2023

Events

MP14 - UF1 - BIG DATA

KEVIN ANDRES HERRERA DOMINGUEZ

Índice.

Justificación del proyecto

En la página web, he creado herramientas que permiten a los usuarios buscar eventos de manera eficiente, personalizada y muy minimalista. A través de nuestra barra de búsqueda, los usuarios pueden encontrar fácilmente eventos según su ubicación o artistas favoritos en cualquier momento y lugar.

El proyecto surge para dar respuesta a las necesidades tanto personales como sociales de las personas. A nivel personal, reconozco lo complicado y consumidor de tiempo que puede ser encontrar información precisa sobre eventos y filtrarlos según las preferencias individuales. Nuestra herramienta de búsqueda de eventos está diseñada para resolver esta necesidad, brindando a los usuarios una forma eficiente y personalizada de encontrar eventos relevantes para ellos.

A nivel social, entiendo que existe una comunidad de amantes de la música, el arte y la cultura en general que busca conectarse y compartir experiencias. Nuestra herramienta de búsqueda de eventos también se enfoca en satisfacer esta necesidad, permitiendo a los usuarios descubrir eventos locales, y explorar nuevos artistas o géneros, nuestro proyecto contribuye a fomentar la comunidad y la participación social en el ámbito de los eventos culturales.

En resumen, mi proyecto busca responder a las necesidades tanto personales como sociales de los usuarios al proporcionar una herramienta minimalista para buscar eventos. Nuestro objetivo es permitir que las personas encuentren y se conecten con actividades culturales, al tiempo que fomentamos la interacción y la comunidad entre los amantes de la música, el arte y la cultura al facilitar el encontrar eventos.

Uso de los datos masivos

En mi aplicación web, utilizo datos masivos abiertos provenientes de la API de Ticketmaster, una plataforma reconocida de venta de entradas y eventos. Ticketmaster proporciona acceso a una amplia gama de información sobre eventos, como conciertos, festivales, conferencias y actividades culturales en todo el mundo.

Para acceder a los datos de Ticketmaster, utilizo su API pública. La API me ofrece endpoints y métodos que me permiten realizar solicitudes de búsqueda y obtención de información sobre eventos específicos, artistas, ubicaciones, fechas y otros detalles relevantes. Estas solicitudes las realizo a través de archivos .py, donde manejo la lógica de mi aplicación.

Una vez que obtengo los datos de la API de Ticketmaster, los analizo y proceso para presentarlos de manera clara y atractiva al usuario en mi aplicación web. Utilizo lenguajes como HTML y CSS para diseñar los templates y las interfaces visuales de la aplicación, lo que me permite mostrar los eventos de manera estructurada y atractiva.

En cuanto al análisis de datos, puedo realizar diferentes procesos dependiendo de los requerimientos de mi aplicación. Puedo filtrar y clasificar los eventos según las preferencias del usuario, ciudad o artista/evento.

Aqui esta el modelo que usa la API que use para este proyecto donde básicamente hice uso de name, price y dates dentro de events y city dentro de venues

Data Model


PRODUCTS
id
type
name
description
OFFERS
id
type
name
products
rank
currency
prices
limit
area
description
price-zones
passwords
EVENTS
id
name
type
locate
url
sales
dates
price range
attractions
please note
extensions
source
promoter
images
venue
info
classifications
field
ATTRACTIONS
id
name
type
images
url
classifications
locale
AREAS
id
type
name
rank
area-type
IMAGES
ratio
url
height
width
fallback
PASSWORDS
id
type
name
exclusive
prompts
text-label
VENUES
id
name
type
locate
url
location
extensions
source
markets
time zone
address
city
country
state
postal code
dmas
SEGMENTS
id
name
PRICE-ZONES
id
type
name
currency
GENRES
id
name
PRICES
type
value
currency
SUB-GENRES
id
name

Explicación del código desarrollado

¿Cómo funciona el código del archivo app.py?

1. Importaciones:

```python
from flask import Flask, render_template, request, redirect, url_for
import pandas as pd
import json
import requests
```

En esta sección, se importan los módulos necesarios para crear la aplicación web y manejar las solicitudes HTTP, trabajar con JSON y realizar solicitudes a la API de Ticketmaster.

2. Creación de la aplicación Flask:

```python

app = Flask(__name__)

```

Se crea una instancia de la clase Flask para inicializar la aplicación web.

3. Rutas de la aplicación:

Las siguientes líneas de código definen diferentes rutas para la aplicación Flask, que corresponden a diferentes páginas de la aplicación web. Por ejemplo:

```python
@app.route('/')
def home():
    return render_template('inicio.html')

```

Esta función decorada define la ruta raíz de la aplicación (`/`) y el controlador asociado `home()`, que renderiza la plantilla `inicio.html` cuando se accede a esa ruta.

4. Controladores y renderizado de plantillas:

```python
@app.route('/acerca-de')
def acerca_de():
    return render_template('acerca_de.html')

```

Este es un ejemplo de un controlador que devuelve una plantilla HTML (`acerca_de.html`) cuando se accede a una determinada ruta.

5. Manejo de solicitudes POST:

```python
@app.route('/buscador', methods=['POST'])
def buscador():
    name = request.form['nombreArtista']
    city = request.form.get('ciudad')
    country = request.form.get('pais')

    api_key = '9a3vA1iPd9GiWCMnWGP28Ggw55E8iOGQ'
    url = f'https://app.ticketmaster.com/discovery/v2/events.json?apikey={api_key}&keyword={name}'

# y el resto del código
```

Este controlador se utiliza para manejar las solicitudes POST enviadas desde el formulario de búsqueda en la página `inicio.html`. Recupera los datos enviados desde el formulario (nombre del artista, ciudad y país) y realiza una solicitud a la API de Ticketmaster utilizando esos datos.

6. Procesamiento de datos:

El código dentro del controlador `buscador()` se encarga de procesar los datos recibidos de la API de Ticketmaster y prepararlos para mostrarlos en la plantilla `buscador.html`. Esto implica realizar solicitudes a la API, procesar la respuesta JSON recibida, extraer los datos relevantes y crear una lista de diccionarios `evento_data` que contiene la información de cada evento encontrado.

7. Renderizado de plantillas y envío de datos:

```python

    return render_template('buscador.html', evento_data=evento_data, nombre_artista=name, ciudad=city, pais=country)

```

Este código devuelve la plantilla `buscador.html` y envía los datos procesados (`evento_data`, `nombre_artista`, `ciudad`, `pais`) a la plantilla para que se muestren correctamente.

8. Comprobación de la API de Ticketmaster:

```python
# Comprobar si la API de Ticketmaster está funcionando y su formato
try:
    api_key = '9a3vA1iPd9GiWCMnWGP28Ggw55E8iOGQ'
    url = f'https://app.ticketmaster.com/discovery/v2/events.json?apikey={api_key}&size=1'
    response = requests.get(url)
    response.raise_for_status()
    data = json.loads(response.content)
    # print(f"La API de Ticketmaster está funcionando. Formato de respuesta: {json.dumps(data, indent=2)}")
except requests.exceptions.RequestException as e:
    print(f'Error en la solicitud a la API de Ticketmaster: {e}')

```

Este código se ejecuta al iniciar la aplicación y realiza una solicitud a la API de Ticketmaster para verificar si está funcionando y para obtener un ejemplo del formato de respuesta. Si hay algún error en la solicitud

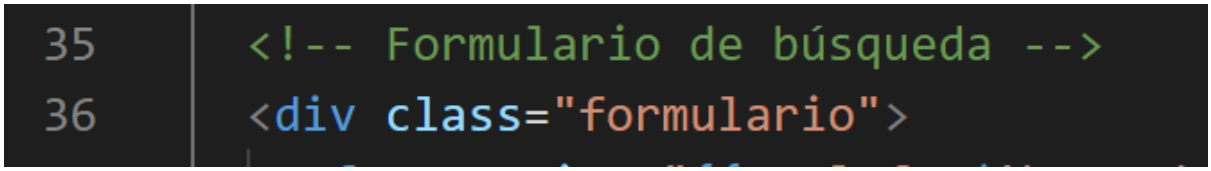
```python
@app.route('/buscador', methods=['POST'])
def buscador():
    name = request.form['nombreArtista']
    city = request.form.get('ciudad')
    country = request.form.get('pais')

    api_key = '9a3vA1iPd9GiWCMnWGP28Ggw55E8iOGQ'
    url = f'https://app.ticketmaster.com/discovery/v2/events.json?apikey={api_key}&keyword={name}'

    if city:
        url += f'&city={city}'
    if country:
        url += f'&countryCode={country}'

    try:
        response = requests.get(url)
        response.raise_for_status()
    except requests.exceptions.RequestException as e:
        print(f'Solicitud fallida con excepción: {e}')
        return redirect(url_for('not_found_error', error=e))

    data = json.loads(response.content)
    if '_embedded' not in data or 'events' not in data['_embedded']:
        print(f'Error: No se encontraron eventos para {name} en {city or country}')
        return redirect(url_for('not_found_error', error=f'No se encontraron eventos para {name} en {city or country}'))

    events = data['_embedded']['events']
    evento_data = []

    for event in events:
        evento_dict = {}
        evento_dict['nombre'] = event['name']

        if '_embedded' in event and 'venues' in event['_embedded']:
            venue = event['_embedded']['venues'][0]
            if 'city' in venue and 'country' in venue:
                evento_dict['lugar'] = f"{venue['city']['name']}, {venue['country']['name']}"
            elif 'city' in venue:
                evento_dict['lugar'] = venue['city']['name']
            elif 'country' in venue:
                evento_dict['lugar'] = venue['country']['name']
            else:
                evento_dict['lugar'] = 'Lugar no disponible'
        else:
            evento_dict['lugar'] = 'Lugar no disponible'

        if 'dates' in event and 'start' in event['dates']:
            evento_dict['fecha'] = event['dates']['start'].get('localDate', 'Fecha no disponible')
        else:
            evento_dict['fecha'] = 'Fecha no disponible'

        if 'priceRanges' in event:
            price_range = event['priceRanges'][0]
            evento_dict['precio'] = f"{price_range['min']} - {price_range['max']} {price_range['currency']}"
        else:
            evento_dict['precio'] = 'Precio no disponible'

        evento_dict['link'] = event.get('url', '#')  # Agregar el enlace del evento al diccionario

        evento_data.append(evento_dict)

    return render_template('buscador.html', evento_data=evento_data, nombre_artista=name, ciudad=city, pais=country)
```

Código completo de la función buscador

¿Cómo funciona el formulario del archivo inicio.html?

El formulario HTML que vamos a ver es parte de la página `inicio.html` y permite a los usuarios ingresar información de búsqueda, como el nombre del artista y la ciudad. Aquí tienes una explicación de cada parte del formulario:

1. Etiqueta `div` con clase "formulario":

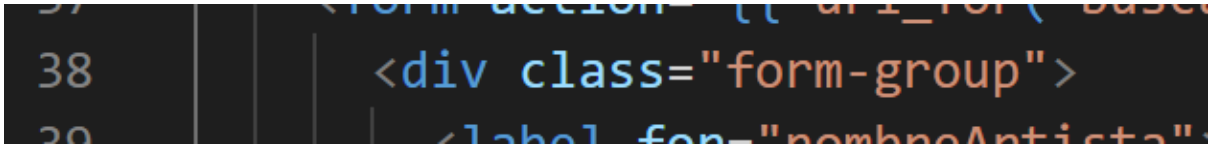


Esta etiqueta `div` crea un contenedor para el formulario y se le asigna la clase "formulario" para aplicar estilos CSS específicos si es necesario.

2. Etiqueta `form`:

La etiqueta `form` representa un formulario HTML. El atributo `action` especifica la URL a la que se enviará la información del formulario cuando se envíe, y utiliza la función `url_for` de Flask para generar la URL dinámicamente en función de la función `buscador()`. El atributo `method` especifica el método HTTP que se utilizará para enviar el formulario, en este caso, "POST".

3. Etiqueta `div` con clase "form-group":

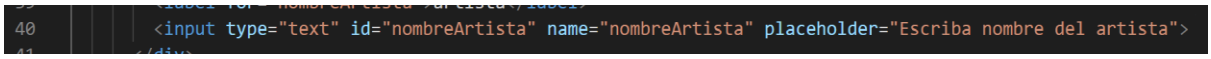


Esta etiqueta `div` se utiliza para agrupar elementos relacionados dentro del formulario, como las etiquetas de texto y los campos de entrada.

4. Etiqueta `label`:

La etiqueta `label` se utiliza para crear una etiqueta de texto descriptiva para un campo de entrada. El atributo `for` se utiliza para asociar el `label` con un campo de entrada específico mediante su atributo `id`.

5. Campo de entrada `input` para el nombre del artista:



El `input` de tipo "text" crea un campo de entrada de texto en el que los usuarios pueden escribir el nombre del artista. El atributo `id` debe coincidir con el valor del atributo `for` de la etiqueta `label` correspondiente. El atributo `name` especifica el nombre del campo que se enviará junto con su valor cuando se envíe el formulario. El atributo `placeholder` muestra un texto de ejemplo en el campo de entrada antes de que el usuario ingrese información.

6. Campo de entrada `input` para la ciudad:

```
44          <input type="text" id="ciudad" name="ciudad" placeholder="Escriba ciudad">
```

Este campo de entrada de texto permite a los usuarios ingresar el nombre de la ciudad. Al igual que el campo del nombre del artista, tiene un `id`, un `name` y un `placeholder` similares.

7. Otro `div` con clase "form-group":

```
46          <div class="form-group">
```

Este `div` se utiliza para agrupar el botón de envío.

8. Botón de envío:

```
47          <button type="submit">Buscar</button>
```

Este botón `submit` permite a los usuarios enviar el formulario. Cuando se hace clic en el botón, los datos del formulario se enviarán a la URL especificada en el atributo `action` del formulario, que en este caso es la función `buscador()`.

Este formulario HTML permite a los usuarios ingresar el nombre del artista y la ciudad para realizar una búsqueda, y cuando se envía el formulario, los datos ingresados se enviarán a la función `

```
35  <!-- Formulario de búsqueda -->
36  <div class="formulario">
37    <form action="{{ url_for('buscador') }}" method="POST">
38      <div class="form-group">
39        <label for="nombreArtista">artista</label>
40        <input type="text" id="nombreArtista" name="nombreArtista" placeholder="Escriba nombre del artista">
41      </div>
42      <div class="form-group">
43        <label for="ciudad">Ciudad </label>
44        <input type="text" id="ciudad" name="ciudad" placeholder="Escriba ciudad">
45      </div>
46      <div class="form-group">
47        <button type="submit">Buscar</button>
48      </div>
49    </form>
50  </div>
```

Código completo del formulario

¿Cómo funciona el cuadro que muestra los resultados del archivo Buscador.html?

Este código es una parte de la plantilla `buscador.html` que se utiliza para mostrar los resultados de búsqueda de eventos. A continuación, trataré de explicar el código:

1. Etiqueta `div` con clase "resultados-container":

```
26      <!-- AQUI ESTA EL CUADRO PARA MOSTRAR LOS EVENTOS -->
27      <div class="resultados-container">
```

Esta etiqueta `div` crea un contenedor para los resultados de búsqueda de eventos y se le asigna la clase "resultados-container" para aplicar estilos CSS específicos si es necesario.

2. Bloque `for` para iterar sobre los eventos en `evento_data`:

```
28          {% for evento in evento_data %}
```

Este bloque `for` de Jinja2 se utiliza para iterar sobre los elementos en la lista `evento_data` y generar dinámicamente el código HTML para cada evento en la lista. Esto permite mostrar múltiples eventos en la página, en lugar de tener que escribir manualmente el código HTML para cada evento.

3. Etiqueta `div` con clase "cuadro-flotanteResultados":

```
29          <div class="cuadro-flotanteResultados">
```

Esta etiqueta `div` crea un cuadro flotante para mostrar la información del evento y se le asigna la clase "cuadro-flotanteResultados" para aplicar estilos CSS específicos si es necesario.

4. Etiqueta `div` con clase "texto":

```
30              <div class="texto">
```

Esta etiqueta `div` se utiliza para agrupar el texto descriptivo del evento, como el título, la ubicación, la dirección, el precio y la fecha.

5. Etiqueta `h1` para el título del evento:

```
<h1 class="tituloTexto">{{ evento['nombre'] }}</h1>
```

Esta etiqueta `h1` muestra el nombre del evento, que se obtiene del diccionario `evento` y su clave correspondiente `'nombre'`. La clase "tituloTexto" se aplica para aplicar estilos CSS específicos si es necesario.

6. Etiqueta `p` para la ciudad y el país:

```
<p class="ciudadPais">{{ evento['lugar'] }}</p>
<p class="Direccion">{{ evento['direccion'] }}</p>
<p class="Precio">{{ evento['precio'] }}</p>
<p class="Fecha">{{ evento['fecha'] }}</p>
```

Esta etiqueta `p` muestra la ubicación del evento (ciudad y país) y se obtiene del diccionario `evento` y su clave correspondiente `'lugar'`. La clase "ciudadPais" se aplica para aplicar estilos CSS específicos si es necesario, y así mismo para la direccion, precio y fecha

7. Etiqueta `a` para el enlace de compra de entradas:

```
<a href="{{ evento['link'] }}" class="botonComprar">Comprar entrada</a>
```

```
<!-- AQUI ESTA EL CUADRO PARA MOSTRAR LOS EVENTOS -->
<div class="resultados-container">
  {% for evento in evento_data %}
  <div class="cuadro-flotanteResultados">
      <div class="texto">
          <h1 class="tituloTexto">{{ evento['nombre'] }}</h1>
          <p class="ciudadPais">{{ evento['lugar'] }}</p>
          <p class="Direccion">{{ evento['direccion'] }}</p>
          <p class="Precio">{{ evento['precio'] }}</p>
          <p class="Fecha">{{ evento['fecha'] }}</p>
          <a href="{{ evento['link'] }}" class="botonComprar">Comprar entrada</a>
      </div>
  </div>
  {% endfor %}
</div>
```

Código completo del cuadro de resultados

Propuestas de mejora

Análisis y mejoras para la aplicación web "Events":

1. Implementación de MySQL: Actualmente, la aplicación no utiliza una base de datos para almacenar y gestionar los datos. Sería beneficioso implementar una base de datos MySQL para almacenar información sobre eventos, usuarios registrados, etc. Esto permitirá una gestión más eficiente de los datos y facilitará operaciones como consultas, inserciones, actualizaciones y eliminaciones.

2. Inicio de sesión: Agregar una funcionalidad de inicio de sesión proporcionará una capa adicional de seguridad y permitirá a los usuarios acceder a características personalizadas, como guardar eventos favoritos, realizar compras, etc. Esto implica la creación de un sistema de autenticación, almacenamiento seguro de contraseñas y manejo de sesiones.

3. Mejorar el diseño responsive: Es importante asegurarse de que la aplicación sea completamente responsive, es decir, que se ajuste y se vea bien en diferentes dispositivos y tamaños de pantalla. Revisar y mejorar el diseño de los cuadros e imágenes en las plantillas, como en la página "acerca_de.html", garantizando que se adapten correctamente a diferentes resoluciones de pantalla.

4. Limpieza del tratamiento del JSON: El código actual que maneja el JSON podría beneficiarse de una refactorización y una estructura más limpia. Esto implica modularizar el código, separando las funciones y responsabilidades en diferentes componentes o archivos, y utilizar buenas prácticas de programación para mejorar la legibilidad del código.

7. Implementación de paginación: Si la cantidad de eventos devueltos por la API de Ticketmaster es grande, considerar la implementación de paginación para mostrar los resultados de manera más eficiente. Esto permitirá cargar y mostrar eventos en lotes más pequeños, mejorando el rendimiento de la aplicación y la experiencia del usuario.

8. Pruebas unitarias: Implementar pruebas unitarias para asegurar el correcto funcionamiento de las diferentes partes de la aplicación. Esto ayudará a identificar y solucionar errores o problemas potenciales antes de implementar nuevas características o cambios importantes.

Estas son algunas sugerencias de mejoras para la aplicación web "Events". La implementación de cada mejora puede llegar a requerir tiempo y conocimientos adicionales, por lo que es importante priorizar según los recursos y requisitos específicos del proyecto.

Adaptar la aplicación para Spark

Para adaptar la aplicación web "Events" para un eventual procesamiento distribuido en Spark, se deben considerar los siguientes aspectos:

En primer lugar, distribuiría el trabajo entre múltiples máquinas para acelerar el procesamiento. En lugar de realizar todas las tareas en una sola máquina, Spark me permite dividir el trabajo en partes más pequeñas y distribuirlo entre varias máquinas. Esto mejoraría la velocidad de procesamiento y permitiría manejar grandes volúmenes de datos de manera más eficiente.

Además, almacenaría los datos en un formato especializado compatible con Spark, como Parquet o Avro. Estos formatos optimizados permiten un acceso más rápido y eficiente a los datos, lo cual es crucial cuando se trabaja con grandes conjuntos de datos.

Otra consideración importante sería dividir el trabajo en tareas más pequeñas y paralelizarlas. Spark divide el trabajo en tareas más pequeñas y las distribuye entre las máquinas del clúster. Cada máquina puede procesar una tarea específica al mismo tiempo, lo que acelera el procesamiento global. Esta división en tareas más pequeñas mejora la eficiencia y permite un procesamiento más rápido.

Aprovecharía el poder de muchas máquinas en el clúster de Spark. Al distribuir el trabajo entre varias máquinas, puedo aprovechar su capacidad de procesamiento combinada para acelerar el procesamiento y manejar cargas de trabajo intensivas en tiempo de computación. Esto es especialmente beneficioso cuando se trabaja con conjuntos de datos grandes o cuando se necesitan realizar operaciones complejas.

Por último, optimizaría el rendimiento de la aplicación en Spark. Ajustaría la distribución de datos entre las máquinas para equilibrar la carga de trabajo y optimizaría las consultas y operaciones para obtener resultados más rápidos. También podría realizar ajustes adicionales en la configuración de Spark para optimizar la ejecución de la aplicación.

En resumen, al adaptar la aplicación web "Events" para utilizar Spark en un entorno de procesamiento distribuido, aprovecharía la capacidad de distribución de Spark, utilizaría formatos de datos optimizados, dividiría el trabajo en tareas más pequeñas y paralelizaría su procesamiento. También aprovecharía el poder combinado de muchas máquinas en el clúster y optimizaría el rendimiento para obtener resultados más rápidos y eficientes.

Referencias

Encontré este artículo que me ayudó ya que era justo lo que estaba buscando, también he de reconocer que me inspire mucho de la página de Apple para ciertos aspectos visuales para ayudar a mantener un lenguaje de diseño limpio, minimalista y agradable a la vista.

https://medium.com/@carlparm/ticketmaster-api-fd5e439c9b3a

También videos que me ayudaban a completar fragmentos de código para facilitarme con muchas cosas para la visualización de contenido o funcionamiento de la aplicación web

https://www.youtube.com/watch?v=_r5wYV8a8O0
https://www.youtube.com/watch?v=bYo-S5F6r8Y
https://www.youtube.com/watch?v=cJ2-a1ITuO4
https://www.youtube.com/watch?v=yWRpbadrs1U
https://www.youtube.com/watch?v=6qko7Nbe8YA
https://www.youtube.com/watch?v=-1DmVCPB6H8
https://www.youtube.com/watch?v=rK0itnj_aBk