RAG-Enabled Chat Application

Technical Documentation & Deployment Guide

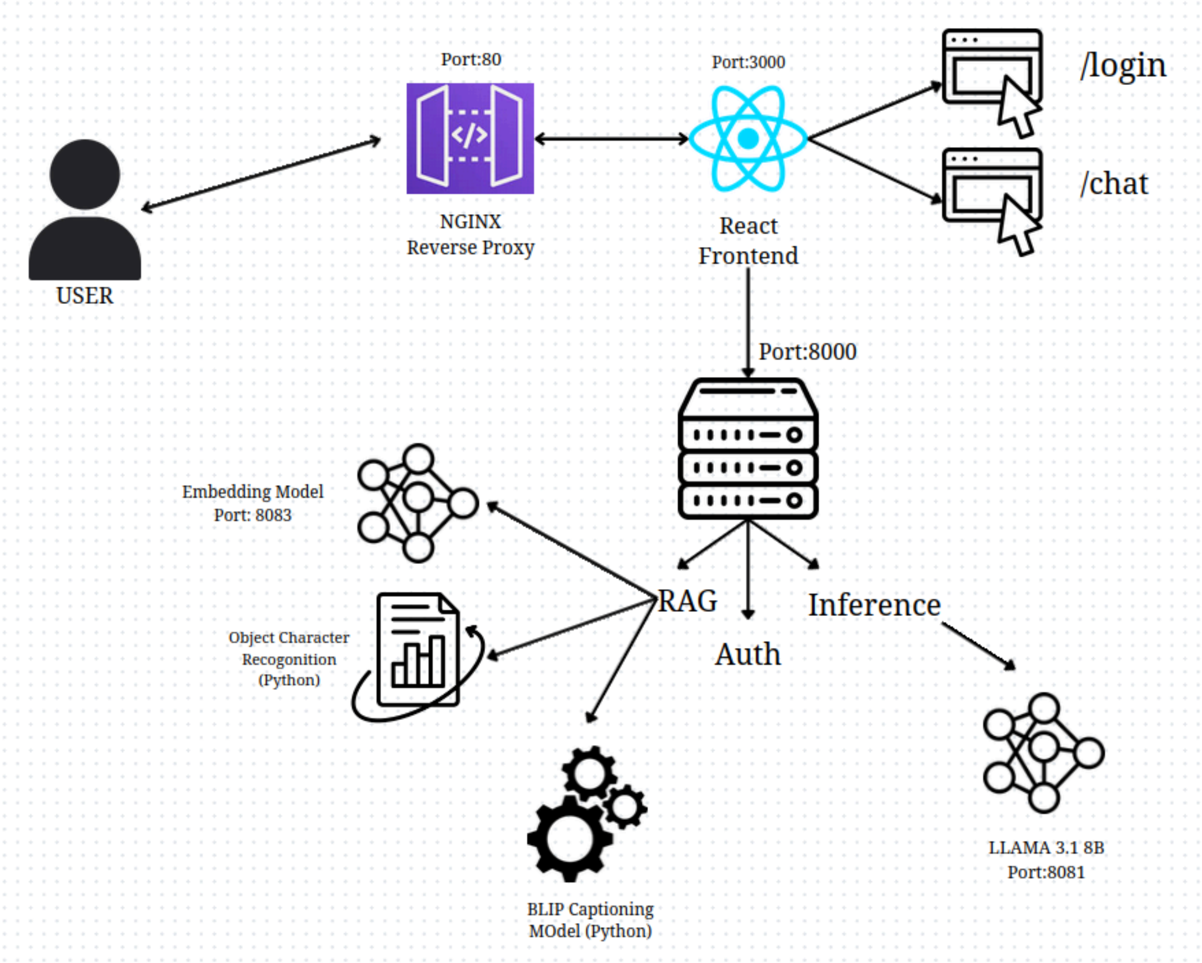


Table of Contents

System Overview

This document provides comprehensive technical documentation for a Retrieval-Augmented Generation (RAG) enabled chat application. The system integrates modern web technologies with advanced language models to deliver intelligent conversational AI capabilities with document-based context awareness.

Technology Stack

Frontend Components

- **Framework**: React
- **Deployment**: Served via NGINX reverse proxy on port 3000

Backend Infrastructure

- **API Server**: FastAPI REST API
- **RAG Engine**: Langchain for Retrieval-Augmented Generation
- **Inference Server**: LLaMA.cpp with Meta LLaMA 3.1 8B model
- **Embedding Model**: nomic-embed-text-v1 for document vectorization
- **Database**: SQLite for user data persistence
- **Vector Database**: FAISS for similarity search

Infrastructure Setup

NGINX Configuration

Installation

Navigate to the offline debian package directory and install NGINX:

```
sudo dpkg -i *.deb
```

Reverse Proxy Configuration

Create the NGINX configuration file:

```
sudo nano /etc/nginx/sites-available/reverse-proxy
```

Add the following configuration:

```
server {
   listen 80;
```

```
    server_name _;

    # Proxy all requests to React frontend on port 3000
    location / {
        proxy_pass http://localhost:3000;
        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;
    }
}
```

Service Configuration

Remove default configuration and enable the reverse proxy:

```
sudo rm /etc/nginx/sites-enabled/default
sudo ln -s /etc/nginx/sites-available/reverse-proxy /etc/nginx/sites-enabled/
```

Test and start the service:

```
sudo nginx -t
sudo systemctl daemon-reload
sudo systemctl enable nginx
sudo systemctl start nginx
```

Deployment Configuration

LLaMA.cpp Server Setup

Environment Configuration

Configure the library path for LLaMA.cpp:

```
export LD_LIBRARY_PATH=build/ggml/src:build/src/:$LD_LIBRARY_PATH
```

Model Server Deployment

Launch the LLaMA inference server:

```
./build/bin/llama-server \
    -m "/path/to/model/Meta-Llama-3.1-8B-Instruct-Q4_K_M.gguf" \
    -np -c 2048 --port 8081 --host 0.0.0.0 \
    --chat-template llama3
```

Launch the embedding server:

```
./build/bin/llama-server \
    -m "/path/to/model/nomic-embed-text-v1.Q4_K_M.gguf" \
    -c 2048 --port 8083 --host 0.0.0.0 \
    --embeddings
```

Automated Service Management

Startup Script Creation

Create an executable startup script:

```
nano startup_services.sh
chmod +x startup_services.sh
```

Startup Script Content

```
#!/bin/bash

# Ubuntu startup script for launching all services
# Ensure paths are updated according to your deployment setup

LOG_DIR="$HOME/logs"
mkdir -p "$LOG_DIR"

echo "Initializing all services..."

# Backend Server
echo "Starting backend server..."
cd /path/to/backend && python3 main.py > "$LOG_DIR/backend.log" 2>&1 &
BACKEND_PID=$!
echo "Backend server started with PID: $BACKEND_PID"

# Frontend Server
echo "Starting frontend server..."
cd /path/to/frontend && PORT=3000 HOST=ip npx serve build -s > "$LOG_DIR/frontend.log"
2>&1 &
FRONTEND_PID=$!
echo "Frontend server started with PID: $FRONTEND_PID"

# LLaMA Inference Server
echo "Starting LLM inference server..."
cd /path/to/llama.cpp && [inference_command] > "$LOG_DIR/llama.log" 2>&1 &
LLM1_PID=$!
```

```
echo "LLaMA inference server started with PID: $LLM1_PID"

# Embedding Server
echo "Starting embedding server..."
cd /path/to/llama.cpp && [embedding_command] > "$LOG_DIR/embed.log" 2>&1 &
LLM2_PID=$!
echo "Embedding server started with PID: $LLM2_PID"

# Process ID Management
echo "$BACKEND_PID" > "$LOG_DIR/pids.txt"
echo "$FRONTEND_PID" >> "$LOG_DIR/pids.txt"
echo "$LLM1_PID" >> "$LOG_DIR/pids.txt"
echo "$LLM2_PID" >> "$LOG_DIR/pids.txt"

echo "All services initialized successfully!"
echo "Log files available in: $LOG_DIR"
echo "Process IDs saved to: $LOG_DIR/pids.txt"
```

SystemD Service Configuration

Create a system service for automated startup:

```
sudo nano /etc/systemd/system/my-services.service
```

Service configuration:

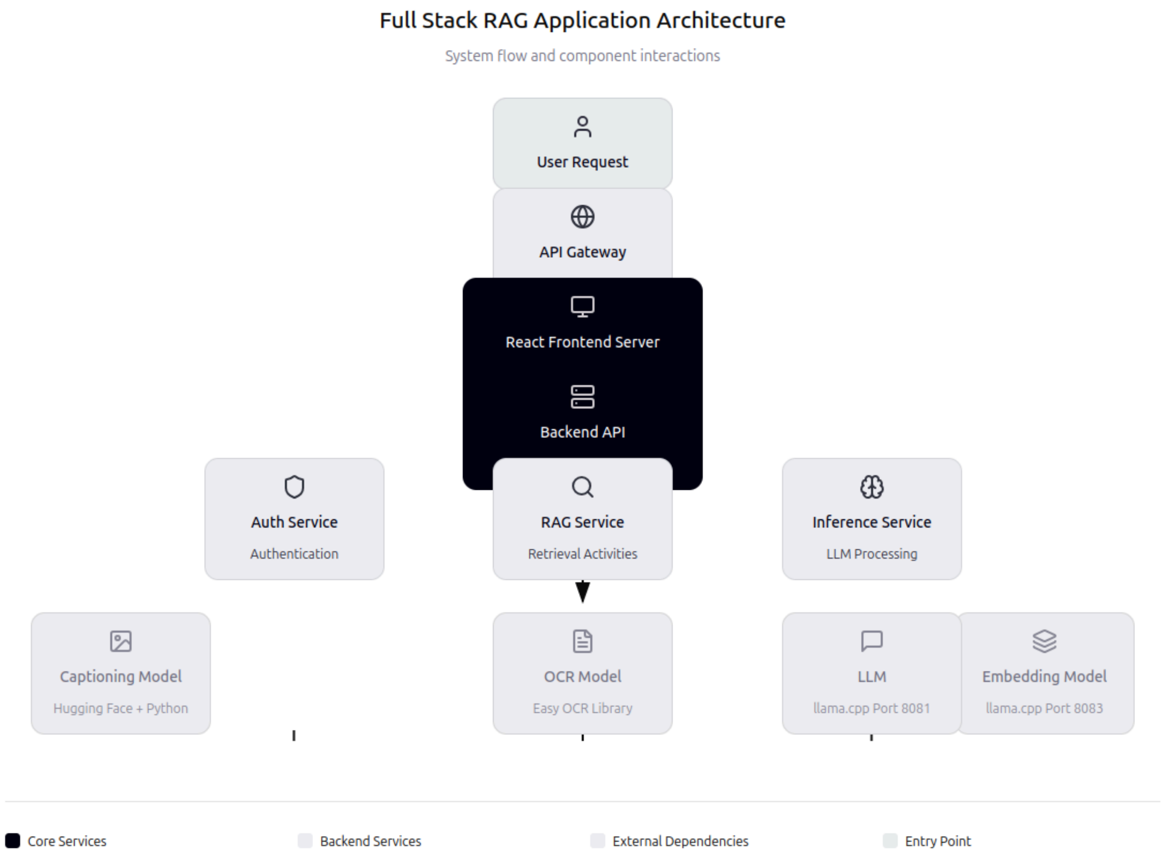
```
[Unit]
Description=RAG Chat Application Services
After=network.target graphical-session.target
Wants=network.target

[Service]
Type=forking
User=your_username
Group=your_username
WorkingDirectory=/home/your_username
ExecStart=/home/your_username/startup_services.sh
ExecStop=/home/your_username/stop_services.sh
Restart=on-failure
RestartSec=5
Environment=HOME=/home/your_username

[Install]
WantedBy=multi-user.target
```

Enable and start the service:

```
sudo systemctl daemon-reload
sudo systemctl enable my-services.service
sudo systemctl start my-services.service
```



Full Stack RAG Application Architecture

System flow and component interactions

Application Architecture

Frontend Components

Authentication Interface

- **Login Page**: Secure user authentication with username/password validation
- **Registration System**: New user account creation with availability checking

Chat Interface Features

- **Session Management**: Clear chat functionality with frontend-only history storage
- **Theme Toggle**: Light/dark mode switching capability
- **Logout Functionality**: Secure session termination with backend cleanup
- **Document Upload**: RAG-enabled file processing with FAISS indexing
- **Real-time Messaging**: Streaming inference with abort capability

Backend Architecture

Service Distribution

- **Frontend**: React application on port 3000 (NGINX reverse proxy from port 80)
- **Backend API**: FastAPI server on port 8000 for authentication and RAG processing
- **LLaMA Inference**: Model server on port 8081 for text generation
- **Embedding Service**: Dedicated server on port 8083 for document vectorization

---

API Documentation

Authentication Endpoints

User Registration

POST /register

Request Body:

```
{
  "username": "string",
  "password": "string"
}
```

Responses:

- `200 OK`: Registration successful
- `409 Conflict`: Username already exists

User Authentication

POST /login

Request Body:

```
{
  "username": "string",
  "password": "string"
}
```

Success Response:

```
{
  "message": "Login successful",
  "username": "string"
}
```

Error Response:

- `401 Unauthorized`: Invalid credentials

Session Termination

POST /logout

Request Body:

```
{
  "username": "string"
}
```

Response:

```
{
  "message": "Logged out successfully."
}
```

Chat and Inference Endpoints

Streaming Message Processing

POST /message/stream

Request Body:

```
{
  "chatHistory": "User: Hi\nAssistant: Hello! How can I help you?",
  "message": "What is Retrieval-Augmented Generation?",
  "model": "llama3.1:8b",
  "ragStatus": true,
  "session_id": "<uuid>",
  "user": "<username>"
}
```

Functionality:

- Processes user queries with optional RAG context
- Performs FAISS similarity search when `ragStatus = true`
- Includes top 4 relevant document chunks in system prompt

- Returns streaming response using `text/event-stream` format

RAG Session Management

Session Creation

POST /rag/create-session

Response:

```
{
  "session_id": "<uuid>",
  "status": "success",
  "message": "RAG session created successfully"
}
```

Session Deletion

DELETE /rag/session/{session_id}

Session Status

GET /rag/session/{session_id}/status

Table Extraction

GET /rag/session/{session_id}/tables

Active Sessions

GET /rag/sessions

Document Processing

Document Upload and Indexing

POST /rag/upload/{session_id}

Form Data:

- `file`: Document file (PDF, DOCX, TXT, PNG, JPG, JPEG)
- `include_tables`: Boolean (default: true)

Processing Pipeline:

1. Text extraction using pdfplumber, docx, and plain text parsers
2. Table extraction and restructuring

3. OCR and image captioning using BLIP + EasyOCR
4. Document chunking via Langchain RecursiveCharacterTextSplitter
5. Embedding generation using LLaMA.cpp model
6. FAISS indexing with Inner Product similarity

Response:

```
{
  "status": "success",
  "message": "Document processed successfully",
  "chunks_created": 24,
  "processing_time": 4.73,
  "filename": "example.pdf",
  "session_id": "<uuid>"
}
```

Document Removal

DELETE /rag/document/{session_id}

System Internals

Document Chunking Strategy

Configuration Parameters:

- `chunk_size`: 800 characters
- `chunk_overlap`: 100 characters
- **Splitting Logic**: Intelligent text segmentation using newlines, punctuation, and whitespace

Embedding Generation

Model Configuration:

- **Server**: LLaMA.cpp on port 8083
- **Model**: nomic-embed-text-v1
- **Preprocessing**: Text prefixed with `"search_query:"` for schema compliance

Similarity Search Implementation

FAISS Configuration:

- **Index Type**: IndexFlatIP with L2-normalized vectors
- **Retrieval Count**: max(4, total_chunks)

- **Similarity Threshold**: 0.75 minimum

Prompt Template System

Message Format Conversion: OpenAI-style messages converted to LLaMA chat format:

<|begin_of_text|><|start_header_id|>role<|end_header_id|>

content<|eot_id|>

Session Lifecycle Management

Memory Management:

- All sessions maintained in-memory
- Each session contains document data, metadata, chunks, and vector index
- **Idle Timeout**: 2 hours
- **Cleanup Schedule**: Background task every 1 hour

Conclusion

This documentation provides a comprehensive guide for deploying and maintaining the RAG-enabled chat application. The system architecture ensures scalability, security, and optimal performance for document-enhanced conversational AI interactions.

For additional support or configuration assistance, please refer to the respective technology documentation or contact the development team.