Tarea case llm

Adrián Yared Armas de la Nuez

Contenido

1. Objetivo..... 1
2. Descripción de la tarea.....2
3. Herramientas utilizadas.....2
4. Flujo de Conversión.....3
5. Componentes clave..... 3
5.1 Traductor.py.....3
5.2 clips_service/.....4
5.2.1 Explicación..... 4
6. Interfaz..... 6

1. Objetivo

Ampliar la funcionalidad de una herramienta CASE de generación automática de código Java a partir de diagramas UML, incorporando un botón que permita traducir el código Java generado a otro lenguaje orientado a objetos, como Python, utilizando un modelo de lenguaje grande (LLM) a través de una API.

2. Descripción de la tarea

A partir de una herramienta web ya existente que permite dibujar diagramas de clases UML y generar automáticamente código Java mediante un sistema experto basado en CLIPS, se propone:
1. Añadir un nuevo botón a la interfaz de usuario que permita traducir el código Java generado en el área de texto a código Python.
2. Para ello, se utilizará un modelo de lenguaje grande (LLM) accesible mediante la API proporcionada por OpenRouter.
3. La aplicación deberá enviar el contenido del área de texto (código Java) al LLM, recibir la respuesta (código equivalente en Python) y mostrarlo en una nueva área de texto o reemplazando el contenido existente.
4. La solución debe manejar los casos de error (por ejemplo, problemas de red o respuestas mal formateadas) de forma adecuada.
5. Se deberá documentar brevemente el funcionamiento de la API utilizada, incluyendo cómo se configura el api_key, api_base y cómo se hace la petición desde JavaScript.

3. Herramientas utilizadas

El núcleo del sistema se construyó utilizando **Python 3.11**, seleccionado por su versatilidad y amplio ecosistema de bibliotecas. Una de las piezas clave del proyecto fue el uso de **CLIPS**, un motor de reglas experto que facilitó la generación automática de código Java a partir de estructuras declarativas, aportando un enfoque basado en reglas para la lógica del sistema.

La comunicación entre los distintos componentes del sistema se estructuró a través de servicios web. Para ello, se utilizó **Flask** como framework principal en el desarrollo del servidor backend, encargado de manejar la lógica de la aplicación web. Paralelamente, se implementó un microservicio independiente con **FastAPI**, especialmente diseñado para ejecutar CLIPS de forma modular y eficiente.

En el lado del cliente, se empleó **JavaScript**, utilizando la **fetch API** para establecer la comunicación asíncrona con el backend. Esto permitió una interacción fluida entre el usuario y la aplicación. La interfaz gráfica fue construida con **HTML5** y **CSS**, brindando una experiencia visual clara, moderna e intuitiva.

Para incorporar funcionalidades basadas en inteligencia artificial, se integró **OpenRouter** como intermediario hacia modelos de lenguaje avanzados, en particular **mistralai/mixtral-8x7b**, lo que permitió enriquecer el sistema con capacidades de comprensión y generación de texto.

Finalmente, la seguridad en el manejo de credenciales y configuraciones sensibles se garantizó mediante el uso de archivos **.env** en conjunto con la biblioteca **python-dotenv**, asegurando así una correcta gestión de variables de entorno en el entorno de desarrollo y producción.

4. Flujo de Conversión

Todo comienza cuando el usuario diseña un **diagrama UML** de forma visual, utilizando una herramienta interactiva que facilita la definición de clases y sus relaciones sin necesidad de escribir código. Una vez completado el diseño, este se exporta automáticamente en formato **XMI**, un estándar que permite representar modelos UML de forma estructurada.

A partir de ese archivo XMI, el sistema genera un archivo **.clp** con hechos que pueden ser procesados por **CLIPS**, un motor de reglas experto. Este archivo representa, en forma lógica y declarativa, la estructura definida por el usuario.

Luego, un **microservicio especializado ejecuta CLIPS**, interpretando esos hechos para generar automáticamente el código correspondiente en **Java**, siguiendo las reglas predefinidas del sistema.

Una vez generado el código Java, entra en juego un **modelo de lenguaje avanzado (LLM)**, al que se accede a través de **OpenRouter**. Este modelo se encarga de **traducir el código Java a Python**, manteniendo la lógica original pero adaptándola al nuevo lenguaje, gracias a sus capacidades de comprensión semántica.

Finalmente, tanto el código Java como su versión en Python se muestran al usuario a través de la interfaz web. Desde ahí, es posible **visualizarlos o descargarlos** fácilmente para su posterior uso.

5. Componentes clave

5.1 Traductor.py

El microservicio en cuestión cumple una función clave dentro del sistema: transformar archivos .clp en código Java. Gracias a su arquitectura basada en microservicios, se logra una mayor portabilidad, se evitan conflictos de dependencias y se facilita la integración con otros componentes.

Este microservicio está desarrollado con FastAPI, lo que permite una implementación rápida y eficiente de servicios web. Dentro de él, el motor CLIPS se ejecuta de forma completamente aislada, lo que garantiza que su funcionamiento no interfiera con otros entornos del sistema.

El servicio expone un endpoint REST accesible desde /run-clips. Este punto de entrada está diseñado para recibir solicitudes HTTP POST que incluyan un archivo .clp. Al recibir el archivo, el servicio:

Ejecuta clips.exe en segundo plano.

Captura la salida estándar generada por CLIPS, que contiene el código Java producido por las reglas definidas.

Devuelve el resultado directamente como texto plano, listo para ser utilizado o transformado en etapas posteriores.

Este enfoque modular no solo aísla la lógica basada en reglas, sino que también facilita el mantenimiento y la escalabilidad del sistema. Además, permite integrar este servicio fácilmente con otros componentes, como el servidor principal desarrollado en Flask, sin necesidad de instalar CLIPS como una dependencia directa en ese entorno.

5.2 clips_service/

5.2.1 Explicación

Construcción del prompt.

```
def construir_prompt(codigo_java: str) -> str:

```

Esta función toma el código Java como texto y lo convierte en un mensaje claro para el modelo.

```
return (
    "Actúa como un ingeniero de software experto en traducción entre lenguajes de programación. "
    "Convierte el siguiente código Java en un script Python funcionalmente equivalente. "
    "Utiliza las mejores prácticas idiomáticas de Python, mantén la lógica original y adapta "
    "las estructuras y convenciones de manera natural al estilo Python.\n\n"
    f"```java\n{codigo_java}\n```\n\n"
    "Devuelve únicamente el código Python resultante, sin explicaciones, encabezados ni comentarios adicionales. "
```

```
    "El formato de salida debe ser limpio y listo para ejecutarse."
)
```

Esta función envía el *prompt* a la API y devuelve la conversión como resultado.

```
def convertir_java_a_python(codigo_java: str, api_key: str = None,

                            modelo: str =
"mistralai/mixtral-8x7b-instruct") -> str:
```

Leer la API key

```
if api_key is None:
    api_key = os.getenv("OPENROUTER_API_KEY")
```

Si no se pasa una API key directamente, intenta leerla de la variable de entorno OPENROUTER_API_KEY.

Verificar disponibilidad

```
if not api_key:
    raise ValueError("...")
```

Lanza un error si no hay clave API.

Preparar la solicitud a la API

```
url = "https://openrouter.ai/api/v1/chat/completions"
prompt = construir_prompt(codigo_java)
```

Define la URL del endpoint de OpenRouter y construye el *prompt* con el código Java.

```
headers = {
    "Authorization": f"Bearer {api_key}",
    "Content-Type": "application/json",
}
```

Prepara los encabezados para la petición HTTP.

```
payload = {
    "model": modelo,
    "messages": [
```

```
        {"role": "system", "content": "Eres un asistente que convierte
código Java a Python."},
        {"role": "user", "content": prompt}
    ],
    "temperature": 0.0
}
```

Envia la solicitud

```
response = requests.post(url, headers=headers, json=payload)
```

Hace una petición POST a la API con los datos anteriores.

Procesar la respuesta

```
if response.status_code != 200:
    raise RuntimeError(...)
```

Lanza error si la petición falla.

```
respuesta = response.json()

if 'choices' not in respuesta:

    raise RuntimeError(...)
```

Verifica que la respuesta tenga el formato esperado

```
return respuesta["choices"][0]["message"]["content"].strip()
```

Extrae y devuelve el código Python generado por el modelo.

6. Interfaz

Código:

```
import os
import requests
from flask import Flask, request

# Importar el módulo de conversión Java → Python
from java_to_python_converter import convertir_java_a_python

# Obtener la API key desde variables de entorno
```

```
api_key = os.getenv("OPENROUTER_API_KEY")

# Inicializar la app Flask
app = Flask(__name__)

# Ruta para archivos generados
UPLOAD_FOLDER = os.path.join(os.getcwd(), 'generated_files')
os.makedirs(UPLOAD_FOLDER, exist_ok=True)

@app.route('/generate-java', methods=['GET'])
def generate_java_code():
    try:
        # Ruta al archivo .clp de entrada
        clp_path = os.path.join(UPLOAD_FOLDER, 'output.clp')

        if not os.path.exists(clp_path):
            return "Archivo output.clp no encontrado", 404

        with open(clp_path, 'rb') as f:
            files = {'file': f}
            response = requests.post("http://localhost:8001/run-clips",
files=files)

        if response.status_code != 200:
            return f"Error desde el microservicio CLIPS:
{response.text}", 500

        # Retorna el código Java generado como texto plano
        return response.text

    except Exception as e:
        return str(e), 500

if __name__ == "__main__":
    # Ejecutar la app Flask localmente en modo debug
    app.run(debug=True, host="127.0.0.1", port=5000)
```

Crear una ruta adicional /convert-python para convertir el Java devuelto a Python usando convertir_java_a_python.

```
aaaaaaaa> python .\run.py
 * Serving Flask app 'run'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a p
roduction WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 117-660-525
```

Clips a Java:

```
// Java code for class Clase1
public class Clase1 extends Clase2 {
}

// Java code for class Clase2
public class Clase2 {
}
```

Descargar .java Cerrar

Java a Python:

```
```python
# Python code for class Clase1
class Clase1(Clase2):
    pass


# Python code for class Clase2
class Clase2:
    pass

```
```

Descargar .py Cerrar

El traductor usa IA avanzada para traducir código de Java a Python. En lugar de depender solo de reglas fijas, incorpora un modelo de lenguaje moderno que entiende mejor el contexto y hace la traducción de forma más flexible y precisa.

Para lograr esto, se conecta a la API de OpenRouter y utiliza un modelo llamado mistralai/mixtral-8x7b, que está especialmente entrenado para manejar varios idiomas y tipos de programación, por lo que es capaz de convertir el código entre Java y Python de manera coherente.

Cuando se hace una traducción, la aplicación envía el código Java junto con instrucciones claras para que el modelo lo convierta a Python, sin agregar explicaciones ni comentarios innecesarios. El resultado en Python se muestra en pantalla y se guarda automáticamente.

La clave para usar la API se guarda de forma segura en un archivo oculto (.env), evitando que quede expuesta en el código o en el sistema.

Para que todo funcione fácilmente en Windows, se incluye un archivo de instalación (setup.bat) que verifica que tengas Python 3.11 instalado, crea los entornos necesarios, instala las dependencias, te pide la clave de la API, y arranca los servicios que hacen posible la traducción. También abre automáticamente una ventana en tu navegador para que puedas usar la interfaz desde tu computadora.