Construye un Bot de Historias con Inteligencia Artificial

Esta entrada del blog fue escrita con la asistencia de ChatGPT-4.

Tabla de Contenidos

Introducción

Esta publicación de blog ofrece una guía completa sobre la arquitectura e implementación de una aplicación de bot de historias impulsada por inteligencia artificial. El proyecto implica la

generación de historias personalizadas utilizando una interfaz web. Utilizamos Python, Flask y React para el desarrollo y desplegamos en AWS. Además, empleamos Prometheus para la monitorización y ElasticSearch, Kibana y Logstash para la gestión de registros. La gestión de DNS se realiza a través de GoDaddy y Cloudflare, con Nginx actuando como puerta de enlace para la gestión de certificados SSL y cabeceras de solicitud.

Arquitectura del Proyecto

Backend El backend del proyecto está construido utilizando Flask, un framework ligero de aplicaciones web WSGI en Python. El backend maneja solicitudes de API, gestiona la base de datos, registra las actividades de la aplicación y se integra con Prometheus para la monitorización.

Aquí tienes un desglose de los componentes del backend:

1. Configuración de la Aplicación Flask:
    - La aplicación Flask se inicializa y configura para utilizar varias extensiones como Flask-CORS para manejar el Intercambio de Recursos de Origen Cruzado (CORS) y Flask-Migrate para gestionar las migraciones de la base de datos.
    - Se inicializan las rutas de la aplicación y se habilita CORS para permitir solicitudes de origen cruzado.
    - La base de datos se inicializa con configuraciones predeterminadas, y se configura un registrador personalizado (logger) para formatear las entradas de registro para Logstash.

```
    from flask import Flask
    from flask_cors import CORS
    from .routes import initialize_routes
    from .models import db, insert_default_config
    from flask_migrate import Migrate
    import logging
    from logging.handlers import RotatingFileHandler
    from prometheus_client import Counter, generate_latest, Gauge

app = Flask(__name__)
app.config.from_object('api.config.BaseConfig')
```

```
    db.init_app(app)
    initialize_routes(app)
    CORS(app)
    migrate = Migrate(app, db)
```

2. Registro y Monitoreo:

    - La aplicación utiliza `RotatingFileHandler` para gestionar archivos de registro y formatea los registros utilizando un formateador personalizado.
    - Las métricas de Prometheus están integradas en la aplicación para rastrear el recuento de solicitudes y la latencia.

```
REQUEST_COUNT = Counter('flask_app_request_count', 'Cantidad total de solicitudes de la aplicación Flas
REQUEST_LATENCY = Gauge('flask_app_request_latency_seconds', 'Latencia de las solicitudes', ['method',

def setup_loggers():
    logstash_handler = RotatingFileHandler('app.log', maxBytes=100000000, backupCount=1)
    logstash_handler.setLevel(logging.DEBUG)
    logstash_formatter = CustomLogstashFormatter()
    logstash_handler.setFormatter(logstash_formatter)

root_logger = logging.getLogger()
root_logger.setLevel(logging.DEBUG)
root_logger.addHandler(logstash_handler)

app.logger.addHandler(logstash_handler)
werkzeug_logger = logging.getLogger('werkzeug')
werkzeug_logger.setLevel(logging.DEBUG)
werkzeug_logger.addHandler(logstash_handler)

setup_loggers()
```

3. Manejo de Solicitudes:

    - La aplicación captura métricas antes y después de cada solicitud, generando un ID de seguimiento para rastrear el flujo de la solicitud.

```python
def generate_trace_id(length=4):
    characters = string.ascii_letters + string.digits
    return ''.join(random.choice(characters) for _ in range(length))

@app.before_request
def before_request():
    request.start_time = time.time()
    trace_id = request.headers.get('X-Trace-Id', generate_trace_id())
    g.trace_id = trace_id

    @app.after_request
    def after_request(response):
        response.headers['X-Trace-Id'] = g.trace_id
        request_latency = time.time() - getattr(request, 'start_time', time.time())
        REQUEST_COUNT.labels(method=request.method, endpoint=request.path, http_status=response.status_
        REQUEST_LATENCY.labels(method=request.method, endpoint=request.path).set(request_latency)
        return response
```

Frontend El frontend del proyecto está construido utilizando React, una biblioteca de JavaScript para construir interfaces de usuario. Interactúa con la API del backend para gestionar los prompts de historias y proporciona una interfaz de usuario interactiva para generar y gestionar historias personalizadas.

1. Componentes de React:
    - El componente principal maneja la entrada del usuario para los prompts de historias e interactúa con la API del backend para gestionar estas historias.

```javascript
import React, { useState, useEffect } from 'react';
import { ToastContainer, toast } from 'react-toastify';
import 'react-toastify/dist/ReactToastify.css';
import { apiFetch } from './api';
import './App.css';

function App() {
  const [prompts, setPrompts] = useState([]);
  const [newPrompt, setNewPrompt] = useState('');
  const [isLoading, setIsLoading] = useState(false);
```

```
useEffect(() => {
  fetchPrompts();
}, []);

const fetchPrompts = async () => {
  setIsLoading(true);
  try {
    const response = await apiFetch('prompts');
    if (response.ok) {
      const data = await response.json();
      setPrompts(data);
    } else {
      toast.error('Error al obtener los prompts');
    }
  } catch (error) {
    toast.error('Ocurrió un error al obtener los prompts');
  } finally {
    setIsLoading(false);
  }
};

      const addPrompt = async () => {
        if (!newPrompt) {
          toast.warn('El contenido del prompt no puede estar vacío');
          return;
        }
        setIsLoading(true);
        try {
          const response = await apiFetch('prompts', {
            method: 'POST',
            headers: {
              'Content-Type': 'application/json',
            },
            body: JSON.stringify({ content: newPrompt }),
          });
          if (response.ok) {
            fetchPrompts();
```

```
      setNewPrompt('');
      toast.success('Prompt añadido exitosamente');
    } else {
      toast.error('Error al añadir el prompt');
    }
  } catch (error) {
    toast.error('Ocurrió un error al añadir el prompt');
  } finally {
    setIsLoading(false);
  }
};

const deletePrompt = async (promptId) => {
  setIsLoading(true);
  try {
    const response = await apiFetch(`prompts/${promptId}`, {
      method: 'DELETE',
    });
    if (response.ok) {
      fetchPrompts();
      toast.success('Prompt eliminado correctamente');
    } else {
      toast.error('Error al eliminar el prompt');
    }
  } catch (error) {
    toast.error('Ocurrió un error al eliminar el prompt');
  } finally {
    setIsLoading(false);
  }
};

return (
  <div className="app">
    <h1>Bot de Historias con IA</h1>
    <div>
      <input
        type="text"
```

```
            value={newPrompt}
            onChange={(e) => setNewPrompt(e.target.value)}
            placeholder="Nuevo Prompt"
          />
          <button onClick={addPrompt} disabled={isLoading}>Agregar Prompt</button>
        </div>
        {isLoading ? (
          <p>Cargando...</p>
        ) : (
          <ul>
            {prompts.map((prompt) => (
              <li key={prompt.id}>
                {prompt.content}
                <button onClick={() => deletePrompt(prompt.id)}>Eliminar</button>
              </li>
            ))}
          </ul>
        )}
        <ToastContainer />
      </div>
    );
  }

  export default App;
  ```

2. Integración de API:
   - El frontend interactúa con la API del backend utilizando solicitudes fetch para gestionar los pro

```javascript
export const apiFetch = (endpoint, options) => {
  return fetch(`https://api.yourdomain.com/${endpoint}`, options);
};
```

Despliegue

El proyecto está desplegado en AWS, con la gestión de DNS manejada a través de GoDaddy y Cloudflare. Se utiliza Nginx como puerta de enlace para la gestión de certificados SSL y cabeceras de solicitud. Utilizamos Prometheus para la monitorización y ElasticSearch, Kibana y Logstash para la gestión de registros.

1. Script de Implementación:
    - Utilizamos Fabric para automatizar tareas de implementación como la preparación de directorios locales y remotos, la sincronización de archivos y la configuración de permisos.

```
from fabric import task
from fabric import Connection

server_dir = '/home/project/server'
web_tmp_dir = '/home/project/server/tmp'

@task
def prepare_remote_dirs(c):
    if not c.run(f'test -d {server_dir}', warn=True).ok:
        c.sudo(f'mkdir -p {server_dir}')
        c.sudo(f'chmod -R 755 {server_dir}')
        c.sudo(f'chmod -R 777 {web_tmp_dir}')
        c.sudo(f'chown -R ec2-user:ec2-user {server_dir}')

@task
def deploy(c, install='false'):
    prepare_remote_dirs(c)
    pem_file = './aws-keypair.pem'
    rsync_command = (f'rsync -avz --exclude="api/db.sqlite3" '
                     f'-e "ssh -i {pem_file}" --rsync-path="sudo rsync" '
                     f'{tmp_dir}/ {c.user}@{c.host}:{server_dir}')
    c.local(rsync_command)
    c.sudo(f'chown -R ec2-user:ec2-user {server_dir}')
```

2. Configuración de ElasticSearch:

- La configuración de ElasticSearch incluye ajustes para el clúster, el nodo y las configuraciones de red.

```
cluster.name: my-application
node.name: node-1
path.data: /var/lib/elasticsearch
path.logs: /var/log/elasticsearch
network.host: 0.0.0.0
http.port: 9200
discovery.seed_hosts: ["127.0.0.1"]
cluster.initial_master_nodes: ["node-1"]
```

3. Configuración de Kibana:
    - La configuración de Kibana incluye ajustes para el servidor y los hosts de ElasticSearch.

```
server.port: 5601
server.host: "0.0.0.0"
elasticsearch.hosts: ["http://localhost:9200"]
```

4. Configuración de Logstash:
    - Logstash está configurado para leer archivos de registro, analizarlos y enviar los registros analizados a ElasticSearch.

    “‘plaintext input { file { path => “/home/project/server/app.log” start_position => “beginning” sincedb_path => “/dev/null” } }

    filter { json { source => “message” } }

    ```
    output {
      elasticsearch {
        hosts => ["http://localhost:9200"]
        index => "flask-logs-%{+YYYY.MM.dd}"
      }
    }
    ```

Configuración de Nginx y Certificado SSL de Let's Encrypt

Para garantizar una comunicación segura, utilizamos Nginx como proxy inverso y Let's Encrypt para los c

1. Define un mapa para manejar los orígenes permitidos:

```nginx
    map $http_origin $cors_origin {
        default "https://example.com";
        "http://localhost:3000" "http://localhost:3000";
        "https://example.com" "https://example.com";
        "https://www.example.com" "https://www.example.com";
    }
    ```

2. Redirigir HTTP a HTTPS:

    ```nginx
    server {
        listen 80;
        server_name example.com api.example.com;
    ```

        return 301 https://$host$request_uri;
    }
    ```

3. Configuración principal del sitio para `example.com`:

    ```nginx
    server {
        listen 443 ssl;
        server_name example.com;
    ```

```nginx
        ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
```
```

```
    ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers on;
    ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";

root /home/proyecto/web;
index index.html index.htm index.php default.html default.htm default.php;

location / {
    try_files $uri $uri/ =404;
}

location ~ .*\.(gif|jpg|jpeg|png|bmp|swf)$ {
    expires 30d;
}

location ~ .*\.(js|css)?$ {
    expires 12h;
}

error_page 404 /index.html;
}
```

4. Configuración de la API para `api.example.com`:

```nginx
server {
    listen 443 ssl;
    server_name api.example.com;

ssl_certificate /etc/letsencrypt/live/example.com-0001/fullchain.pem;
ssl_certificate_key /etc/letsencrypt/live/example.com-0001/privkey.pem;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers on;
    ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
```

```
    location / {
        # Limpiar cualquier encabezado Access-Control preexistente
        more_clear_headers 'Access-Control-Allow-Origin';

            # Manejar solicitudes de preflight CORS
            if ($request_method = 'OPTIONS') {
                add_header 'Access-Control-Allow-Origin' $cors_origin;
                add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, PUT, DELETE';
                add_header 'Access-Control-Allow-Headers' 'Origin, Content-Type, Accept, Authorization,
                add_header 'Access-Control-Max-Age' 3600;
                return 204;
            }

add_header 'Access-Control-Allow-Origin' $cors_origin always;
add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, PUT, DELETE' always;
add_header 'Access-Control-Allow-Headers' 'Origin, Content-Type, Accept, Authorization, X-Client-Info, 

nginx             proxy_pass http://127.0.0.1:5000/;             proxy_set_header Host
$host;             proxy_set_header X-Real-IP $remote_addr;             proxy_set_header
X-Forwarded-For $proxy_add_x_forwarded_for;             proxy_set_header X-Forwarded-Proto
$scheme;             proxy_connect_timeout 600s;             proxy_send_timeout 600s;
proxy_read_timeout 600s;             send_timeout 600s;         }     }
```

Conclusión

Este proyecto muestra una arquitectura robusta para una aplicación de bot de historias impulsada por IA, utilizando prácticas y herramientas modernas de desarrollo web. El backend está construido con Flask, asegurando un manejo eficiente de solicitudes y una integración con varios servicios para registro y monitoreo. El frontend, desarrollado con React, proporciona una interfaz de usuario interactiva para gestionar los prompts de historias. Al aprovechar AWS para la implementación, Nginx para la comunicación segura y la pila ELK para la gestión de registros, garantizamos escalabilidad, confiabilidad y mantenibilidad. Esta configuración integral demuestra el poder de combinar tecnologías de vanguardia para ofrecer una experiencia de usuario fluida.