AI を活用したストーリーボットの構築

このブログ記事は、ChatGPT-4 の助けを借りて書かれました。

目次

はじめに

このブログ記事では、AI を活用したストーリーボットアプリケーションのアーキテクチャと実装について包括的なガイドを提供します。このプロジェクトでは、ウェブインターフェースを使

用してパーソナライズされたストーリーを生成します。開発には Python、Flask、React を使用し、AWS にデプロイします。さらに、監視には Prometheus を、ログ管理には ElasticSearch、Kibana、Logstash を使用します。DNS 管理は GoDaddy と Cloudflare を通じて行われ、Nginx が SSL 証明書とリクエストヘッダー管理のゲートウェイとして機能します。

プロジェクトのアーキテクチャ

バックエンド　このプロジェクトのバックエンドは、Python の軽量な WSGI ウェブアプリケーションフレームワークである Flask を使用して構築されています。バックエンドは API リクエストを処理し、データベースを管理し、アプリケーションの活動を記録し、Prometheus との統合を通じて監視を行います。

以下にバックエンドコンポーネントの詳細を示します：

1. Flask アプリケーションのセットアップ:
    - Flask アプリが初期化され、Cross-Origin Resource Sharing（CORS）を扱うための Flask-CORS やデータベースマイグレーションを管理するための Flask-Migrate など、さまざまな拡張機能を使用するように設定されます。
    - アプリケーションのルートが初期化され、CORS が有効化されてクロスオリジンリクエストが許可されます。
    - データベースがデフォルト設定で初期化され、Logstash 用にログエントリをフォーマットするカスタムロガーが設定されます。

```
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. ロギングとモニタリング:
    - アプリケーションは RotatingFileHandler を使用してログファイルを管理し、カスタムフォーマッタを使用してログをフォーマットします。
    - Prometheus メトリクスがアプリケーションに統合されており、リクエスト数とレイテンシを追跡します。

```
REQUEST_COUNT = Counter('flask_app_request_count', 'Flask アプリの総リクエスト数', ['method', 'endpoint
REQUEST_LATENCY = Gauge('flask_app_request_latency_seconds', 'リクエストのレイテンシ', ['method', 'endp

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. リクエスト処理:
    - アプリケーションは各リクエストの前後にメトリクスをキャプチャし、リクエストの流れを追跡するためのトレース ID を生成します。

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

```python
@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_code
    REQUEST_LATENCY.labels(method=request.method, endpoint=request.path).set(request_latency)
    return response
```

フロントエンド　プロジェクトのフロントエンドは、ユーザーインターフェースを構築するための JavaScript ライブラリである React を使用して構築されています。フロントエンドは、バックエンドの API と連携してストーリーのプロンプトを管理し、パーソナライズされたストーリーを生成および管理するためのインタラクティブなユーザーインターフェースを提供します。

1. React コンポーネント:
    - メインコンポーネントは、ストーリーのプロンプトに対するユーザー入力を処理し、バックエンド API と連携してこれらのストーリーを管理します。

```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('プロンプトの取得に失敗しました');
    }
  } catch (error) {
    toast.error('プロンプトの取得中にエラーが発生しました');
  } finally {
    setIsLoading(false);
  }
};

const addPrompt = async () => {
  if (!newPrompt) {
    toast.warn('Prompt の内容が空です');
    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 が正常に追加されました');
    } else {
      toast.error('Prompt の追加に失敗しました');
```

```
    }
  } catch (error) {
    toast.error('Prompt の追加中にエラーが発生しました');
  } finally {
    setIsLoading(false);
  }
};

const deletePrompt = async (promptId) => {
  setIsLoading(true);
  try {
    const response = await apiFetch(`prompts/${promptId}`, {
      method: 'DELETE',
    });
    if (response.ok) {
      fetchPrompts();
      toast.success('プロンプトが正常に削除されました');
    } else {
      toast.error('プロンプトの削除に失敗しました');
    }
  } catch (error) {
    toast.error('プロンプトの削除中にエラーが発生しました');
  } finally {
    setIsLoading(false);
  }
};

return (
  <div className="app">
    <h1>AI 搭載ストーリーボット</h1>
    <div>
      <input
        type="text"
        value={newPrompt}
        onChange={(e) => setNewPrompt(e.target.value)}
        placeholder=" 新しいプロンプト"
      />
```

```
      <button onClick={addPrompt} disabled={isLoading}>プロンプトを追加</button>
    </div>
    {isLoading ? (
      <p>読み込み中...</p>
    ) : (
      <ul>
        {prompts.map((prompt) => (
          <li key={prompt.id}>
            {prompt.content}
            <button onClick={() => deletePrompt(prompt.id)}>削除</button>
          </li>
        ))}
      </ul>
    )}
    <ToastContainer />
  </div>
);

    export default App;
    ```

2. API 統合：
    - フロントエンドは、ストーリープロンプトを管理するために、fetch リクエストを使用してバックエント

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

デプロイ

このプロジェクトは AWS 上にデプロイされており、DNS 管理は GoDaddy と Cloudflare を通じて行われています。Nginx は SSL 証明書とリクエストヘッダー管理のためのゲートウェイとして使用されています。監視には Prometheus を、ログ管理には ElasticSearch、Kibana、Logstash を使用しています。

1. デプロイスクリプト:

- Fabric を使用して、ローカルおよびリモートディレクトリの準備、ファイルの同期、権限の設定などのデプロイタスクを自動化しています。

```
 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. ElasticSearch の設定:
    - ElasticSearch のセットアップには、クラスター、ノード、およびネットワーク設定の構成が含まれます。

```
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. Kibana の設定:
    - Kibana のセットアップには、サーバーと ElasticSearch ホストの設定が含まれています。

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

4. Logstash の設定:
    - Logstash は、ログファイルを読み取り、それらを解析し、解析されたログを ElasticSearch に出力するように設定されています。

```
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}"
  }
}
```

Nginx の設定と Let's Encrypt SSL 証明書

安全な通信を確保するために、Nginx をリバースプロキシとして使用し、Let's Encrypt で SSL 証明書を設定しています。以下は、HTTP から HTTPS へのリダイレクトと SSL 証明書の設定を行うための Nginx 設定です。

1. 許可されたオリジンを処理するためのマップを定義します:

    ```
    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. HTTPをHTTPSにリダイレクトする:

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

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

3. example.com のメインサイト設定:

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

        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/project/web;
        index index.html index.htm index.php default.html default.htm default.php;
```
```

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

この Nginx の設定は、リクエストされた URI に対応するファイルが存在するかどうかをチェックします。もしファイルが存在しない場合、404 エラーを返します。

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

この設定は、Nginx サーバーにおいて、指定された画像ファイル形式（GIF、JPG、JPEG、PNG、BMP、SWF）に対するキャッシュの有効期限を 30 日に設定するものです。これにより、これらのファイルがブラウザにキャッシュされ、30 日間は再ダウンロードされずに表示されるようになります。

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

この Nginx の設定は、JavaScript (`js`) と CSS (`css`) ファイルに対するキャッシュの有効期限を 12 時間に設定しています。これにより、ブラウザはこれらのファイルを 12 時間キャッシュし、その間はサーバーに再度リクエストを送らずにキャッシュされたファイルを使用します。これにより、ページの読み込み速度が向上し、サーバーの負荷が軽減されます。

```
        error_page 404 /index.html;
    }
    ```

4. `api.example.com` のAPI設定:

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

```nginx
        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 / {
            # 既存のAccess-Controlヘッダーをクリアする
            more_clear_headers 'Access-Control-Allow-Origin';

            # CORSプリフライトリクエストの処理
            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, X
```

このコードは、Nginx サーバーで CORS（Cross-Origin Resource Sharing）を設定するためのものです。各ヘッダーは、異なるオリジンからのリクエストを許可するために使用されます。

- `Access-Control-Allow-Origin`: 許可するオリジンを指定します。`$cors_origin` は変数で、許可するオリジンの値が格納されています。
- `Access-Control-Allow-Methods`: 許可する HTTP メソッドを指定します。ここでは `GET`, `POST`, `OPTIONS`, `PUT`, `DELETE` が許可されています。
- `Access-Control-Allow-Headers`: 許可するリクエストヘッダーを指定します。ここでは、`Origin`, `Content-Type`, `Accept`, `Authorization`, `X-Client-Info`, `X-Trace-Id`, `X-Requested-With`, `X-HTTP-Method-Override`, `DNT`, `Keep-Alive`, `User-Agent`, `If-Modified-Since`, `Cache-Control`, `Content-Range`, `Range` が許可されています。

`always` キーワードは、これらのヘッダーが常にレスポンスに含まれることを保証します。

```
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;         }     }
```

結論

このプロジェクトは、AI を活用したストーリーボットアプリケーションの堅牢なアーキテクチャを紹介し、現代のウェブ開発手法とツールを活用しています。バックエンドは Flask で構築されており、効率的なリクエスト処理と、ロギングやモニタリングのためのさまざまなサービスとの統合を実現しています。フロントエンドは React で構築され、ストーリープロンプトを管理するためのインタラクティブなユーザーインターフェースを提供します。AWS をデプロイメントに、Nginx を安全な通信に、ELK スタックをログ管理に活用することで、スケーラビリティ、信頼性、保守性を確保しています。この包括的なセットアップは、最先端のテクノロジーを組み合わせることで、シームレスなユーザーエクスペリエンスを提供する力を示しています。