Tutorial para crear un checador con Python y Flask

Por: Luis Antonio García

Índice

Antes de iniciar

En este tutorial vamos a utilizar Python y Flask, nos estaremos basando con el tutorial de la página oficial de Flask, no seré el blog que viene ahí, si no un checador donde podremos registrar entradas y salidas de los usuarios y agregar/editar usuarios.

Se asume que ya están familiarizados con Python 3, si no es así les recomiendo el tutorial de la página oficial.

Diseño del proyecto

Vamos a crear un directorio llamado ***ProyectoChecador*** e ingresamos a la carpeta:

```
$ mkdir ProyectoChecador
$ cd ProyectoChecador
```

Después crearemos un entorno virtual con virtualenv, llamado ***entorno*** que utilizará Python 3:

```
$ virtualenv -p python3 entorno
```

Activamos el entorno virtual y procedemos a instalar flask:

```
$ source entorno/bin/activate
(entorno) $ pip install flask
```

Configuración de la aplicación

Una aplicación de Flask es una instancia de la clase ***Flask***. Todo lo relacionado con la aplicación, como la configuración y las URL's, se registrará en esta clase.

La forma más sencilla de crear una aplicación Flask es crear una instancia global de Flask directamente en la parte superior de su código. Si bien esto es simple y útil en algunos casos, puede causar algunos problemas difíciles a medida que el proyecto crece.

En lugar de crear una instancia de Flask globalmente, vamos a crearla dentro de una función. Esta función se conoce como la fábrica de aplicaciones (*Application Factory*). Cualquier configuración, registro y otra configuración que la aplicación necesite ocurrirá dentro de la función, luego se devolverá la aplicación.

Fábrica de aplicaciones (*Application Factory*)

Empecemos con el código, vamos a crear una carpeta llamada ***checador*** que contenga el archivo ***__init__.py***, este archivo cumple doble propósito: contendrá la fábrica de aplicaciones y le indica a Python que el directorio ***checador*** debe ser tratado como un paquete.

```
$ mkdir checador
```

checador/__init__.py

```
import os
from flask import Flask

def create_app(test_config=None):
    # Crea y configura la app
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_mapping(
        SECRET_KEY='dev',
        DATABASE=os.path.join(app.instance_path, 'checa.sqlite'),
    )

    if test_config is None:
        # Carga la instancia config.py
        app.config.from_pyfile('config.py', silent=True)
    else:
        app.config.from_mapping(test_config)

    # Nos aseguramos que exista la carpeta instance
    try:
        os.makedirs(app.instance_path)
    except OSError:
        pass

    # Una página simple para probar que funcione
```

```
@app.route('/')
def hola():
    return 'Servidor funcionando!'
return app
```

create_app es la función de la fábrica de aplicaciones. Se le agregará mas en el tutorial, pero ya hace bastante.

1. ***app = Flask(__name__, instance_relative_config=True)*** crea una instancia de Flask.
    - ***__name__*** es le nombre del módulo actual de Python. La aplicación necesita saber dónde se encentra para configurar algunas rutas, y ***__name__*** es una forma conveniente de decirlo.
    - ***instance_relative_config = True*** le dice a la aplicación que los archivos de configuración son relativos a la carpeta de instancias. La carpeta de instancias se encuentra fuera del paquete checador y puede contener datos locales que no deberían comprometerse con el control de versiones, como los secretos de configuración y el archivo de la base de datos.
2. ***app.config.from_mapping()*** establece algunas configuraciones por default que la plaicación utilizará:
    - ***SECRET_KEY*** es utilizado por Flask y extensiones para mantener los datos seguros. Está configurado en 'dev' para proporcionar un valor conveniente durante el desarrollo, pero debe anularse con un valor aleatorio durante la implementación.
    - ***DATABSE*** es la ruta donde se guardará el archivo de base de datos SQLite. Está en ***app.instance_path***, que es la ruta que Flask ha elegido para la carpeta de instancia (que se llamará ***instance***). Aprenderá más sobre la base de datos en la siguiente sección.
3. ***app.config.from_pyfile()*** anula la configuración predeterminada con valores tomados del archivo ***config.py*** en la carpeta de la instancia, si existe. Por ejemplo, cuando se implementa, esto se puede usar para establecer un ***SECRET_KEY*** real.
    - ***test_config*** también se puede pasar a la fábrica y se usará en lugar de la configuración de la instancia. Esto es por si usted hace pruebas, se puedan configurar independientemente de los valores de desarrollo que hayamos configurado.
4. ***os.makedirs()*** asegura que ***app.instance_path*** exista. Flask no crea la carpeta de instancia automáticamente, pero debe crearse porque su proyecto creará el archivo de base de datos SQLite allí.
5. ***@app.route()*** crea una ruta simple para que pueda ver la aplicación funcionando antes de acceder al resto del tutorial. Crea una conexión entre la URL */* y una función que devuelve una respuesta, en este caso la cadena ***'Servidor funcionando!'***.

Correr la aplicación

Ahora vamos a correr nuestra aplicación usando el comando ***flask***. Desde el terminal, le indicaremos a Flask dónde encontrar la aplicación, luego lo ejecutamos en modo de desarrollo (development). Recuerden, que tenemos que estar en el directorio ***ProyectoChecador***, no en el de nuestro paquete ***checador*** con nuestro entorno activado.

El modo de desarrollo muestra un depurador interactivo cada vez que una página genera una excepción, y reinicia el servidor cada vez que realiza cambios en el código. Podemos dejarlo funcionando y volver a cargar la página del navegador mientras sigue el tutorial.

Para Linux y Mac:

```
$ export FLASK_APP=checador
$ export FLASK_ENV=development
$ flask run
```

Para Windows PowerShell, utilice ***set*** en vez de ***export***:

```
> set FLASK_APP = "checador"
> set FLASK_ENV = "development"
> flask run
```

Veremos una salida similar a esto:

```
* Serving Flask app "flaskr"
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 855-212-761
```

Si visita al página http://127.0.0.1:5000/ en su navegador, verá el mensaje 'Servidor funcionando!'

Definir y accesar a la Base de Datos

Nuestra aplicación usará la base de datos SQLite para almacenar nuestra información. Python viene con soporte incorporado para SQLite en el módulo sqlite3.

SQLite es conveniente porque no requiere configurar un servidor de base de datos por separado y está integrado en Python. Sin embargo, si las solicitudes concurrentes intentan escribir en la base de datos al mismo tiempo, se ralentizarán a medida que cada escritura se realice secuencialmente. Las aplicaciones pequeñas no notarán esto. Una vez que crezca, es posible que desee cambiar a una base de datos diferente.

El tutorial no entra en detalles sobre SQL. Si no está familiarizado con él, los documentos de SQLite describen el lenguaje.

Conexión a la Base de Datos

Lo primero que haremos al trabajar con nuestra base de datos SQLite (y la mayoría de las otras librerías de base de datos en Python) será crear una conexión hacia ella. Todas las consultas y operaciones se realizarán utilizando una conexión, que se cierra una vez finalizado el trabajo.

En las aplicaciones web, esta conexión suele estar vinculada a la solicitud (request). Se crea en algún momento mientras se maneja la solicitud (request) y se cierra antes de enviar la respuesta (response).

checador/db.py

```
import sqlite3
import click
from flask import current_app, g
from flask.cli import with_appcontext
from werkzeug.security import generate_password_hash

def get_db():
    # Crea conexión a la base de datos
    if 'db' not in g:
        g.db = sqlite3.connect(
            current_app.config['DATABASE'],
            detect_types=sqlite3.PARSE_DECLTYPES
        )
        g.db.row_factory = sqlite3.Row
    return g.db


def close_db(e=None):
    # Cierra la base de datos
    db = g.pop('db', None)
    if db is not None:
        db.close()
```

g es un objeto especial que es único para cada solicitud. Se utiliza para almacenar datos a los que pueden acceder múltiples funciones durante la solicitud. La conexión se almacena y se reutiliza en lugar de crear una nueva conexión si se llama a ***get_db*** por segunda vez en la misma solicitud.

current_app es otro objeto especial que apunta a la aplicación Flask que maneja la solicitud. Como utilizamos una fábrica de aplicaciones, no hay ningún objeto de aplicación al escribir el resto de su código. Se llamará a ***get_db*** cuando la aplicación se haya creado y esté manejando una solicitud, por lo que se puede usar current_app.

sqlite3.connect() establece una conexión con el archivo al que apunta la clave de configuración ***DATABASE***. Este archivo aún no tiene que existir, y no lo hará hasta que se inicialice la base de datos más tarde.

sqlite3.Row le dice a la conexión que devuelva filas que se comportan como dicts (colecciones de datos organizados por pares clave-valor). Esto permite acceder a las columnas por nombre.

close_db comprueba si se creó una conexión verificando si se configuró ***g.db***. Si la conexión existe, se cerrará. Después le indicaremos a nuestra aplicación sobre la función ***close_db*** en la fábrica de aplicaciones para llamarla después de cada solicitud.

Creando las tablas

En SQLite, los datos se almacenan en tablas y columnas. Estos deben crearse antes de que pueda almacenar y recuperar datos. Nosotros vamos a almacenar usuarios en la tabla de usuarios y checadas en la tabla de checadas. Cree un archivo con los comandos SQL necesarios para crear tablas vacías:

checador/schema.sql

```
DROP TABLE IF EXISTS user;
DROP TABLE IF EXISTS checks;

CREATE TABLE user (
  id INTEGER PRIMARY KEY AUTOINCREMENT,                    -- ID del usuario
  username TEXT UNIQUE NOT NULL,                           -- Usuario
  password TEXT NOT NULL,                                  -- Password de la cuenta
  email TEXT,                                              -- Correo electrónico
  nombre TEXT NOT NULL,                                    -- Nombre del usuario
  ap_1 TEXT NOT NULL,                                      -- Primer apelido
  ap_2 TEXT,                                               -- Segundo apellido
  fecha_creado DATETIME NOT NULL DEFAULT (datetime(CURRENT_TIMESTAMP,
'localtime')),     -- Fecha de creación
  userid_crea INTEGER,                                     -- ID de usuario que creó
  baja INTEGER NOT NULL DEFAULT 0                          -- Indica si el usuario está dado de baja
);

CREATE TABLE checks (
  id INTEGER PRIMARY KEY AUTOINCREMENT,                    -- ID de la checada
  user_id INTEGER NOT NULL,                                -- ID de usuario que checa
```

```
    checa_entrada DATETIME NOT NULL DEFAULT (datetime(CURRENT_TIMESTAMP,
'localtime')),     -- Fecha y hora de entrada
    checa_salida DATETIME,                              -- Fecha y hora de salida
    baja INTEGER NOT NULL DEFAULT 0                     -- Inidca si la checada está dada de baja
);
```

Agregue las funciones de Python que ejecutarán estos comandos SQL al archivo db.py:

checador/db.py

```
def init_db():
    # Inicializa la BD
    db = get_db()
    with current_app.open_resource('schema.sql') as f:
        db.executescript(f.read().decode('utf8'))



def init_user():
    # Creamos el usuario admin
    db = get_db()
    db.execute(
        'INSERT INTO user ( username, password, nombre, ap_1 )'
        ' VALUES ( ?, ?, ?, ? )', ('admin', generate_password_hash('admin'), 'Administrador', 'Del
Sistema'))
    db.commit()



@click.command('init-db')
@with_appcontext
def init_db_command():
    '''Borra la información existente y crea nuevas tablas.'''
    init_db()
    click.echo('Base de datos inicializada...')
    init_user()
    click.echo('Usuario "admin" creado.')
```

open_resource() abre un archivo relativo al paquete ***checador***, lo cual es útil ya que no necesariamente sabrá dónde está esa ubicación cuando implemente la aplicación. ***get_db*** devuelve una conexión de base de datos, que se usa para ejecutar los comandos leídos del archivo.

click.command() define un comando de línea de comando llamado ***init-db*** que llama a la función ***init_db*** y muestra un mensaje de éxito al usuario. Puede leer la ***interfaz de línea de comandos*** para obtener más información sobre cómo escribir comandos.

Agregamos una función que se llama ***init_user()*** la cual nos crea el usuario "***admin***" y la contraseña "***admin***" por default, ya que más adelante le indicaremos a nuestro programa que solo los usuarios registrados puedan "***registrar***" a otros usuarios.

Registrarse con la aplicación

Las funciones ***close_db*** e ***init_db_command*** deben registrarse con la instancia de la aplicación; de lo contrario, la aplicación no los utilizará. Sin embargo, dado que está utilizando una función de fábrica, esa instancia no está disponible al escribir las funciones. En su lugar, escriba una función que tome una aplicación y realice el registro.

checador/db.py

```
def init_app(app):
    app.teardown_appcontext(close_db)
    app.cli.add_command(init_db_command)
```

app.teardown_appcontext() le dice a Flask que llame a esa función al limpiar después de devolver la respuesta.

app.cli.add_command() agrega un nuevo comando que se puede llamar con el comando ***flask***.

Vamos importar y llamar a esta función desde la fábrica de aplicacions. Coloque el nuevo código al final de la función de fábrica antes de devolver la aplicación (return app).

checador/__init__.py

```
def create_app(test_config=None):
    # Crea y configura la app
    app = …….
    # código existente

    # Registramos la función init_app
    from . import db
    db.init_app(app)

    return app
```

Inicializar el archivo de base de datos

Ahora que ***init-db*** se ha registrado con la aplicación, se puede llamar con el comando ***flask***, similar al comando ***run*** que vimos anteriormente (flask run).

Nota: Si todavía está corriendo el servidor, puede detenerlo o ejecutar este comando en una nueva terminal. Si usa una nueva terminal, recuerde cambiar al directorio de su proyecto y activar el entorno como se describe en Activar el entorno. También deberá configurar ***FLASK_APP*** y ***FLASK_ENV*** como se mostró anteriormente.

Ejecute el comando ***init-db***:

```
$ flask init-db
Base de datos inicializada…
Usuario "admin" creado.
```

Ahora habrá un archivo ***checa.sqlite*** en la carpeta ***instance*** de su proyecto.

Planos (Blueprints) y Vistas (Views)

Una función de vista es el código que escribimos para responder a las solicitudes de su aplicación. Flask usa patrones para hacer coincidir la URL de solicitud entrante con la vista que debería manejarlo. La vista devuelve datos que Flask convierte en una respuesta saliente. Flask también puede ir en la otra dirección y generar una URL a una vista basada en su nombre y argumentos.

Crear un Blueprint

Un Blueprint es una forma de organizar un grupo de vistas relacionadas y otro código. En lugar de registrar vistas y otro código directamente con una aplicación, se registran con un plano. Luego, el plano se registra con la aplicación cuando está disponible en la función de fábrica (nuestro __init__.py).

Nuestro ***checador*** tendrá dos planos, uno para las funciones de autenticación y otro para las funciones de checadas de los usuarios. El código para cada plano irá en un módulo separado.

checador/auth.py

```
import functools
from flask import (
    Blueprint, flash, g, redirect, render_template, request, session, url_for
)
from werkzeug.security import check_password_hash, generate_password_hash
from checador.db import get_db

bp = Blueprint('auth', __name__, url_prefix='/auth')
```

Esto crea un ***Blueprint*** llamado ***'auth'***. Al igual que el objeto de la aplicación, el blueprint necesita saber dónde está definido, por lo que ***__name__*** se pasa como segundo argumento. ***url_prefix*** se antepondrá a todas las URL asociadas con el blueprint.

Ahora vamos a importar y registrar nuestro blueprint en nuestra fábrica (__init_.py) usando ***app.register_blueprint()***. Coloque el nuevo código al final de la función de fábrica antes de devolver la aplicación (return app).

checador/__init__.py

```
def create_app(test_config=None):
    # Crea y configura la app
    app = …….
    # código existente

    # Registramos nuestro Blueprint
    from . import auth
    app.register_blueprint(auth.bp)

    return app
```

El blueprint de autenticación tendrá vistas para registrar nuevos usuarios y para iniciar y cerrar sesión.

La primera vista: Registrarse

Cuando el usuario visita la URL ***/auth/register***, la vista de ***register*** devolverá HTML con un formulario para que lo completen. Cuando envían el formulario, validará su entrada y mostrará el formulario nuevamente con un mensaje de error o creará el nuevo usuario e irá a la página de inicio de sesión.

Por ahora solo escribiremos el código de vista. Más adelante escribiremos las plantillas para generar el formulario HTML.

checador/auth.py

```
@bp.route('/register', methods=('GET', 'POST'))
def register():
    # Validamos si el método fue POST
    if request.method == 'POST':
        # Igualamos las variables del formulario
        username = request.form['username']
        password = request.form['password']
        email = request.form['email']
        nombre = request.form['nombre']
        ap_1 = request.form['ap_1']
        ap_2 = request.form['ap_2']

        db = get_db()
        error = None
        # Verificamos que no haya errores
        if not username:
            error = 'El usuario es requerido.'
        elif not password:
            error = 'El password es requerido.'
        elif not nombre:
            error = 'El nombre es requerido.'
        elif not ap_1:
            error = 'El primer apellido es requerido.'
        elif db.execute(
            'SELECT id FROM user WHERE username = ?', (username,)
        ).fetchone() is not None:
            error = 'El usuario “{}” ya se encuentra registrado.'.format(username)

        if error is None:
            # Si no hubo errores procedemos insertar la info en la base de datos
            db.execute(
                'INSERT INTO user (username, password, email, nombre, ap_1, ap_2)'
                ' VALUES (?, ?, ?, ?, ? ,?)', (username,
                generate_password_hash(password), email, nombre, ap_1, ap_2)
            )
```

```
            db.commit()
            # Retornamos la vista para hacer login
            return redirect(url_for('auth.login'))
        flash(error)
    # Si el método no fue POST, regresamos el formulario de registro
    return render_template('auth/register.html')
```

Esto es lo que está haciendo la función de vista de ***register***:

1. ***@bp.route*** asocia la URL ***/register*** con la función de vista ***register***. Cuando Flask recibe una solicitud para ***/auth/register***, llamará a la vista de registro y usará el valor de retorno como respuesta.
2. Si el usuario envió el formulario, ***request.method*** será ***'POST'***. En este caso, comienza a validar la entrada.
3. ***request.form*** es un tipo especial de asignación de ***dict*** con claves y valores de formulario enviados. El usuario ingresará su nombre de usuario, contraseña, correo, nombre, primer apellido y segundo apellido.
4. Valida que el nombre de usuario, contraseña, nombre y primer apellido no estén vacíos.
5. Valida que el nombre de usuario aún no esté registrado consultando la base de datos y verificando si se devuelve un resultado. ***db.execute*** toma una consulta SQL con el marcador **?** para cualquier entrada del usuario y una tupla de valores para reemplazar los marcadores. La biblioteca de la base de datos se encargará de escapar de los valores para que no sea vulnerable a un ataque de inyección SQL.

   fetchone() devuelve una fila de la consulta. Si la consulta no devolvió resultados, devuelve **None**. Más adelante, se usrá ***fetchall()***, que devuelve una lista de todos los resultados.
6. Si la validación se realiza correctamente, inserta los nuevos datos de usuario en la base de datos. Por seguridad, las contraseñas nunca deben almacenarse directamente en la base de datos. En su lugar, ***generate_password_hash()*** se usa para hacer un hash de forma segura a la contraseña, y ese hash se almacena. Como esta consulta modifica los datos, se debe llamar a ***db.commit()*** para guardar los cambios.
7. Después de almacenar al usuario, son redirigidos a la página de inicio de sesión. ***url_for()*** genera la URL para la vista de inicio de sesión en función de su nombre. Esto es preferible a escribir la URL directamente, ya que le permite cambiar la URL más adelante sin cambiar todo el código que la vincula. ***redirect()*** genera una respuesta de redireccionamiento a la URL generada.
8. Si la validación falla, los errores se muestran al usuario. ***flash()*** almacena mensajes que se pueden recuperar al representar la plantilla.
9. Cuando el usuario navega inicialmente a ***auth/register***, o hubo un error de validación, se debe mostrar una página HTML con el formulario de registro. ***render_template()*** mostrará una plantilla que contiene el HTML, que escribiremos mas adelante en el tutorial.

Login

Esta vista sigue el mismo patrón que la vista de registro anterior.

checador/auth.py

```
@bp.route('/login', methods=('GET', 'POST'))
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None
        user = db.execute(
            'SELECT * FROM user WHERE username = ?', (username,)
        ).fetchone()

        if user is None:
            error = 'La información proporcionada no es correcta.'
        elif not check_password_hash(user[‘password’], password):
            error = 'La información proporcionada no es correcta.'

        if error is None:
            session.clear()
            session['user_id'] = user['id']
            return redirect(url_for('index'))
        flash(error)
    return render_template('auth/login.html')
```

Existen algunas diferencias con respecto a la vista ***register***:

1. Primero se consulta al usuario y se almacena en una variable para su uso posterior.
2. ***check_password_hash()*** codifica la contraseña enviada de la misma manera que el hash almacenado y las compara de forma segura. Si coinciden, la contraseña es válida.
3. ***session*** es un ***dict*** que almacena datos entre solicitudes. Cuando la validación se realiza correctamente, el ***id*** del usuario se almacena en una nueva sesión. Los datos se almacenan en una cookie que se envía al navegador, y el navegador luego lo devuelve con solicitudes posteriores. Flask firma de forma segura los datos para que no puedan ser manipulados.

Ahora que el ***id*** del usuario está almacenada en la sesión (***session***), estará disponible en solicitudes posteriores. Al comienzo de cada solicitud, si un usuario ha iniciado sesión, su información debe cargarse y ponerse a disposición de otras vistas.

checador/auth.py

```
@bp.before_app_request
def load_loged_in_user():
    user_id = session.get('user_id')
```

```
    if user_id is None:
        g.user = None
    else:
        g.user = get_db().execute(
            'SELECT * FROM user WHERE id = ?', (user_id,)
        ).fetchone()
```

bp.before_app_request() registra una función que se ejecuta antes de la función de vista, sin importar qué URL se solicite. ***load_logged_in_user*** comprueba si un ***id*** de usuario está almacenado en la ***sesión*** y obtiene los datos de ese usuario de la base de datos, almacenándolos en ***g.user***, que dura el tiempo de la solicitud. Si no hay una identificación de usuario, o si la identificación no existe, ***g.user*** será ***None***.

Logout

Para cerrar sesión, debe eliminar el id de usuario de la ***sesión***. Después, ***load_logged_in_user*** no cargará al un usuario en solicitudes posteriores.

checador/auth.py

```
@bp.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('index'))
```

Requerir autenticación en otras vistas

Crear, editar y eliminar usuarios requerirá que un usuario inicie sesión. Se puede usar un decorador para verificar esto para cada vista a la que se aplica.

checador/auth.py

```
def login_required(view):
    @functools.wraps(view)
    def wrapped_view(**kwargs):
        if g.user is None:
            return redirect(url_for('auth.login'))
        return view(**kwargs)
    return wrapped_view
```

Este decorador devuelve una nueva función de vista que envuelve la vista original a la que se aplica. La nueva función comprueba si un usuario está cargado (en ***g.user***) y, de lo contrario, redirige a la página de inicio de sesión. Si se carga un usuario, se llama a la vista original y continúa normalmente. Usaremos este decorador cuando hagamos las vistas del checador.

Endpoints y URLs

La función ***url_for()*** genera la URL a una vista basada en un nombre y argumentos. El nombre asociado con una vista también se denomina ***endpoint*** y, por default, es el mismo que el nombre de la función de vista.

Por ejemplo, la vista ***hola()*** que se agregó a la fábrica de aplicaciones anteriormente en el tutorial tiene el nombre ***'hola'*** y se puede vincular con ***url_for ('hola')***. Si tomó un argumento, que veremos más adelante, se vincularía al uso de ***url_for ('hola', quien='Mundo')***.

Cuando se usa un blueprint, el nombre del blueprint se antepone al nombre de la función, por lo que el ***endpoint*** para la función de inicio de sesión que escribimos anteriormente es ***'auth.login'*** porque lo agregó al blueprint ***'auth'***.

Plantillas (Templates)

Hemos escrito las vistas de autenticación para su aplicación, pero si está ejecutando el servidor e intenta acceder a cualquiera de las URL, verá un error ***TemplateNotFound***. Esto se debe a que las vistas están llamando a ***render_template()***, pero aún no hemos escrito las plantillas. Los archivos de plantilla se almacenarán en el directorio ***templates*** dentro del paquete ***checador***.

Las plantillas son archivos que contienen datos estáticos, así como marcadores de posición para datos dinámicos. Una plantilla se representa con datos específicos para producir un documento final. Flask usa la biblioteca de plantillas ***Jinja*** para renderizar plantillas.

En nuestra aplicación, utilizaremos plantillas para representar ***HTML*** que se mostrará en el navegador del usuario. En Flask, Jinja está configurado para ***escapar automáticamente*** cualquier dato que se represente en plantillas HTML. Esto significa que es seguro representar la entrada del usuario; los caracteres que hayan ingresado y que puedan interferir con el HTML, como < y >, se escaparán con valores seguros que se ven iguales en el navegador pero no causan efectos no deseados.

Jinja se ve y se comporta principalmente como Python. Se usan delimitadores especiales para distinguir la sintaxis de Jinja de los datos estáticos en la plantilla. Cualquier cosa entre ***{{*** y ***}}*** es una expresión que se enviará al documento final. ***{%*** y ***%}*** denota una declaración de flujo de control como ***if*** y ***for***. A diferencia de Python, los bloques se denotan mediante etiquetas de inicio y fin en lugar de sangría, ya que el texto estático dentro de un bloque podría cambiar la sangría.

El diseño base

Cada página de la aplicación tendrá el mismo diseño básico alrededor de un cuerpo diferente. En lugar de escribir la estructura HTML completa en cada plantilla, cada plantilla ***extenderá*** una plantilla base y se antepondrán secciones específicas.

checador/templates/base.html

```
<!DOCTYPE html>
<title>{% block title %}{% endblock %} - Checador</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<nav>
  <h1>Checador</h1>
  <ul>
    {% if g.user %}
      <li><span>{{ g.user['username'] }}</span></li>
      <li><a href="{{ url_for('auth.logout') }}">Cerrar sesión</a></li>
    {% else %}
      <li><a href="{{ url_for('auth.register') }}">Registrarse</a></li>
      <li><a href="{{ url_for('auth.login') }}">Iniciar sesión</a></li>
    {% endif %}
  </ul>
</nav>
<section class="content">
  <header>
```

```
    {% block header %}{% endblock %}
  </header>
  {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
  {% endfor %}
  {% block content %}{% endblock %}
</section>
```

g está disponible automáticamente en las plantillas. Según si se establece ***g.user*** (de ***load_logged_in_user***), se muestran el nombre de usuario y un enlace de cierre de sesión, o se muestran enlaces para registrarse e iniciar sesión. ***url_for()*** también está disponible automáticamente y se usa para generar URLs para vistas en lugar de escribirlas manualmente.

Después del título de la página y antes del contenido, la plantilla recorre cada mensaje devuelto por ***get_flashed_messages()***. Utilizamos ***flash()*** en las vistas para mostrar mensajes de error, y este es el código que los mostrará.

Hay tres bloques definidos aquí que se antepondrán en las otras plantillas:

1. ***{% block title %}*** cambiará el título que se muestra en la pestaña del navegador y el título de la ventana.
2. ***{% block header %}*** es similar al título pero cambiará el título que se muestra en la página.
3. ***{% block content %}*** es donde va el contenido de cada página, como el formulario de inicio de sesión o para hacer una checada.

La plantilla base está directamente en el directorio ***templates***. Para mantener lo demás organizados, las plantillas para un blueprint se colocarán en un directorio con el mismo nombre que el blueprint.

Registro

checador/templates/auth/register.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Registro de usuarios{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="username">*Usuario</label>
    <input name="username" id="username" required>
    <label for="password">*Contraseña</label>
    <input type="password" name="password" id="password" required>
    <label for="email">Correo electrónico</label>
    <input type="email" name="email" id="email">
    <label for="nombre">*Nombre</label>
    <input type="text" name="nombre" id="nombre" required>
```

```
  <label for="ap_1">*Primer apellido</label>
  <input name="ap_1" id="ap_1" required>
  <label for="ap_2">Segundo apellido</label>
  <input name="ap_2" id="ap_2">
  <input type="submit" value="Registrarse">
 </form>
{% endblock %}
```

{% extends 'base.html' %} le dice a Jinja que esta plantilla debería reemplazar los bloques de la plantilla base. Todo el contenido procesado debe aparecer dentro de las etiquetas ***{% block %}*** que anteponen los bloques de la plantilla base.

Un patrón útil utilizado aquí es colocar ***{% block title %}*** dentro de ***{% block header %}***. Esto establecerá el bloque de título y luego generará el valor en el bloque de encabezado, de modo que tanto la ventana como la página compartan el mismo título sin escribirlo dos veces.

Las etiquetas de ***input*** están usando aquí el atributo ***required***. Esto le indica al navegador que no envíe el formulario hasta que se completen esos campos. Si el usuario está usando un navegador más antiguo que no admite ese atributo, o si está usando algo además de un navegador para hacer solicitudes, aún queremos validar los datos en la vista de Flask. Es importante validar siempre completamente los datos en el servidor, incluso si en el cliente también se realiza alguna validación.

Log In

Es similar a la plantilla de registro, excepto por el título y el botón de submit.

checador/templates/auth/login.html

Registrar un usuario

Ahora que las plantillas de autenticación están escritas, podemos registrar un usuario. Asegúrese de que el servidor aún se esté ejecutando (***flask run*** si no lo está), luego vaya a http://127.0.0.1:5000/auth/register.

Intente hacer clic en el botón "Registrarse" sin completar el formulario y verifique que el navegador muestre un mensaje de error. Intente eliminar los atributos ***required*** de la plantilla ***register.html*** y haga clic en "Registrarse" nuevamente. En lugar de que el navegador muestre un error, la página se volverá a cargar y se mostrará el error de ***flash()*** en la vista.

Complete un nombre de la información de registro y será redirigido a la página de inicio de sesión. Intente ingresar un nombre de usuario incorrecto o el nombre de usuario y la contraseña incorrectos. Si inicia sesión, recibirá un error porque todavía no hay una vista del ***index*** a la que nos redirige.

Archivos estáticos

Las vistas y plantillas de autenticación funcionan, pero se ven muy simples en este momento. Se puede agregar algo de CSS para agregar estilo al diseño HTML que hemos construido. El estilo no cambiará, por lo que es un archivo estático en lugar de una plantilla.

Flask agrega automáticamente una vista estática que toma una ruta relativa al directorio ***checador/static*** y la sirve. La plantilla ***base.html*** ya tiene un enlace al archivo style.css:

```
{{url_for ('static', filename = 'style.css')}}
```

Además de CSS, otros tipos de archivos estáticos pueden ser archivos con funciones de JavaScript o una imagen de logotipo. Todos se colocan en el directorio ***checador/static*** y se hace referencia con ***url_for ('static', filename = '...')***.

Este tutorial no nos centraremos en cómo escribir CSS, yo utilizaré ***Bootstrap***, si desean personalizar a su gusto lo pueden hacer dentro del archivo ***style.css***, yo no utilizaré ese archivo, para cualquier duda que tengan la documentación de Bootstrap es bastante buena, recuerden descargar el CSS de bootstrap y guardarlo en la carpeta staic, les muestro cómo quedaron mis archivos de html:

checador/templates/base.html

```
<!DOCTYPE html>
<title>{% block title %}{% endblock %} - Checador</title>
<link rel="stylesheet" href="{{ url_for('static', filename='bootstrap.min.css') }}">

<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <h1>Checador</h1>
  <div class="collapse navbar-collapse" id="navbarNavAltMarkup"></div>
    <ul class="navbar-nav">
      {% if g.user %}
      <li class="nav-item active">
        <span class="nav-link">{{ g.user['username'] }}</span>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="{{ url_for('auth.logout') }}">Cerrar sesión</a>
      </li>
      {% else %}
      <li class="nav-item">
        <a class="nav-link" href="{{ url_for('auth.register') }}">Registrarse</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="{{ url_for('auth.login') }}">Iniciar sesión</a>
      </li>
      {% endif %}
    </ul>
  </div>
</nav>
```

```
<section class="container">
  <header>
    {% block header %}{% endblock %}
  </header>
  <div class="form-row">
    {% for message in get_flashed_messages() %}
    <div class="alert alert-danger col-md-6" role="alert">{{ message }}</div>
    {% endfor %}
  </div>
  <div class="container">
  {% block content %}{% endblock %}
  </div>
</section>
```

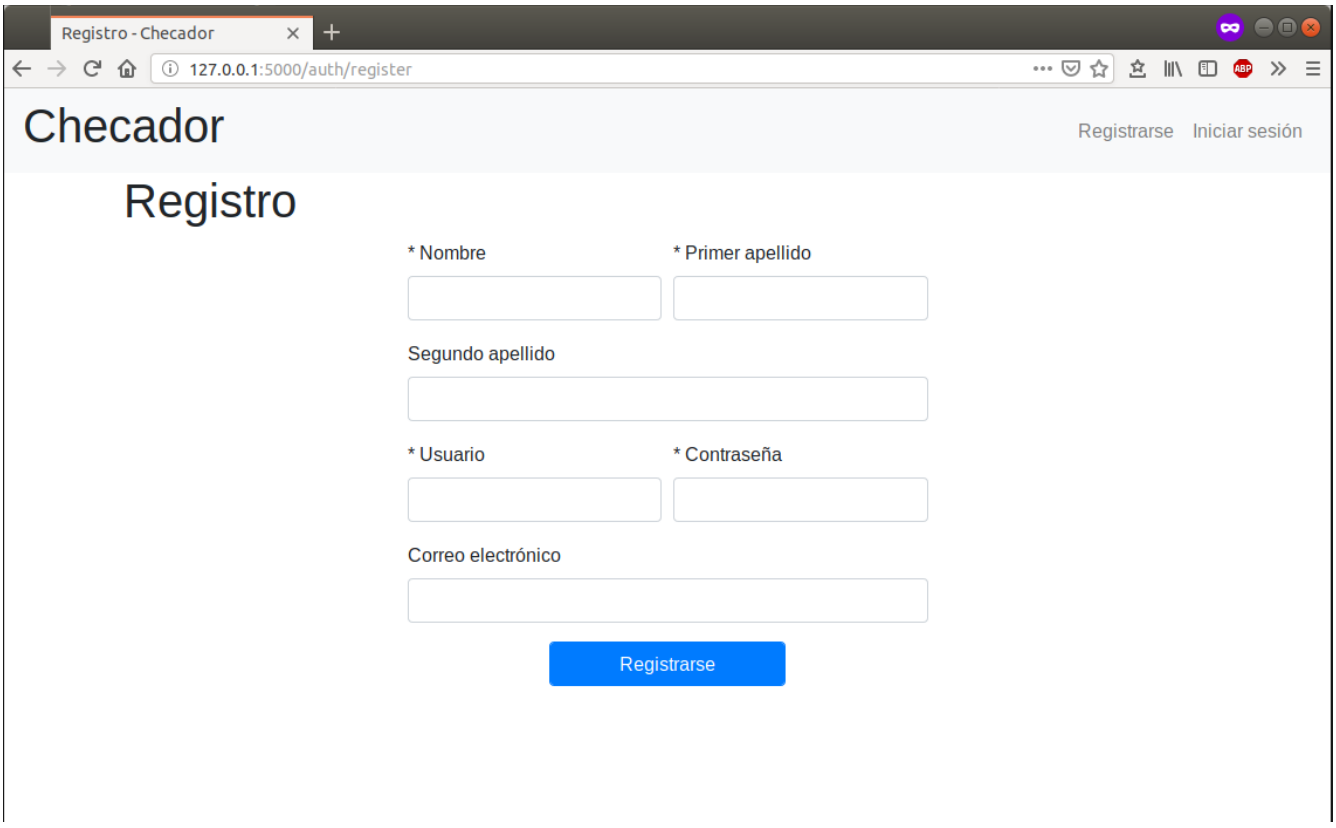
checador/templates/auth/register.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Registro de usuarios{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <div class="form-row justify-content-center">
      <div class="form-group col-md-3">
        <label for="nombre">* Nombre</label>
        <input class="form-control" type="text" name="nombre" id="nombre" required>
      </div>
      <div class="form-group col-md-3">
        <label for="ap_1">* Primer apellido</label>
        <input class="form-control" name="ap_1" id="ap_1" required>
      </div>
    </div>
    <div class="form-row justify-content-center">
      <div class="form-group col-md-6">
        <label for="ap_2">Segundo apellido</label>
        <input class="form-control" name="ap_2" id="ap_2">
      </div>
    </div>
    <div class="form-row justify-content-center">
      <div class="form-group col-md-3">
        <label for="username">* Usuario</label>
        <input class="form-control" name="username" id="username" required>
      </div>
```

```
    <div class="form-group col-md-3">
      <label for="password">* Contraseña</label>
      <input class="form-control" type="password" name="password" id="password" required>
    </div>
  </div>
  <div class="form-row justify-content-center">
    <div class="form-group col-md-6">
      <label for="email">Correo electrónico</label>
      <input class="form-control" type="email" name="email" id="email">
    </div>
  </div>
  <div class="form-group row justify-content-center">
    <div class="col-md-3">
      <input type="submit" class="btn btn-primary form-control" value="Registrarse">
    </div>
  </div>
 </form>
{% endblock %}
```

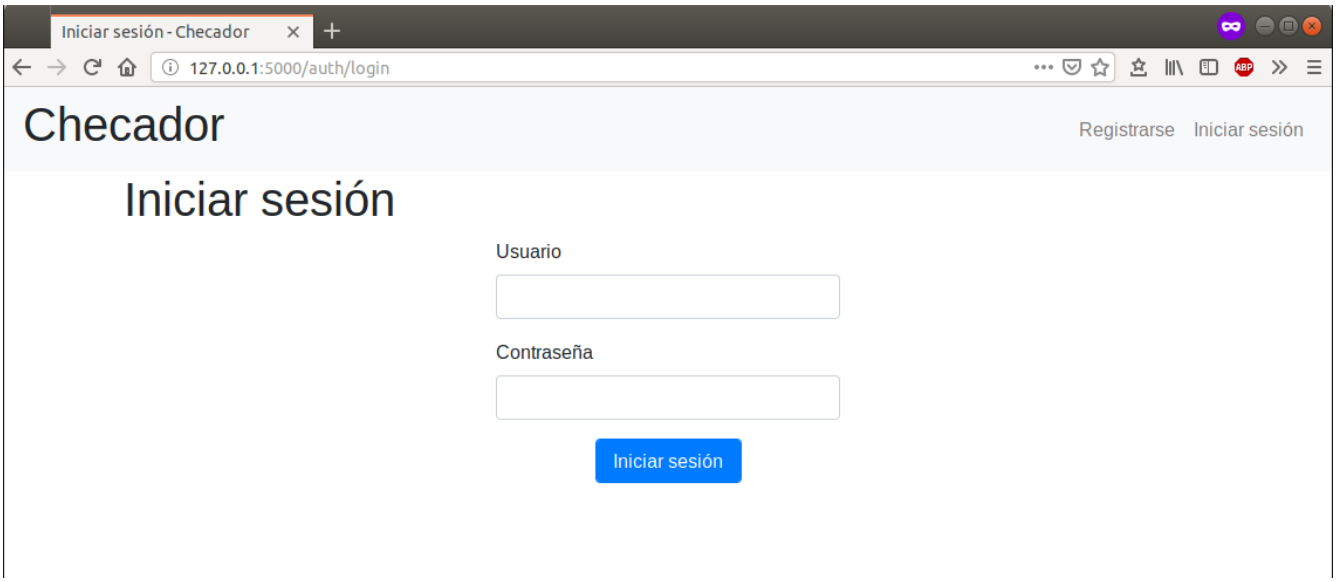


checador/templates/auth/login.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Iniciar sesión{% endblock %}</h1>
{% endblock %}
```

```
{% block content %}
 <form method="post">
  <div class="form-row justify-content-center">
   <div class="form-group col-md-4">
    <label for="username">Usuario</label>
    <input class="form-control" name="username" id="username" required>
   </div>
  </div>
  <div class="form-row justify-content-center">
   <div class="form-group col-md-4">
    <label for="password">Contraseña</label>
    <input class="form-control" type="password" name="password" id="password" required>
   </div>
  </div>
  <div class="form-group row justify-content-center">
   <div class="col-md-3">
    <input type="submit" class="btn btn-primary form-control" value="Iniciar sesión">
   </div>
  </div>
 </form>
{% endblock %}
```



Blueprint del checador

Utilizaremos las mismas técnicas que cuando escribimos el blueprint de autenticación para el blueprint del checador. El checador debe permitir registrar las entradas y salidas de los usuarios registrados, el ***admin*** puede registrar más usuarios y editar/eliminar a los usuarios registrados.

A medida que se implementa cada vista, mantengamos el servidor de desarrollo en ejecución. Mientras guardamos los cambios, vamos a la URL en el navegador y hacemos la prueba.

El Blueprint

Definimos el blueprint y lo registramos en la fábrica de aplicaciones.

checador/checar.py

```
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort
from checador.db import get_db

bp = Blueprint('checar', __name__)
```

Importamos y registrámos el blueprint desde la fábrica usando ***app.register_blueprint()***. Colocamos el nuevo código al final de la función de fábrica antes de devolver la aplicación (***return app***).

checador/__init__.py

```
def create_app():
    app = ...
    # código existente omitido

    from . import checar
    app.register_blueprint(checar.bp)
    app.add_url_rule('/', endpoint='index')

    return app
```

A diferencia del blueprint de autenticación, el blueprint del checador no tiene un ***url_prefix***. Por lo tanto, la vista del ***index*** estará en /. Checar es la característica principal de Checador, por lo que tiene sentido que el index del checador sea el index principal.

Sin embargo, el punto final para la vista de ***index*** definida a continuación será ***checar.index***. Algunas de las vistas de autenticación se referían a un punto final de ***index*** simple. ***app.add_url_rule()*** asocia el nombre de punto final ***'index'*** con la url / para que ***url_for('index')*** o ***url_for('checar.index')*** funcionen, generando el mismo URL / de cualquier manera.

En otra aplicación, puede darle al blueprint del checador un ***url_prefix*** y definir una vista de ***index*** separada en la fábrica de aplicaciones, similar a la vista de ***hola*** (generada al inicio del tutorial para probar el servidor). Entonces, los puntos finales y URL de ***index*** y ***checar.index*** serían diferentes.

Index

El index mostrará el formulario para registrar entradas/salidas al igual que un registro de las checadas de los usuarios del día.

checador/checar.py

```
@bp.route('/', methods=('GET', 'POST'))
def index():
    db = get_db()
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        error = None
        ultima_checada = None
        if username is None:
            error = 'El usuario es requerido.'
        elif password is None:
            error = 'La contraseña es requerida.'
        else:
            user = db.execute(
                'SELECT * FROM user WHERE username = ? AND baja = 0', (username,)
            ).fetchone()
            if user is None:
                error = 'El usuario no es correcto.'
            elif not check_password_hash(user['password'], password):
                error = 'La contraseña no es correcta.'

        if error is None:
            ultima_checada = db.execute(
                'SELECT * FROM checks'
                ' WHERE user_id = ?'
                    ' AND date(checa_entrada) = date("now", "localtime")'
                    ' AND baja = 0'
                    ' AND checa_salida IS NULL'
                ' ORDER BY checa_entrada DESC', (user['id'],)
            ).fetchone()
            if ultima_checada is not None:
                # El usuario ya registró entrada
                db.execute(
                    'UPDATE checks'
                    ' SET checa_salida = datetime(CURRENT_TIMESTAMP, "localtime")'
```

```
                ' WHERE id = ?', (ultima_checada['id'],)
            )
        else:
            # El usuario no tiene una entrada
            db.execute(
                'INSERT INTO checks (user_id)'
                ' VALUES (?)', (user['id'],)
            )
        db.commit()
    else:
        flash(error)
  checadas = db.execute(
    'SELECT u.username, time(c.checa_entrada) as checa_entrada,'
        ' time(c.checa_salida) as checa_salida'
    ' FROM checks c'
    ' JOIN user u ON c.user_id = u.id'
    ' WHERE date(c.checa_entrada) = date("now", "localtime")'
        ' or date(c.checa_salida) = date("now", "localtime")'
        ' and c.baja = 0'
  ).fetchall()
  return render_template('checar/index.html', checadas=checadas)
```

Lo primero que hacemos es obtener una conexión a la base de datos con ***get_db()*** y guardarla en ***db,*** si el método de petición fue ‘GET’ solamente se mandan a ***index.html*** las checadas del día, pero si el método fue ‘POST’ procedemos a validar usuario y contraseña, si no existen errores procedemos a consultar la última entrada del día que no tenga un registro de salida, si nuestra consulta no es ***None*** actualizamos el registro existente con la hora de salida y si la consulta regresa algo, procedemos a insertar una nueva entrada.

Ahora vamos a crear la vista para los resultados, como ya lo comentamos antes, esta será nuestro index y tendrá que estar en la carpeta “checar” dentro de templates.

checador/templates/checar/index.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Registro de entradas y salidas{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <div class="form-row justify-content-center">
      <div class="form-group col-md-4">
        <label for="username">Usuario</label>
        <input class="form-control" name="username" id="username" required>
```

```
      </div>
    </div>
    <div class="form-row justify-content-center">
      <div class="form-group col-md-4">
        <label for="password">Contraseña</label>
        <input class="form-control" type="password" name="password" id="password" required>
      </div>
    </div>
  <div class="form-group row justify-content-center">
    <div class="col-md-3">
      <input type="submit" class="btn btn-primary form-control" value="Checar">
    </div>
  </div>
</form>
<table class="table table-striped">
  <thead>
    <tr>
      <th scope="col">Usuario</th>
      <th scope="col">Entrada</th>
      <th scope="col">Salida</th>
    </tr>
  </thead>
  <tbody>
    {% for checada in checadas %}
    <tr>
      <th>{{ checada['username'] }}</th>
      <th>{{ checada['checa_entrada'] }}</th>
      {% if checada['checa_salida'] %}
        <th>{{ checada['checa_salida'] }}</th>
      {% else %}
        <th> </th>
      {% endif %}
    </tr>
    {% endfor %}
  </tbody>
</table>
{% endblock %}
```

Nuestra vista se conforma de dos campos que son usuario y contraseña y el botón de checar, y en seguida se muestran los registros del día.

Para obligar al usuario a que esté autenticado, como en el caso de querer registrar algún usuario, es necesario agregar el decorador ***@login_required*** entre el decorador de la ruta y la definición de la función. Muy importante como en el caso de la función de registro es necesario mover la función de ***login_required(view)*** antes de la función de ***register()***.

checador/auth.py

```
@bp.route('/register', methods=('GET', 'POST'))
@login_required
def register():
    # Código existente…
```

Para utilizar la función de ***login_required()*** en otro módulo se importa de la siguiente manera:

```
from checador.auth import login_required
…
…
@bp.route……
@login_required()
def ….
```

Haciendo nuestro proyecto instalable

Hacer nuestro proyecto instalable significa que podemos crear un archivo de distribución e instalarlo en otro entorno, así como instalamos Flask en nuestro entorno del proyecto.

Describir el proyecto

El archivo ***setup.py*** describe nuestro proyecto y los archivos que le pertenecen. El archivo setup.py va en la misma carpeta que está nuestro proyecto, en nuestro caso ***ProyectoChecador***

setup.py

```
from setuptools import find_packages, setup

setup(
    name='checador',
    version='1.0.0',
    packages=find_packages(),
    include_package_data=True,
    zip_safe=False,
    install_requires=[
        'flask',
    ],
)
```

packags le dice a Python qué paquete de directorios incluir (y los archivos Python que contienen). ***find_packages()*** encuentra estos directorios automáticamente, por lo tanto no tenemos que escribirlos. Para incluir otros archivos, como los directorios ***static*** y ***teplates***, ***include_package_data*** está en verdadero. Python necesita otro archivo llamado ***MANIFEST.in*** para indicarle los otros datos.

MANIFEST.in

```
include checador/schema.sql
graft checador/static
graft checador/templates
global-exclude *.pyc
```

Esto le indica a Python que copie todo lo que esté en los directorios ***static*** y ***templates***, y elarchivo ***schema.sql***, pero que excluya todos los archivos de bytecode (.pyc).

Para más información visite la ***documentación oficial de empaquetado.***

Instalar el proyecto

Vamos a usar el comando ***pip*** con nuestro entorno virtual activado (servidor detenido).

```
$ pip install -e .
```

Esto le indica a pip que encuentre el archivo ***setup.py*** en el directorio actual y los instale en modo ***editable*** o ***development***. El modo ***editable*** significa que como vayas haciendo cambios en el código

local, solo necesitaras re-instalar si cambiamos la metadata acerca del proyecto, así como sus dependencias.

Podemos observar que nuestro proyecto ahora está instalado con el comando ***pip list***.

```
$ pip list
Package         Version      Location
--------------- -------      ----------------------------------------
checador        1.0.0        /home/user/Documentos/ProyectoChecador
Click           7.0
Flask           1.1.1
itsdangerous    1.1.0
Jinja2          2.10.3
MarkupSafe      1.1.1
pip             19.3.1
pkg-resources   0.0.0
setuptools      41.4.0
Werkzeug        0.16.0
wheel           0.33.6
```

Nada cambia de cómo hemos estado ejecutando nuestro proyecto hasta ahora. ***FLASK_APP*** todavía está configurado en ***checado*** y ***flask run*** todavía ejecuta la aplicación, pero podemos llamarla desde cualquier lugar, no solo desde el directorio ***ProyectoChecador***.

Implementar en producción

Esta parte del tutorial asume que tiene un servidor en el que desea implementar su aplicación. Ofrece una descripción general de cómo crear el archivo de distribución e instalarlo, pero no entrará en detalles sobre qué servidor o software usar. Puede configurar un nuevo entorno en su computadora de desarrollo para probar las instrucciones a continuación, pero probablemente no debería usarlo para alojar una aplicación pública real. Consulte ***Opciones de implementación*** para obtener una lista de muchas formas diferentes de alojar su aplicación.

Construir e instalar

Cuando vayamos a implementar nuestra aplicación en otro lugar, creamos un archivo de distribución. El estándar actual para la distribución de Python es el formato de ***wheel***, con la extensión ***.whl***. Con nuestro entorno virtual activado, nos aseguramos de instalar primero la biblioteca de ***wheel***:

```
$ pip install wheel
```

Ejecutar ***setup.py*** con Python nos brinda una herramienta de línea de comandos para emitir comandos relacionados con la compilación. El comando ***bdist_wheel*** creará un archivo de distribución de ***wheel***.

```
$ python setup.py bdist_wheel
```

Podemos encontrar el archivo en ***dist/checador-1.0.0-py3-none-any.whl***. El nombre del archivo es el nombre del proyecto, la versión y algunas etiquetas sobre el archivo de instalacón.

Copie este archivo a otra máquina, configuramos y activamos un nuevo entorno virtual, luego instalamos el archivo con ***pip***.

```
$ pip install checador-1.0.0-py3-none-any.whl
```

Pip instalará nuestro proyecto junto con sus dependencias.

Como se trata de una máquina diferente, debe ejecutar ***init-db*** nuevamente para crear la base de datos en la carpeta de instancias.

```
$ export FLASK_APP=checador
$ flask init-db
```

Cuando Flask detecta que está instalado (no en modo editable), utiliza un directorio diferente para la carpeta de instancia. Podemos encontrarlo en entorno/var/checador-instance en su lugar.

Configurar la clave secreta

Al inicio del tutorial, le dimos un valor predeterminado para ***SECRET_KEY***. Esto debemos cambiarlo a algunos bytes aleatorios en producción. De lo contrario, los atacantes podrían usar la clave pública ***'dev'*** para modificar la cookie de sesión, o cualquier otra cosa que utilice la clave secreta.

Puede usar el siguiente comando para generar una clave secreta aleatoria:

```
$ python -c 'import os; print(os.urandom(16))'
b'_5#y2L"F4Q8z\n\xec]/'
```

Creamos el archivo ***config.py*** en la carpeta de instancias, de donde la fábrica leerá si existe. Copiamos dentro del archivo el valor generado.

entorno/var/checador-instance/config.py

```
SECRET_KEY = b'_5 # y2L "F4Q8z \ n \ xec] / '
```

También podemos establecer cualquier otra configuración necesaria aquí, aunque ***SECRET_KEY*** es la única necesaria para el ***Checador***.

Ejecutar con un servidor de producción

Cuando se ejecuta públicamente en lugar de en desarrollo, no debe usar el servidor de desarrollo incorporado (***flask run***). Werkzeug proporciona el servidor de desarrollo por conveniencia, pero no está diseñado para ser particularmente eficiente, estable o seguro.

En su lugar, use un servidor WSGI de producción. Por ejemplo, para usar ***Waitress***, primero instálelo en el entorno virtual:

```
$ pip install waitress
```

Tenemos que decirle a Waitress sobre nuestra aplicación, pero no utiliza ***FLASK_APP*** como lo hace ***flask run***. Debemos indicarle que importe y llame a la fábrica de aplicaciones para obtener un objeto de aplicación.

```
$ waitress-serve --call 'checador: create_app'
Serving on http://0.0.0.0:8080
```

Consulte ***Opciones de implementación*** para obtener una lista de muchas formas diferentes de alojar su aplicación. Waitress es solo un ejemplo, elegido para el tutorial porque es compatible con Windows y Linux. Hay muchos más servidores WSGI y opciones de implementación que puede elegir para su proyecto.

Ejecutando servidor en CentOS 8

Una vez instalado CentOS 8, vamos a crear una carpeta en ***documentos*** y accedemos a ella, ahí crearemos nuestro entorno virtual y copiaremos nuestro archivo creado anteriormente ***checador-1.0.0-py3-none-any.whl***

```
$ cd Documentos
$ mkdir checador
$ python3 -m venv entorno
$ . entorno/bin/activate
```

Como lo mencionamos anteriormente ahí instalamos con ***pip*** nuestra aplicación así como waitress.

```
$ pip install checador-1.0.0-py3-none-any.whl
$ pip install waitress
```

Vamos a escribir los comandos para que nos cree nuestra base de datos:

```
$ export FLASK_APP=checador
$ flask init-db
```

Generamos nuestra ***SECRET_KEY*** y procedemos a guardarla en el archivo ***config.py***

```
$ python -c 'import os; print(os.urandom(16))'
b'^\x1f\xc0]\x86\x98Fj0^t\x90\x9d\xcf\x1b\x1c'
```

entorno/var/checador-instance/config.py

```
SECRET_KEY = b'^\x1f\xc0]\x86\x98Fj0^t\x90\x9d\xcf\x1b\x1c'
```

Ahora procedemos a correr nuestro servidor

```
$ waitress-serve --call 'checador: create_app'
Serving on http://0.0.0.0:8080
```

Si tratamos de ingresar a nuestra aplicación desde otra computadora dentro de nuestra red local, probablemente no logremos obtener una conexión hacia nuestro servidor por la cuestión de los puertos.

En una nueva terminal escribimos el comando

```
$ firewall-cmd --list-all
```

si no están como usuario ***root*** (que es lo recomendable) les pedirá contraseña, una vez que pusimos la contraseña en mi caso no mostró nada (pero no dio error). Esto indica que no tenemos el puerto abierto, por lo cual procederemos a abrirlo con el siguiente comando:

```
$ firewall-cmd --zone=public --add-port=8080/tcp --permanent
```

Una vez hecho esto procedemos a listar de nuevo los puertos:

```
$ firewall-cmd –list-all
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: enp0s3
  sources:
  services: cockpit dhcpv6-client ssh
  ports: 8080/tcp
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

Ahora ya vemos nuestro puerto abierto y podremos acceder desde cualquier computadora de nuestra red local poniendo la ip y el puerto del servidor ***192.168.1.XX:8080***

Espero haberles ayudado un poco de lo que estoy aprendiendo, me ha sevido bastante hacer este tutorial ya que he comprendido cosas que no lograba comprender solo leyendo. El repositorio del proyecto se encuentra hasta andes de ***Hacerlo instalable*** y se encuentra en https://github.com/lgarciao/Tutorial-Checador.