SESIÓN N° 08:

REFLEX

I
OBJETIVOS

- Analizar la estructura de una aplicación web creada con Reflex, identificando los componentes principales, como páginas, estados y estilos, y comprendiendo cómo interactúan entre sí.
- Evaluar la eficiencia y la escalabilidad de las aplicaciones web creadas con Reflex, considerando factores como el rendimiento, la reutilización de código y la facilidad de mantenimiento.
- Desarrollar aplicaciones web funcionales utilizando Reflex, incluyendo la creación de interfaces de usuario interactivas, la gestión de estados de la aplicación y la integración con APIs externas.
- Implementar estilos personalizados en las aplicaciones web creadas con Reflex, utilizando tanto el atributo 'style' en los componentes como archivos de estilo separados, y aplicando conceptos de diseño web para crear interfaces atractivas y usables.
- Valorar la importancia de la simplicidad y la eficiencia en el desarrollo web, reconociendo las ventajas de utilizar un framework como Reflex para crear aplicaciones web con Python.
- Demostrar una actitud proactiva y de autoaprendizaje hacia las nuevas tecnologías, estando dispuesto a explorar las funcionalidades de Reflex y mantenerse actualizado con sus últimas novedades.

II
TEMAS A TRATAR

- Introducción a reflex
- Ventajas de usar Reflex
- Funcionamiento de Reflex
- Explorando Reflex: Un Framework Python para el Desarrollo Web
- Primeros Pasos con Reflex
- Creación de un Proyecto Básico en Reflex
- Componentes y Estructura de la Aplicación en Reflex
- Manejo de Estados en Reflex
- Vinculando el Estado a la Interfaz en Reflex
- Componentes Reutilizables en Reflex
- Estilos y Personalización en Reflex
- Resumen

III
MARCO TEORICO

1. INTRODUCCIÓN A REFLEX

Reflex es un framework de Python en fase beta diseñado para la creación rápida y sencilla de aplicaciones web utilizando únicamente Python. A diferencia de otros frameworks que requieren el uso de JavaScript, HTML y CSS para el desarrollo frontend, Reflex permite construir la interfaz de usuario, la lógica del lado del servidor y la gestión de datos utilizando exclusivamente Python. Este enfoque simplifica el proceso de desarrollo y reduce la curva de aprendizaje para aquellos que ya están familiarizados con Python, eliminando la necesidad de dominar múltiples lenguajes de programación.

Reflex se basa en la transformación del código Python a código JavaScript en segundo plano, aprovechando tecnologías robustas como React, Chakra UI y Next.js. Esta integración transparente permite a los desarrolladores beneficiarse de la potencia y flexibilidad de estas

tecnologías sin necesidad de interactuar directamente con ellas.

La metodología de trabajo de Reflex se centra en la modularidad y la reutilización de código a través de una estructura de componentes similar a la de React. Cada componente representa una parte independiente de la interfaz de usuario, que puede ser desde un elemento simple como un texto hasta un formulario complejo.

Reflex ofrece una biblioteca de más de 60 componentes visuales predefinidos, como botones, formularios y elementos de diseño, que se pueden integrar fácilmente en las aplicaciones. Estos componentes se basan en Chakra UI, una librería popular para construir interfaces de usuario con React, lo que garantiza una apariencia moderna y atractiva. Además, Reflex permite la integración con componentes de otras librerías de JavaScript, como Tailwind CSS, brindando aún más opciones de estilo y personalización.

Las páginas en Reflex se definen como métodos de Python que devuelven un componente raíz, que a su vez puede contener otros componentes anidados. Estos métodos se decoran con '@rx.page', especificando la ruta de la página y su título.

La gestión del estado de la aplicación, es decir, las variables que controlan el comportamiento de la interfaz de usuario se realizan mediante clases que heredan de 'rx.State'. Estas clases almacenan las variables de estado y definen métodos para actualizarlas. Reflex se encarga de sincronizar automáticamente los cambios en el estado con la interfaz de usuario gracias a su función de "hot reloading", que actualiza la aplicación en tiempo real sin necesidad de recargar la página manualmente.

Reflex ofrece diversas formas de aplicar estilos a las aplicaciones. Se pueden utilizar atributos de estilo directamente en los componentes, crear archivos de estilo separados o integrar librerías CSS como Tailwind CSS. Esta flexibilidad permite a los desarrolladores elegir el enfoque que mejor se adapte a sus necesidades y preferencias.

La filosofía detrás de Reflex se basa en la idea de que Python, como lenguaje de programación versátil y popular, puede ser una herramienta poderosa para el desarrollo web completo. Al permitir a los desarrolladores usar Python tanto para el frontend como para el backend, Reflex elimina la necesidad de cambiar de contexto entre diferentes lenguajes, lo que simplifica el flujo de trabajo y reduce la probabilidad de errores.

Esta filosofía se refleja en la facilidad de uso y la curva de aprendizaje amigable de Reflex. Los desarrolladores que ya conocen Python pueden comenzar a crear aplicaciones web rápidamente, sin necesidad de aprender nuevos lenguajes o frameworks complejos.

Para ilustrar el funcionamiento de Reflex, los ejemplos presentados en las fuentes se centran en la creación de aplicaciones sencillas, como un contador y un chat. A través de estos ejemplos, se muestra cómo definir páginas, crear componentes, gestionar el estado de la aplicación y aplicar estilos utilizando únicamente código Python.

2. VENTAJAS DE USAR REFLEX

Simplicidad: Permite desarrollar aplicaciones web completas utilizando únicamente Python, lo que facilita el aprendizaje y reduce la curva de aprendizaje para aquellos que ya están familiarizados con el lenguaje.

Componentes: Reflex ofrece una variedad de componentes predefinidos, como botones, formularios, contenedores y elementos de diseño, que se pueden integrar fácilmente en las aplicaciones. Además, permite la integración con componentes de otras librerías populares de JavaScript, como React y Chakra UI, lo que amplía aún más las posibilidades de diseño y funcionalidad.

Backend Integrado: Reflex trabaja en conjunto con FastAPI, un framework moderno y eficiente para la creación de APIs y backends, lo que simplifica la creación de la lógica del lado del servidor y la gestión de datos.

Hot Reloading: Reflex cuenta con la función de "hot reloading", lo que significa que los cambios en el código se ven reflejados en la aplicación web en tiempo real, sin necesidad de reiniciar el servidor.

Despliegue Sencillo: Reflex facilita el despliegue de aplicaciones web, tanto en servidores locales

como en plataformas de alojamiento web populares. Además, se está desarrollando una plataforma propia para el despliegue automatizado de proyectos de Reflex.

3. FUNCIONAMIENTO DE REFLEX

Reflex funciona interpretando el código Python y transformándolo en código JavaScript, utilizando tecnologías como React, Chakra UI y Next.js en segundo plano. Los desarrolladores no necesitan interactuar directamente con estas tecnologías, ya que Reflex se encarga de la traducción y la integración.

A. PRINCIPALES CARACTERÍSTICAS DE REFLEX

Estructura de Proyecto: Un proyecto de Reflex generalmente incluye carpetas para los assets (imágenes, archivos estáticos), el código principal de la aplicación y la configuración.

Páginas: Las páginas en Reflex se definen como métodos de Python que devuelven un componente. Estos métodos se decoran con '@rx.page' para indicar la ruta de la página y su título.

Componentes: Reflex utiliza una estructura de componentes similar a React, donde cada componente representa una parte independiente de la interfaz de usuario. Los componentes pueden ser tan simples como un texto o tan complejos como un formulario completo.

Estados: El manejo de estados en Reflex se realiza mediante clases que heredan de 'rx.State'. Estas clases almacenan las variables que controlan el comportamiento de la aplicación y proporcionan métodos para actualizarlas. Los cambios en el estado de la aplicación se reflejan automáticamente en la interfaz de usuario gracias a la función de "hot reloading".

Estilos: Los estilos en Reflex se pueden aplicar directamente a los componentes utilizando el atributo 'style' o mediante la creación de archivos de estilo separados. Reflex permite utilizar Python para definir los estilos, así como integrar librerías como Tailwind CSS.

4. EXPLORANDO REFLEX: UN FRAMEWORK PYTHON PARA EL DESARROLLO WEB

Reflex es un framework de Python que está ganando popularidad por su enfoque innovador al desarrollo web. Permite crear aplicaciones web completas utilizando únicamente Python, sin necesidad de escribir código JavaScript, HTML o CSS directamente. Esta característica lo hace especialmente atractivo para desarrolladores Python que buscan incursionar en el desarrollo web sin tener que aprender nuevos lenguajes.

A. VENTAJAS CLAVE DE REFLEX

Facilidad de Aprendizaje: Una de las principales ventajas de Reflex es su baja barrera de entrada para desarrolladores Python. La sintaxis familiar de Python y la ausencia de la necesidad de cambiar de contexto entre lenguajes simplifica el proceso de aprendizaje y permite a los desarrolladores concentrarse en la lógica de la aplicación en lugar de la sintaxis.

Flexibilidad y Extensibilidad: Reflex no se limita a sus componentes predefinidos. Los desarrolladores pueden integrar fácilmente componentes de React, lo que amplía significativamente las posibilidades de diseño y funcionalidad. Esta característica lo hace adecuado tanto para proyectos simples como para aplicaciones más complejas y personalizadas.

Documentación Completa y Recursos Abundantes: Reflex cuenta con una documentación detallada y bien estructurada que facilita el aprendizaje y la resolución de problemas. Además, la comunidad de Reflex es activa y ofrece soporte a través de foros y canales de comunicación.

B. GALERÍA DE APLICACIONES DE EJEMPLO: INSPIRACIÓN Y

APRENDIZAJE PRÁCTICO

Para facilitar aún más el aprendizaje y la exploración de las capacidades de Reflex, el framework ofrece una galería de aplicaciones de ejemplo con código fuente completo. Estas aplicaciones, que van desde ejemplos sencillos como un contador hasta proyectos más avanzados como un generador de emails con integración de ChatGPT, permiten a los desarrolladores:

Comprender la estructura de un proyecto de Reflex: Al examinar el código fuente de las aplicaciones de ejemplo, los desarrolladores pueden familiarizarse con la organización de archivos, la definición de páginas y la creación de componentes en Reflex.

Aprender mediante la práctica: La mejor forma de dominar un nuevo framework es utilizándolo en proyectos reales. Las aplicaciones de ejemplo proporcionan un punto de partida para que los desarrolladores puedan modificarlas, extenderlas y experimentar con diferentes funcionalidades.

Encontrar inspiración para nuevos proyectos: La galería de aplicaciones de ejemplo ofrece una visión general de las posibilidades de Reflex, lo que puede inspirar a los desarrolladores a crear sus propias aplicaciones innovadoras.

5. PRIMEROS PASOS CON REFLEX

Para comenzar a utilizar Reflex, es necesario tener instalada una versión de Python 3.7 o superior. Se recomienda utilizar un entorno virtual para evitar conflictos con otras dependencias.

Para instalar Reflex, se puede utilizar el siguiente comando:

```
pip install reflex
```

Una vez instalado Reflex, se puede crear un nuevo proyecto utilizando:

```
reflex init
```

Esto creará una estructura de proyecto básica con todo lo necesario para empezar a desarrollar una aplicación web. Para ejecutar el proyecto, se utiliza el comando:

```
reflex run
```

Esto iniciará un servidor de desarrollo que permitirá visualizar la aplicación en el navegador.

6. CREACIÓN DE UN PROYECTO BÁSICO EN REFLEX

A. INICIALIZANDO UN NUEVO PROYECTO

Para comenzar un nuevo proyecto en Reflex, se utiliza el comando ‘reflex init’ en la terminal. Este comando crea la estructura básica del proyecto, incluyendo los directorios necesarios y un archivo de configuración.

Ejemplo:

```
reflex init mi_proyecto
```

Este comando creará un directorio llamado "mi_proyecto" con la siguiente estructura:

B. ESTRUCTURA BÁSICA DE UN PROYECTO REFLEX

Un proyecto Reflex generalmente se organiza en los siguientes directorios:

web: Este directorio contiene el código compilado a JavaScript, listo para ser desplegado en un servidor web. No es necesario modificar directamente el contenido de este directorio, ya que Reflex se encarga de la compilación automáticamente.

assets: Aquí se almacenan los recursos estáticos del proyecto, como imágenes, archivos CSS personalizados y otros archivos que se necesitan cargar directamente en el navegador. Se puede acceder a estos archivos desde el código Python utilizando el módulo 'rx' de Reflex. Por ejemplo, para mostrar una imagen ubicada en 'assets/logo.png', se usaría 'rx.Image(src="logo.png")'.

Directorio del proyecto: Este directorio, que lleva el nombre del proyecto (en el ejemplo anterior, "mi_proyecto"), contiene el código fuente principal de la aplicación escrito en Python. Dentro de este directorio se encuentran los archivos y subdirectorios que definen la estructura y el comportamiento de la aplicación.

__init__.py: Este archivo, ubicado tanto en el directorio raíz del proyecto como en algunos subdirectorios, indica a Python que trate esos directorios como módulos. En el contexto de Reflex, este archivo se utiliza para inicializar las páginas y otros elementos de la aplicación.

Archivos de Python (.py): Estos archivos contienen el código Python que define las páginas, componentes, estados y estilos de la aplicación.

C. ARCHIVO DE CONFIGURACIÓN

El archivo de configuración principal de un proyecto Reflex se encuentra en la raíz del proyecto y usualmente se llama 'reflex.yaml'. Este archivo permite configurar aspectos importantes del proyecto, como:

Nombre de la aplicación ('app_name'): Define el nombre de la aplicación, que se utiliza para generar el nombre del módulo principal y otros elementos del proyecto.

Configuración de la base de datos: Reflex ofrece integración con bases de datos, y la configuración de la conexión a la base de datos se puede definir en este archivo.

Puerto del servidor: Permite especificar el puerto en el que se ejecutará el servidor de desarrollo local.

Variables de entorno: Se pueden definir variables de entorno específicas del proyecto en este archivo.

En resumen, la creación de un nuevo proyecto Reflex con 'reflex init' genera una estructura de directorios y un archivo de configuración que proporcionan las bases para comenzar a desarrollar aplicaciones web utilizando únicamente Python. Los directorios 'web' y 'assets' contienen el código compilado y los recursos estáticos, respectivamente, mientras que el directorio del proyecto alberga el código fuente principal de la aplicación.

7. COMPONENTES Y ESTRUCTURA DE LA APLICACIÓN EN REFLEX

A. INTRODUCCIÓN AL CONCEPTO DE COMPONENTES

En Reflex, el desarrollo de la interfaz de usuario se basa en componentes. Un componente es un fragmento de código reutilizable que encapsula la lógica, el estado y la apariencia de un elemento específico de la interfaz de usuario.

Reflex proporciona una amplia gama de componentes predefinidos, como botones, encabezados, formularios, imágenes, entre otros. Estos componentes se importan desde el módulo 'reflex' (normalmente importado como 'rx') y se utilizan como bloques de construcción para crear la estructura de la interfaz de usuario.

a) RX.HSTACK: ORGANIZACIÓN HORIZONTAL DE COMPONENTES

RX.hstack es un componente de diseño que permite organizar componentes hijos horizontalmente, uno al lado del otro. Es útil para crear filas y distribuir elementos en una línea.

Aquí tienes un ejemplo sencillo de cómo usar 'RX.hstack':

```
import reflex as rx

def mi_componente():
    return rx.hstack(
        rx.Button("Botón 1"),
        rx.Button("Botón 2"),
        rx.Button("Botón 3")
    )
```

En este ejemplo, se crean tres botones utilizando el componente 'RX.button' y se organizan horizontalmente dentro de un 'RX.hstack'.

b) CREACIÓN DE BOTONES (RX.BUTTON)

El componente RX.button se utiliza para crear botones interactivos en la interfaz de usuario.

Atributos comunes de 'RX.button':

children: El contenido del botón, que puede ser texto u otros componentes.

onclick: Una función que se ejecuta cuando se hace clic en el botón.

color_scheme: Define el esquema de color del botón.

bg: Permite establecer un color de fondo personalizado.

Ejemplo:

```
rx.Button("Haz clic aquí", color_scheme="blue", onclick=lambda: print("¡Botón presionado!"))
```

Este código crea un botón azul con el texto "Haz clic aquí". Al hacer clic en el botón, se ejecutará la función 'lambda' que imprimirá "¡Botón presionado!" en la consola.

c) CREACIÓN DE ENCABEZADOS (RX.HEADING)

El componente RX.heading se utiliza para crear encabezados de diferentes niveles (H1 a H6).

Atributos comunes de 'RX.heading':

children: El contenido del encabezado, generalmente texto.

size: Define el tamaño del encabezado (un número del 1 al 6, donde 1 es el más grande).

Ejemplo:

```
rx.Heading("Este es un encabezado H1", size=1)
rx.Heading("Este es un encabezado H3", size=3)
```

Este código crea un encabezado H1 con el texto "Este es un encabezado H1" y un encabezado H3 con el texto "Este es un encabezado H3".

8. MANEJO DE ESTADOS EN REFLEX

A. CREACIÓN DE UNA CLASE STATE

En Reflex, el manejo de estados se realiza mediante clases especiales que heredan de ‘rx.State’. Estas clases permiten definir variables que almacenan el estado de la aplicación y métodos para actualizar dicho estado.

Ejemplo de una clase State:

```
import reflex as rx

class Contador(rx.State):
  """Clase para manejar el estado de un contador."""

  def __init__(self):
    super().__init__()
    self.count = 0  # Variable de estado 'count' inicializada en 0

  def incrementar(self):
    """Incrementa el valor de 'count' en 1."""
    self.count += 1

  def disminuir(self):
    """Disminuye el valor de 'count' en 1."""
    self.count -= 1
```

En este ejemplo:

Se define una clase ‘Contador’ que hereda de ‘rx.State’.

En el método ‘__init__’, se inicializa la variable de estado ‘count’ con el valor 0.

Se definen dos métodos, ‘incrementar’ y ‘disminuir’, que modifican el valor de ‘count’.

B. UTILIZANDO LA CLASE STATE EN UN COMPONENTE

Para utilizar la clase ‘Contador’ dentro de un componente Reflex, primero debes instanciarla. Luego, puedes acceder a la variable de estado ‘count’ y a los métodos ‘incrementar’ y ‘disminuir’ a través del objeto instanciado.

Ejemplo de uso en un componente:

```
import reflex as rx
from typing import Callable

class Contador(rx.State):
  # ... (Código de la clase Contador del ejemplo anterior)

def mi_componente():
  """Componente que muestra un contador."""
  contador = Contador()

  return rx.vstack(
      rx.heading(f"Contador: {contador.count}", size=2),
      rx.hstack(
          rx.button("Incrementar", on_click=contador.incrementar),
          rx.button("Disminuir", on_click=contador.disminuir),
      ),
  )

@rx.page(route="/")
def index():
    return mi_componente()

# ... (Resto del código de la aplicación)
```

En este ejemplo:

Se crea una instancia de la clase ‘Contador’ llamada ‘contador’.

El valor de ‘contador.count’ se utiliza para mostrar el valor actual del contador en un encabezado.

Los métodos ‘contador.incrementar’ y ‘contador.disminuir’ se asignan a los eventos ‘on_click’ de los botones "Incrementar" y "Disminuir", respectivamente.

De esta manera, al hacer clic en los botones, se ejecutan los métodos correspondientes de la clase ‘Contador’, lo que modifica el valor de la variable de estado ‘count’. Reflex detecta automáticamente el cambio en el estado y actualiza la interfaz de usuario en consecuencia, mostrando el nuevo valor del contador.

9. VINCULANDO EL ESTADO A LA INTERFAZ EN REFLEX

La vinculación de estados con la interfaz es crucial para crear aplicaciones web dinámicas con Reflex. Esto implica conectar las variables de estado definidas en la clase ‘State’ con los componentes de la interfaz de usuario, de modo que cualquier cambio en el estado se refleje automáticamente en la interfaz y viceversa.

A. ACCEDIENDO AL ESTADO DESDE LOS COMPONENTES

Como se mencionó en la conversación anterior, para acceder al estado de la aplicación desde un componente, primero se necesita una instancia de la clase ‘State’ que maneja ese estado.

Consideremos el ejemplo de la clase ‘Contador’ que se definió anteriormente:

```
import reflex as rx

class Contador(rx.State):
  """Clase para manejar el estado de un contador."""

  def __init__(self):
    super().__init__()
    self.count = 0  # Variable de estado 'count' inicializada en 0

  def incrementar(self):
    """Incrementa el valor de 'count' en 1."""
    self.count += 1

  def disminuir(self):
    """Disminuye el valor de 'count' en 1."""
    self.count -= 1
```

Para usar esta clase dentro de un componente y que este tenga acceso a la variable ‘count’ y a los métodos ‘incrementar’ y ‘disminuir’, se puede hacer lo siguiente:

```
import reflex as rx

# ... (Código de la clase Contador)

def mi_componente():
  """Componente que muestra un contador."""
  contador = Contador()

  return rx.vstack(
      rx.heading(f"Contador: {contador.count}", size=2),
      rx.hstack(
          rx.button("Incrementar", on_click=contador.incrementar),
          rx.button("Disminuir", on_click=contador.disminuir),
      ),
  )

@rx.page(route="/")
def index():
    return mi_componente()
```

En este ejemplo, la variable ‘contador.count’ se utiliza directamente dentro del componente ‘rx.heading’ para mostrar su valor.

B. VINCULANDO FUNCIONES A EVENTOS

La vinculación de funciones a eventos, como ‘on_click’ en los botones, es esencial para que la interfaz de usuario pueda interactuar con el estado de la aplicación.

En el ejemplo anterior, los métodos ‘contador.incrementar’ y ‘contador.disminuir’ se asignan a los eventos ‘on_click’ de los botones "Incrementar" y "Disminuir" respectivamente:

```
rx.button("Incrementar", on_click=contador.incrementar),
rx.button("Disminuir", on_click=contador.disminuir),
```

Al hacer clic en el botón "Incrementar", se ejecuta el método ‘contador.incrementar’, lo que incrementa el valor de la variable de estado ‘count’. Reflex detecta automáticamente este cambio y actualiza la interfaz de usuario para reflejar el nuevo valor. Lo mismo ocurre al hacer clic en el botón "Disminuir".

10. COMPONENTES REUTILIZABLES EN REFLEX

El concepto de **componentes reutilizables** es fundamental en Reflex y en muchos otros frameworks de desarrollo frontend. Permite crear unidades de código independientes y encapsuladas que representan elementos específicos de la interfaz de usuario. Estos componentes se pueden reutilizar en diferentes partes de la aplicación, lo que facilita el mantenimiento del código y acelera el desarrollo.

Aunque las fuentes proporcionadas no brindan ejemplos concretos de creación de un componente de chat, sí mencionan que Reflex ofrece una galería de ejemplos y componentes predefinidos.

A. CREANDO UN COMPONENTE DE CHAT BÁSICO

Basándonos en la información de las fuentes y en los principios generales de desarrollo con componentes, podemos deducir la estructura de un componente de chat básico en Reflex:

```
import reflex as rx

class Mensaje(rx.Component):
  """Componente que muestra un mensaje de chat."""

  def __init__(self, usuario: str, texto: str):
    super().__init__()
    self.usuario = usuario
    self.texto = texto

  def render(self):
    return rx.box(
        rx.text(f"{self.usuario}: {self.texto}"),
        bg="lightblue",
        margin="5px",
        padding="10px",
        border_radius="5px",
    )

class Chat(rx.Component):
  """Componente que muestra una conversación básica."""

  def __init__(self, mensajes: list = []):
```

```
    super().__init__()
    self.mensajes = mensajes

  def render(self):
    return rx.vstack(
        [Mensaje(mensaje, mensaje) for mensaje in self.mensajes],
        width="400px",
        height="300px",
        border="1px solid gray",
        overflow_y="scroll",
    )

@rx.page(route="/")
def index():
    return Chat(mensajes=[
        ("Usuario 1", "¡Hola!"),
        ("Usuario 2", "¿Cómo estás?"),
    ])

# ... (Resto del código de la aplicación)
```

En este ejemplo:

'Se define el componente 'Mensaje' que recibe el usuario y el texto del mensaje y los muestra dentro de un 'rx.box' con estilo.

'Se define el componente 'Chat' que recibe una lista de mensajes.

'El componente 'Chat' itera sobre la lista de mensajes y crea una instancia del componente 'Mensaje' para cada uno.

'Los mensajes se muestran en un 'rx.vstack' con una barra de desplazamiento vertical.

B. MOSTRANDO UNA LISTA DE CONVERSACIONES

Para mostrar múltiples instancias del componente 'Chat', se puede simplemente crear una lista de conversaciones y luego iterar sobre ella, creando un componente 'Chat' para cada conversación:

```
import reflex as rx

# ... (Código de los componentes Mensaje y Chat)

@rx.page(route="/")
def index():
    conversaciones = [
        [("Usuario 1", "¡Hola!"), ("Usuario 2", "¿Cómo estás?")],
        [("Persona A", "Buenos días"), ("Persona B", "Hola, ¿qué tal?")]
    ]

    return rx.hstack(
        [Chat(mensajes=conversacion) for conversacion in conversaciones],
        spacing="20px",
    )

# ... (Resto del código de la aplicación)
```

En este ejemplo, se crea una lista 'conversaciones' donde cada elemento es una lista de mensajes. Luego, se utiliza 'rx.hstack' para mostrar los componentes 'Chat' de cada conversación horizontalmente, uno al lado del otro.

11. ESTILOS Y PERSONALIZACIÓN EN REFLEX

Reflex ofrece una manera única de manejar estilos CSS directamente desde Python, lo que simplifica el proceso de personalización de la interfaz de usuario. Si bien las fuentes proporcionadas no explican cómo crear un archivo `styles.py` independiente, sí muestran cómo definir estilos en línea dentro del código de la aplicación y cómo usar algunas características de estilo predefinidas de Reflex.

A. CREANDO UN ARCHIVO DE ESTILOS (STYLES.PY)

Basándonos en el conocimiento general sobre la aplicación de estilos en frameworks de desarrollo web y en la forma en que Reflex maneja los estilos en línea, podemos deducir una posible estructura para un archivo `styles.py`:

```
import reflex as rx
# Estilos para sombras
sombra_suave = rx.shadow(blur="4px", color="rgba(0, 0, 0, 0.2)")
sombra_intensa = rx.shadow(blur="8px", color="rgba(0, 0, 0, 0.5)")

# Estilos para márgenes
margen_pequeno = rx.margin("5px")
margen_mediano = rx.margin("10px")
margen_grande = rx.margin("20px")

# Estilos para bordes
borde_redondeado = rx.border_radius("5px")
borde_cuadrado = rx.border("1px solid gray")
```

En este ejemplo:

- Se definen diferentes estilos utilizando las funciones de estilo de Reflex, como `rx.shadow`, `rx.margin` y `rx.border_radius`.
- Se asignan nombres descriptivos a cada estilo para facilitar su uso posterior.

B. APLICANDO LOS ESTILOS A LOS COMPONENTES

Para aplicar los estilos definidos en `styles.py` a los componentes, se puede importar el archivo y luego usar los nombres de los estilos dentro del atributo `style` de los componentes:

```
import reflex as rx
from .styles import sombra_suave, margen_mediano, borde_redondeado

def mi_componente():
  return rx.box(
      "¡Hola, mundo!",
      style={
          "padding": "10px",
          "background-color": "lightblue",
          **sombra_suave,
          **margen_mediano,
          **borde_redondeado,
      }
  )
```

En este ejemplo:

- Se importa el archivo `styles.py`.
- Se utiliza el operador de desempaquetado `**` para aplicar los estilos definidos en `styles.py` al atributo `style` del componente `rx.box`.
- Se combinan estilos personalizados con estilos definidos en línea.

Consideraciones Adicionales

Reflex ofrece una amplia gama de funciones de estilo predefinidas que se pueden usar para personalizar la apariencia de los componentes. Se recomienda consultar la documentación oficial de Reflex para obtener una lista completa de estas funciones y sus opciones.

Para un control más preciso sobre los estilos, Reflex permite la integración con frameworks CSS populares como Tailwind CSS.

12. RESUMEN

Este resumen abarca desde la introducción a Reflex hasta los temas de estilos y personalización, componentes reutilizables y vinculación de estados con la interfaz, basándose en la información proporcionada en las fuentes y en la conversación previa.

Introducción a Reflex

Reflex es un framework de desarrollo web relativamente nuevo que permite crear aplicaciones web utilizando únicamente Python.

Puntos Clave:

- Simplicidad: Reflex se enfoca en la facilidad de uso y en reducir la curva de aprendizaje para desarrolladores Python que desean crear aplicaciones web.
- Python Puro: Se promociona la capacidad de construir el frontend y el backend utilizando únicamente Python, sin necesidad de aprender otros lenguajes como JavaScript o HTML.
- Componentes Predefinidos: Reflex proporciona una colección de componentes predefinidos para elementos comunes de la interfaz de usuario, lo que acelera el desarrollo.
- Integración con Backend: Se menciona la estrecha integración con FastAPI para el desarrollo del backend, lo que sugiere la posibilidad de crear aplicaciones web completas con Python.
- Despliegue Sencillo: Se destaca la facilidad para desplegar aplicaciones Reflex, tanto localmente como en plataformas de alojamiento web.

Vinculación de Estados con la Interfaz

La vinculación de estados permite conectar las variables de estado de la aplicación con los componentes de la interfaz de usuario, asegurando que cualquier cambio en el estado se refleje en la interfaz y viceversa.

Puntos Clave:

- Clase State: Reflex utiliza una clase llamada State para manejar el estado de la aplicación. Esta clase puede contener variables y métodos para actualizar el estado.
- Acceso al Estado desde Componentes: Los componentes pueden acceder a la instancia de la clase State para obtener el valor de las variables de estado y ejecutar los métodos que las modifican.
- Vinculación de Funciones a Eventos: Los métodos de la clase State se pueden vincular a eventos de los componentes, como on_click, para actualizar el estado en respuesta a las interacciones del usuario.

Componentes Reutilizables

Los componentes reutilizables son unidades de código independientes que representan elementos específicos de la interfaz de usuario. Permiten una mayor organización del código, facilitan el mantenimiento y agilizan el desarrollo.

Puntos Clave:

- Encapsulamiento: Los componentes encapsulan la lógica y la presentación de un elemento de la interfaz de usuario, lo que los hace independientes y reutilizables.
- Composición: Los componentes se pueden componer para crear estructuras de interfaz de usuario más complejas a partir de elementos más simples.
- Flexibilidad: Los componentes pueden recibir datos a través de props (propiedades) para personalizar su comportamiento y apariencia.

Estilos y Personalización

Reflex permite definir estilos CSS directamente desde Python, lo que simplifica la personalización de la interfaz de usuario.

Puntos Clave:

- Estilos en Línea: Se pueden definir estilos directamente dentro del atributo style de los componentes utilizando un diccionario de Python.
- Funciones de Estilo: Reflex ofrece funciones de estilo predefinidas, como rx.shadow, rx.margin y rx.border_radius, para aplicar estilos comunes.
- Integración con Frameworks CSS: Reflex permite la integración con frameworks CSS populares como Tailwind CSS para un control más preciso sobre los estilos.

IV

(La práctica tiene una duración de 4 horas) ACTIVIDADES

CONTEXTO DEL PROBLEMA

El contexto general del problema que Reflex busca resolver es la complejidad inherente al desarrollo web tradicional, que a menudo requiere que los desarrolladores aprendan y utilicen múltiples lenguajes de programación, como HTML, CSS y JavaScript, además del lenguaje del lado del servidor. Reflex propone una solución simplificada al permitir la creación de aplicaciones web utilizando únicamente Python, lo que podría resultar atractivo para desarrolladores Python que buscan una forma más directa de ingresar al desarrollo web.

A continuación, se presentan cuatro experiencias de práctica que se basan en las características clave de Reflex, desarrolladas de manera incremental y articulada para guiar la comprensión del framework:

1. EXPERIENCIA DE PRÁCTICA N° 01: CONTADOR SIMPLE CON ESTADO LOCAL

Objetivo: Familiarizarse con la estructura básica de una aplicación Reflex, la creación de componentes simples y el manejo de estados locales.

Descripción:

1. Crear un componente `Contador`: Este componente mostrará un número que representa el conteo actual y dos botones: uno para incrementar el conteo y otro para disminuirlo.
2. Implementar el estado local: Utilizar la clase `rx.State` para crear una variable de estado llamada `conteo` inicializada en 0.
3. Vincular botones a funciones de estado: Definir dos funciones dentro de la clase `rx.State`: `incrementar` para aumentar `conteo` en 1 y `disminuir` para disminuirlo en 1. Vincular estas funciones a los eventos `on_click` de los botones correspondientes.

Ejemplo de Código:

```
import reflex as rx

class EstadoContador(rx.State):
    def __init__(self):
        super().__init__()
        self.conteo = 0

    def incrementar(self):
        self.conteo += 1

    def disminuir(self):
        self.conteo -= 1

def contador():
    return rx.fragment(
        rx.hstack(
            rx.button("Incrementar", on_click=EstadoContador.incrementar),
            rx.text(EstadoContador.conteo),
            rx.button("Disminuir", on_click=EstadoContador.disminuir),
        )
```

```
    )
```

2. EXPERIENCIA DE PRÁCTICA N° 02: LISTA DE TAREAS CON ESTADO GLOBAL

Objetivo: Comprender el concepto de estado global en Reflex y cómo compartir datos entre diferentes componentes.

Descripción:

1. Crear dos componentes: Un componente `ListaTareas` para mostrar una lista de tareas y un componente `AgregarTarea` para agregar nuevas tareas a la lista.
2. Implementar el estado global: Utilizar una instancia global de `rx.State` para almacenar la lista de tareas. Esto permitirá que ambos componentes accedan y modifiquen la misma lista.
3. Sincronizar componentes con el estado global: Asegurar que `ListaTareas` se actualice automáticamente cada vez que se agregue una nueva tarea a través de `AgregarTarea`.

Ejemplo de Código:

```
import reflex as rx

class EstadoTareas(rx.State):
    def __init__(self):
        super().__init__()
        self.tareas = ["Tarea 1", "Tarea 2"]

    def agregar_tarea(self, nueva_tarea):
        self.tareas.append(nueva_tarea)

def lista_tareas():
    return rx.fragment(
        rx.heading("Lista de Tareas"),
        rx.ul([rx.li(tarea) for tarea in EstadoTareas.tareas]),
    )

def agregar_tarea():
    nueva_tarea = rx.input(placeholder="Agregar tarea...")
    return rx.fragment(
        nueva_tarea,
        rx.button("Agregar",                                    on_click=lambda:
EstadoTareas.agregar_tarea(nueva_tarea.value))
    )
```

3. EXPERIENCIA DE PRÁCTICA N° 03: FORMULARIO SIMPLE CON VALIDACIÓN

Objetivo: Aprender a crear formularios con Reflex, manejar la entrada del usuario y realizar una validación básica de los datos.

Descripción:

1. Crear un componente `FormularioContacto`: Este componente incluirá campos para el nombre, la dirección de correo electrónico y un mensaje.
2. Implementar validación básica: Agregar validación a los campos del formulario, por ejemplo, comprobar que el campo de correo electrónico tenga un formato válido.
3. Manejar el envío del formulario: Capturar los datos del formulario cuando se envíe y mostrar un mensaje de éxito o error al usuario.

Ejemplo de Código:

```
import reflex as rx

class EstadoFormulario(rx.State):
    def __init__(self):
        super().__init__()
        self.nombre = ""
        self.email = ""
        self.mensaje = ""

    def enviar_formulario(self):
        if "@" not in self.email:
            rx.notify("Error: Dirección de correo electrónico no válida")
        else:
            rx.notify("Formulario enviado correctamente")

def formulario_contacto():
    return rx.fragment(
        rx.heading("Formulario de Contacto"),
        rx.form(
            rx.input(
                placeholder="Nombre",
                on_change=lambda valor: setattr(EstadoFormulario, "nombre", valor)
            ),
            rx.input(
                placeholder="Correo electrónico",
                on_change=lambda valor: setattr(EstadoFormulario, "email", valor)
            ),
            rx.textarea(
                placeholder="Mensaje",
                on_change=lambda   valor:   setattr(EstadoFormulario,   "mensaje",
valor)
            ),
            rx.button("Enviar", on_click=EstadoFormulario.enviar_formulario),
        ),
    )
```

4. EXPERIENCIA DE PRÁCTICA N° 04: CONSUMO DE UNA API EXTERNA

Objetivo: Integrar una aplicación Reflex con una API externa para obtener y mostrar datos dinámicos.

Descripción:

1. Elegir una API pública: Seleccionar una API pública que proporcione datos en un formato adecuado (como JSON) y que no requiera autenticación compleja.
2. Crear un componente para mostrar los datos: Este componente se encargará de realizar la solicitud a la API, procesar los datos y mostrarlos al usuario.
3. Implementar la lógica de la solicitud: Utilizar la biblioteca `httpx` (o similar) para realizar una solicitud GET a la API y obtener los datos.
4. Manejar la respuesta de la API: Procesar los datos JSON recibidos de la API y mostrarlos de manera adecuada en la interfaz de usuario.

Ejemplo de Código:

```
import reflex as rx
import httpx

async def obtener_datos_api():
    async with httpx.AsyncClient() as cliente:
```

```
        respuesta = await cliente.get("https://api.example.com/datos")
        respuesta.raise_for_status()
        return respuesta.json()

class EstadoDatosAPI(rx.State):
    def __init__(self):
        super().__init__()
        self.datos = []

    async def cargar_datos(self):
        self.datos = await obtener_datos_api()

def mostrar_datos_api():
    return rx.fragment(
        rx.button("Cargar Datos", on_click=EstadoDatosAPI.cargar_datos),
        rx.ul([rx.li(dato) for dato in EstadoDatosAPI.datos]),
    )
```

Nota: El código proporcionado es un ejemplo básico y es posible que deba adaptarse en función de la API específica que se esté utilizando. Se recomienda consultar la documentación de la API para obtener información detallada sobre sus endpoints, parámetros y formato de respuesta.

V

EJERCICIOS PROPUESTOS

1. **Filtro de Tareas Pendientes en el Tablero Kanban**

Objetivo: En lugar de arrastrar y soltar, que puede resultar más complejo para empezar, este ejercicio se centra en crear un filtro dinámico para mostrar solo las tareas pendientes en el tablero Kanban.

Descripción:

a) Variable de Estado mostrar_solo_pendientes: En la clase State del componente del tablero Kanban, crea una variable booleana llamada mostrar_solo_pendientes inicializada en False. Esta variable controlará la visibilidad de las tareas.

b) Botón "Mostrar Pendientes": Implementa un botón que, al ser presionado, cambie el valor de la variable mostrar_solo_pendientes a True.

c) Lógica de Filtrado: En el componente ColumnaKanban, antes de renderizar las tarjetas de tareas con rx.for_each, aplica una condición que verifique si mostrar_solo_pendientes es True. Si lo es, filtra las tareas para mostrar solo aquellas cuyo estado sea "Pendiente".

Ejemplo de Código (Conceptual):

```
import reflex as rx

class State(rx.State):
    def __init__(self):
        super().__init__()
        self.mostrar_solo_pendientes = False

    def mostrar_pendientes(self):
      self.mostrar_solo_pendientes = True

def tarjeta_tarea(tarea):
    return rx.div(
        tarea["titulo"],
        # ... otros detalles de la tarea
    )

def columna_kanban(nombre, tareas):
    if state.mostrar_solo_pendientes:
        tareas = [t for t in tareas if t["estado"] == "Pendiente"]

    return rx.div(
```

```
        rx.heading(nombre),
        rx.div(
            [tarjeta_tarea(tarea) for tarea in tareas]
        )
    )

def index():
    return rx.div(
        rx.button("Mostrar Pendientes", on_click=State.mostrar_pendientes),
        columna_kanban("En Progreso", tareas_en_progreso),
        columna_kanban("Completadas", tareas_completadas)
    )
```

2. Contador de Tareas por Estado en el Tablero Kanban

Objetivo: En lugar de integrar una biblioteca de gráficos externa, este ejercicio se enfoca en calcular y mostrar la cantidad de tareas por cada estado en el tablero Kanban, utilizando las capacidades nativas de Reflex para la manipulación de datos.

Descripción:

a) Función contar_tareas_por_estado: Crea una función que reciba una lista de tareas y devuelva un diccionario donde las claves son los estados de las tareas y los valores son la cantidad de tareas con ese estado.

b) Mostrar Contadores: En el componente del tablero Kanban, después de las columnas, utiliza la función contar_tareas_por_estado para calcular los contadores y mostrarlos en elementos rx.div separados.

Ejemplo de Código (Conceptual):

```
import reflex as rx

def contar_tareas_por_estado(tareas):
    contadores = {}
    for tarea in tareas:
        estado = tarea["estado"]
        if estado in contadores:
            contadores[estado] += 1
        else:
            contadores[estado] = 1
    return contadores

def index():
    contadores = contar_tareas_por_estado(todas_las_tareas)
    return rx.div(
        # ... Columnas Kanban ...
        rx.div(f"Pendientes: {contadores.get('Pendiente', 0)}"),
        rx.div(f"En Progreso: {contadores.get('En Progreso', 0)}"),
        rx.div(f"Completadas: {contadores.get('Completada', 0)}")
    )
```

V
CUESTIONARIO

1. ¿Cuál es la principal ventaja de utilizar Reflex para el desarrollo web, según lo destacado en las fuentes?
2. Menciona tres características clave de Reflex que lo diferencian de otros frameworks web.
3. ¿Qué significa que Reflex te permite construir aplicaciones web con "Python puro"? ¿Qué implicaciones tiene esto para los desarrolladores?
4. Describe el concepto de "componentes" en Reflex y proporciona un ejemplo específico de un componente utilizado en las fuentes.
5. ¿Cómo se manejan los eventos, como hacer clic en un botón, en una aplicación Reflex? Proporciona un ejemplo de código que ilustre cómo un evento puede desencadenar una acción.

6. Explica la función del archivo `rx.State` en una aplicación Reflex. ¿Qué tipo de información se almacena típicamente en este archivo?
7. ¿De qué manera Reflex facilita la creación de interfaces de usuario dinámicas? Menciona al menos dos mecanismos o características que contribuyan a esta facilidad.
8. ¿Qué son los "estados globales" en el contexto de una aplicación Reflex y cómo se relacionan con los componentes individuales?
9. Describe, en términos generales, el proceso para integrar una biblioteca externa, como una biblioteca de gráficos, en una aplicación Reflex.
10. ¿Qué beneficios ofrece el "hot reloading" durante el desarrollo con Reflex?
11. ¿Qué alternativas existen para desplegar (deploy) una aplicación Reflex, según se menciona en las fuentes?
12. ¿Qué tipo de aplicaciones web son ideales para construir con Reflex, considerando sus fortalezas y limitaciones?
13. Las fuentes mencionan algunas tecnologías y bibliotecas de Javascript que se integran o utilizan en segundo plano con Reflex. Nombra al menos tres de estas tecnologías y explica brevemente su función.
14. Basándote en tu comprensión de Reflex a partir de las fuentes, ¿cuáles son algunas de las limitaciones potenciales o desafíos que podrías encontrar al desarrollar una aplicación web compleja con este framework?
15. Reflex está en constante desarrollo. ¿Qué nuevas características o mejoras te gustaría ver en futuras versiones de este framework, teniendo en cuenta las tendencias actuales en el desarrollo web?

VI
BIBLIOGRAFIA Y REFERENCIAS

Ciencia, P. (6 de julio de 2023). *FRAMEWORK REFLEX - Instalación y primeros pasos*. Obtenido de YouTube: https://www.youtube.com/watch?v=YDc-EfbMacE

CodingEntrepreneurs. (19 de abril de 2023). *Build Full Stack Web Apps in Pure Python with Reflex - No Javascript Required*. Obtenido de YouTube: https://www.youtube.com/watch?v=zBhhQa_za24

Moure, M. b. (26 de junio de 2023). *Crea una WEB usando solo PYTHON*. Obtenido de YouTube: https://www.youtube.com/watch?v=YDc-EfbMacE