Cambios e implementación de requerimientos

INFORME

GR3

Proyecto Transcriptor Braille - Análisis de Arquitectura e Implementación

---

1. INTRODUCCIÓN Y OBJETIVOS

El Proyecto Transcriptor Braille es una aplicación de código abierto diseñada para facilitar la conversión de texto en español hacia su representación en braille, así como la generación de señalética accesible. La arquitectura implementada sigue un modelo de microservicios con separación clara entre componentes backend y frontend, permitiendo escalabilidad y mantenibilidad del sistema.

Los objetivos principales del proyecto incluyen: la transcripción automática de texto a braille español, la traducción inversa de braille a texto, la generación de señalética en braille, y la representación visual mediante impresión en espejo para facilitar la lectura táctil.

---

2. ARQUITECTURA GENERAL DEL SISTEMA

2.1 Modelo de Arquitectura

La aplicación implementa una arquitectura de capas organizada en los siguientes niveles:

Capa de Presentación (Frontend): Aplicación React que proporciona la interfaz de usuario para interactuar con los servicios de transcripción.

Capa de Aplicación (Backend): Servidor FastAPI que expone endpoints REST para las operaciones de transcripción y traducción inversa.

Capa de Lógica de Negocio: Módulos especializados en la conversión de caracteres a braille español e impresión en espejo.

Capa de Persistencia y Configuración: Gestión de variables de entorno y configuraciones globales.

2.2 Infraestructura de Contenedores

Se ha implementado Docker Compose como orquestador de servicios, definiendo dos contenedores principales:

Backend Container: Ejecuta el servidor Uvicorn en el puerto 8000 con capacidad de recarga automática en modo desarrollo.

Frontend Container: Ejecuta la aplicación React en el puerto 3000 con watcher polling habilitado para detección de cambios en entornos Docker.

Ambos servicios se comunican a través de una red bridge denominada braille_network, asegurando aislamiento de la infraestructura y comunicación interna optimizada.

---

3. COMPONENTES DEL BACKEND

3.1 Estructura General

El backend está organizado bajo la carpeta app con la siguiente estructura:

main.py: Punto de entrada de la aplicación FastAPI que configura las rutas iniciales y middleware.

config.py: Gestión centralizada de configuraciones y variables de entorno.

logger.py: Sistema de logging estructurado para trazabilidad de operaciones.

exceptions.py: Definición de excepciones personalizadas para manejo de errores específicos del dominio.

utils.py: Funciones utilitarias compartidas entre módulos.

3.2 Módulo Core de Lógica Braille

La carpeta api/core contiene los algoritmos fundamentales:

braille_logic.py: Implementa la lógica de conversión de caracteres españoles a braille, incluyendo:

- Mapeo de caracteres alfabéticos a sus equivalentes en braille español.
- Gestión de caracteres especiales y acentuados propios del español.
- Implementación de reglas de impresión en espejo para facilitar la lectura táctil desde la perspectiva del lector vidente que asiste al usuario.
- Procesamiento de números y símbolos comunes.

3.3 Capas de Servicios

services/generator.py: Servicio encargado de la generación de braille a partir de texto plano, coordinando el flujo de procesamiento y aplicando optimizaciones.

services/translator.py: Servicio especializado en la traducción inversa de braille a texto legible, permitiendo validación y verificación de conversiones.

3.4 Rutas API

routes/generation.py: Endpoints REST para solicitudes de transcripción de texto a braille. Incluye validación de entrada y manejo de errores específicos.

routes/translation.py: Endpoints REST para operaciones de traducción inversa, permitiendo conversión de braille nuevamente a texto.

3.5 Esquemas de Datos

schemas/translation.py: Definición de modelos Pydantic para validación de payloads en peticiones y respuestas, asegurando tipado fuerte y documentación automática en Swagger.

3.6 Testing

Se implementan pruebas unitarias e integración:

tests/test_logic.py: Pruebas de la lógica central de conversión a braille español.

tests/test_generation.py: Pruebas de los servicios de generación y endpoints.

test_generator_debug.py: Herramienta de depuración para validar funcionalidad de generación.

---

4. COMPONENTES DEL FRONTEND

La aplicación frontend está construida con React y proporciona una interfaz usuario amigable para acceder a los servicios del backend.

Configuración de conectividad mediante variable de entorno REACT_APP_API_URL que apunta a http://localhost:8000 en desarrollo.

Implementación de watcher polling (WATCHPACK_POLLING=true) para garantizar la detección de cambios en entornos Docker en sistemas Windows y macOS.

Volumen anónimo para node_modules que preserva las dependencias instaladas dentro del contenedor, evitando conflictos con sincronización de archivos.

---

5. FUNCIONALIDAD DE TRANSCRIPCIÓN A BRAILLE ESPAÑOL

5.1 Especificaciones

El sistema implementa la conversión de caracteres españoles siguiendo el estándar de braille español, que incluye:

Caracteres alfabéticos: a-z con sus correspondencias en celdas braille de 6 puntos.

Caracteres acentuados: á, é, í, ó, ú, ñ y sus variantes mayúsculas.

Números: Representación numérica con prefijo específico en braille.

Signos de puntuación: Punto, coma, punto y coma, dos puntos, interrogación, exclamación y otros.

Símbolos especiales: Caracteres matemáticos, monetarios y símbolos de uso común.

5.2 Algoritmo de Conversión

El módulo braille_logic.py implementa un mapeo directo de caracteres a sus representaciones en braille mediante estructuras de datos diccionario que asocian cada carácter con su equivalente en formato braille.

La conversión procesa el texto carácter por carácter, aplicando reglas de contextualización cuando es necesario (mayúsculas, números, cambios de modo).

Se incluye validación de entrada para garantizar que los caracteres sean procesables, rechazando entrada que no corresponda a caracteres españoles válidos.

---

6. FUNCIONALIDAD DE IMPRESIÓN EN ESPEJO

6.1 Propósito y Utilidad

La impresión en espejo es una característica crítica que invierte horizontalmente la representación braille, permitiendo que una persona vidente pueda leer el patrón táctil desde la perspectiva del usuario no vidente.

Esta funcionalidad facilita la verificación visual de documentos braille por parte de personal de apoyo sin necesidad de equipamiento especializado.

6.2 Implementación Técnica

El módulo braille_logic.py incluye la lógica de espejo que:

Invierte la posición de los puntos dentro de cada celda braille de 6 puntos.

En braille estándar, los puntos se numeran del 1-3 (columna izquierda) y 4-6 (columna derecha).

La transformación en espejo mapea: punto 1 ↔ punto 4, punto 2 ↔ punto 5, punto 3 ↔ punto 6.

Invierte el orden secuencial de celdas dentro de cada línea, de modo que la lectura proceda de derecha a izquierda.

Mantiene la integridad del contenido semántico del braille mientras presenta la visualización requerida.

6.3 Aplicación en Flujos

La impresión en espejo se aplica como opción en el endpoint de generación, permitiendo que el cliente solicite explícitamente esta transformación.

El servicio generator.py coordina la invocación de la función de espejo después de la transcripción inicial, procesando el resultado antes de devolverlo al cliente.

---

7. CONFIGURACIÓN Y GESTIÓN DE DEPENDENCIAS

7.1 Backend

El archivo requirements.txt especifica todas las dependencias Python:

fastapi: Framework web moderno para construcción de APIs REST.

uvicorn: Servidor ASGI de alto rendimiento para ejecutar aplicaciones FastAPI.

pydantic: Librería de validación de datos y gestión de esquemas.

pytest: Framework de testing para pruebas unitarias e integración.

Variables de entorno gestionadas mediante .env.example con configuración de logging, debug y parámetros de conexión.

7.2 Frontend

Gestión de dependencias mediante npm con configuración en package.json (información no visible en archivos adjuntos pero inferida de estructura Docker).

7.3 Dockerfile Backend

Construye imagen Python con instalación de dependencias desde requirements.txt y copia de código fuente.

Expone puerto 8000 para comunicación HTTP.

7.3 Dockerfile Frontend

Construye imagen Node.js con instalación de dependencias npm y compilación de aplicación React.

---

8. PRUEBAS Y VALIDACIÓN

8.1 Framework de Testing

Implementación con pytest permitiendo:

Pruebas unitarias de funciones de conversión.

Pruebas de integración de endpoints API.

Fixtures compartidas mediante conftest.py.

8.2 Requerimientos de Calidad

El proyecto define requerimientos explícitos documentados en la carpeta Docs/documentacion/requerimientos:

req_01_transcripcion.py: Especificaciones de transcripción a braille español.

req_02_traduccion_inversa.py: Especificaciones de traducción braille a texto.

req_03_generacion_señaletica.py: Generación de señalética accesible.

req_04_docstrings.py: Requisitos de documentación de código.

req_05_casos_prueba.py: Casos de prueba y cobertura.

req_06_diseño_arquitectonico.py: Requisitos arquitectónicos.

req_07_documentacion_ambiente.py: Documentación de configuración de ambiente.

8.3 Reportes

Se genera report.html en la carpeta Pruebas como resultado de ejecución de suite de testing, proporcionando métricas de cobertura y resultados detallados.

---

9. DOCUMENTACIÓN Y DOCSTRINGS

9.1 Estándares Implementados

Implementación exhaustiva de docstrings siguiendo estándar Python con descripción de funciones, parámetros, retornos y ejemplos.

Archivo DOCSTRINGS_IMPLEMENTADOS.py documenta todos los docstrings generados en el backend.

9.2 Documentación Arquitectónica

Archivo arquitectura.md en carpeta Docs proporciona diagrama de componentes y flujos principales del sistema.

---

10. INTEGRACIÓN CONTINUA

10.1 Configuración CI/CD

Archivo ci.yml define pipeline de integración continua con:

Ejecución automática de tests en cada push.

Validación de código.

Generación de reportes de cobertura.

---

11. CAMBIOS Y MEJORAS IMPLEMENTADAS

11.1 Orquestación con Docker Compose

Implementación de docker-compose.yml que define:

Servicio backend con hot reload automático para desarrollo iterativo.

Servicio frontend con watcher polling para detección de cambios.

Red aislada braille_network para comunicación entre servicios.

Dependencias de servicio para garantizar orden de inicialización.

Volúmenes estratégicos que permiten desarrollo local con contenedores.

11.2 Separación de Responsabilidades

Arquitectura en capas con módulos especializados:

Lógica core (braille_logic.py) independiente de detalles de transportación.

Servicios que coordinan flujos complejos.

Rutas que manejan solo aspectos HTTP.

11.3 Validación de Entrada

Implementación de esquemas Pydantic para validación automática de datos en endpoints.

Excepciones personalizadas para manejo específico de errores del dominio braille.

11.4 Testing Exhaustivo

Cobertura de testing en lógica core, servicios y endpoints.

Herramientas de depuración para validación de funcionalidad.

Reportes HTML de resultados de testing.

---

12. CONSIDERACIONES TÉCNICAS IMPORTANTES

12.1 Compatibilidad Multiplataforma

Configuración de WATCHPACK_POLLING en frontend para garantizar funcionamiento en Windows, macOS y Linux dentro de contenedores Docker.

12.2 Desarrollo vs Producción

Modo desarrollo con hot reload en backend y watcher en frontend.

Comando de ejecución configurado específicamente para desarrollo con --reload en Uvicorn.

12.3 Gestión de Dependencias en Contenedores

Volumen anónimo /app/node_modules en frontend para preservar instalación de paquetes npm.

Evita conflictos entre bind mounts y gestión de dependencias dentro del contenedor.

---

13. CONCLUSIONES

El Proyecto Transcriptor Braille implementa una solución completa y bien arquitecturada para transcripción de texto español a braille, con énfasis particular en:

Precisión de conversión a braille español conforme a estándares internacionales.

Funcionalidad de impresión en espejo que facilita verificación y accesibilidad.

Arquitectura escalable con separación clara de responsabilidades.

Infraestructura containerizada que simplifica desarrollo y despliegue.

Cobertura exhaustiva de testing y documentación.

La implementación actual proporciona una base sólida para futura expansión, permitiendo agregar características adicionales de accesibilidad y localización a otros idiomas manteniendo la integridad arquitectónica del sistema.