1)
st.write(df2)
df1 = df1.merge(df2,how='inner',on='Name')
st.subheader('Mutated dataframe')
st.write(df1)
```

Rendering Static and Interactive Charts

In this section, we will develop several instances of static and interactive charts using data from a Pandas dataframe.

Static Bar Chart

Can be generated by inputting a Pandas dataframe into a Matplotlib figure by using the method shown here. We can specify the chart type by setting kind='bar'. Other Matplotlib parameters can be found at https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html.

```
import pandas as pd
import matplotlib.pyplot as fig
df = pd.DataFrame(data={'Name':['Jessica','John','Alex'], 'Score 1':
[77,56,87],'Score 2':[76,97,82]}
)
```

```
df.set_index('Name').plot(kind='bar',stacked=False,xlabel='Name',
ylabel='Score')
st.pyplot(fig)
```

Static Line Chart

Similarly, a static line chart can be generated by inputting a Pandas dataframe into a Matplotlib figure by using the method shown here. We can specify the chart type and the option of having subplots by setting kind='line', subplots=True. Other Matplotlib parameters can be found at https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html.

```
import pandas as pd
import matplotlib.pyplot as fig
df = pd.DataFrame(data={'Exam':['Exam 1','Exam 2','Exam 3'], 'Jessica':
[77,76,87],'John':[56,97,95],'Alex':[87,82,93]})
df.set_index('Exam').plot(kind='line',xlabel='Exam',ylabel=
'Score',subplots=True)
st.pyplot(fig)
```

Interactive Line Chart

An interactive line chart can be generated by inputting a Pandas dataframe into a Plotly figure by using the method shown here. We can simply declare the chart type and its associated properties by using the JSON notation used with all Plotly charts and figures. Other Plotly parameters can be found at https://plotly.com/python/line-charts/

```
import pandas as pd
import plotly.graph_objects as go
df = pd.DataFrame(data={'Exam':['Exam 1','Exam 2','Exam 3'],
'Jessica':[77,76,87],'John':[56,97,95],'Alex':[87,82,93]}
)
fig = go.Figure(data=[
    go.Line(name='Jessica', x=df['Exam'], y=df['Jessica']),
    go.Line(name='John', x=df['Exam'], y=df['John']),
    go.Line(name='Alex', x=df['Exam'], y=df['Alex'])
])
fig.update_layout(
    xaxis_title='Exam',
    yaxis_title='Score',
    legend_title='Name',
)
st.plotly_chart(fig)
```

Interactive Map

An interactive geospatial map can be generated by inputting a Pandas dataframe containing the longitude and latitude of a set of points into a Plotly figure by using the method shown below. In addition, we can declare the exact geospatial location to zoom into by setting geo_ scope='usa'. Other Plotly parameters can be found at https://plotly.com/python/scatter-plots-on-maps/

```
import pandas as pd
import plotly.graph_objects as go
df = pd.DataFrame(data={'university':['Harvard University','Yale
University',
'Princeton University','Columbia University','Brown University',
'Dartmouth University','University of Pennsylvania','Cornell University'],
'latitude':[42.3770,41.3163,40.3573,40.8075,41.8268,
43.7044,39.9522,42.4534],
'longitude':[-71.1167,-72.9223,-74.6672,-73.9626,-71.4025, -72.2887,
-75.1932,-76.4735]}
)
fig = go.Figure(data=go.Scattergeo(
         lon = df['longitude'],
         lat = df['latitude'],
         text = df['university'],
         ))
fig.update_layout(
        geo_scope='usa',
    )
st.plotly_chart(fig)
```

Developing the User Interface

Streamlit: intuitive, responsive, and standardized interface, whereby the developer can render a web page without having to worry about the intricate details of design. Streamlit interfaces are plug and play, allowing the developer to focus on the logic of their program while leaving the visual implementation to Streamlit. Should the user require something more bespoke and customized, they are welcome to integrate their own HTML and/or JavaScript components. Customizability and external components will be addressed in subsequent chapters.

```
import pandas as pd
import plotly.express as px
program = st.sidebar.selectbox('Select program',['DataframeDemo','Other
Demo'])
code = st.sidebar.checkbox('Display code')
if program == 'Dataframe Demo':
    df = px.data.stocks()
    st.title('DataFrame Demo')
    stocks = st.multiselect('Select stocks',df.columns[1:],
[df.columns[1]])
    st.subheader('Stock value')
    # Mutating the dataframe to keep selected columns only
    st.write(df[['date'] + stocks].set_index('date'))
    # Creating a Plotly timeseries line chart
```

```
        fig = px.line(df, x='date', y=stocks,
                      hover_data={"date": "|%Y %b %d"}
                )
    st.write(fig)
    if code:
        st.code(
                """
        import streamlit as st
        import pandas as pd
        import plotly.express as px
        df = px.data.stocks()
        st.title('DataFrame demo')
        program = st.sidebar.selectbox('Select
        program',['Dataframe Demo'])
        code = st.sidebar.checkbox('Display code')
        stocks = st.multiselect('Select stocks',df.columns[1:],[df.
        columns[1]])
        st.subheader('Stock value')
        st.write(df[['date'] + stocks].set_index('date'))
        fig = px.line(df, x='date', y=stocks,
                      hover_data={"date": "|%Y %b %d"}
        )
        st.write(fig)
                """
    )
    elif program == 'Other Demo':
        st.title('Other Demo')
```

Architecting the User Interface

Should one need to develop more bespoke and tailored applications, Streamlit affords a considerable degree of customization of its frontend design to the developer without soliciting any knowledge of HTML, CSS, or JavaScript. One may configure their application using a multitude of color schemes, fonts, and appearances both graphically and programmatically.

Furthermore, Streamlit gives you the ability to immaculately structure and organize the web page using a combination of a sidebar, columns, expanders, and containers. Together, these elements can offer the user an enhanced experience while making efficient use of the space on the page.

Furthermore, with placeholders and progress bars, you can render dynamic content on demand or in response to an event. And most importantly, with Streamlit you can create as many pages and nested subpages as required for your application in a highly modularized and scalable way.

Designing the Application

We can make use of the multitude of native methods it provides to customize our page to be exactly the way we want. From color schemes and themes to columns, expanders, sidebar, and placeholders, the number of design permutations we can achieve is endless.

Configuring the Page

You have the ability to configure several attributes of the web page, namely, configuring the page layout, initial sidebar state, page title (displayed on the browser), icon, hamburger menu state, footer, and more. While some of these items can be configured within your script, the rest must be set within the global config file /.streamlit/config.toml

Basic page configuration

The following listing can be used to configure the page title, icon, layout, initial sidebar state, and menu items. Please note that the page icon supports .ico files. You may use the package Pillow to import and handle images. The page layout can be 'centered' or 'wide', and the initial sidebar state can be 'auto', 'expanded', or 'collapsed'. In addition, you may choose to customize one or more of the following pages in the hamburger menu: 'Get help', 'Report a bug', or 'About'. The Get help and Report a bug pages can only be instantiated with a URL that will redirect the user to another web page. The About page can be instantiated as a modal window.

```
from PIL import Image
icon = Image.open('favicon.ico')
st.set_page_config(
    page_title='Hello world',
    page_icon=icon,
    layout='centered',
    initial_sidebar_state='auto',
    menu_items={
        'Get Help': 'https://streamlit.io/',
        'Report a bug': 'https://github.com',
        'About': 'About your application: **Hello world**'
        }
    )
st.sidebar.title('Hello world')
st.title('Hello world')
```

Removing Footer and Hamburger Menu

You may choose to remove the default footer provided by Streamlit as well as the entire hamburger menu by adding the commands. Please note that while modifications to the CSS may work in the current version of Streamlit, this exact method however is not guaranteed to work in subsequent versions, should the internal implementation be changed.

```
hide_footer_style = "<style>.reportview-container .main footer
{visibility: hidden;}</style>"
st.markdown(hide_footer_style, unsafe_allow_html=True)
hide_menu_style = "<style>#MainMenu {visibility: hidden;}</style>"
st.markdown(hide_menu_style, unsafe_allow_html=True)
```

Adding a Customized Footer

Furthermore, you may add your own customized footer by adding the markdown command shown below:

```
st.sidebar.markdown(
    f"'<div class="markdown-text-container stText"
    style="width: 698px;">
    <footer><p></p></footer><div style="font-size: 12px;">
    Hello world v 0.1</div>
    <div style="font-size: 12px;">Hello world LLC.</div>
    </div>"', unsafe_allow_html=True)
```

Advanced Page Configuration

Aside from the basic configurable settings of a Streamlit web page, there is a multitude of other parameters that can be configured as required by modifying the global config file /.streamlit/config.toml. Parameters that can be configured pertain to the global, logger, client, runner, server, browser, mapbox, deprecation, AWS S3, and theme settings. For further information, please refer to https://docs.streamlit.io/library/advanced-features/configuration.

Config.toml settings:

```
[global]
disableWatchdogWarning = false
showWarningOnDirectExecution = true
dataFrameSerialization = 'arrow'

[logger]
level = 'debug'
messageFormat = "%(asctime)s %(message)s".

[client]
caching = true
displayEnabled = true
showErrorDetails = true

[runner]
magicEnabled = true
installTracer = false
fixMatplotlib = true
postScriptGC = true
fastReruns = false

[server]
folderWatchBlacklist = [ ]
fileWatcherType = 'auto'
cookieSecret = 'key'
headless = false
runOnSave = false
address =
port = 8501
```

```
baseUrlPath = 'URL'
enableCORS = true
enableXsrfProtection = true
maxUploadSize = 200
enableWebsocketCompression = false

[browser]
serverAddress = 'localhost'
gatherUsageStats = true
serverPort = 8501

[mapbox] # zie mapbox.com
token = ' '

[deprecation]
showfileUploaderEncoding = 'True'
showPyplotGlobalUse = 'True'

[s3]
bucket =
url =
accessKeyId =
secretAccessKey =
keyPrefix =
region =
profile =

[theme]
base =
primaryColor =
backgroundColor =
secondaryBackgroundColor =
textColor =
font =
```

Developing Themes and Color schemes

Streamlit gives both the developer and the user the ability to customize the theme and color scheme of the application both graphically and programmatically.

Customizing the Theme Graphically

You may select one of the available theme appearances, Light or Dark, from the settings menu located within the hamburger menu. In addition, you may select colors for each of the following areas: Primary color, Background color, Text color, and Secondary background color, and select one of the available fonts, Sans serif, Serif, or Monospace. Below more information about each color setting and the Streamlit elements that each one alters. Please note that the interactive widgets such as st.slider will use the Secondary background color as their background color when the widget is used within the body of the application; however, when the widget is called within the sidebar, the Background color is used instead.

primary color => interactive widgets such as st.slider Background color => Main body of application text color => all text elements secondary background color => sidebar background color

Customizing the Theme Programmatically

Alternatively, you may choose to customize the theme's appearance and colors programmatically by modifying the global config file /.streamlit/config.toml discussed earlier. You may specify the theme settings by modifying the [theme] parameters.

```
[theme]
base = 'light'
primaryColor = '#7792E3'
backgroundColor = '#273346'
secondaryBackgroundColor = '#B9F1C0'
textColor = '#FFFFFF'
font = 'sans serif'
```

Using Themes with Custom Components

If you are developing your own custom Streamlit components, it may be necessary to pass the application's theme settings to your component. Please ensure that you have installed the latest version of streamlit-component-lib by using the following command: npm install streamlit-component-lib

This package will automatically update the colors of your custom component to reflect the theme settings of your Streamlit application. In addition, it will allow you to read the theme settings in your JavaScript and/or CSS script as shown in the following. As CSS variables, the object settings can be exposed as follows:

```
--primary-color
--background-color
--secondary-background-color
--text-color
--font
```

Accordingly, these settings can be accessed as follows:

```
.mySelector {
  color: var(--primary-color);
}
```

You may also choose to expose the object settings as a ReactJS prop, which may be accessed as follows:

```
{
  "primaryColor": ""
```

```
  "backgroundColor": "",
  "secondaryBackgroundColor": "",
  "textColor": "",
  "font": "",
}
```

Organizing the Page

Streamlit provides several methods to organize and customize the frontend design of an application. Namely, as a developer you have the ability to enable a sidebar, divide the web page into columns, display expander boxes to hide content, and create containers to bundle several widgets together. Combined with one another, these features allow you to customize your application to a tangibly high degree and offer a tailored experience to your users.

Sidebar: you have the ability of subdividing your page with an elegant sidebar that can be expanded and contracted on demand with the st.sidebar command. In addition you may configure Streamlit to keep the sidebar expanded, contracted, or automatically adjusted at the start based on the screen size. Practically, every Streamlit element and widget with the exception of st.echo and st.spinner can be invoked within the sidebar. You may even render other features related to page organization such as st.columns, st.expander, and st.container within the sidebar.

Expanders: you have the ability to collapse content into expanders, should you wish to make more efficient use of the space on the main body or sidebar of your application. Expanders can be expanded and contracted on demand or can be set to either state at the start. In addition, expanders can contain any element including columns and containers but not nested expanders.

Columns: you can divide the main body and sidebar with Streamlit's columns feature that can be called with the st.columns command. You may specify the number of columns you require by writing st.columns(2), or alternatively you may use a list to set the number and width of each column arbitrarily by writing st.columns([2,1]). Columns can be invoked within a with statement and can be used concurrently with expanders and containers; however, they may not be nested within one another.

Containers: Should you require to bundle several widgets or elements together, you can do so with Streamlit's container feature that can be called with the st.container command in the main body or sidebar. Containers can be invoked within a with statement and can be used with columns, expanders, and even nested containers. They can also be altered out of order, for instance, if you display some text out of the container and then display some more text within the container, the latter will be displayed first.

```
from datetime import datetime
#Expander in sidebar
st.sidebar.subheader('Expander')
with st.sidebar.expander('Time'):
    time = datetime.now().strftime("%H:%M:%S")
    st.write('**%s**' % (time))
#Columns in sidebar
st.sidebar.subheader('Columns')
col1, col2 = st.sidebar.columns(2)
with col1:
```

```
    option_1 = st.selectbox('Please select option 1',['A','B'])
with col2:
    option_2 = st.radio('Please select option 2',['A','B'])
#Container in sidebar
container = st.sidebar.container()
container.subheader('Container')
option_3 = container.slider('Please select option 3')
st.sidebar.warning('Elements outside of container will be
displayed externally')
container.info('**Option 3:** %s' % (option_3))
#Expander in main body
st.subheader('Expander')
with st.expander('Time'):
    time = datetime.now().strftime("%H:%M:%S")
    st.write('***%s**' % (time))
#Columns in main body
st.subheader('Columns')
col1, col2 = st.columns(2)
with col1:
    option_4 = st.selectbox('Please select option 4',['A','B'])
with col2:
    option_5 = st.radio('Please select option 5',['A','B'])
#Container in main body
container = st.container()
container.subheader('Container')
option_6 = container.slider('Please select option 6')
st.warning('Elements outside of container will be displayed
externally')
container.info('**Option 6:** %s' % (option_6))
```

Placeholders: one of the most versatile and powerful features brandished by Streamlit. By using the st.empty or st.sidebar.empty command, you have the ability to quite literally reserve space at any location on the main body or sidebar of your application. This can be particularly useful for displaying content out of order or on demand after a certain event or trigger. A placeholder can be invoked by writing placeholder = st.empty(), and subsequently any widget or element can be attached to it whenever required. For instance, you may attach text to it by writing placeholder. info('Hello world'), and then the same placeholder can be replaced by equating it to another element. And finally when it is no longer needed, the placeholder can be cleared using the placeholder.empty() command.

Displaying Dynamic Content

To display dynamic content such as a constantly updating map, chart, or clock, you may place an element within a placeholder and invoke it within a for loop or while loop to iterate over many instances of that element. As a result, the element will appear to be dynamic with a constantly changing state with each iteration of the loop. An example of dynamic content is the clock application built using a placeholder. A while loop is being used to constantly update the st.info element to display current time until the time reaches a predefined point, at which placeholder is cleared and the while loop is ended.

```
from datetime import datetime st.title('Clock')
clock = st.empty()
```

```
while True:
    time = datetime.now().strftime("%H:%M:%S")
    clock.info('**Current time: ** %s' % (time))
    if time == '16:09:50': clock.empty()
        st.warning('Alarm!!')
        break
```

Creating a Real-Time Progress Bar

We can make use of Streamlit's st.progress widget that will render a progress bar showing the value provided to it between 1 and 100 as an integer or 0.0 and 1.0 as a float. As an example we will visualize the progress made in downloading a file from a URL using the package wget. If you have not already done so, please proceed by downloading and installing it using pip install wget. From wget, we can read the total size of a file in bytes and the current amount downloaded in bytes which we will then feed to our progress bar for visualization.

```
import wget
progress_text = st.empty()
progress_bar = st.progress(0)
def streamlit_progress_bar(current,total,width):
    percent = int((current/total)*100)
    progress_text.subheader('Progress: {}%'.format(percent))
    progress_bar.progress(percent)
wget.download('file URL', bar=streamlit_progress_bar)
```

Provisioning Multipage Applications

Pagination and scalability are inherent needs for any web application, and with Streamlit there are indeed multiple ways to address such needs in intuitive and accessible ways.

Creating Pages

You will require one main script that will house the main page of your application with routes to all the other pages In it you will initially import all of the other pages' scripts by writing from pages. page_1 import main_page_1. Please note that if the scripts for the pages are located within another folder, then you will need to append the name of the folder to the script name to import it, that is, from pages.page_1. Subsequently, you will create a drop-down menu using a widget such as st. selectbox with the names of each page and create a dictionary that contains the name of each page as the key and the function to render the page as the value. When the user selects a page from the menu, the name is passed to the dictionary to acquire the corresponding function for that page, and then that function is invoked to render the page.

```
# main_page.py
from pages.page_1 import func_page_1
from pages.page_2 import func_page_2
```

```
def main():
    st.sidebar.subheader('Page selection')
    page_selection = st.sidebar.selectbox('Please select a page',['Main
Page', 'Page 1','Page 2'])
    pages_main = {
        'Main Page': main_page,
        'Page 1': run_page_1,
        'Page 2': run_page_2
    }
    # Run selected page
    pages_main[page_selection]()

def main_page():
    st.title('Main Page')

def run_page_1():
    func_page_1()

def run_page_2():
    func_page_2()

if __name__ == '__main__':
    main()
```

```
# page_1.py
def func_page_1():
    st.title('Page 1')
```

```
# page_2.py
def func_page_2():
    st.title('Page 2')
```

Enabling URL Paths

You may add unique URL paths for each page within your multipage Streamlit application if you download and use the Extra-Streamlit-Components package that will be divulged in great depth later.

Creating Subpages

You also have the ability to create nested subpages within other pages to as many levels as you possibly need. The process for creating subpages is in fact very similar to that of creating pages. Once again, you will need to begin by importing the scripts for all of the pages and subpages into your main script. Once that is done, you will create a separate dictionary of subpages for each page; the dictionary contains the name of the subpage as the key and the associated function to run it as the value. Subsequently, you will create a drop-down menu for the pages using the st.selectbox widget, and then you can use a series of if and elif statements to determine which page has been selected by the user. Within each if statement, you will then

create another drop-down menu for the subpages. Once the user has selected a subpage, it will be passed to the dictionary to acquire the associated function to invoke and render the subpage.

```
from pages.page_1 import func_page_1
from pages.subpages_1.subpage_1_1 import func_subpage_1_1
from pages.subpages_1.subpage_1_2 import func_subpage_1_2

def main():
    st.sidebar.subheader('Page selection')
    page_selection = st.sidebar.selectbox('Please select a page', ['Main
Page','Page 1'])
    pages_main = {
        'Main Page': main_page,
        'Page 1': run_page_1
    }
    subpages_page_1 = {
        'Subpage 1.1': run_subpage_1_1,
        'Subpage 1.2': run_subpage_1_2
    }

    if page_selection == 'Main Page': # Run selected page
        pages_main[page_selection]()
    elif page_selection == 'Page 1':
        st.sidebar.subheader('Subpage selection')
        subpage_selection_1 = st.sidebar.selectbox("Please select a
subpage", tuple(subpages_page_1.keys()))
        # Run selected subpage
        subpages_page_1[subpage_selection_1]()

def main_page():
    st.title('Main Page')

def run_page_1():
    func_page_1()

def run_subpage_1_1():
    func_subpage_1_1()

def run_subpage_1_2():
    func_subpage_1_2()

if __name__ == '__main__':
    main()
```

```
# subpage_1_1.py
def func_subpage_1_1():
    st.title('Subpage 1.1')
```

```
# subpage_1_2.py
def func_subpage_1_2():
    st.title('Subpage 1.2')
```

Modularizing Application Development

In almost every web project, there is a need for visual components and code parts that manages the overall experience which is not directly seen by the end user, rather experienced. This is what is usually referred to as the business logic of the application that is responsible for controlling and managing intermodule communication, specifically whenever there is a reaction to be made upon user action, or to initiate an action by the user such as prompting the user to sign in.

Example: Developing a Social Network Application

For instance, a simple social network application will need components to make a post and to read fields. These two seemingly simple requirements can be broken down into three main parts: views, action handling services, database connection or API (application programming interface) client of another backend service. For that example, everything will be post-centric, which means this should be a reusable, shared resource.

To build the mentioned project, a bottom-top approach shall be followed by starting with the most dependent object of the whole design, which is the Post class, and topping it with the user-visible views. The post shall be a class as shown below because it encapsulates related data in one form; in Python, we can use the dataclasses decorator to denote this is just a class to hold data, and we can give it a default initialization function to fill in values for the declared variable.

```
# Models/Post.py
from dataclasses import dataclass
import datetime

@dataclass(init=True)
class Post:
    creator_name: str
    content: str
    posting_date: datetime.datetime
```

```
# Models/__init__.py
from .Post import Post
```

Following that, there has to be a data source access mechanism with its only job being to store and write new posts, whether directly on a database or with an external service which can be communicated with HTTP methods or a messaging service, that is, Kafka, RabbitMQ, or AWS's SQS. For this example, we will assume a backend service is already built with two exposed methods, one to add posts and the other to get posts between two timestamps.

```
# API.py
from Models import Post
import datetime
class API:
    def __init__(self, config=None):
            self.config = config
    def add_post(self, post: Post):
        # POST HTTP request to backend to add the post # Returns true as
in post has been added
        return True
    def get_posts(self, start_date: datetime.datetime, end_data:
datetime.datetime):
        # GET HTTP request to backend to posts within a time period
        # Returns a list of Posts
        return [
            Post("Adam", "Python is a snake",datetime.datetime(year=2021,
month=5, day=1)
            ),
            Post("Sara", "Python is a programming
language",datetime.datetime(year=2021, month=5, day=3)
        )]
```

Once we have our API ready, we can start building an internal service to act the middleware between the visual components and the API, usually referred to as “Services” among seasoned developers.

```
# Services/AddPost.py
from API import api_instance
from Models import Post
def add_post(post: Post):
    if post is None or len(post.creator_name) == 0 or len(post.content) ==
0:
        return None
    did_add = api_instance.add_post(post)
    return did_add
```

```
# Services/GetFeed.py
from API import api_instance
import datetime
def get_feed():
    to_date = datetime.datetime.now()
    from_date = to_date - datetime.timedelta(days=1)
    posts = api_instance.get_posts(from_date, to_date)
    return posts
```

```
# Services/__init__.py
from .AddPost import add_post
```

```
from .GetFeed import get_feed
```

Even though the application is not considered complete due to missing the mostly awaited components that convert it to an interactive web application by users, it can still be used as a stand-alone service by any software as it is structured end to end to serve a clear purpose of adding and getting posts with filtering applied on passing data. Wrapping it up, we will insert our Streamlit visual components in a class-like structure for consistency with the most of the code base so far.

Furthermore, instead of using the service function right away by importing them, we think it is a great opportunity to introduce “dependency injection” which is a concept widely used in strongly typed languages such as C# and Java. This mechanism provides the capability of providing different implementations of the same function to be used by the depending class if needed, like when making test cases where making an actual post should be avoidable. Apart from this being a benefit to testable code, this coding pattern is preferred in many frameworks due to its easy readability.

```
# FeedView.py
import streamlit as st
from Models import Post
from typing import Callable

class FeedView:
    def __init__(self, get_feed_func: Callable[[], list]):
        posts = get_feed_func()
        for post in posts:
            _PostView(post)

class _PostView:
    def __init__(self, post: Post):
        st.write(f"**{post.creator_name}**: {post.content} |
        _{post.posting_date}_")
```

```
# AddPostView.py
import datetime
import streamlit as st
from Models import Post
from typing import Callable

class AddPostView:
    def __init__(self, add_post_func: Callable[[Post], bool]):
        user_name_text = st.text_input("Displayed name?")
        post_text = st.text_input("What’s in your mind?")
        clicked = st.button("Post")
        if clicked:
                    post = Post(
                        creator_name=user_name_text,
                        content=post_text,
                        posting_date=datetime.datetime.now()
                    )
```

```
                    did_add = add_post_func(post)
            if did_add:
                st.success("Post added!")
            else:
                st.error("Error adding post")
```

```
# main.py
import streamlit as st
from Views import FeedView, AddPostView
from Services import get_feed, add_post

AddPostView(add_post)
st.write("___")
FeedView(get_feed)
```

Best Practices for Folder Structuring

The example discussed in the section above can have all the files placed in the project's root folder. Even though this can give us a bug-free application, it can cause entanglements to the code reader if the application continues growing. Hence, we need to structure the files in folders and expose them as modules to be plugged in easily and professionally in other Python scripts. A folder structure such as described below groups similar files together. The *init*.py script is included in every subfolder to export the files within it as modules. And the Views/*init*.py script exposes an instance of the class instead of the class itself. Note that the underscore before the class instance name is just to rename it in this scope to indicate that this is a private property for this script.

```
# Views/__init__.py
from .AddPostView import AddPostView
from .FeedView import FeedView
```

```
# API/__init__.py
# from .API import API as _API
    api_instance = _API()
```

It is prudent to note that the imports are from a relative path to a file not absolute. We can know this due to the presence of the "dot" in front of ".AddPostView" and ".FeedView"; this means using the file with the corresponding name in the local folder of the importing file, instead of searching in the project's root folder.

```
/Path/To/SocialNetwork
    API
        __init__.py
        API.py
    Model
```

```
        __init__.py
        Post.py
    Services
        __init__.py
        AddPost.py
        GetFeed.py
    Views
        __init__.py
        AddPostView.py
        FeedView.py
    main.py
```

Data Management and Visualization

Here we will address some of the key methods used to manage big data. We will cover encoding large multimedia files and dataframes into bytes data that can then be stored more robustly on database systems or in memory. Subsequently, we will demonstrate the utility of Streamlit's in-house caching capabilities that can be used to cache data, function executions, and objects in order to drastically reduce execution time on reruns of our application. Finally, we will cover techniques to mutate dataframes and tables in our application on demand.

In the second part of this chapter, we will dive into great depths of visualizing data, with a special focus on the Plotly visualization library for Python. We will provide boilerplate scripts that can be used to render basic, statistical, time-series, geospatial charts, and animated data visualizations.

Data Management

The need to wrangle with data is inherent for most if not all web applications. While in this section we will not go into the depths of pre- and postprocessing data, we will however delve into some of the most profound ways of managing (big) data.

Processing Bytes Data

You may need to grapple with binary/ bytes data. For instance, you may need to stream multimedia content or store your files on a database system. Streamlit attends to a lot of the overhead when it comes to dealing with such data. By using the st.image, st.video, and st.audio commands, we are able to natively process not only saved files on disk but also Numpy arrays, URLs, and most importantly bytes data.

While structured databases will be covered in the next chapter it is worthy to note here that quite literally any bytes data can be saved and retrieved into a PostgreSQL table as long as the column data type is specified as bytea. This is particularly handy when working with large objects such as image, video, and audio files that need to be stored as blob (binary large object) storage. For such purposes, you will need to encode your data as follows.

Text

String and text can simply be encoded as follows for storage. Subsequently, encoded string or text may be decoded as follows: bytes_data.decode(). The default encoding used in the preceding method will be UTF-8 unless otherwise specified.

```
bytes_data = b'Hello world'
# Or alternatively
text = 'Hello world'
bytes_data = text.encode()
```

Multimedia

To convert any uploaded image, video, or audio file to bytes data, simply use the following:

```
uploaded_file = st.file_uploader('Please upload a multimedia file')
if uploaded_file is not None:
    bytes_data = uploaded_file.read()
```

You may render the bytes data as an image, video, or audio by using the following commands:

```
# Image
st.image(bytes_data)
# Video
st.video(bytes_data)
# Audio
st.audio(bytes_data)
```

Dataframes

To read and display dataframes, you may use the following method:

```
import pandas as pd
uploaded_file = st.file_uploader('Please upload a CSV file')
if uploaded_file is not None:
    df = pd.read_csv(uploaded_file)
    st.write(df)
```

Alternatively, to encode dataframes (for storage as BLOB data on databases, for instance), you may use the Python module StringIO which stores content such as CSV files on memory as a file-like object, also known as a string-based IO. These objects can then be accessed by other functions and libraries such as Pandas as shown in the following:

```
from io import StringIO
import pandas as pd
uploaded_file = st.file_uploader('Please upload a CSV file')
if uploaded_file is not None:
```

```
    stringio = StringIO(uploaded_file.getvalue().decode())
    st.write(pd.read_csv(stringio))
```

Please note that while you may use the preceding method to store Pandas dataframes as a string-based IO, you may however not store the string IO into a database. To store a Pandas dataframe, you are better off saving it as a table using the Pandas command dataframe.to_sql, which will be covered in great depth in later sections.

Caching Big Data

Given the sheer magnitude of data exposed to us, it may sometimes be necessary to cache data in volatile storage to access it with reduced latency later on. Streamlit provides us with a native method to cache data, function executions, and objects on memory using the @st.cache, @st.experimental_memo, and @st.experimental_singleton decorators, respectively. You may simply write a function that returns data or an object and precede it with the @st.cache, @st.experimental_memo, or @stexperimental_singleton decorator to make use of this feature. The first time you invoke the function, the returned data or object will be cached on memory, and for every subsequent invocation, the return will be from the cache and not the function itself, unless you alter the arguments of the function.

You can utilize Streamlit caching for yourself to benchmark the percentage of runtime saved by recalling your dataframe from the cache. There is a drastic positive effect on the runtime saved especially as you approach dataframes with over 100,000 rows of data. The effect starts to level off at 100,000,000 rows of data with around 70% runtime saved.

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

@st.cache
def dataframe(rows):
    df = pd.DataFrame(
        np.random.randn(rows, 5),
        columns=('col %d' % i for i in range(5)))
    return df

runtime = pd.DataFrame(data={'Number of rows':[10,100,100
    0,10000,100000,1000000, 10000000,100000000], 'First runtime
    (s)':None, 'Second runtime (s)':None,'Runtime saved (%)':None})

for i in range(0,len(runtime)):
    start = time.time() dataframe(runtime.loc[i]['Number of rows'])
    stop = time.time()
    runtime.loc[i,'First runtime (s)'] = stop - start
    start = time.time()
    dataframe(runtime.loc[i]['Number of rows'])
    stop = time.time()
    runtime.loc[i,'Second runtime (s)'] = stop - start
    runtime.loc[i,'Runtime saved (%)'] = 100 - int(100*
```

```
(runtime.loc[i,'Second runtime (s)']/runtime. loc[i,'First runtime (s)']))

st.write(runtime)
```

Mutating Data in Real Time

You may be required to mutate data and specifically dataframes on demand within your application. Whether it is to filter out a time-series dataset based on a given date-time range or to append data to an existing column, the need to mutate data is inherent. To that end, with Streamlit we have the option of mutating data natively or by using third-party toolkits as shown in the following sections.

Native Data Mutation

Streamlit provides an intuitive and native method to add data to existing tables with its st.add_rows command. With this method, you can easily append a dataframe to a table created previously and immediately regenerate and view any associated charts in real time using the method below:

```
import streamlit as st
import pandas as pd
import random

def random_data(n):
    y = [random.randint(1, n) for value in range(n)]
    return y

if __name__ == '__main__':
    df1 = pd.DataFrame(data={'y':[1,2]})
    col1, col2 = st.columns([1,3])
    with col1:
        table = st.table(df1)
    with col2:
        chart = st.line_chart(df1)
        n = st.number_input('Number of rows to add',0,10,1)
        if st.button('Update'):
            y = random_data(n)
            df2 = pd.DataFrame(data={'y':y}) table.add_rows(df2)
chart.add_rows(df2)
```

Advanced and Interactive Data Mutation

While Streamlit's native method of mutating data allows you to append rows to existing dataframes and charts, it does not however provide other advanced methods of mutating data, such as modifying individual cells, removing data, or filtering.

Luckily, a highly versatile and rich third-party component called streamlit-aggrid fills that gap. Built on top of the AG Grid library for JavaScript frameworks, streamlit-aggrid displays data in an interactive grid widget, allowing the user to manipulate data with filtering, sorting, selecting, updating, pivoting, aggregating,

querying, and a host of other methods. For further information on other features, you may refer to www.ag-grid.com/.

To use streamlit-aggrid on your Streamlit application, you must first initiate the widget and configure the features that you require, and then you may insert your Pandas dataframe into it using the AgGrid() command. Subsequently, the widget will be rendered, and the return value if invoked will be provided as a dict. To access the data within the widget, you must retrieve the 'data' key of the dictionary, and similarly to access the selected rows, you have to retrieve the 'selected_rows' key. The data will be returned as a table, whereas the selected rows will be returned as a list of dictionaries.

MEER INFO: zie boek pp. 112-118!

Exploring Plotly Data Visualizations

There is a plethora of data visualization libraries in Python of which many can be rendered at your fingertips in Streamlit. Whether you use one of the more native commands such as st.vega_lite_chart or resort to using the Swiss army knife command otherwise known as st.write, either way you have at your disposal the ability to visualize aggressively. Among the multitude of visualization libraries, one stands out the most from the crowd. And that is none other than Plotly, arguably one of the most versatile, interactive, and aesthetically ornate visualization stacks out there. In this section, we will showcase some of the most relevant types of charts that one may use for web development. However, the following list will in no way be exhaustive, and should you need to peruse for other charts, you may refer to https://plotly.com/python/ for a complete list.

Rendering Plotly in Streamlit

you have at your disposal two native options to display Plotly and other types of charts. Specifically, you may use the st.write command, also known as the Swiss army knife of commands, to render the chart by simply writing the Plotly chart object (hereafter invoked as fig) as follows: st.write(fig)

Alternatively, you may utilize the st.plotly_chart command that provides greater functionality when rendering Plotly charts:

```
st.plotly_chart(fig, use_container_width=True, sharing='streamlit')
```

You may use the st.plotly_chart command with the additional arguments of use_container_width to specify whether the chart width should be restricted to the encapsulating column width or not, and you may use the sharing argument to specify whether the chart should be rendered in Plotly's offline mode ('streamlit') or whether to send the chart to Plotly's chart studio ('private', 'secret', or 'public') for which you will require a Plotly account; for further information, please refer to plotly.com/ chart-studio/. In addition, you may supply any other keyword argument that is supported by Plotly's plot() function.

In this section, we will cover Plotly line, scatter, bar, and pie charts. Before we proceed any further, we will initially import all of the necessary libraries for this section as listed in the following:

```
import streamlit as st
import numpy as np
```

```
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
```

Basic Charts

For the sake of uniformity, we will be using the same randomly generated Pandas dataframe (shown as follows) as our dataset to generate each of the charts:

```
data = np.random.randint(0, 10, size=(40,2))
df = pd.DataFrame(data, columns=['Column 1', 'Column 2'])
```

Line Charts

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=df.index, y=df['Column 1'],
                    mode='lines',
                    name='Column 1'))
fig.add_trace(go.Scatter(x=df.index, y=df['Column 2'],
                    mode='lines',
                    name='Column 2'))
```

Scatter Chart

```
fig = go.Figure(data=go.Scatter( y = df['Column 1'], mode='markers',
    marker=dict(
        size=10,
        color=df['Column 2'], # Set color equal to a variable
colorscale='Viridis', # Select colorscale showscale=True
    )
))
```

Bar Chart

```
fig = go.Figure(data=[
    go.Bar(name='Column 1', x=df.index, y=df['Column 1']),
    go.Bar(name='Column 2', x=df.index, y=df['Column 2'])
])
```

Pie Chart

```
fig = px.pie(df, values=df.sum(), names=df.columns)
```

Chart Layout

To update the properties and layout of the chart, you may use the update_ layout declaration as shown in the following:

```
fig = go.Figure(data=[
    go.Bar(name='Column 1', x=df.index, y=df['Column 1']),
    go.Bar(name='Column 2', x=df.index, y=df['Column 2'])
])
fig.update_layout(
    title='Column 1 vs. Index',
    xaxis_title='Index',
    yaxis_title='Value',
    legend_title='Columns',
    font=dict(
        family='Arial',
        size=10,
        color='black'
    )
)
```

Statistical Charts

In this section, we will generate a Plotly histogram and box plot. The following randomly generated dataframe will be used for both charts:

```
data = np.random.randn(40, 2)
df = pd.DataFrame(data, columns=['Column 1', 'Column 2'])
```

Histogram

```
fig = go.Figure()
fig.add_trace(go.Histogram(name='Column 1', x=df['Column 1']))
fig.add_trace(go.Histogram(name='Column 2', x=df['Column 2']))
fig.update_layout(barmode='overlay')
fig.update_traces(opacity=0.75)
```

Box Plot

```
fig = go.Figure()
fig.add_trace(go.Box(
    y=df['Column 1'],
    name='Column 1',
    boxmean='sd' # Display mean, median and standard deviation ))
fig.add_trace(go.Box(
    y=df['Column 2'],
    name='Column 2',
    boxmean='sd' # Display mean, median and standard deviation
))
```

Time-Series Charts

Time-series charts can be generated using the same line chart function used in Section 4.2; the only difference is that the provided index must be in a date-time format. You can use the following function to create a Pandas dataframe with randomly generated values indexed between a range of specified dates:

```
data = np.random.randn(40, 2)
df = pd.DataFrame(data, columns=['Column 1', 'Column 2'])
df.index = pd.date_range(start='1/1/2018', end='2/9/2018',
freq='D')
```

Subsequently, the line chart function can be invoked as follows:

```
fig = px.line(df, x=df.index, y=df.columns)
```

Geospatial Charts

Depending on your application, you may need to render interactive maps with geospatial data. Luckily, Plotly offers geospatial charts with a wealth of features and attributes. In this section, we will cover one type of such charts, namely, the choropleth map, using a dataset of world GDP per capita from 1990 to 2020 [22]:

```
df = pd.read_csv('gdp-per-capita-worldbank.csv').sort_
values(by='Year', ascending=False)
fig = px.choropleth(df, locations=df['Code'],
    color=df['GDP per capita, PPP (constant 2017
    international $)'],
    hover_name=df['Entity'])
```

Animated Visualizations

With Plotly, you have the option of incorporating simple animations into your charts. This can be particularly handy when displaying a time-varying value in a time-series dataset. However, you are not restricted to time- series data and may indeed animate other types of numeric data should you wish to do so. In this section, we will animate the same previously used dataset of world GDP per capita from 1990 to 2020 [22] using both an animated bubble map and bar chart as shown in the following.

Animated Bubble Map

```
df = pd.read_csv('gdp-per-capita-worldbank.csv').sort_
values(by=['Year', 'Entity'])
fig = px.scatter_geo(df, locations=df['Code'],
    color=df['GDP per capita, PPP (constant 2017
international $)'],
    hover_name=df['Entity'],
    size=df['GDP per capita, PPP (constant 2017
international $)'],
    animation_frame=df['Year'])
```

Animated Bar Chart

```
df = pd.read_csv('gdp-per-capita-worldbank.csv').sort_
values(by=['Year', 'Entity'])
df = df[df['GDP per capita, PPP (constant 2017 international
$)'] > 50000]
fig = px.bar(df, x=df['Entity'],
    y=df['GDP per capita, PPP (constant 2017 international $)'],
animation_frame=df['Year'])
```

Database integration

Boek behandelt PostgreSQL en MongoDB. Ik laat Mongo voorlopig achterwege. Gebruikt pgAdmin4 als GUI.

Connecting a PostgreSQL Database to Streamlit

Credentials, including the username and password, can be written as environment variables in a .env file, or in a secrets.yaml which will then be read and parsed by Streamlit. Ideally, however, they should be written to the secrets.toml file in a folder called .streamlit which is recommended.

.streamlit/secrets.toml:

```
[db_postrgres]
host = "127.0.0.1"
port = "5432"
user = "postgres"
password = "admin"
dbname = "CompanyData"
```

To interface Python with PostgreSQL, we need to use a capable library to do so. For that example, we will use psycopg2, but other libraries such as sqlalchemy can do the job, and it will be introduced in later chapters.

Streamlit reruns the python script upon user actions. So a new database connection will be established with every rerun. To avoid that unnecessary call, we can cache the first established connection. Streamlit allows caching function calls out of the box, by using its native function decorator st.cache. It can also accept some other parameters, for instance, an expiration date of the cache, which will cause the next function call to reexecute the function body if the cache is invalidated by then. After the new connection is established, we will need a cursor to use for querying SQL commands, also known as SQL queries. The cursor needs to be disposed after the query finishes; otherwise, it can retain in memory, and with every new query, the memory can be bloated and cause memory leaks, which is every software developer's nightmare. The developer can choose to close it manually or use a context manager which will close it once its usage scope is exited by the interpreter.

```
import streamlit as st
import psycopg2

@st.cache(allow_output_mutation=True,
    hash_funcs={"_thread.RLock": lambda _: None})
def init_connection():
    return psycopg2.connect(**st.secrets["db_postrgres"])

conn = init_connection()

def run_query(query_str):
    cur = conn.cursor()
    cur.execute(query_str)
    data = cur.fetchall()
    cur.close()
    return data

def run_query_with_context_manager(query_str):
    with conn.cursor() as cur:
        cur.execute(query_str)
        return cur.fetchall()

query = st.text_input("Query")

c1, c2 = st.columns(2)

output = None

with c1:
    if st.button("Run with context manager"):
        output = run_query_with_context_manager(query)
with c2:
    if st.button("Run without context manager"):
        output = run_query(query)
```

```
st.write(output)
```

Displaying Tables in Streamlit

After querying data from the database, we can display it in text format, or we can use more visually entertaining tools from Streamlit, which will require a modification of the representation of the data.

Among data scientists and developers, it is usually known to parse structured data, whether it is sensor values, identification information, or any repeating data with a structure in the form of a Pandas dataframe. Dataframes are generally Numpy arrays with extra capability such as storing column values and SQL-like querying techniques. That being said, it also shares the same fast vectorization capabilities with normal Numpy arrays, which is essentially a parallelized way to do mathematical computations on an array as a whole instead of doing it one by one.

Streamlit allows printing dataframes right away from a single command in two different st.table displays a noninteractive representation of the dataframe as shown in Figure 5-6. And Figure 5-7 displays st. dataframe, rendering an interactive representation of the dataframe, where the user can sort any column just by clicking it. As a trade-off, this makes the web application slower as more CPU and/or memory usage is required, since the complexity of the sorting algorithm grows- in an O(n*log(n)) manner

```
import streamlit as st
import pandas as pd
df = pd.DataFrame([["Adam", "01/01/1990", 2],
                   ["Sara", "01/01/1980", 1],
                   ["Bob", "01/01/1970", 1],
                   ["Alice", "01/01/2000", 3]
                   ], columns=["Name", "DOB", "Paygrade ID"])
st.table(df)
st.dataframe(df)
```

Leveraging Backend servers

Here we will introduce a more sophisticated and scalable way of designing web applications. Specifically, this chapter will present the process to offload the overhead associated with managing databases from the Streamlit server and onto an independent backend server as it should be in a full-stack environment. To that end, this chapter will walk the developer through the entire process of provisioning a backend server in Python that acts as the middleman between the database and the frontend. And finally, the developer will be acquainted with the workings of a highly modular, versatile, and secure architecture with additional security layers between the application and database.

The Need for Backend Servers

As part of building a scalable and robust Streamlit application, some of the tasks which are executed within the Streamlit application, are better off being executed in an isolated system that is easy to communicate with. Such environment is referred to as a backend, and it is responsible for managing authentication,

authorization, databases, and other gateway connections – all while including managing core business logic which shouldn't be done on the frontend side, that is, Streamlit.

Even though Streamlit is a server-side web framework, it's highly recommended to still isolate from other system aspects for modularity as security. As if everything from authentication to database management were run from Streamlit, this can possess multiple security threats, from XSS to SSRF and possibly RCE if not engineered correctly.

Not claiming that a backend-frontend architecture is invulnerable, but having this architecture adds another layer of protection which is made with the API. That needs to be broken or bypassed by malicious actors to reach other protected system aspects. And with modern API designing methodologies, it is attack proof, as it pipelines every request through the same routing mechanism which decreases human error in making a vulnerable code.

Frontend-Backend Communication

The backend is usually referred to as the server, and the frontend is always known to be the client. Usually, the client triggers a resource or information request upon the user's actions. The request is eventually followed by a response by the server including the solicited data. The request-response communication is how HTTP protocol works. HTTP generally has two major components, headers and a body. The request or response headers include information about the request itself and payload carried in the body. Some of these information include, but are not limited to, cookies, request identifiers, keys, tokens, data type of the body, host or IP address of the server, and data encoding or compressing mechanisms. For the sake of this book's scope, we will focus on the general aspects of keys, tokens, cookies, and the body's content type. Since backends are mainly responsible for sending and receiving information, a good and the most widely used format is JSON.

Provisioning a Backend Server

To stick with the Python theme of the book, we will build a Pythonic backend server. Looking to the potential options for such requirements, we can mainly find Flask and Django good for the job. Both can be used as frontend application servers as both can serve HTML for browsers to render, but Django is built with that task in mind due to the presence of a web template engine called Jinja. But Flask is flexible and configurable to the developer's need, plus it being lighter in weight. To get started, install it with Pip.

API Building

A backend will run a single or multiple methods or functions upon a request to the server. Those methods and functions depend on the URL and can be configured to also take the headers in consideration. A simple example for that is like the following listing, which will serve the user the page if http:///server_status was requested, and 404 Not Found otherwise.

```
from flask import Flask

app = Flask(__name__)

@app.route('/server_status')
def welcome_controller():
    return {
```

```
        "message": "Welcome to your Flask Server",
        "status": "up",
        "random": 1 + 1
    }
    app.run()
```

To trigger a specific function call when a route is requested, a function decorator needs to be added before it with the route name. It is not necessary to have static routes, they can also be dynamic, by specifying a specific string format which will be mapped by Flask to that function. For instance, /text/1 and /text/3 map to /text/. Calling a specific method depending on the HTTP method can also be configured through the same function decorator by adding an extra parameter as follows: @app.route('/text/', methods=['GET', 'PUT'])

We will use SQLAlchemy instead of psycopg2, to make use of the ORM which represents SQL commands with Python class objects. To start, we would first need to represent our tables as classes and a Base class which the other two classes will inherit SQL properties from. These properties include parameterization of SQL queries which prevent SQLI. The classes in the following listings shall point to already existing tables in the database, and the *tablename* property shall be the table name.

```
# Base.py
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

```
# PayGrades.py
from sqlalchemy import Column, Integer, String
from .Base import Base

class PayGrades(Base):
    __tablename__ = 'paygrades'
    id = Column(Integer, primary_key=True)
    base_salary = Column(String)
    reimbursement = Column(String, default=True)
    bonuses = Column(String)

    def to_dict(self):
        return {
            "id": self.id,
            "base_salary": self.base_salary,
            "reimbursement": self.reimbursement,
            "bonuses": self.bonuses
        }
```

```
# Employees.py
from sqlalchemy import Column, Integer, String
from .Base import Base

class Employees(Base):
```

```
    __tablename__ = 'persons'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    date_of_birth = Column(String, default=True)
    paygrade_id = Column(Integer, unique=True, index=True)

    def to_dict(self):
        return {
            "id": self.id,
            "name": self.name,
            "date_of_birth": self.date_of_birth,
            "paygrade_id": self.paygrade_id
        }
```

Subsequently we can support adding and getting employee data over HTTP requests using Flask. This backend server includes two routes, the first is to query all employees using the database connection made with SQLAlchemy, and the second is to insert or add a new employee to the employees table with the user-supplied properties sent in the HTTP body as a JSON document.

```
# main.py
from flask import Flask, request
from DataBase import Connection
from DataBase import Employees

app = Flask(__name__)

@app.route('/employees')
def get_all_employees():
    with connection.use_session() as session:
        employees = session.query(Employees).all()
        employees = [employee.to_dict() for employee in employees]
        return {"data": employees}

@app.route('/employee', methods=["POST"])
def add_employee():
    body = request.json
    with connection.use_session() as session:
        session.add(Employees(**body))
        session.commit()
    return {"message": "New employee added successfully"}

connection = Connection("postgresql://
postgres:admin@127.0.0.1:5432/CompanyData")
app.run()
```

API Testing

In this example, the user will be a human interfacing with the backend using an API testing platform such as Postman. In a later section, Streamlit will be the user of this server directly without the need for an API

testing platform.

Wrapping this section up, we need to point out that we didn't set any headers, including the content type for the "POST/employee" route which uses a JSON payload, because Postman took care of that behind the scenes. And Flask took care of also adding the JSON content type to the response as Python's lists and dictionaries are easily parseable to JSON as mentioned before.

Multithreading and Multiprocessing Requests

Once an application scales or when one of its initial requirements is to do heavy multiple independent computations of processes, a lot of power is wasted whether on the backend's Flask side or Streamlit's side. This is due to not harnessing the power of the CPUs which gets even stronger over time. Modern CPU strength is not due to them being quicker with faster clock cycles, but with the more cores it packs. Streamlit and Flask are single-threaded, single-processed applications by default. And to speed them up, we can try either or both of multiprocessing and multithreading, which introduces true parallelization to our application. The usage of both approaches shall be controlled by the developer to run a function a multiple number of times which then will be executed in parallel by the CPU.

```
# streamlit_main.py
import streamlit as st
from multiprocessing import Pool, cpu_count
import threading
import time

def func(iterations, id):
    i= 0
    for i in range(iterations):
        i += 1
    print("Finished job id =", id)
    if __name__ == "__main__":
        pool = Pool(cpu_count())
        st.title("Speed You Code!")
        jobs_count = 5
        iterations = 10 ** 3
        c1, c2 = st.columns(2)
        with c1:
            if st.button("multiprocess"):
                inputs = [(iterations, i) for i in range(jobs_count)]
                t11 = time.time() pool.starmap(func, inputs)
                t21 = time.time()
                st.write(f"Finished after {t21 - t11} seconds")
        with c2:
            if st.button("multithread"):
                threads = [threading.Thread(target=func, args=(iterations,
i)) for i in range(10)]
                t12 = time.time()
                for thread in threads:
                    thread.start()
                for thread in threads:
                    thread.join()
```

```
                t22 = time.time()
                st.write(f"Finished after {t22 - t12} seconds")
```

Notice that in this listing, the first code to be executed is after line 13 which is required for the whole example to work error-free due to Streamlit knowing that this code block shall be executed only once, which means the processing pool won't be initialized again in reruns. Similar precaution shall be taken even in Flask applications. The main difference between multiprocessing and multithreading is that multithreading reuses the already existing memory space and spawns new threads within the same process, which is a faster operation than spawning totally new processes which adds an overhead of CPU context switching. Moreover, every new process in the multiprocessing pool requires an entirely new memory space. In addition, each processes' inputs need to be copied or cloned, thereby introducing a memory-greedy application. It may seem that multithreading is the better option, however, that is not true due to the fact that multiprocessing is more CPU efficient when executing hefty tasks as the CPU scheduler allocates more time to those processes.

Connecting Streamlit to a Backend Server

Once we have an optimized the backend server with multiprocessing and/ or multithreading, we are ready to connect our Streamlit application to it. For this, we will need to use an HTTP client library to communicate with the backend API.

```
# streamlit_api.py
import streamlit as st import requests
import datetime
url = "http://127.0.0.1:5000"

def add_employee(name, dob, paygrade):
    data = {
        "name": name,
        "date_of_birth": dob,
        "paygrade_id": paygrade
    }
    response = requests.post(url + "/employee", json=data)
    if response.status_code == 200:
        return True
    return False

def get_employees():
    response = requests.get(url + "/employees")
    return response.json()['data']

form = st.form("new_employee")
name = form.text_input("Name")
dob = str(form.date_input("DOB", min_value=datetime. datetime(year=1920,
day=1, month=1)))
paygrade = form.number_input("paygrade", step=1)

if form.form_submit_button("Add new Employee"):
    if add_employee(name, dob, paygrade):
        st.success("Employee Added")
```

```
    else:
        st.error("Error adding employee")

st.write("___")
employees = get_employees()
st.table(employees)
```

Implementing session state

It is vital to establish session-specific data that can be utilized to deliver a more enhanced experience to the user. Specifically, the application will need to preserve the user's data and entries using what is referred to as session states. These states can be set and accessed on demand whenever necessary, and they will persist whenever the user triggers a rerun of the Streamlit application or navigates from one page to another. In addition, we will establish the means to store state across multiple sessions with the use of cookies that can store data on the user's browser to be accessed when they restart the associated Streamlit application.

Implementing Session State Natively

Since Streamlit version 0.84, a native way to store and manage session- specific data including variables, widgets, text, images, and objects has been introduced. The values of session states are stored in a dictionary format, where every value is assigned to a unique key to be indexed with. With session state, users can receive an enhanced and more personalized experience by being able to access variables or entries that were made previously on other pages within the application. For instance, users can enter their username and password once and continue to navigate through the application without being prompted to reenter their credentials again until they log out.

Note that the first two key-value entries (KeyInput1 and KeyInput2) are present even though they haven't been created by the user. Those keys are present to store the state of the user-modified components, which are the defined text input components. This means that the developer also has the capability of modifying the values of any component as long as it has a unique key set with its definition. Another caveat is that each session state must be invoked before it can be read; otherwise, you will be presented with an error. To avoid this, ensure that you always initialize the state with a null or fault value.

```
def get_state_value(key):
    return st.session_state.get(key)

def set_state_value(key, value):
    st.session_state[key] = value

c1, c2, c3 = st.columns(3)

with c1:
    st.subheader("All")
    st.write(st.session_state)
with c2:
    st.subheader("Set Key")
    key = st.text_input("Key", key="KeyInput1")
    value = st.text_input("Value")
```

```
    if st.button("Set"):
        st.session_state[key] = value
        st.success("Success")
with c3:
    st.subheader("Get Key")
    key = st.text_input("Key", key="KeyInput2")
    if st.button("Get"):
        st.write(st.session_state.get(key))
```

Building an Application with Session State

To demonstrate the utility of session states, in the following example we will create a trivial multipage application where the user can use states to store the key of the selected page, an uploaded dataframe, and the value of a slider widget. In our main page we first initialize the state of our page selection, and then we use buttons to change the state to the key of the requested page. Subsequently, the associated function of the selected page is invoked directly from the session state to render the page.

In Page One of the application we will use session states to store an uploaded dataframe and the value of a slider that is used to filter the number of rows shown in the dataframe. The user can navigate back and forth between the pages and still be able to access a previously uploaded dataframe with the same number of rows set on the slider.

```
# main_page
from page_1 import func_page_1
def main():
# Initializing session state for page selection if 'page_state' not in
st.session_state:
        st.session_state['page_state'] = 'Main Page'
    # Writing page selection to session state
st.sidebar.subheader('Page selection') if st.sidebar.button('Main Page'):
st.session_state['page_state'] = 'Main Page' if st.sidebar.button('Page
One'):
        st.session_state['page_state'] = 'Page 1'
pages_main = {
        'Main Page': main_page,
        'Page 1': run_page_1
}
    # Run selected page
    pages_main[st.session_state['page_state']]()
def main_page(): st.title('Main Page')
def run_page_1(): func_page_1()
if __name__ == '__main__': main()
```

```
# page_1.py
import pandas as pd def func_page_1():
    st.title('Page One')
    # Initializing session states for dataframe and slider
if 'df' not in st.session_state: st.session_state['df'] = None
```

```
if 'rows' not in st.session_state: st.session_state['rows'] = None
file = st.file_uploader('Upload file')
    # Writing dataframe to session state
if file is not None:
df = pd.read_csv(file) st.session_state['df'] = df
if st.session_state['df'] is not None:
# Creating slider widget with default value from session state
rows = st.slider('Rows to display',value=st.session_ state['rows'],
min_value=1,max_value=len(st.session_ state['df']))
    # Writing slider value to session state
    st.session_state['rows'] = rows
    # Rendering dataframe from session state
    st.write(st.session_state['df'].iloc[:st.session_
    state['rows']])
```

Introducing Session IDs

Session IDs are unique identifiers of a new connection to Streamlit's HTML serving WebSocket. A new WebSocket connection is established if a new browser page is opened even if another connection is established. But both are treated independently by the server. These WebSocket connections are used to transfer data from the server to the client's browser and vice versa. Those unique identifiers can be used to provide the end user with a personalized experience. And to do so, the server needs to map users' progress and updates to their corresponding identifiers. This mapping can be done natively in an effortless way using Streamlit's native API.

```
from streamlit.scriptrunner.script_run_context import get_script_run_ctx
from streamlit.server.server import Server

all_sessions = Server.get_current()._session_info_by_id
session_id = get_script_run_ctx().session_id
session_number = list(all_sessions.keys()). index(session_id) + 1

st.title("Session ID #"+str(session_number))
st.header("Id of this session is: " + session_id) st.subheader("All
sessions ("+str(len(all_sessions))+") :") st.write(all_sessions)
```

Implementing Session State Persistently

Streamlit's native method to store session state will more than suffice for most if not all applications. However, it may be necessary to store session state on an accessible database to retrieve later on or, in other words, store session state persistently. This can be especially useful for generating user insights and traffic metrics for your application.

For the example shown below, we will be making use of PostgreSQL to store and retrieve a variable and a dataframe while our user navigates through the pages of the application. Specifically, the entered name and uploaded dataset will be written to the database in tables named with the unique session ID and read/rendered with the previous value each time the user refers back to Page Two, as long as the user is still within the same session. Once the application is refreshed and a new session ID generated, the user will no

longer have access to the variables; however, the database administrator can access the historical states in the database should they need to do so. Given that Streamlit reruns the script with every user interaction, without session state both the name and dataframe would be reset each time the script is run; however, with this implementation, we can save the data persistently and access it in the associated database on demand.

This method is scalable and can be extended to as many variables, and even files (in the form of byte strings) if required, using the four read and write functions shown as follows:

```
# Read session variables
read_state('column name', database_engine, session_ID)
# Read session dataframes
read_state_df(database_engine, session_ID)
# Write and overwrite session variables
write_state('column name', value, database_engine, session_ID)
# Write and overwrite session dataframes
write_state_df(dataframe, database_engine, session_ID)
```

```
import pandas as pd
import psycopg2
from sqlalchemy import create_engine
from streamlit.scriptrunner.script_run_context import get_script_run_ctx

def get_session_id():
    session_id = get_script_run_ctx().session_id
    session_id = session_id.replace('-','_')
    session_id = '_id_' + session_id
    return session_id

def read_state(column,engine,session_id):
    state_var = engine.execute("SELECT %s FROM %s" % (column,session_id))
    state_var = state_var.first()[0]
    return state_var

def read_state_df(engine,session_id):
    try:
        return pd.read_sql_table(session_id,engine)
    except:
        return pd.DataFrame([])

def write_state(column,value,engine,session_id):
    engine.execute("UPDATE %s SET %s='%s'" % (session_id, column,value))

def write_state_df(df,engine,session_id):
    df.to_sql('%s' % (session_id),engine,index=False, if_exists='replace')

def page_one(engine,session_id):
    st.title('Hello world')

def page_two(engine,session_id):
```

```
    name = st.text_input('Name',read_state('name',engine,session_id))
    write_state('name',name,engine,session_id)

    file = st.file_uploader('Upload dataset')

    if file is not None:
        df = pd.read_csv(file)
        write_state_df(df,engine,session_id + '_df')

    if read_state_df(engine,session_id + '_df').empty is False:
        df = read_state_df(engine,session_id + '_df')
        df = df[df['Name'] == name]
        st.write(df)

if __name__ == '__main__':
# Creating PostgreSQL engine
    engine = create_engine('postgresql://<username>:<password> @localhost:
<port>' '/<database')
    # Getting session ID
    session_id = get_session_id()
    # Creating session state tables
    engine.execute("CREATE TABLE IF NOT EXISTS %s (name text)" %
(session_id))
    len_table = engine.execute("SELECT COUNT(*) FROM %s" % (session_id));
    len_table = len_table.first()[0]
    if len_table == 0:
        engine.execute("INSERT INTO %s (name) VALUES (“)" % (session_id));
    # Creating page selector
    pages = {
        'Page One': page_one,
        'Page Two': page_two
        }
    page_selection = st.sidebar.selectbox('Select page', ['Page One','Page
Two'])
    pages[page_selection](engine,session_id)
```

User Insights

The simplest allows you to read the timestamp each time the user engages with a subsection of your code, such as clicking a button or uploading a dataset and record it in a PostgreSQL database. Similarly, the number of rows of the uploaded dataset can also be recorded. Each insight is stored in a separate column in a table whose primary key is the unique session ID. Once the application has been restarted, a new row with a different session ID will be created. By inserting the following update function, you can record any value at any step in your program: update_row(column,new_value,session_id,mutable,engine)

Please note that insights can be overwritten multiple times by setting the mutable argument to True or left as False if you want to record a value only the first time it was generated.

```
from streamlit.scriptrunner.script_run_context import get_ script_run_ctx
from datetime import datetime
import pandas as pd
```

```
import psycopg2
from sqlalchemy import create_engine

def get_session_id():
    session_id = get_script_run_ctx().session_id
    session_id = session_id.replace('-','_')
    session_id = '_id_' + session_id
    return session_id

def insert_row(session_id,engine):
    if engine.execute("SELECT session_id FROM user_insights WHERE
session_id = '%s'"
    % (session_id)).fetchone() is None:
        engine.execute("""INSERT INTO user_insights (session_id) VALUES
('%s')""" % (session_id))

def update_row(column,new_value,session_id,mutable,engine):
    if mutable:
        engine.execute("UPDATE user_insights SET %s = '%s' WHERE
session_id = '%s'"
        % (column,new_value,session_id))
    elif engine.execute("SELECT %s FROM user_insights WHERE session_id =
'%s'"
    % (column,session_id)).first()[0] is None:
        engine.execute("UPDATE user_insights SET %s = '%s' WHERE
session_id = '%s'"
        % (column,new_value,session_id))

if __name__ == '__main__':
    # Creating PostgreSQL engine
    engine = create_engine('postgresql://<username>:<password>@
localhost:''<port>/<database>')
    # Getting session ID
    session_id = get_session_id()
    # Creating session state tables
    engine.execute("'CREATE TABLE IF NOT EXISTS user_insights (session_id
text, step_1 text, step_2 text, no_rows bigint)"')
    insert_row(session_id,engine)
    st.title('Hello world')
    st.subheader('Step 1')
    if st.button('Click'):
        st.write('Some content')
        update_row('step_1',datetime.now().strftime("%H:%M:%S
        %d/%m/%Y"),session_id, True,engine)
    st.subheader('Step 2')
    file = st.file_uploader('Upload data')
    if file is not None:
        df = pd.read_csv(file)
        st.write(df)
        update_row('step_2',datetime.now().strftime("%H:%M:%S
%d/%m/%Y"),session_id, False,engine)
update_row('no_rows',len(df),session_id,True,engine)
```

Visualizing User Insights

Now that we have established how to read insights from a Streamlit application and record them on a PostgreSQL database, the next step will be to visualize the data on demand. Initially, to extract the data we can run Listing 7-7 to import the insights table into a Pandas dataframe and save it locally on disk as an Excel spreadsheet if you wish to render your own customized charts. Otherwise, we can visualize the data with Listing 7-8, where we import the Excel spreadsheet generated earlier into a Pandas dataframe, convert the timestamps into hourly and daily values, sum the number of rows that are within the same hour or day, and finally visualize them with Plotly charts as shown in Figure 7-9. In addition, we can filter the data by using a st.selectbox to select the column from the insights table to visualize.

```
# read_user_insights.py
import pandas as pd
import psycopg2
from sqlalchemy import create_engine

def read_data(name,engine):
    try:
        return pd.read_sql_table(name,engine)
    except:
        return pd.DataFrame([])

if __name__ == '__main__':
    # Creating PostgreSQL engine
    engine = create_engine('postgresql://<username>:<password>@
localhost:'
    '<port>/<database>')
    df = read_data('user_insights',engine)
    df.to_excel('C:/Users/.../user_insights.xlsx',index=False)
```

```
# plot_user_insights.py
import streamlit as st
import pandas as pd
import plotly.express as px

st.set_page_config(layout='wide')
st.title('User Insights')
df = pd.read_excel('C:/Users/.../user_insights.xlsx')
column_selection = st.selectbox('Select column',df.columns[1:-2])
df = df[column_selection]
df = pd.to_datetime(df,format='%H:%M:%S %d/%m/%Y')
df_1h = df.copy()
df_1d = df.copy()
col1, col2 = st.columns(2)
with col1:
    st.subheader('Hourly chart')
    df_1h = df_1h.dt.strftime('%Y-%m-%d %I%p')
    df_1h = pd.DataFrame(df_1h.value_counts())
    df_1h.index = pd.DatetimeIndex(df_1h.index)
```

```
    df_1h = df_1h.sort_index()
    fig = px.bar(df_1h, x=df_1h.index, y=df_1h[column_selection])
    st.write(fig)

with col2:
    st.subheader('Daily chart')
    df_1d = df_1d.dt.strftime('%Y-%m-%d')
    df_1d = pd.DataFrame(df_1d.value_counts())
    df_1d.index = pd.DatetimeIndex(df_1d.index)
    df_1d = df_1d.sort_index()
    fig = px.line(df_1d, x=df_1d.index, y=df_1d[column_selection])
    st.write(fig)
```

Cookie Management

We have discussed how to store and manage data within a session using native and workaround approaches. However, what might be missing is the ability to manage data between sessions. For instance, storing a counter of how many times a button has been clicked or even more usefully not prompting the user to log in each and every time they open a new session. To do so, we require the utility of cookies. Cookies can be used to track a user's actions across many websites or store their personal information such as authentication tokens. Cookies are stored and managed on the user's end, specifically, in their browser. This means the server doesn't know about its content by default. In order to check out the cookies on any web application, we can simply open developer tools from the browser and head to the console tab. Then type 'document.cookie', and then the cookies will be displayed.

In a typical Streamlit application, more unknown cookies will appear apart from the one in Figure 7-10; those might be for advertisement tracking or other purposes. Which might require the developer to remove them depending on the cookie policy they adopt. Or in other cases, the developer might want to add other cookies to enhance the application experience. Both actions need a way to manage cookies on any web app. To manipulate cookies from a Streamlit application, we need to use a third-party module or library to make this happen. For this example, we will use Extra-Streamlit-Components which can be installed with pip install extra-streamlit-components and an import naming convention as stx, where the X denotes the extra capabilities that can be

provided on a vanilla Streamlit application. Within this library, there is a module called Cookie Manager which will be our tool for such task. Listing 7-9 builds a simple Streamlit application with the capability of setting, getting, and deleting cookies. The controls are even customizable based on the developer needs. For instance, an expiration date can be set to any new cookie added which will autodelete itself after the set date is reached. Figures 7-11 and 7-12 show adding and getting an example authentication token, respectively.

```
# cookie_management.py
import extra_streamlit_components as stx
st.title("Cookie Management Demo")
st.subheader("_Featuring Cookie Manager from Extra-Streamlit-Components_")
cookie_manager = stx.CookieManager()
st.subheader("All Cookies:")
cookies = cookie_manager.get_all()
```

```
st.write(cookies)

c1, c2, c3 = st.columns(3)

with c1:
    st.subheader("Get Cookie:")
    cookie = st.text_input("Cookie", key="0")
    clicked = st.button("Get")
    if clicked:
        value = cookie_manager.get(cookie)
        st.write(value)
with c2:
    st.subheader("Set Cookie:")
    cookie = st.text_input("Cookie", key="1")
    val = st.text_input("Value")
    if st.button("Add"):
        cookie_manager.set(cookie, val)
with c3:
    st.subheader("Delete Cookie:")
    cookie = st.text_input("Cookie", key="2")
    if st.button("Delete"):
        cookie_manager.delete(cookie)
```

Please note that the All Cookies section in Figures 7-11 and 7-12 is displayed in a well-structured JSON format but has some redacted cookies for privacy concern purposes. Good to note that there is not a visual aspect to this Streamlit application from the newly introduced module as it is classified as a service, hence the name Cookie Manager. However, this doesn't mean all other Streamlit-compatible libraries share the same behavior, as they can contain a visual aspect to them.

Authentication and Application security

Once all users requesting access to the application are authenticated, we can guarantee a secure user experience whereby private data is safe and any unwelcome or malicious requests are formidably denied. In this chapter, we will learn how to establish user accounts, verify user actions, and implement other housekeeping measures that are expected of any well-versed software engineer.

Developing User Accounts

In this chapter, we will build on the prevous example by introducing HR admins who get to see and add employees and their pay grades. Assume there are admins responsible to do those actions, and the company keeps changing and assigning new admins. In this case, we need our application to support making more admin accounts and authorize them.

Now those actions need authorized people to execute them, so we mainly need three main additions: adding an admin table in our database, allowing admin account creations, and authorizing users with admin accounts to use the rest of the service.

Hashing

To add a new table to the database, we will need to follow a step similar to what was done previously. Notice that we are storing mainly two pieces of information per admin, username and password hash. The hash or a nonguessable representation of the password is stored instead of the password itself. By doing this, we are protecting our users' privacy and credentials in case of a data breach. As if this happens, the attacker will have to spend billions of years to brute-force all hashes to find a single user's actual password. So what hashing mainly does is a one-way transformation of data that is not reversible.

After creating the new table, we will need to create a corresponding Python class to make an ORM for SQLAlchemy as shown in Listing 8-1. Hashing a password can be done in multiple ways, a few of which can be MD5, SHA256, SHA512, and others. However, the most commonly used algorithm by modern systems is Bcrypt. Infact Bcrypt is used by default to protect users' passwords in Linux environments. Before explaining how Bcrypt works, we first need to know what methods are used to make a hash more secure.

Salting

Including extra bytes in the password, also known as adding a salt, gives a totally different hash. This helps in password reusability cases by users across different websites, and one has been breached. By doing so, attackers won't know whether the same user uses the same password among multiple domains. And it will give them a harder time to break the hash. However, this trick will not be useful if the attacker knows the hashing salt and how it is applied. Hence, Bcrypt falls under the spotlight by introducing a cryptographic way to store randomly generated salts within the hash. This makes it possible to check if a Bcrypt hash is generated from a plain text, using an abstracted Bcrypt library's function.

```
# Flask/DataBase/Models/Admins.py
from sqlalchemy import Column, Integer, String
from .Base import Base

class Admins(Base):
    __tablename__ = 'admins'
    id = Column(Integer, primary_key=True)
    username = Column(String)
    password_hash = Column(String, default=True)

    def to_dict(self):
        return {
            "id": self.id,
            "username": self.username,
            "password_hash": self.password_hash
        }
```

```
# Flask/DataBase/Services/HashingService.py
import bcrypt

class HashingService:
    def __init__(self, bcrypt_gen_salt: int = 12):
        self.gen_salt = bcrypt_gen_salt

    def hash_bcrypt(self, plain_text: bytes) -> bytes:
```

```
        return bcrypt.hashpw(plain_text, bcrypt.gensalt(self. gen_salt))

    def check_bcrypt(self, plain_text: bytes, hashed_password: bytes) ->
bool:
        try:
            return bcrypt.checkpw(plain_text, hashed_password)
        except:
            return False
```

Verifying User Credentials

Now after we have the needed service and storage support to manage passwords, we can proceed with the backend refactoring to support authentication for every route. This means we need to intercept every request to the server and decide whether it is authenticated or not. In other words, we need to have an independent piece of software to sit between the client's request and the access controller; this is usually referred to as middleware among backend developers. The authentication process has to be checking for a specific identifier of the request that the server can trust; such identifier is referred to as "authentication token," or "token" for short. This token shall be issued by the server, and it shall be verified.

Tokens are mainly either of the two: custom session IDs or JWTs. For this example, we will proceed with JWTs as it doesn't require the server to store it, which makes it stateless. JWTs consist of three main parts encoded in base64 and separated by a period. The first part contains information about the payload signing mechanism, the second holds the raw payload, and lastly the third contains a password-protected signature of the payload using the same hashing mechanisms in the first part. Check also jwt.io: JSON Web Token (JWT) content.

```
Flask/DataBase/Services/JWTService.py
from jwt import PyJWT from time import time from typing import Union
class JWTService: expires_in_seconds = 2592000 signing_algorithm = "HS256"
def __init__(self, signing_key: str, expires_in_seconds: int = 2592000):
        self.signing_key = signing_key
        self.expires_in_seconds = expires_in_seconds
def generate(self, data: dict,
expires_in_seconds: int = expires_in_seconds)
-> Union[str, None]: try:
            instance = PyJWT()
curr_unix_epoch = int(time()) data['iat'] = curr_unix_epoch
if isinstance(expires_in_seconds, int):
data['exp'] = curr_unix_epoch + expires_in_seconds
            token = instance.encode(
                payload=data,
                key=self.signing_key,
                algorithm=self.signing_algorithm)
if type(token) == bytes:
token = token.decode('utf8') # Needed for some versions of PyJWT
return token
except BaseException as _:
return None
def is_valid(self, token: str, verify_time: bool = True) -> bool:
try:
```

```
payload = self.get_payload(token) if payload is None:
return False
if verify_time and 'exp' in payload and payload['exp'] < int(time()):
return False
return True except:
return False
def get_payload(self, token: str): try:
            instance = PyJWT()
            payload = instance.decode(
                jwt=token,
                key=self.signing_key,
                algorithms=[self.signing_algorithm])
return payload except Exception as e:
return None
```

Since we have a way to issue and validate any token, we can integrate it with our middleware class in Listing 8-4, which has a function responsible to check if the requested route shall be authenticated or not.

If authentication is needed, it will check if the JWT passed is valid. If it is not, it will return a famous 401 error which is the status code equivalent to “Not Authorized”; otherwise, a None is returned which means proceed to the next step in the backend’s code, which will be the controller in our case. And as seen on line 8, we are declaring login and sign-up routes – which will be introduced later – don’t need to be authenticated, because after a successful login, a token will be supplied. The same for signing up, but another layer of protection will be introduced later, to avoid abuse by externals making new accounts without control and supervision.

```
# Flask/DataBase/Middleware.py
from flask import Request
from Services.JWTService import JWTService from werkzeug import exceptions
class Middleware:
def __init__(self, jwt_service: JWTService):
        self.unauthenticated_route_names = {"/api/auth/login",
        "/api/auth/sing_up"}
        self.jwt_service = jwt_service
def auth(self, request: Request): is_route_unauthenticated = request.path
in self. unauthenticated_route_names
if is_route_unauthenticated: return None
if "token" in request.headers:
token = request.headers['token']
is_valid = self.jwt_service.is_valid(token) if is_valid:
return None
else:
return exceptions.Unauthorized()
return exceptions.Unauthorized()
```

Finally, we need to initialize the previously made services and make three more routes for logging in, signing up, and checking login status. We need the last route to allow the frontend to make the decision whether to display the login page or not. So the server’s main file shall look like Listing 8-5. We can notice that secrets

and keys are read from an external YAML file and then parsed. One of those secrets is to make sure only who knows it can make new accounts as shown in Figures 8-3 and 8-4 using Postman.

```
# Flask/flask_main.py
from flask import Flask, request
from DataBase import Connection, Employees, Admins from Services import
JWTService, HashingService from Middleware import Middleware
from werkzeug import exceptions
import yaml
app = Flask(__name__)
with open("secrets.yaml") as f:
yaml_dict = yaml.safe_load(f) sing_up_key = yaml_dict['sing_up_key']
jwt_secret = yaml_dict['jwt_secret']
jwt_service = JWTService(jwt_secret)
middleware = Middleware(jwt_service)
hashing_service = HashingService()
app.before_request(lambda: middleware.auth(request))
@app.route('/api/employees')
def get_all_employees():
    with connection.use_session() as session:
employees = session.query(Employees).all() employees = [employee.to_dict()
for employee in employees]
return {"data": employees}
@app.route('/api/employee', methods=["POST"]) def add_employee():
    body = request.json
    with connection.use_session() as session:
        session.add(Employees(**body))
session.commit()
return {"message": "New employee added successfully"}
@app.route('/api/auth/login', methods=["POST"]) def log_in():
    username, password = request.json['username'], request.
    json['password']
    with connection.use_session() as session:
admin_account = session.query(Admins).filter( Admins.username ==
username).first()
if admin_account is None:
# Username doesn't exist. But don't inform the client with that as
# they can use it to bruteforce valid usernames return
exceptions.Unauthorized(
                description="Incorrect username/password
                combination")
# Checking if such hash can be generated from that password
        is_password_correct = hashing_service.check_bcrypt(
            password.encode("utf8"), admin_account.password_
            hash.encode("utf8"))
if not is_password_correct:
return exceptions.Unauthorized(
                description="Incorrect username/password
                combination")
        token_payload = {"username": username}
        token = jwt_service.generate(token_payload)
if token is None:
```

```
return exceptions.InternalServerError(description=" Login failed")
return {"token": token}
@app.route('/api/auth/sing_up', methods=["POST"]) def sign_up():
username, password = request.json['username'], request. json['password']
if request.headers.get("sing_up_key") != "sing_up_key":
        exceptions.Unauthorized(description="Incorrect Key")
    with connection.use_session() as session:
        password_hash = hashing_service.hash_bcrypt(
password.encode("utf-8")).decode("utf-8")
admin = Admins(username=username, password_ hash=password_hash)
session.add(admin)
session.commit()
return {"message": "Admin account created successfully"}
@app.route('/api/auth/is_logged_in')
def is_logged_in():
# If this controller is reached this means the # Auth middleware
recognizes the passed token return {"message": "Token is valid"}
connection = Connection("postgresql://
postgres:admin@127.0.0.1:5432/CompanyData")
app.run()
```

After creating the account, we can manually check the database if the new credentials are present. This can be verified in Figure 8-5, where the username is as supplied and the other column is a valid Bcrypt hash for the supplied password.

Now as we have admin accounts ready, we can test logging in with Postman before moving on to the next steps. By inserting the same username and password used in signing up, in a JSON format to a POST of the responsible route, we get a token back as seen in Figure 8-6.

Taking this to the next phase of development on Streamlit's side, we will first refactor Listing 8-6 to support initialization with an authentication token. Then use this token as part of every request, except the one used for logging in as it is not needed. Good to note that the is logged in function is implemented for a quick check of the current token's validity if supplied.

```
# Streamlit/API.py
import requests
class API:
def __init__(self, base_url: str, token: str):
        self.base_url = base_url
        self.base_headers = {"token": token}
def add_employee(self, name, dob, paygrade): try:
            data = {
                "name": name,
                "date_of_birth": dob,
                "paygrade_id": paygrade
            }
response = requests.post(self.base_url + "/employee", json=data,
headers=self.base_headers)
if response.status_code == 200: return True
except:
return False
```

```
def get_employees(self): try:
response = requests.get(self.base_url + "/employees",
headers=self.base_headers)
return response.json()['data']
except:
return None
def login(self, username, password): try:
            response = requests.post(self.base_url + "/auth/
            login", json={
                "username": username,
                "password": password
            })
body = response.json()
token = body.get("token") if type(body) == dict else None
return token except:
return None
def is_logged_in(self):
return requests.get(self.base_url + "/auth/is_logged_ in",
headers=self.base_headers).status_code == 200
```

Having our API adapted to authentication tokens as shown in Listing 8-7, we can take it to the actual frontend side by implementing cookie support to store the authentication tokens and use it wherever needed as shown in Listing 8-8. Whenever the Streamlit renders, it will check the local cookies and look for authentication tokens; if the token is valid, it will display the management portal as shown in Figure 8-8 with a customized welcome message. Otherwise, it will prompt for an obligatory login, which can be seen in Figure 8-7.

```
# Streamlit/API.py
import requests
class API:
def __init__(self, base_url: str, token: str):
        self.base_url = base_url
        self.base_headers = {"token": token}
def add_employee(self, name, dob, paygrade): try:
            data = {
                "name": name,
                "date_of_birth": dob,
                "paygrade_id": paygrade
            }
response = requests.post(self.base_url + "/employee", json=data,
headers=self.base_headers)
if response.status_code == 200: return True
except:
return False
def get_employees(self): try:
response = requests.get(self.base_url + "/employees",
headers=self.base_headers)
return response.json()['data'] except:
return None
def login(self, username, password): try:
            response = requests.post(self.base_url + "/auth/
```

```
            login", json={
 "username": username,
                "password": password
            })
body = response.json()
token = body.get("token") if type(body) == dict else None
return token except:
return None
def is_logged_in(self):
return requests.get(self.base_url + "/auth/is_logged_ in",
headers=self.base_headers).status_code == 200
```

```
# Streamlit/streamlit_main.py
import streamlit as st
from Views import AddEmployee, DisplayEmployees, Login from API import API
import extra_streamlit_components as stx
import base64, json
cookie_manager = stx.CookieManager()
cookies = cookie_manager.get_all()
authentication_token = cookies.get("token")\
if type(cookies) == dict else cookies
api = API("http://127.0.0.1:5000/api", authentication_token)
def get_username_from_token(auth_token): b64 = str(auth_token).split(".")
[1] b64 = b64 + "=" * (4 - (len(b64) % 4))

data = base64.b64decode(b64).decode("utf8") username = json.loads(data)
['username'] return username
def manage_login(username, password): token = api.login(username,
password) cookie_manager.set("token", token) return token is not None
st.title("Company Management Portal")
if api.is_logged_in(): st.subheader(f"_Welcome "
                f"**{get_username_from_
                token(authentication_token)}**_")
    st.write("_____")
    AddEmployee(api.add_employee)
    st.write("___")
    DisplayEmployees(api.get_employees)
else: Login(manage_login)
```

Looking closely into the Streamlit side code, we see that almost the same coding pattern to the backend – dependency injection – has been used. This makes the code coherent end to end. Simply the actions of the API are passed down to the views which are abstracted with a class as shown in Listings 8-9, 8-10, and 8-11.

```
# Streamlit/Views/AddEmployee.py
import streamlit as st
from typing import Callable import datetime
class AddEmployee:
```

```
def __init__(self, on_submit: Callable[[str, str, int], bool]):
        st.header("Add a new employee")
form = st.form("new_employee") name = form.text_input("Name") dob =
str(form.date_input("DOB",
                    min_value=datetime.datetime(year=1920,
                    day=1, month=1)))
        paygrade = form.number_input("paygrade", step=1)
if form.form_submit_button("Add new Employee"): success = on_submit(name,
dob, paygrade)
if success:
st.success("New employee added") else:
                st.error("Employee not added")
```

```
# Streamlit/Views/DisplayEmployees.py import streamlit as st
from typing import Callable
class DisplayEmployees:
def __init__(self, get_employees: Callable[[], list]):
        st.header("Current Employees")
        employees = get_employees()
if employees is None:
st.error("Error getting employees")
else: st.table(employees)
```

```
# Streamlit/Views/Login.py import streamlit as st
from typing import Callable
class Login:
def __init__(self, on_login: Callable[[str, str], bool]):
st.header("Login")
username = st.text_input("Username")
password = st.text_input("Password",type="password")
if st.button("Login"):
success = on_login(username, password) if success:
st.success("Login successful") else:
                st.error("Incorrect username and password
                combination")
```

Secrets Management

As we have already discussed how to keep a Streamlit application's secret credentials safe from an external's reach, we now will introduce another way to be used in Flask and that can also be applied in a Streamlit context. Basically, we need a file to add the secrets in. This means the JWT signing key and the signing up header key need to be stored somewhere on disk and then loaded into our application's memory usage. Secrets and keys can be stored in different ways, but one of the most user-friendly ways

is using YAML files, as shown in Listing 8-12, which are then parsed and converted to a Python dictionary.

```
# Flask/secrets.yaml
jwt_secret: "A RANDOM TEXT HERE"
sing_up_key: "ANOTHER RANDOM TEXT HERE"
```

Anti-SQL Injection Measures with SQLAlchemy

As a final code implemented protection, we aim to protect the backend's SQL queries by preventing a nonintended action to happen. First, we need to identify what a SQL injection is. It is mostly when a user-controlled text changes the SQL command behavior. For instance, assume we want to support searching for employees starting with a string of character to be input by the end user; it will result in a final query as such: SELECT * FROM Employees WHERE name = 'input%'. This poses a threat if the input was OR 1=1 -- as it would result in a final query of SELECT * FROM Employees WHERE name = '%' OR 1=1 -- which orders the database to select all employees, instead of treating the input as the search string. To overcome this problem, we need to use parameterization which is a technique to isolate the original SQL command from the changing variables. So, for the example from before, it would rather look something likeSELECT * FROM Employees WHERE name = '@name%'where@name is a SQL variable initialized before submitting the query. As a developer, this might be an overhead toward a more secure SQL. Thus, we use libraries and/or packages to do this on our behalf. For this scenario, we are using SQLAlchemy, which is a library that can connect to many types of databases and change SQL command formats depending on the architecture, origin, and version, all by following an intuitive API that is documented in docs.sqlalchemy.org.

Configuring Gitignore Variables

Having all files tracked with a version control system like Git is a must for big projects, as it adds more simplicity in managing what files are important and their modification history. However, not all files must be tracked, and some will pose security threats if tracked, as the code base can be accessed by anyone if it is public, or broken into if it is private. This makes the secrets managed under threat; hence, it is widely agreed among software developers to not track the secrets file and store the actual secrets in a security vault, which is accessible using multiple methods of authentication. As this adds one more layer of security over the application's secrets, it gives a direct hit to code readability and understanding if an unaware developer started working on the code. This can be fixed by adding another file with the following naming conventions: Flask/secrets.example.yaml which will host similar content to what is shown in Listing 8-12 albeit with a replacement of the actual key values with something vague yet understandable, as shown in Listing 8-13. Subsequently, we can remove the original secrets file from Git by updating the .gitignore file as shown in Listing 8-14. If needed, any file and folder in or under the same folder as .gitignore can be set to be ignored or excluded depending on the syntax.

```
# Flask/secrets.example.yaml
# Copy the content of this file to secrets.yaml
# and replace its contents with the correct values
jwt_secret: "<INSERT TEXT>"
sing_up_key: "<INSERT TEXT HERE>"
```

```
# Flask/.gitignore
secrets.yaml
```

Deploying locally and to the cloud

Deployment to Streamlit Cloud

Streamlit however is democratizing this one last frontier and making deployment quite literally a one-click process. With Streamlit Cloud, you can simply connect your GitHub repository and then click deploy. Streamlit will then provision the application with all of its required dependencies for you and will update it each time you push a new version of your source code. No additional intervention is required by the developer whatsoever. Furthermore, if you require more than one private application, additional computing resources, or enterprise-grade features, you may upgrade to Streamlit's premium packages.

One-Click Deployment to the Streamlit cloud

Before you proceed to deploy your first application to Streamlit Cloud, you should open a GitHub account and push your script to a repository. Subsequently, you can follow these steps to deploy:

Navigate to share.streamlit.io and follow along!

Another benefit associated with using Streamlit Cloud is that you can securely store private data on Streamlit's servers and readily access them in your application. This feature is exceptionally useful for storing user credentials, database connection strings, API keys, and other passwords, without having to enter them in plain text form in your code (which you should never do under any circumstances). Instead, you can execute the following steps to store and access private data with Streamlit's Secrets Management.

Add your secrets in the form of a TOML file! If you wish to replicate Secrets Management locally on your own server, you can simply add the same TOML file as secrets.toml in the .streamlit folder in your root directory. Ensure that such files are ignored in Git commits by adding the folder to your .gitignore file.

```
import streamlit as st
st.write('**Secret:**',st.secrets['secret'])
st.write('**Password:**',st.secrets['section']['password'])
```

Deployment to Linux

pp.240-245

Streamlit components

263-307

Streamlit use cases

309-361

Streamlit at work

363-379