Build an AI-Powered Story Bot with Flask, React, and ELK

This blog post was written with the assistance of ChatGPT-4.

Table of Contents

Introduction

This blog post provides a comprehensive guide to the architecture and implementation of an AI-powered story bot application. The project involves generating personalized stories using a web interface. We use Python, Flask, and React for development and deploy on AWS. Additionally, we use Prometheus for monitoring and ElasticSearch, Kibana, and Logstash for log management. DNS management is handled through GoDaddy and Cloudflare, with Nginx serving as a gateway for SSL certificate and request header management.

Project Architecture

Backend The backend of the project is built using Flask, a lightweight WSGI web application framework in Python. The backend handles API requests, manages the database, logs application activities, and integrates with Prometheus for monitoring.

Here' s a breakdown of the backend components:

1. **Flask Application Setup**:
   - The Flask app is initialized and configured to use various extensions like Flask-CORS for handling Cross-Origin Resource Sharing and Flask-Migrate for managing database migrations.
   - The application routes are initialized, and CORS is enabled to allow cross-origin requests.
   - The database is initialized with default configurations, and a custom logger is set up to format log entries for 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. **Logging and Monitoring**:
   - The application uses RotatingFileHandler to manage log files and formats logs using a custom formatter.
   - Prometheus metrics are integrated into the application to track request count and latency.

```
REQUEST_COUNT = Counter('flask_app_request_count', 'Total request count of the Flask App', ['metho
REQUEST_LATENCY = Gauge('flask_app_request_latency_seconds', 'Request latency', ['method', 'endpoi

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. **Request Handling**:
   - The application captures metrics before and after each request, generating a trace ID to track request flow.

```
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 The frontend of the project is built using React, a JavaScript library for building user interfaces. It interacts with the backend API to manage story prompts and provides an interactive user interface for generating and managing personalized stories.

1. **React Components**:
   - The main component handles user input for story prompts and interacts with the backend API to manage these stories.

```
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('Failed to fetch prompts');
      }
    } catch (error) {
      toast.error('An error occurred while fetching prompts');
    } finally {
      setIsLoading(false);
    }
```

```
};

const addPrompt = async () => {
  if (!newPrompt) {
    toast.warn('Prompt content cannot be empty');
    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 added successfully');
    } else {
      toast.error('Failed to add prompt');
    }
  } catch (error) {
    toast.error('An error occurred while adding the 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 deleted successfully');
      } else {
        toast.error('Failed to delete prompt');
      }
    } catch (error) {
      toast.error('An error occurred while deleting the prompt');
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="app">
      <h1>AI-Powered Story Bot</h1>
      <div>
        <input
          type="text"
          value={newPrompt}
          onChange={(e) => setNewPrompt(e.target.value)}
          placeholder="New Prompt"
        />
        <button onClick={addPrompt} disabled={isLoading}>Add Prompt</button>
      </div>
      {isLoading ? (
        <p>Loading...</p>
      ) : (
        <ul>
          {prompts.map((prompt) => (
            <li key={prompt.id}>
              {prompt.content}
              <button onClick={() => deletePrompt(prompt.id)}>Delete</button>
            </li>
          ))}
        </ul>
      )}
      <ToastContainer />
    </div>
  );
}

export default App;
```

2. **API Integration**:
   - The frontend interacts with the backend API using fetch requests to manage story prompts.

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

Deployment

The project is deployed on AWS, with DNS management handled through GoDaddy and Cloudflare. Nginx is used as a gateway for SSL certificate and request header management. We use Prometheus for monitoring and ElasticSearch, Kibana, and Logstash for log management.

1. **Deployment Script**:
   - We use Fabric to automate deployment tasks such as preparing local and remote directories, syncing files, and setting permissions.

```
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 Configuration**:
    - The ElasticSearch setup includes configurations for the cluster, node, and network settings.

```
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 Configuration**:
    - The Kibana setup includes configurations for the server and ElasticSearch hosts.

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

4. **Logstash Configuration**:
    - Logstash is configured to read log files, parse them, and output the parsed logs to ElasticSearch.

```
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 Configuration and Let’ s Encrypt SSL Certificate

To ensure secure communication, we use Nginx as a reverse proxy and Let’ s Encrypt for SSL certificates. Below is the Nginx configuration for handling HTTP to HTTPS redirection and setting up the SSL certificates.

1. **Define a map to handle the allowed origins**:

   ```
   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. **Redirect HTTP to HTTPS**:

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

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

3. **Main site configuration for `example.com`**:

   ```
   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;
    }

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

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

    error_page 404 /index.html;
}
```

4. **API configuration for `api.example.com`:**

```
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 / {
        # Clear any pre-existing Access-Control headers
        more_clear_headers 'Access-Control-Allow-Origin';

        # Handle CORS preflight requests
        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-C

        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;
    }
}
```

Conclusion

This project showcases a robust architecture for an AI-powered story bot application, utilizing modern web development practices and tools. The backend is built with Flask, ensuring efficient request handling and integration with various services for logging and monitoring. The frontend, built with React, provides an interactive user interface for managing story prompts. By leveraging AWS for deployment, Nginx for secure communication, and the ELK stack for log management, we ensure scalability, reliability, and maintainability. This comprehensive setup demonstrates the power of combining cutting-edge technologies to deliver a seamless user experience.