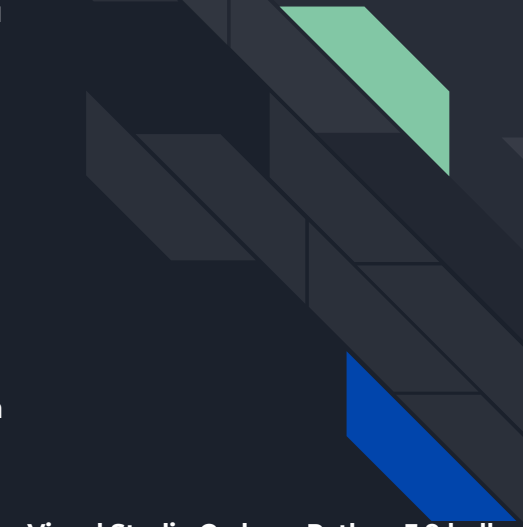
Flask ve Dash İle Multi Page Web Application Yapımı

Muhammed Enes Baysan
menesbaysan@gmail.com
@mebaysan
28/12/2020

İçindekiler

Giriş

Bu dökümanı bu projeyi yaparken zorlandığım için başkalarına faydalı olabilmek adına hazırladım. Bir veri ambarının canlı olarak görselleştirilip yöneticilere sunulması amacıyla bir web projesi geliştirmem istendi. Dash paketleri ile bu işi çok rahat halledebilirdim fakat şöyle bir problemim vardı; ekranları oluştururken yetkilendirme işlemlerini de hesaba katmam gerekiyordu. Dash'in authentication paketi ile bu sorunu aşamadım. Çünkü Dash'in kendi dökümanında gösterilen çok sayfalı uygulama aslında url path'ine göre değişen bir sayfa içeriğiydi. Bu sebeple kendi authentication paketini kullanırsam sadece bir path'e izin vermek gibi bir seçeneğim yoktu. Uygulamaya erişim için bilgilere sahip olan birisi tüm path'lere erişebiliyordu. Bu sebeple her path'i ayrı bir Dash uygulaması haline getirdim. Daha sonrasında tüm path'leri (Dash uygulamalarını) tek bir Flask uygulaması altında toplayıp Dash ekranlarını iframe olarak Flask uygulamasında gösterdim. Ufak bir rol yönetim ekranı ile de kimlerin hangi dashboard'ları görüp göremeyeceğini halledebildim. Dash uygulamalarını Flask içerisinde iframe olarak kullanmak internette arandığında bulunabilecek bir kaynak fakat authentication'ın olduğu bir örneğe denk gelemedim ve bu konudaki örnekler de hep yabancı kaynaklardaydı. Bu konu hakkında Türkçe kaynak oluşturabilmek ve uygulamalı bir örnek ile başlangıç şablonu olmasını dilediğim bir çalışma yapmak istedim. Umarım faydalı olur.



***Döküman boyunca Visual Studio Code ve Python 3.9 kullanacağım.**

***Temel Flask ve Dash bilgisinin olduğu varsayılmıştır.**

Kullanılan Paketler

- Flask
- Dash
- Dash Bootstrap Components
- Plotly
- Flask Login
- Flask SQLAlchemy

Proje Yapısı

- Proje **wsgi.py** dosyasından ayağa kalkar
- Kök dizindeki **fetch_all.py** dosyası sayesinde tüm dash uygulama dizinlerinin altındaki fetch_data.py dosyası çalıştırılır.
- **__init__.py** ana FLASK uygulamamızı oluşturur
- **app.py** ana FLASK uygulamamıza ait bileşenleri barındırır (blueprints, routes, handlers vb)
- Ana dizindeki **myconfig.py** dosyası Flask uygulamasına ait ayarları tutmaktadır (static path, development mode vb.)
- Proje ile ilgili bütün fonksiyonlar, modüller, paketler vb **flaskapp** klasörü altında toplanmıştır
- **flaskapp/dashboard** klasörü altından DASH uygulamalarına erişebiliriz
- **flaskapp/dashboard/utilites/database_config.py** dosyası ambara bağlanılacak olan pymssql config bilgilerini içerir
- **flaskapp/dashboard/utilities** klasörü altında işimizi kolaylaştıracak fonksiyonlar bulunmaktadır, varsa kendi yazdığımız componentler vb, veya dash uygulamalarının uyumluluğu için CSS sabitleri

- **flaskapp/static** klasörü altında FLASK uygulamasına ait static dosyalar (bootstrap, jquery vb) tutulmaktadır
- **flaskapp/templates** klasörü altında FLASK uygulamasına ait route'ların döndürdüğü template (jinja2, html vb) dosyaları tutulmaktadır
- **flaskapp/dashboard/apps** klasörü altında **her bir** dash uygulamasına ait klasörler bulunmaktadır
    - **app.py** altında DASH uygulaması ayağa kalkar
        - **BASE_URL** sabitleri iframe olarak kullanılacak linki bize vermektedir. -> ÖR: www.xyz.com/BASE_URL
        - **APP_NAME** sabitleri DASH uygulamasını FLASK'a gönderirken (navbar, context processors vb) hangi isimle anılacağını belirler
    - **data.py** altında DASH uygulamasına gönderilecek olan veri okunur/hazırlanır
    - **fetch_data.py** altında veri sunucuya çekilir, işlenir ve bulunduğu dizine csv olarak yazılır
    - ***.xlsx/*.csv** dosyaları eğer veri seti SQL'den çekilmeyecekse aynı dizin altına eklenir

Proje Dizin Yapısı

```
.
├── README.md
├── config.py
├── flaskapp
│   ├── __init__.py
│   ├── context_processors.py
│   ├── dashboard
│   │   ├── apps
│   │   │   ├── dash_uygulamasi_1
│   │   │   │   ├── veriseti.csv
│   │   │   │   ├── app.py
│   │   │   │   └── data.py
│   │   │   │   └── fetch_data
│   │   ├── layout.py
│   │   ├── urls.py
│   │   └── utilities
│   │       ├── style.py
│   │       └── tables.py
│   ├── routes.py
│   ├── static
│   └── templates
│       ├── dashboards
│       │   ├── dashboard_basic.jinja2
│       │   └── dashboard_layout.jinja2
│       ├── index.jinja2
│       └── utilities
│           ├── layout.jinja2
│           └── navbar.jinja2
├── req.txt
└── wsgi.py
└── fetch_all.py
```

Flask Uygulamamızı Oluşturalım

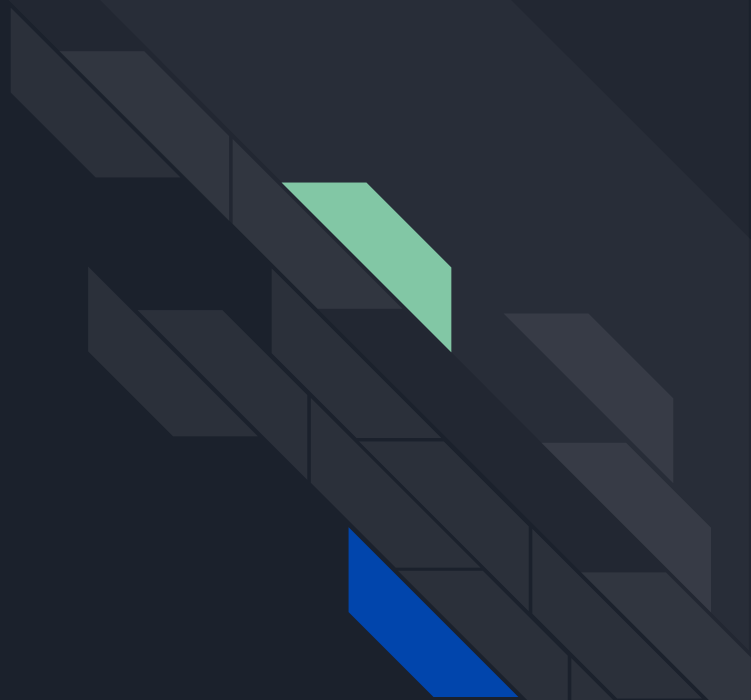
Proje için DASH-FLASK-MultiPageApp klasörü oluşturuyorum. python3 -m virtualenv venv komutu ile bulunduğum dizine (DASH-FLASK-MultiPageApp) bir virtualenv oluşturuyorum. Sanal ortamı aktif ettikten sonra pip install flask diyerek paketi sanal ortamıma yüklüyorum. Projeye ait tüm kodlar bu klasörde bulunacak. Kök dizine **myconfig.py** **ve** **wsgi.py** dosyalarını oluşturup ardından Flask uygulamasını oluşturmak için flaskapp adında bir klasörü oluşturup içerisine **__init__.py** ve **app.py** dosyalarını oluşturuyorum.

- myconfig.py içerisinde Flask uygulamasına ait configleri tutacağım
- __init__.py içerisinde; myconfig'den aldığım ayarlar ile Flask uygulamasını ayağa kaldıracağım. İlerleyen aşamalarda burada Dash uygulamalarını Flask ile bind edeceğiz.
- app.py içerisinde route'ları ve blueprint'leri yazacağım.
- wsgi.py dosyasından ise __init__.py altında oluşturduğum Flask uygulamasını serve edeceğim.

*** BASE_PATH = DASH-FLASK-MultiPageApp olarak döküman boyunca kullanacağım.**

BASE_PATH/myconfig.py altında Config adında bir class oluşturuyorum ve Flask uygulamam için ayarları set ediyorum.

```
class Config:
    FLASK_APP = 'wsgi.py'
    FLASK_ENV = 'development'
    SECRET_KEY = 'cvhuylfd123qxpmrdm47681hds12'
    STATIC_FOLDER = 'static'
    TEMPLATES_FOLDER = 'templates'
```



BASE_PATH/app.py altında ilk route'umu yazıyorum. Tabii BASE_PATH/flaskapp/templates altında da index.jinja2 dosyamı oluşturuyorum.
current_app sayesinde o anda çalışan Flask uygulamasına erişebiliyorum.

```python
from flask import render_template
from flask import current_app as app


@app.route('/')
def index():
    return render_template('index.jinja2', title='Anasayfa')
```

app.py

BASE_PATH/flaskapp/templates altında da index.jinja2 dosyamı yazıyorum. Bunun için aynı dizin altında bir de layout.jinja2 adında bir dosya oluşturuyorum. Bu dosya layout dosyam olacak ve Flask uygulama ait şablonlar bu dosyadan miras alacak.

flaskapp > templates > layout.jinja2

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      name="viewport"
      content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0"
    />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />

    <title>
      {% if title %} Dash & Flask | {{ title }} {% else %} Dash & FLask {% endif
      %}
    </title>
    {% block custom_css %} {% endblock %}
    <style>
      .my-navbar {
        list-style-type: none;
        margin: 0;
        padding: 0;
      }
      .my-navbar li{
        display: inline;
      }
    </style>
  </head>
  <body>
    <ul class="my-navbar">
      <li><a href="/">Anasayfa</a></li>
    </ul>
    {% block content %} {% endblock %} {% block custom_js %} {% endblock %}
  </body>
</html>
```

layout.jinja2

index.jinja2 dosyam da layout.jinja2 dosyasından miras alıyor ve sadece content bloğunu dolduruyor. Bu şablon app.py içerisindeki index fonksiyonu tarafından return edilecek.

flaskapp > templates > index.jinja2

```
1   {% extends 'layout.jinja2' %}
2   {%block content %}
3
4   <h1>Anasayfa</h1>
5
6   {%endblock content %}
```

index.jinja2

BASE_PATH/flaskapp/__init__.py dosyası sayesinde myconfig.py içerisindeki ayarlar ve init_flask_app fonksiyonu sayesinde Flask uygulamamızı oluşturuyoruz.
Buradaki core_app oluşturduğumuz Flask uygulamasını temsil eder. with app_context altında bu uygulamayı oluştururken import edilmesi gereken modülleri belirtiyoruz.
app.py dosyasını da burada dahil ediyoruz. Bu sayede route'larımızı (ve bu dosya içerisindeki diğer bileşenleri; blueprints vd.) aktif ediyoruz diyebiliriz. İleride bu blok altında Dash uygulamalarımızı da import edip Flask uygulaması ile bind edeceğiz.

```
from flask import Flask
from myconfig import Config

def init_flask_app():
    core_app = Flask(__name__, instance_relative_config=False)
    core_app.config.from_object(Config)
    with core_app.app_context():
        from flaskapp import app
    return core_app
```

__init__.py

BASE_PATH/wsgi.py dosyası içerisinde de __init__.py dosyası altında oluşturduğumuz init fonksiyonunu çağırıyoruz ve o dosyada oluşturduğumuz Flask uygulamasını ayağa kaldırıp bu dosya içerisinde oluşmuş Flask uygulamasını serve ediyoruz.

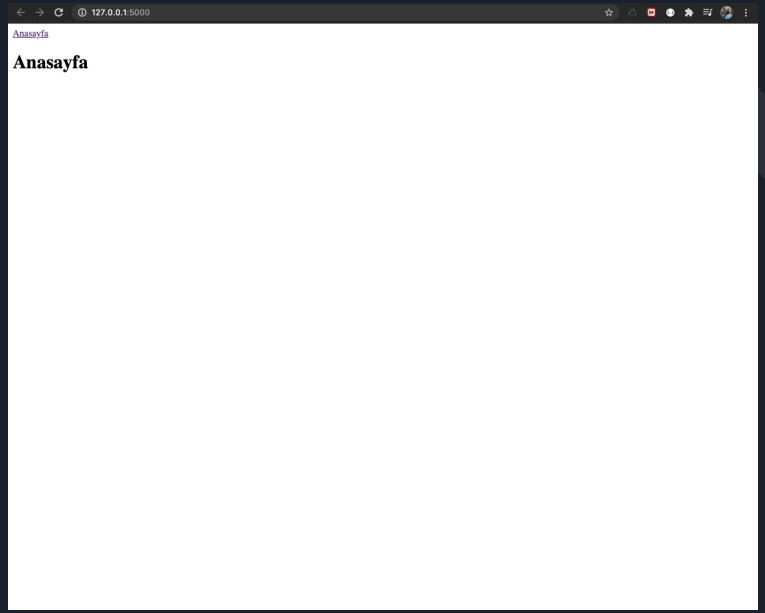
```
from flaskapp import init_flask_app


if __name__ == '__main__':
    app = init_flask_app()
    app.run()
```

wsgi.py

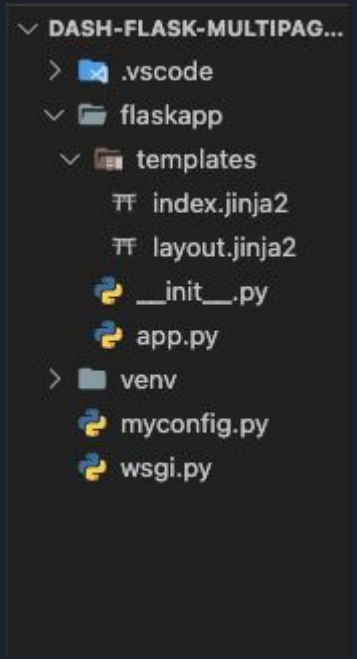
BASE_PATH içerisinde python wsgi.py komutunu çalıştırarak uygulamamızı ayağa kaldırıyoruz. Bu şekilde bir çıktı alacağız.

```
(venv) mebaysan@Baysans-MacBook-Air DASH-FLASK-MultiPageApp % python wsgi.py
 * Serving Flask app "flaskapp" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Linke gittiğimizde ise bizi app.py altındaki index route'u karşılayacak.



Buraya kadar adımları izlediğimizde dizin yapımızın son hali aşağıdaki gibi olacaktır. Buraya kadar basit bir Flask uygulaması inşa ettik. Şimdi basit bir Dash uygulaması oluşturup bununla Flask uygulamasını birleştirelim. Adımları takip ettiyseniz dizin yapınızın son yapısı şu şekilde olacaktır.



Dash Uygulamamızı Oluşturalım

BASE_PATH/flaskapp altında dashboard adında bir klasör oluşturuyorum. Proje yapısında anlattığım gibi bu klasör altında Dash uygulamalarımı ve kendi fonksiyonları vb tutmak için gerekli dosyaları oluşturacağım.

- apps klasörü altında Dash uygulamalarımı oluşturacağım. Her bir Dash uygulaması için ayrı klasörler oluşturup içlerini dolduracağım.
  - fetch_data.py ile veriyi localime çekeceğim
  - data.py ile localden veriyi okuyacağım
  - app.py ise ilgili Dash uygulamamı barındıracak olan dosya olacak
- utilities klasörü altında kendi yazdığım fonksiyonları (tarih formatlayıcılar, döviz çevirme fonksiyonları, stil sabitleri vb.) tutacağım.
- urls.py içerisinde ise her bir Dash uygulamasına ait CONFIG sabitlerini bir liste olarak tutup url şemamı oluşturacağım.

İlk dashboard için seaborn ile gelen tips veri setini kullanacağım. Senaryomuz gereği bu veri setini fetch_data.py ile localime çekip data.py ile Dash uygulamama göndereceğim. Önce locale çekip sonra Dash'e göndermemin sebebi ise her restart attığımızda veri setini tekrar web'den/sql'den çekmek istemiyor olmamdır. İleride yazacağımız fetch_all.py dosyası sayesinde bu dosya üzerinden tüm Dash'lere ait fetch_data.py'ları çalıştırıp veri setlerini güncelleyeceğiz. (Çalıştığım sorgularda 50m küsür satırı sql'den çekiyorum. Büyük bir veri olmadığının farkındayım fakat güncellemeler vb. için her restart attığımda 50m satır veriyi baştan çekmemek için fetc_data.py ile data.py'ı ayırdım. data.py her seferinde çalışıyor fakat fetch_data.py'lar fetch_all.py sayesinde bir crontab üzerinden sadece gece 3'te çalışıyor. Bu sayede yorulmadan günlük olarak verileri taze tutabiliyorum.)

BASE_PATH/flaskapp/dashboard/apps/tips_dash klasörünün altında app.py, fetch_data.py ve data.py dosyalarımı oluşturuyorum. Normalde DB'den çekmek için utilities altında DB ayarlarını tanımlayabiliriz. Fakat bu uygulamamızda csv dosyaları ile çalışacağımız için buna gerek olmayacaktır.

```
import pandas as pd
import os


def fetch_data():
    df = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/master/tips.csv')
    df.to_csv(os.path.join(os.getcwd(), 'flaskapp', 'dashboard', 'apps', 'tips_dash', 'data.csv'))
```

os modülü sayesinde tips veri setine ait Dash'imin bulunduğu dizine fetch ettiğim veri setini csv olarak yazıyorum. Bu sayede her Dash'e ait veri seti kendi bulunduğu dizine yazılmış olacaktır. Fakat burada direkt olarak df.to_csv('data.csv') fonksiyonunu çalıştırsaydım bu csv BASE_PATH altına yazılmış olacaktı. Bunun sebebi ise bu fonksiyonu çalıştırdığımız fonksiyonun wsgi.py dosyası altından çağrılmasıdır.

fetch_data.py

BASE_PATH/flaskapp/dashboard/apps/tips_dash/data.py dosyasını oluşturup yine os modülü sayesinde ilgili dizin altındaki data.csv dosyasını okuyup return eden bir fonksiyon yazıyorum. Bu fonksiyonu Dash altından çağıracağız ve veri setini Dash uygulamasında kullanılabilir hale getireceğiz.

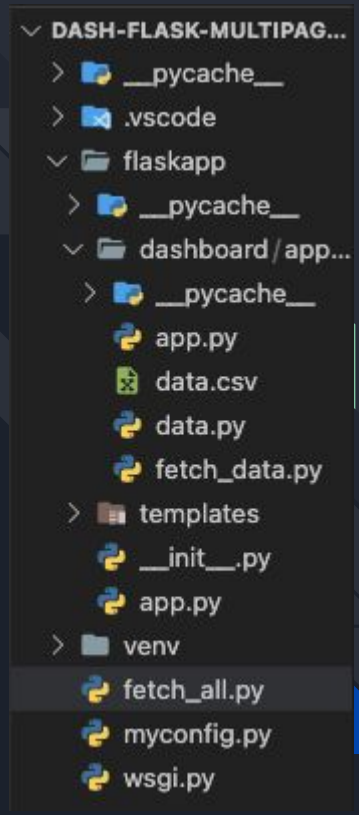
```
import pandas as pd
import os

def get_data():
    df = pd.read_csv(os.path.join(os.getcwd(), 'flaskapp', 'dashboard', 'apps', 'tips_dash', 'data.csv'))
    return df
```

tips_dash dizini altındaki data.csv dosyasını okuyup return ediyoruz.

data.py

Şimdi app.py dosyasını oluşturmadan önce fetch_all.py dosyasını oluşturalım.
Bahsettiğimiz gibi bu dosya altında tüm fetch_data.py dosyalarını çağırıp fetch_data fonksiyonlarını çalıştıracağız. Bu sayede veri setleri ilgili dizinler altına yazılacak.
Bu dosyamızı da BASE_PATH altında oluşturuyorum.
Bu örneğimizde kullanmayacağız fakat bir cronjob yazarak bu dosyayı çalıştırarak belirli periyotlarla verilerinizi güncel tutabilirsiniz.
***Tabi bu dosyayı çalıştırabilmek dolayısıyla fetch_data.py ve data.py dosyalarını çalıştırabilmek için pip install pandas komutu ile Pandas'ı sanal ortamıma yüklüyorum.**

```
from flaskapp.dashboard.apps.tips_dash.fetch_data import fetch_data as fetch_tips


fetch_tips()
```



Daha sonra python fetch_all.py komutu ile dosyamı çalıştırarak veri setini tips_dash altına çekiyorum.

- **Bunu yaparken muhtemelen SSL hatası alacağız bu konu bu dökümanın konusu olmadığından bunu şimdilik geçip veri setini elimle (ilgili dizine terminalden giderek ipython yardımı ile veriyi çekip yazıyorum) ilgili dizine yerleştiriyorum.**

Eğer siz sql'den veyahutta başka bir kaynaktan veriyi çekip bu komutu çalıştırırsanız veri setinin ilgili dizine yazılmış olduğunu göreceksiniz.

BASE_PATH/fetch_all.py

pip install dash komutu sayesinde Dash paketini sanal ortamıma yüklüyorum. Otomatik olarak plotly vd paketler de yüklenecektir. Fakat ekstra olarak Dash Bootstrap paketini de yüklemek istiyorum. Kısaca bahsetmek gerekirse Bootstrap componentlerini kullanmamızı sağlayan bir pakettir kendileri. pip install dash-bootstrap-components komutu ile de bu paketi sanal ortamıma yüklüyorum. Ardından basit bir Dash uygulaması oluşturmak için aşağıdaki kod bloğunu app.py içerisine yazıyorum (basit bir Dash örneği).

```
import dash
import dash_core_components as dcc
import dash_html_components as html
import dash_bootstrap_components as dbc
import plotly.express as px
from flaskapp.dashboard.apps.tips_dash.dataimport get_data


CONFIG = {
    'BASE_URL': '/h9M5hdnUFRE8qffkqDUrWdK/tips/',
    'APP_URL': 'tips',
    'APP_NAME': 'Tips Veri Setine Ait Dashboard',
    'MIN_HEIGHT': 1500,
}
```

```
def add_dash(server):

    def get_fig(df):
        return px.scatter(df, x="total_bill",
                          y="tip",
                          color='time',
                          log_x=True,
                          size_max=60,
                          title='Toplam Hesaba Göre Verilen Bahşiş Miktarı',
                          labels={'time': 'Hangi Öğün',
                                  'total_bill': 'Toplam Hesap', 'tip': 'Bahşiş'}
                         )


    DF = get_data()

    app = dash.Dash(server=server,
                    routes_pathname_prefix=CONFIG['BASE_URL'],
                    suppress_callback_exceptions=True,
                    external_stylesheets=[dbc.themes.BOOTSTRAP])
```

```
    day_dropdown = dcc.Dropdown(
        id='day-dropdown',
        options=[

            {'label': f'Gün: {day}', 'value': day} for day in DF['day'].unique()
        ],
        searchable=False,
        placeholder='Gün Seçebilirsiniz...',
    )

    smoker_dropdown = dcc.Dropdown(
        id='smoker-dropdown',
        options=[
            {'label': f'Sigara: {smoker}', 'value': smoker} for smoker in
DF['smoker'].unique()
        ],
        searchable=False,
        placeholder='Sigara Durumunu Seçebilirsiniz...'
    )
```

```
LAYOUT = html.Div(children=[
        html.H1('Basit Bir Dash Örneği', style={
            'textAlign': 'center',
            'color': 'red',
        }),
        html.Div(children=[
            dbc.Row(children=[
                dbc.Col(children=[
                    day_dropdown,
                smoker_dropdown
                ]),
                dbc.Col(children=[
                    dcc.Graph(
                id='scatter-chart',
                figure=get_fig(DF)
            )
                ])
            ]),
        ]),
    ])
```

```
app.layout = LAYOUT
@app.callback(
        dash.dependencies.Output('scatter-chart', 'figure'),
        [dash.dependencies.Input('day-dropdown', 'value'),
        dash.dependencies.Input('smoker-dropdown', 'value')]
    )

    def day_filtrele(day_value, smoker_value):
        if not day_value and not smoker_value:
            return get_fig(DF)
        elif day_value and not smoker_value:
            return get_fig(DF.query(f'day == "{day_value}"'))
        elif smoker_value and not day_value:
            return get_fig(DF.query(f'smoker == "{smoker_value}"'))
        else:
            return get_fig(DF.query(f'day == "{day_value}"').query(f'smoker == "{smoker_value}"'))

    return server
```

tips_dash/app.py

Şimdi bu kodu açıklamaya çalışayım.

- CONFIG sabiti içerisinde bu Dash uygulamasına ait unique değerler set edilir.
    - BASE_URL keyi sayesinde Flask içerisinde oluşturduğumuz iframe'in linkini bu key olarak vereceğiz. Bu sayede bu Dash görüntülenebilecek. Fakat unutmayalım bu Dash'i iframe olarak göstereceğimizden bu BASE_URL keyine sahip olan herkes Flask sistemine login olmadan direkt istek atarak bu iframe'i görebilir. Bunun önüne geçmek için her restart atıldığında burada random string oluşturup bu key'i generate edebilirsiniz. Bu sayede sistem her gece yeniden başlatıldığında (her gece verileri tazeleyeceğinizi düşünerek) yeni bir random url'e sahip olmuş olacak.
- APP_URL keyi sayesinde rol bazlı yönetim yapabileceğiz. Bu APP_URL ile aynı isimde roller üretip bunları kullanıcılara atayacağız.
- APP_NAME keyi sayesinde Flask uygulamamızda yazacağımız context_processor ile navbarda gösterdiğimiz linklerin isimleri bu keyden gelecek
- MIN_HEIGHT keyi sayesinde de iframe'in minimum boyu set edilmiş olacak.

Aslında burada bu sabitten ayrı olarak çok önemli olan bir diğer bileşen ise add_dash fonksiyonudur. Gördüğümüz gibi bu fonksiyon parametre olarak bir server alıyor. Aldığı server'in üzerine Dash uygulamasını oluşturuyor. Bu sayede Flask uygulaması ile Dash uygulamasını bind edebiliyoruz. Bu fonksiyon aldığı server parametresini return ediyor. Return işlemi gerçekleşene kadar aldığı serverin üzerine Dash uygulamasını ekliyor ve güncellenmiş olarak bu serveri döndürüyor. Bu fonksiyonu da __init__.py dosyası altında çağıracağız ve parametre olarak oluşturduğumuz core_app değişkeni olan Flask uygulamamızı göndereceğiz.

Flask ile Dash'ı Birleştirelim

Oluşturduğumuz add_dash fonksiyonunu __init__.py içerisinde import etmeden önce Dash uygulamalarımız için de url şemamızı oluşturacağımız urls.py dosyamızı oluşturalım. BASE_PATH/flaskapp/dashboard/urls.py içerisinde Dash uygulamalarımıza ait olan CONFIG sabitlerini bir liste olarak tutacağız. Bu sayede context processor içerisinde bu listeyi dönerek Dash uygulamalarına ait url şemasını Flask tarafında rahat bir şekilde kullanabileceğiz. Aynı zamanda yetkilendirme işlemini yaparken de işimiz kolaylaşacak.

```
from flaskapp.dashboard.apps.tips_dash import app as tips_dash

URL_PATHS = [
    tips_dash.CONFIG,
]
```

urls.py

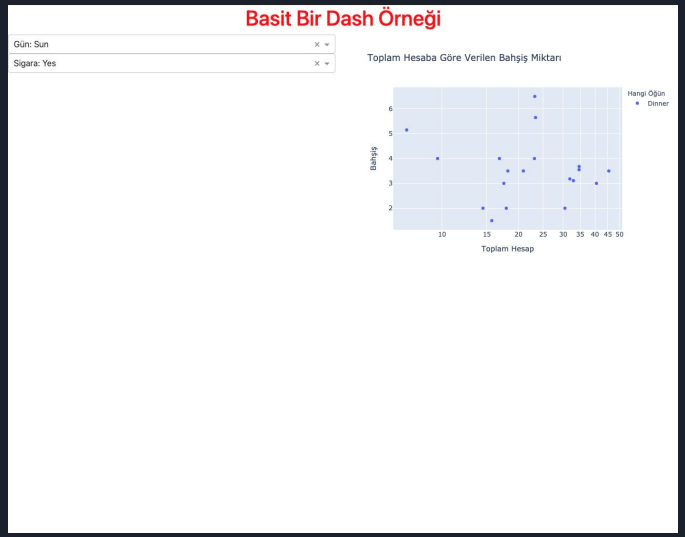
Şimdi __init__.py içerisinde tips_dash altındaki add_dash fonksiyonunu import edelim.

```
from flask import Flask
from myconfig import Config
from flaskapp.dashboard.apps.tips_dash.app import add_dash as add_tips_dash


def init_flask_app():
    core_app = Flask(__name__, instance_relative_config=False)
    core_app.config.from_object(Config)
    with core_app.app_context():
        from flaskapp import app
        core_app = add_tips_dash(core_app)
    return core_app
```

__init__.py

Hatırlayalım; add_dash fonksiyonu parametre olarak aldığı Flask uygulamasının üzerine bir Dash uygulaması inşa edip güncellenmiş Flask uygulamasını return ediyordu. Burada da core_app değişkenimizi (ana Flask uygulamamız) üzerine Dash inşa edilmiş Flask uygulaması ile güncelliyoruz ve return ediyoruz. Bu sayede burada istediğimiz kadar Dash uygulamasını import edip core_app'i güncelleyerek tek bir Flask uygulamasının üzerine Dash uygulamaları inşa edebiliriz.

python wsgi.py komutu ile uygulamamızı çalıştıralım ve tips_dash altında set ettiğimiz BASE_URL'e istek atalım. Eğer Dash uygulamasını görebilirsek başarılı olmuşuz demektir.



Evet Dash uygulamamızı başarılı bir şekilde Flask uygulamamız ile birleştirip ayağa kaldırabildik. Şimdi de context_processor yazarak urls.py altındaki tüm url'leri Flask layoutuna gönderelim ve bu Dash'leri birer iframe içerisinde gösterelim.

Dash Url'lerini Flask Şablonlarına Gönderelim

Yukarıda Dash uygulamamıza kendi url'i üzerinden ulaştık. Şimdi bunları context_processor sayesinde layout.jinja2'ye gönderelim. BASE_PATH/flaskapp/context_processors.py dosyasını oluşturuyorum. get_dashboard adındaki fonksiyon urls.py altındaki URL_PATHS listesini bir sözlüğe çevirerek return edecek. Template'lerden bu listeye get_dashboards değişkeni ile erişebileceğiz.

```
from flask import current_app as app
from flaskapp.dashboard.urls import URL_PATHS


@app.context_processor
def get_dashboards():
    return dict(get_dashboards=URL_PATHS)
```

Bu dosyamızı da __init__.py altında import etmemiz lazım. with app context sayesinde uygulamamız ayağa kalkarken context processorlerimiz de uygulamamız ile birlikte aktif edilmiş olacak.

context_processors.py

Flask uygulamasını oluştururken context_processors dosyasını da import ediyoruz. Bu sayede context processorlerimiz de aktif edilmiş olacak.

```
from flask import Flask
from myconfig import Config
from flaskapp.dashboard.apps.tips_dash.app import add_dash as add_tips_dash

def init_flask_app():
    core_app = Flask(__name__, instance_relative_config=False)
    core_app.config.from_object(Config)
    with core_app.app_context():
        from flaskapp import app,context_processors
        core_app = add_tips_dash(core_app)
    return core_app
```

templates klasörü altındaki layout.jinja2 dosyasında ufak bir güncelleme yapmamız gerekiyor. Yazdığımız get_dashboard context processor'u sayesinde tüm Dash'lere ait url'leri navbarda gösteriyoruz.

```
<ul class="my-navbar">
    <li><a href="/">Anasayfa</a></li>
    {% for dashboard in get_dashboards %}
        <a href="/dashboard/{{ dashboard['APP_URL'] }}">{{ dashboard['APP_NAME'] }}</a>
    {% endfor %}
  </ul>
```

layout.jinja2

Fakat bu linke tıkladığımızda 404 hatası ile karşılaşacağız. Bunun sebebi ise /dashboard/APP_URL path'ini karşılayacak bir route yazmamamızdan kaynaklanmakta.

app.py altında Dash uygulamalarını iframe olarak göstermek için bir route yazalım. Aynı zamanda ilerleyen aşamalarda bu route içerisinde yetki kontrol işlemleri de yapacağız.
app.py dosyasında URL_PATHS değişkenini import ediyoruz.

```
from flaskapp.dashboard.urls import URL_PATHS
```

Şimdi route'umuzu yazabiliriz. Eğer /dashboard/ pathine parametre olarak gelen dash_url değişkeni URL_PATHS içerisindeki elemanlardan birinin APP_URL key'i ile eşleşirse dashboard_basic.jinja2 template'ini döneceğiz ve iframe için gerekli olan Dash url, Dash Name ve Dash Min Height parametrelerini göndereceğiz. Unutmayalım URL_PATHS değişkeni Dash uygulamaları altındaki CONFIG (dict) sabitlerini tutmakta.

```
@app.route('/dashboard/<string:dash_url>')
def get_dash(dash_url):
    CONFIG = next(filter(lambda x: x['APP_URL'] == dash_url, URL_PATHS),None)
    if not CONFIG:
        return '404'
    return render_template('dashboard_basic.jinja2', dash_url=CONFIG['BASE_URL'],
                           dash_min_height=CONFIG['MIN_HEIGHT'],
                           title=CONFIG['APP_NAME'])
```

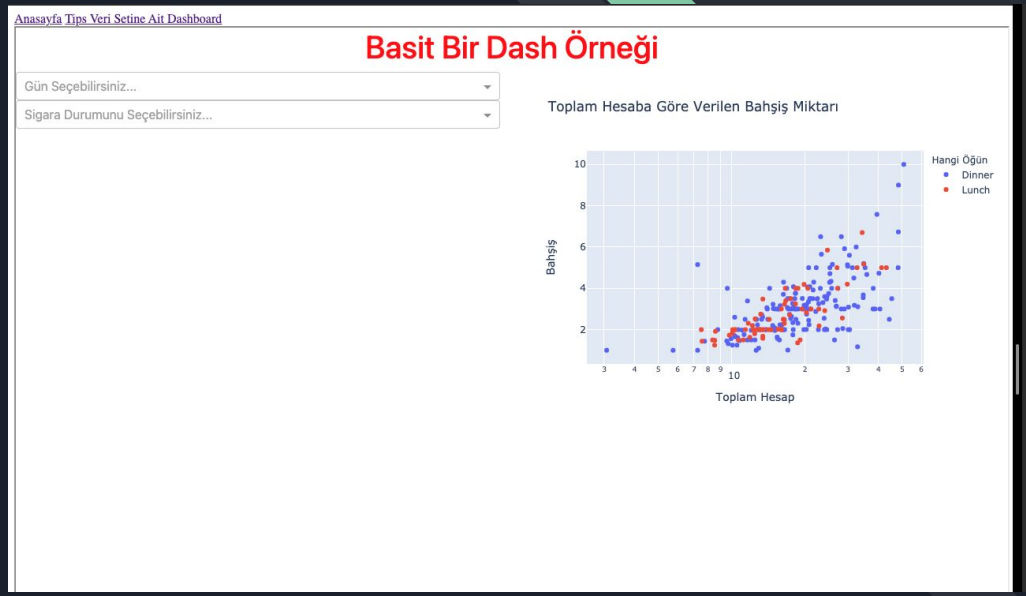
app.py

Şimdi bu route'muz için bir şablon dosyası hazırlayalım. templates klasörü altında dashboard_basic.jinja2 dosyasını oluşturuyorum. Route'dan anladığımız gibi şablona gönderdiğimiz 3 parametreyi iframe'in ilgili özelliklerine set ediyorum. Bu sayede Dash uygulamasına ait APP_URL iframe'in source'u oluyor ve Dash uygulaması iframe olarak gözüküyor.

```
{% extends "layout.jinja2" %}

{% block content %}
    <iframe src="{{ dash_url }}" width="100%" height="{{ dash_min_height }}"></iframe>
{% endblock %}
```

dashboard_basic.jinja2

Bu örneğimizde Dash ve Flask'ı birleştirmeye odaklandığımdan stile önem vermedim. Lütfen stilsizliği dikkate almayınız :)
Yanda gözüktüğü gibi layout'ta yazdığımız navbar dashboard üzerinde gözüktü. Artık Dash uygulamasını Flask uygulaması içerisinde göstermiş olduk. Eğer url'i sallarsak 404 ile karşılaşacağız. Burada Flask flash'ları kullanarak hoş mesajlar gösterebilirsiniz. Fakat konumuz bu olmadığından bunu da şimdilik geçeceğim.



Authentication İçin Modellerimizi Oluşturalım

Artık Flask-SqlAlchemy kullanarak rol ve kullanıcı modellerimizi oluşturabiliriz. pip install flask-sqlalchemy komutu ile ilgili paketi sanal ortamıma kuruyorum. myconfig.py dosyası içerisinde SqlAlchemy için gerekli olan key'leri ekliyorum.
*Ben bu örneği yaparken PostgreSQL kullanıyorum ve db'nin bağlantısını veriyorum. Siz diğer db bağlantıları için paketin dökümantasyonuna bakabilirsiniz.

```
class Config:
    FLASK_APP = 'wsgi.py'
    FLASK_ENV = 'development'
    SECRET_KEY = 'cvhuylfd123qxpmrdm47681hds12'
    STATIC_FOLDER = 'static'
    TEMPLATES_FOLDER = 'templates'
    SQLALCHEMY_DATABASE_URI = 'postgresql://youtube:123456@localhost/Dash-FlaskDB'
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

myconfig.py

*PostgreSQL ile çalışmak için pip install psycopg2 ile gerekli paketi kurmamız gerekmektedir.

BASE_PATH/flaskapp altında db.py adında bir dosya oluşturuyorum. Bu dosya içerisinde SqlAlchemy'e ait instance'ı tutacağım.

```
from flask_sqlalchemy import SQLAlchemy


db = SQLAlchemy()
```

db.py

Ardından __init__.py dosyama gidiyorum ve oluşturduğum db instance'ı ile core_app (Flask uygulaması)'ı bind ediyorum. Önce db.py dosyasından db instance'ı import ediyorum.

```
from flaskapp.db import db
```

Flask uygulamasının ayarlarını set ettiğim satırın bir altında db instance'ını initialize edeceğim. Bunun için parametre olarak oluşturduğum core_app değişkenini gönderiyorum. Bu sayede bir SqlAlchemy uygulamasını Flask uygulamamın üzerine inşa ediyor.

```
core_app.config.from_object(Config)
    db.init_app(core_app)
    with core_app.app_context():
```

__init__.py

Artık modellerimizi oluşturabiliriz. Bunun için flaskapp klasörü altında models klasörünü ve içerisine de user.py dosyasını oluşturuyorum. pip install flask_login komutu ile Flask Login paketini sanal ortamıma kuruyorum. Login ve Logout işlemlerini bu paket sayesinde gerçekleştireceğiz. Aynı zamanda paket içerisindeki UserMixin sınıfı sayesinde de User modelimizi oluştururken miras alacağız ki oluşturduğumuz User sınıfı ile oturum işlemlerini yapacağımızı set edeceğiz.

Gerekli fonksiyonları ve sınıfı dosyama import ediyorum.

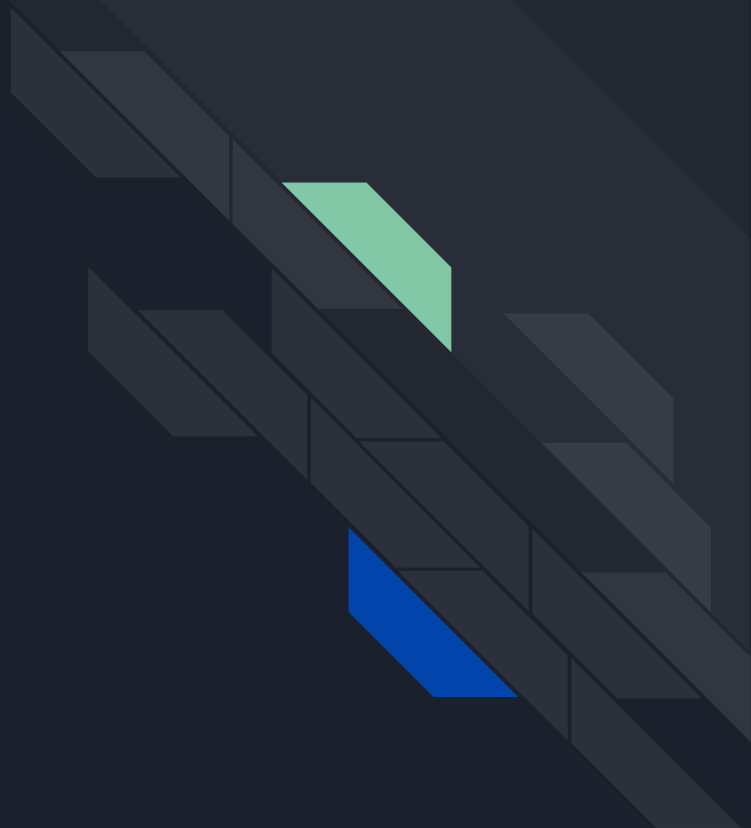
```
from flaskapp.db import db
from flask_login import UserMixin
from werkzeug.security import generate_password_hash, check_password_hash
```

user.py

Konumuz SqlAlchemy ve Login olmadığından burayı da hızlıca geçeceğim. Fakat aşağıdaki kod bloklarını kısaca özetlemem gerekirse UserModel ve RoleModel sınıfları arasında çoka çok bir ilişki tanımlıyoruz. Bunu da role_user tablosu üzerinde gerçekleştiriyoruz. Role ve User modellerimiz temel crud işlemlerini gerçekleştirmelerini sağlayan metodlara sahipler. User modelimiz ekstra olarak is_my_role adında bir metoda sahip. Bu metod sayesinde hangi Dash'lere erişebileceğini kontrol edebileceğiz ve bir template-filter yazarak jinja2 şablon dosyalarımızda linklerin görünürlüğünü ayarlayabileceğiz.

```python
from flaskapp.db import db
from flask_login import UserMixin
from werkzeug.security import generate_password_hash, check_password_hash

role_user = db.Table(
    'role_user',
    db.Column('user_id', db.Integer, db.ForeignKey('users.id'), primary_key=True),
    db.Column('role_id', db.Integer, db.ForeignKey('roles.id'), primary_key=True)
)


class RoleModel(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False, unique=True)

    def __init__(self, _name):
        self.name = _name

    def __repr__(self):
        return self.name
```

```python
    @classmethod
    def find_by_name(cls, name):
        return cls.query.filter_by(name=name).first()

    @classmethod
    def find_by_id(cls, _id):
        return cls.query.filter_by(id=_id).first()

    @classmethod
    def get_all(cls):
        return cls.query.all()

    def save_to_db(self):
        db.session.add(self)
        db.session.commit()

    def delete_from_db(self):
        db.session.delete(self)
        db.session.commit()

    def get_users_count(self):
        return len(self.users)
```



```python
class UserModel(UserMixin, db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String, unique=True, nullable=False)
    password = db.Column(db.String, nullable=False)
    is_admin = db.Column(db.Boolean, nullable=False)
    roles = db.relationship('RoleModel', secondary=role_user, lazy='subquery',
                            backref=db.backref('users', lazy=True))

    def __init__(self, username, password, is_admin):
        self.username = username
        self.password = password
        self.is_admin = is_admin

    def __repr__(self):
        return self.username

    @classmethod
    def find_by_username(cls, username):
        return cls.query.filter_by(username=username).first()
```

```python
    @classmethod
    def find_by_id(cls, _id):
        return cls.query.filter_by(id=_id).first()
    @classmethod
    def get_all(cls):
        return cls.query.all()

    def save_to_db(self):
        db.session.add(self)
        db.session.commit()

    def delete_from_db(self):
        db.session.delete(self)
        db.session.commit()

    def set_password(self, _password):
        self.password = generate_password_hash(_password)

    def check_password(self, _password):
        return check_password_hash(self.password, _password)

    def is_my_role(self, role):
        return role in self.roles
```

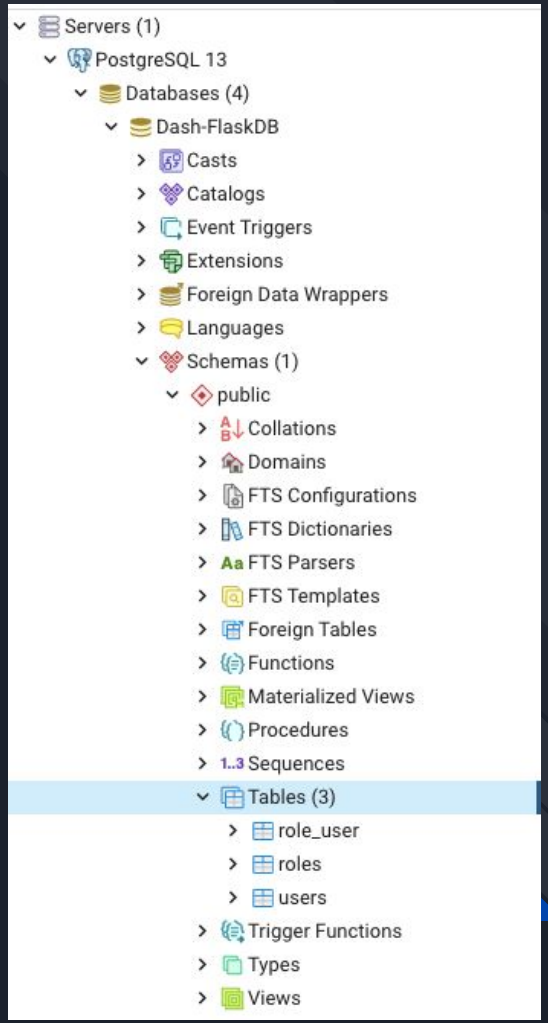
user.py

Modellerimizi yazdığımıza göre şimdi de veri tabanını oluşturmamız lazım. Bunun için flaskapp/app.py dosyama gidiyorum ve Flask-SqlAlchemy ile gelen bir decorator yardımı ile uygulama ilk istek atıldığında istepi henüz karşılamadan önce veri tabanını oluşturuyorum. Bunun için önce oluşturduğum db instance'ını bu dosyaya da import ediyorum.

```
from flaskapp.db import db


@app.before_first_request
def create_tables():
    db.create_all()
```

app.py

Uygulamamı tekrar çalıştırıp (python wsgi.py) ana sayfaya gittikten sonra veri tabanıma baktığımda tabloların oluşmuş olduğunu görmem gerekmektedir (yanda).



Oturum İşlemleri İçin Flask-Login Ayarlayalım

Db instance'ı oluşturduğumuz gibi oturum işlemlerini yönetmek için de bir Flask-Login instance'ı oluşturacağız. Bunun için flaskapp/login.py dosyasını oluşturuyorum.

```
from flask_login import LoginManager


login = LoginManager()
```

login.py

Aynı şekilde __init__.py dosyası içerisinde de Login app'imi initialize edeceğim. Parametre olarak core_app'i (Flask uygulamamız) gönderiyorum. Bu sayede mevcut Flask uygulamasının üzerine bir Flask-Login uygulaması inşa ediyor diyebiliriz.

```
from flaskapp.login import login

    .
    .
    db.init_app(core_app)
    login.init_app(core_app)
    with core_app.app_context():
```

__init__.py

Flask-Login uygulamamızı başarıyla oluşturduk. Şimdi flaskapp/app.py dosyamıza gidip oturum işlemleri için hangi modeli kullanacağımızı ve eğer unauthorize bir işlem (login olmadan) gerçekleşirse nereye redirect olması gerektiğini set edeceğiz. Öncelikle login instance'ımı ve User modelimi dosyama import ediyorum.

```
from flaskapp.models.user import UserModel
from flaskapp.login import login
```

Şimdi de decoratorler yardımı ile hangi modeli kullanacağımı (user_loader) ve unauthorize durumunda nereye yönleneceğini (unauthorized_handler) set ediyorum.

```
from flask import render_template,redirect,url_for

@login.user_loader
def load_user(id):
    return UserModel.query.get(int(id))
@login.unauthorized_handler
def unauthorized_callback():
    return redirect(url_for('login'))
```

app.py

redirect ve url_for fonksiyonlarını kullanabilmek için import ediyorum.

Tabiki henüz redirect olacağımız bir login route'umuz yok. Sırasıyla login ve logout route'larımızı da yazıyorum.
İşimize yarayacak bir kaç fonksiyonu import ediyorum.

```
from flask_login import current_user, login_required, login_user, logout_user
```

Sırasıyla; oturumdaki mevcut kullanıcıyı yakalamaya, oturum açmadan route'a erişememeye, kullanıcıya oturum açtırmaya ve kullanıcının oturumunu sonlandırmaya yararlar.
flask altından flash fonksiyonunu da import ediyorum. Şablonlarımızda bu fonksiyonu kullanmayacağız fakat bu fonksiyon sayesinde şablonlarda mesajlar gösterebildiğimizi bilsek yeter.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')
        user = UserModel.find_by_username(username)
        if user is None or not user.check_password(password):
            flash('Kullanıcı Adı veya Parola Hatalı!', 'danger')
            return render_template('login.jinja2', title='Giriş')
        login_user(user)
        return redirect(url_for('index'))
    return render_template('login.jinja2', title='Giriş')
```

```python
@app.route('/logout')
@login_required
def logout():
    logout_user()
    flash('Başarılı bir şekilde çıkış yapıldı!', 'success')
    return redirect(url_for('index'))
```

app.py

templates klasörü altında login.jinja2 şablon dosyamı oluşturuyorum. Logout olduğunda zaten oturum sonlanacak ve tekrar ana sayfaya redirect olacak.

```
{% extends 'layout.jinja2' %}
{% block content %}
          <div><form id="loginForm" method="post" action="/login">
                    <div><label>Kullanıcı Adı</label><input type="text" id="loginUsername" name="username" required></div>
                    <div><label>Parola</label><input type="password" id="loginPassword" name="password" required></div>
                     <button type="submit">Giriş</button>
              </form></div>
{% endblock %}
```

login.jinja2

Şimdi layout.jinja2 dosyamıza giriş butonu ekleyelim. Eğer kullanıcı sisteme giriş yapmış ise buton çıkış olarak gözüksün ve bizi logout route'una yönlendirsin. Ve eğer kullanıcı giriş yapmamışsa hiç bir Dash ekranına erişemesin. Bunun için navbarımızı biraz güncelliyoruz.

```
<ul class="my-navbar">
    <li><a href="/">Anasayfa</a></li>
    {% if current_user.is_anonymous %}
    <li><a href="/login">Giriş</a></li>
    {% else %}
    <li><a href="/logout">Çıkış</a></li>
     {% for dashboard in get_dashboards %}
          <a href="/dashboard/{{ dashboard['APP_URL'] }}">{{ dashboard['APP_NAME'] }}</a>
    {% endfor %}
    {% endif %}
  </ul>
```

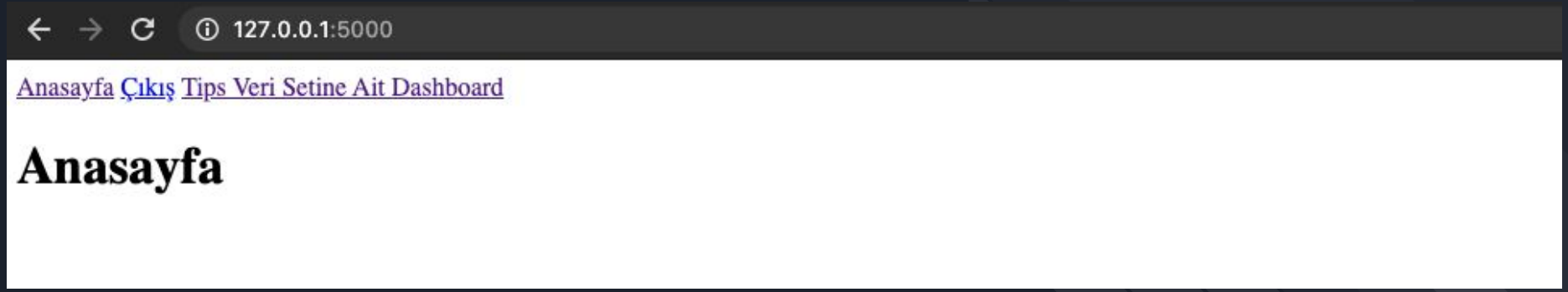
Admin İşlemleri İçin Admin Panelimizi Oluşturalım

Artık kullanıcı ve rol ekleyebileceğimiz admin panelini yazabiliriz. Burası için de bir blueprint oluşturacağız. Bu kısmı da Flask ve Dash'i birleştirmenin konusu olmadığından hızlı bir şekilde geçeceğim. flaskapp/admin klasörünü oluşturuyorum. İçerisine app.py dosyasını oluşturup bir blueprint oluşturuyorum ve aşağıdaki kodu yazıyorum. Kodu özetlemek gerekirse Kullanıcı ve Rol ekleyebildiğimiz ve kullanıcılara roller atayabildiğimiz route'lardan oluşmakta. Bu bölümde devam edebilmek adına flaskapp/app.py dosyasında index route'unda manuel bir admin User oluşturuyorum. UserModel'i import edip idnex fonksiyonu altında manuel bir user oluşturabilirsiniz.

```
from flask import Blueprint, render_template, redirect, url_for, flash, request
from flask_login import current_user, login_required
from flaskapp.models.user import UserModel, RoleModel
from flaskapp.db import db

app = Blueprint('admin', __name__)
```

flaskapp/admin/app.py

Kodu yazmadan önce manuel olarak oluşturduğum kullanıcı ile oturum açıyorum. Bunun sebebi ise admin blueprint'ine login olmadan erişemeyecek olmamızdır (en azından kodun şu anki hali ile). Her şey doğru ise aşağıdaki gibi bir çıktı ile karşılaşacaksınız.



Yandaki gibi manuel olarak User oluşturabiliriz.

```python
@app.route('/')
def index():
    user = UserModel('test','test',True)
    user.set_password('test')
    user.save_to_db()
    return render_template('index.jinja2', title='Anasayfa')
```

```python
@app.route('/')
@login_required
def index():
    if not current_user.is_admin:
        flash('Bu sayfayı görüntülemek için gerekli izne sahip değilsiniz', 'warning')
        return redirect(url_for('index'))
    users = UserModel.get_all()
    roles = RoleModel.get_all()
    return render_template('admin/index.jinja2', users=users, roles=roles)
```

```python
@app.route('/user/add', methods=['POST'])
@login_required
def user_add():
    if not current_user.is_admin:
        flash('Bu sayfayı görüntülemek için gerekli izne sahip değilsiniz', 'warning')
        return redirect(url_for('index'))
    username = request.form.get('username')
    password = request.form.get('password')
    is_admin = True if request.form.get('is_admin') == 'True' else False
    if UserModel.find_by_username(username):
        flash(f'{username} Zaten mevcut! Başka bir kullanıcı adı deneyin', 'warning')
        return redirect(url_for('admin.index'))
    new_user = UserModel(username=username, password=password, is_admin=is_admin)
    new_user.set_password(password)
    new_user.save_to_db()
    flash(f'{username} Kullanıcısı başarıyla eklendi!', 'success')
    return redirect(url_for('admin.index'))
```

```python
@app.route('/user/delete/<int:id>', methods=['POST'])
@login_required
def user_delete(id):
    if not current_user.is_admin:
        flash('Bu sayfayı görüntülemek için gerekli izne sahip değilsiniz', 'warning')
        return redirect(url_for('index'))
    user = UserModel.find_by_id(id)
    user.delete_from_db()
    flash(f'{user.username} Kullanıcısı başarıyla silindi!', 'success')
    return redirect(url_for('admin.index'))
```

```python
@app.route('/user/detail/<int:id>', methods=['GET', 'POST'])
@login_required
def user_detail(id):
    if not current_user.is_admin:
        flash('Bu sayfayı görüntülemek için gerekli izne sahip değilsiniz', 'warning')
        return redirect(url_for('index'))
    user = UserModel.find_by_id(id)
    roles = RoleModel.get_all()
    if request.method == 'POST':
        user.username = request.form.get('username')
        user.is_admin = True if request.form.get('is_admin') == 'True' else False
        if request.form.get('password'):
            user.set_password(request.form.get('password'))
        user.roles.clear()
        roles = request.form.getlist('roles')
        for role in roles:
            user.roles.append(RoleModel.find_by_id(role))
        user.save_to_db()
        flash(f'{user.username} Başarıyla güncellendi!', 'success')
        return redirect(url_for('admin.index'))
    return render_template('admin/user_detail.jinja2', user=user, roles=roles)
```

```python
@app.route('/role/add', methods=['POST'])
@login_required
def role_add():
    if not current_user.is_admin:
        flash('Bu sayfayı görüntülemek için gerekli izne sahip değilsiniz', 'warning')
        return redirect(url_for('index'))
    role_name = request.form.get('role_name')
    if RoleModel.find_by_name(role_name):
        flash(f'{role_name} Zaten mevcut! Başka bir rol adı deneyin', 'warning')
        return redirect(url_for('admin.index'))
    new_role = RoleModel(role_name)
    new_role.save_to_db()
    flash(f'{new_role.name} Rolü başarıyla eklendi!', 'success')
    return redirect(url_for('admin.index'))
```

```python
@app.route('/role/delete/<int:id>', methods=['POST'])
@login_required
def role_delete(id):
    if not current_user.is_admin:
        flash('Bu sayfayı görüntülemek için gerekli izne sahip değilsiniz', 'warning')
        return redirect(url_for('index'))
    role = RoleModel.find_by_id(id)
    role.delete_from_db()
    flash(f'{role.name} Rolü başarıyla silindi!', 'success')
    return redirect(url_for('admin.index'))
```

flaskapp/admin/app.py

Şimdi de templates/admin altında şablon dosyalarımı oluşturuyorum. Routelardan anlaşılacağı gibi 2 adet şablon dosyam olacak. index.jinja2, user_detail.jinja2 Kolay olması için anasayfada hem kullanıcı hem rol ekleyeceğim. Rol silme işlemleri de anasayfadaki rol tablosu üzerinden yapılacak. user_detail üzerinde ise kullanıcılara rol ataması yapacağım. Unutmamaız gereken bir şey var; rol isimleri ile Dash CONFIG sabitlerinde yazdığımız APP_URL keyleri aynı olmalıdır. Buna daha sonra tekrar geleceğiz.

```
{% extends 'layout.jinja2' %}
{% block content %}
<br>
        <div>
            <form method="post" action="/admin/user/add">
                <div>
                    <label>Kullanıcı Adı</label>
                    <input type="text" id="adminUsername" name="username">
                </div>
                <div>
                    <label>Parola</label>
                    <input type="password" id="adminPassword" name="password">
                </div>
```

```
                    <div>
                        <label>Parola</label>
                        <input type="password" id="adminPassword" name="password">
                    </div>
                    <div>
                        <label for="adminAdmin">Admin mi?</label>
                        <select id="adminAdmin" name="is_admin">
                            <option value=False selected>Hayır</option>
                            <option value=True>Evet</option>
                        </select>
                    </div>
                    <button type="submit">Kullanıcı Ekle</button>
                </form>
            </div>
            <br>
            <div>
                <form method="post" action="/admin/role/add">
                    <div>
                        <label>Rol Adı</label>
                        <input type="text" id="adminRoleName" name="role_name">
                        <small><b>Rol adları dash CONFIG sabitleri içerisindeki APP_URL değişkenleri ile
aynıolmalıdır</b></small>
                    </div>
```

```
                    <button type="submit">Rol Ekle</button>
                </form>
            </div>
        <br><br>
            <div>
             <h4>Kullanıcılar</h4>
             <table>
                <thead>
                <tr>
                    <th>ID</th>
                    <th>Kullanıcı Adı</th>
                    <th>Admin mi?</th>
                    <th></th>
                    <th></th>
                </tr>
                </thead>
                <tbody>
                {% for user in users %}
                    <tr>
                        <td>{{ user.id }}</td>
                        <td>{{ user.username }}</td>
```

```
                        <td>
                            {% if user.is_admin %}
                                Evet
                            {% else %}
                                Hayır
                            {% endif %}
                        </td>
                        <td><a href="/admin/user/detail/{{ user.id }}">Detay</a></td>
                        <td>
                            <form action="/admin/user/delete/{{ user.id }}" method="post">
                                <button type="submit">Sil</button>
                            </form>

                        </td>
                    </tr>
                {% endfor %}
                </tbody>
            </table>
        </div>
        <div>
```

```
                <h4>Roller</h4>
                <table>
                    <thead>
                    <tr>
                        <th>ID</th>
                        <th>Rol Adı</th>
                        <th></th>
                    </tr>
                    </thead>
                    <tbody>
                    {% for role in roles %}
                        <tr>
                            <td>{{ role.id }}</td>
                            <td>{{ role.name }}</td>
                            <td>
                                <form action="/admin/role/delete/{{ role.id }}" method="post">
                                    <button type="submit">Sil</button>
                                </form>
                            </td></tr>{% endfor %}</tbody></table></div>
{% endblock %}
```

templates/admin/index.jinja2

```
{% extends 'layout.jinja2' %}
{% block content %}
            <div>
                <form method="post" action="/admin/user/detail/{{ user.id }}">
                    <div>
                        <label>Kullanıcı Adı</label>
                        <input type="text" id="adminUsername" name="username"
                               value="{{ user.username }}">
                    </div>
                    <div>
                        <label>Parola</label>
                        <input type="password" id="adminPassword" name="password">
                    </div>
                    <div>
                        <label>Admin mi?</label>
                        <select id="adminAdmin" name="is_admin">
                            {% if user.is_admin %}
                                <option value=True selected>Evet</option>
                                <option value=False>Hayır</option>
                            {% else %}
                                <option value=False selected>Hayır</option>
                                <option value=True>Evet</option>
                            {% endif %}
```

```
                        </select>
                    </div>
                    <div>
                        <label>Roller</label>
                        <select id="userRoles" name="roles" multiple>
                            {% for role in roles %}
                                <option value="{{ role.id }}" {% if role in user.roles %} selected {%
endif %}>{{ role.name }}</option>
                            {% endfor %}
                        </select>
                    </div>
                    <button type="submit">Kullanıcı Güncelle
                    </button>
                </form>
            </div>
{% endblock %}
```

templates/admin/user_detail.jinja2

Şimdi admin sayfamıza ulaşmak için templates/layout.jinja2 dosyamızı biraz güncelleyelim ve admin linkini ekleyelim. Kullanıcı eğer admin ise admin sayfasına ait linkleri görebilsin.

```
<ul class="my-navbar">
    <li><a href="/">Anasayfa</a></li>
    {% if current_user.is_anonymous %}
    <li><a href="/login">Giriş</a></li>
    {% else %}
    {% if current_user.is_admin %}
    <li><a href="/admin">Admin İşlemleri</a></li>
    {% endif %}
     {% for dashboard in get_dashboards %}
          <a href="/dashboard/{{ dashboard['APP_URL'] }}">{{ dashboard['APP_NAME'] }}</a>
    {% endfor %}
    <li><a href="/logout">Çıkış</a></li>
    {% endif %}
  </ul>
```

templates/layout.jinja2

Tabii bu blueprinti aktif etmek için ana Flask uygulamamıza tanıtmamız gerekmekte. Bunun için flaskapp/app.py dosyasına gidiyorum ve blueprinti import ediyorum.

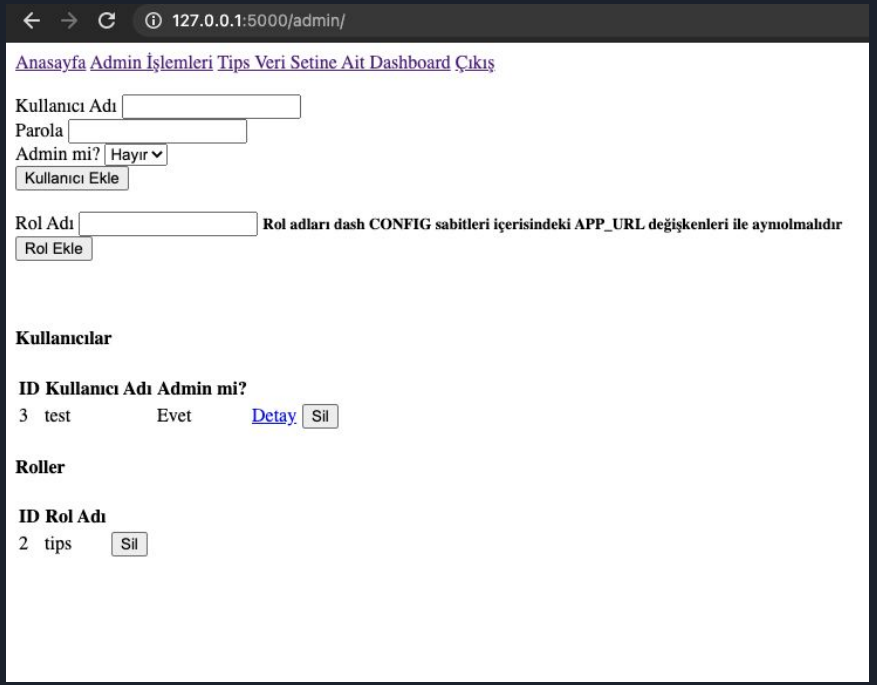
```
from flaskapp.admin.app import app as admin_bp
```

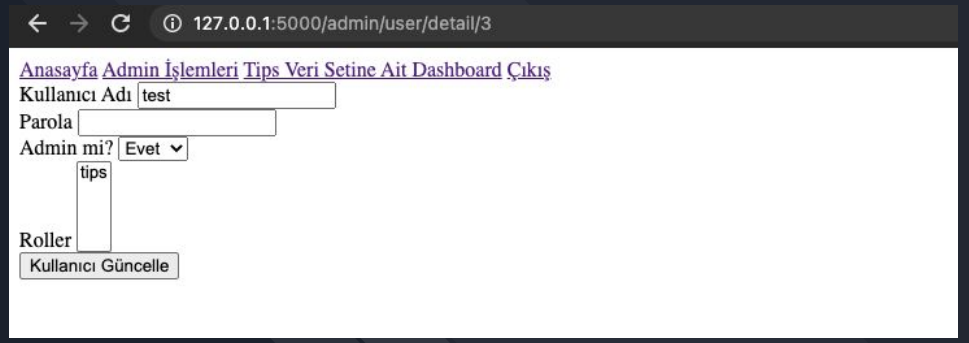
blueprinti Flask uygulamama kayıt ediyorum.

```
.
.
.
app.register_blueprint(admin_bp, url_prefix='/admin')
```

flaskapp/app.py

Bu uygulamada stillerin önceliğimiz olmadığından boş geçtiğimizi tekrar hatırlatmak istiyorum. Lütfen stil dosyalarına takılmayınız. Eğer bu adıma kadar geldiysek admin index ve user detail sayfalarının çıktısı şu şekilde olacaktır. Önceki sayfalarda bir user ekleyip login olduğumuzu hatırlatmak isterim.





Ben tips adında bir rol ekledim. Bu string bize nereden tanıdık gelmekte? Dash uygulaması oluştururken APP_URL key'i içerisinde bu stringi set etmiştik. Aslında APP_URL keyleri bizim rollerimize karşılık gelmekte. Şimdi bu rol kontrolünü yapalım.

Dash Ekranlarına Erişim İçin Template Filter Yazalım

flaskapp klasörü altında template_filters.py dosyasını oluşturuyorum. Bu dosya altında template filterlerimi yazacağım. Bu kodu özetleyecek olursak; bu filtreye parametre olarak gelen rol adı (APP_URL) mevcut oturum açmış kullanıcının rolleri içerisinde var mı ona bakacak. Eğer varsa True dönecek (is_my_role metodundan). Bu sayede eğer rol kullanıcıya tanımlı ise navbarda bu Dash'e ait link gözükecek.

```
from flask import current_app as app
from flask_login import current_user
from flaskapp.models.user import RoleModel


@app.template_filter()
def is_my_role(role_url):
    role = RoleModel.find_by_name(role_url)
    return current_user.is_my_role(role)
```

flaskapp/template_filters.py

Aynı zamanda yazdığımız bu template_filters.py dosyasını uygulama çalıştırılırken import etmemiz gerektiğinden __init__.py dosyası içerisinde import ediyorum.

```
with core_app.app_context():
        from flaskapp import app,context_processors,template_filters
```

flaskapp/__init__.py

Şimdi yazdığımız tempalte_filter'i navbar içinde uygulayalım. get_dashboards context processor'u bize dict'lerden oluşan bir liste dönüyordu bunu biliyoruz. Döndüğümüz dict'in APP_URL key'i is_my_role template filter'ına gönderiyoruz. Eğer True dönerse bu link bize gözükecektir.

```
<ul class="my-navbar">
    <li><a href="/">Anasayfa</a></li>
    {% if current_user.is_anonymous %}
    <li><a href="/login">Giriş</a></li>
    {% else %}
    {% if current_user.is_admin %}
    <li><a href="/admin">Admin İşlemleri</a></li>
    {% endif %}
     {% for dashboard in get_dashboards %}
      {% if  dashboard['APP_URL'] | is_my_role %}
          <a href="/dashboard/{{ dashboard['APP_URL'] }}">{{ dashboard['APP_NAME'] }}</a>
          {% endif %}
    {% endfor %}
    <li><a href="/logout">Çıkış</a></li>
    {% endif %}
  </ul>
```

Bu ekranda gördüğümüz gibi navbarda Tips Dash'i için link gözükmemekte. Çünkü henüz bu rolü (APP_URL) test user'ına (mevcut oturum açmış) eklemedik.

Şimdi tips'i seçip kullanıcıyı güncelliyoruz ve navbar'da Dash'e ait linkin açıldığını görüyoruz.

Peki neden tips adında bir rol ekledik?
Dash uygulamasını oluştururken yazdığımız CONFIG sabiti içerisindeki APP_URL'leri üzerinden yetkilendirme işlemlerini yapacağımızı söylemiştik. Bu sebeple hangi Dash'e ait izin vermek istiyorsak o APP_URL'i kullanıcıya eklememiz gerekmektedir.

```
CONFIG = { # bu sabit içerisinde bu Dash uygulamas
    'BASE_URL': '/h9M5hdnUFRE8qffkqDUrWdK/tips/',
    'APP_URL': 'tips', # bu key sayesinde rol bazl
    'APP_NAME': 'Tips Veri Setine Ait Dashboard',
    'MIN_HEIGHT': 1500, # iframe'in boyutunu set e
}
```

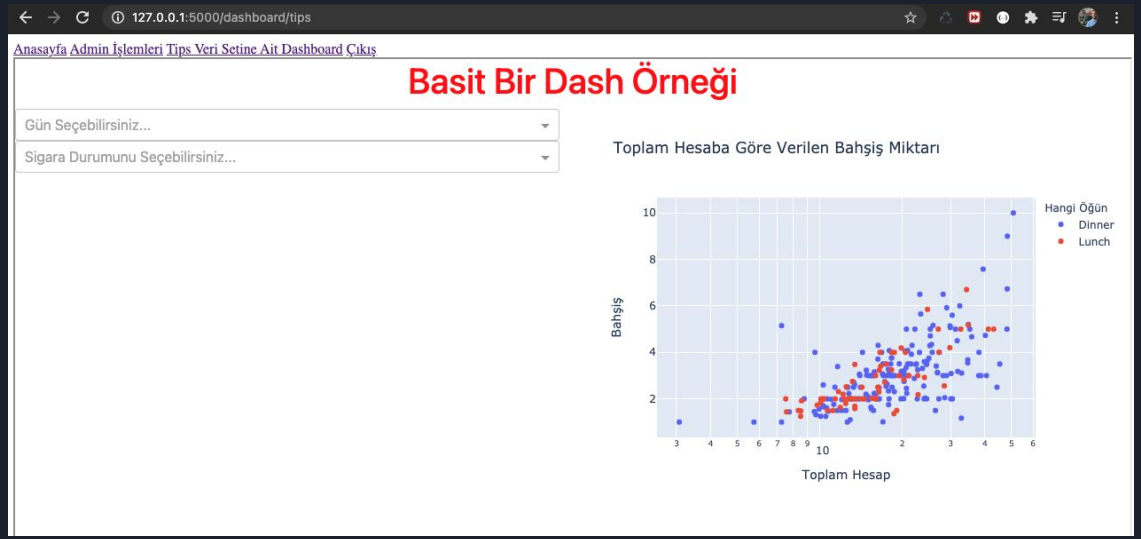
Aynı zamanda her Dash için o APP_URL'i ile aynı isimde bir rol eklememiz gerekmektedir. Çünkü yazdığımız kodlarda Dash url şeması urls.py içerisinden oluşturulmaktadır. Ve bu dosya da tüm CONFIG sabitlerinden oluşan bir listedir. Rolü olmayan bir Dash'e ulaşamayız diyebiliriz.

Dash Ekranlarına Rol Kontrol Uygulayalım

Evet template filter'larımızı yazdık. Fakat bunlar sadece frontend tarafında görünürlüğü kapatıyor. Url üzerinden tekrar bu Dash'lere erişebiliriz. Bunun sebebi ise henüz Dash'lerin döndürüldüğü route'a (`get_dash`) henüz bir rol kontrolü uygulamadık. Şimdi bu route'a kontrol bloğumuzu ekleyelim.

```
@app.route('/dashboard/<string:dash_url>')
@login_required
def get_dash(dash_url):
    CONFIG = next(filter(lambda x: x['APP_URL'] == dash_url, URL_PATHS),None)
    if not CONFIG:
        flash('Aradığınız Dashboard bulunamadı', 'danger')
        return redirect(url_for('index'))
    role = RoleModel.find_by_name(dash_url)
    if current_user.is_my_role(role):
        return render_template('dashboards/dashboard_basic.jinja2', dash_url=CONFIG['BASE_URL'],
                               dash_min_height=CONFIG['MIN_HEIGHT'],title=CONFIG['APP_NAME'])
    else:
        flash('Bu dashboardu görmek için yetkiye sahip değilsiniz', 'warning')
        return redirect(url_for('index'))
```

Bu bloğumuzu açıklayacak olursak; zaten biz CONFIG değişkenine ulaşmak istediğimiz Dash'ın CONFIG sabitini atıyorduk. Aslında APP_URL key'leri tam olarak burada işe yarıyor. RoleModel.find_by_name(dash_url) diyerek parametre olarak gelen APP_URL keyini (çünkü navbarda bu listeyi dönerek dashboard/APP_URL olarak link oluşturuyorduk) roller içerisinde aratıyoruz. Ardından mevcut oturum açmış kullanıcının rolleri içerisinde bu rolün varlığını kontrol ediyoruz. Eğer var ise Dash ekranına erişebiliyor. Eğer parametre olarak gelen APP_URL oturum açmış kullanıcının rolleri arasında yer almıyorsa kullanıcıya bir uyarı döndürüp onu ana sayfaya redirect ediyoruz. Bu sayede rol bazlı olarak Dash'lere erişim sağlamış oluyoruz.



Umarım faydalı olmuştur.

Kodlar: mebaysan/DashVeFlaskBirlestirmek