UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERIA

ESCUELA DE CIENCIAS Y SISTEMAS

MANEJO E IMPLEMENTACIÓN DE ARCHIVOS

PRIMER SEMESTRE 2025

SECCION A

MANUAL TECNICO

HAROLD BENJAMIN OXLAJ MANGANDI

202100543

INDICE

- ESPECIFICACIONES:
  - Frontend: Vue.js
  - Backend: Go
  - Es necesario Tener instalado Graphviz para generar los reportes
- Dependencias Frontend:
  - "@codemirror/lang-javascript": "^6.2.2",
  - "@codemirror/state": "^6.4.1",
  - "@codemirror/theme-one-dark": "^6.1.2",
  - "@codemirror/view": "^6.33.0",
  - "axios": "^1.7.7",
  - "codemirror": "^6.0.1",
  - "core-js": "^3.8.3",
  - "vue": "^3.2.13",
  - "vue-codemirror": "^6.1.1",
  - "vue-router": "^4.4.5"

ESTRUCTURA DE LA APLICACIÓN:

Esta aplicación simula un sistema de archivos EXT2 por medio de archivos binarios que servirán como los discos donde se podrán crear particiones tanto primarias como extendidas y lógicas, también se podrá formatear las particiones primarias para hacer en ellas el sistema de archivos EXT2, teniendo usuarios, grupos, la posibilidad de crear nuevos directorios y archivos en el sistema, también se contará con la creación de varios reportes para visualizar mejor el sistema de archivos.

Todas las funcionalidades se harán por medio de comandos que se detallan mas adelante

BACKEND:

Archivo Main.go

- **Endpoints:**
  - **POST /scannear**:

    Valida que el JSON sea correcto. Si es válido, pasa los comandos a Scanner.Scan().

- **Puerto**: :7777.

Funcionamiento: Inicia un servidor web usando el framework Gin y se define un endpoint /scannear para procesar solicitudes POST con datos de entrada.

CARPETA: Utilities

Archivo: utilities.go:

En este archivo se implementan herramientas generales para la lectura u escritura de los archivos binarios y demás funcionalidades usadas frecuentemente en el proyecto.

Función: CreateFile

Parámetros

- name (string): Ruta del archivo a crear.

Funcionamiento: Crea un archivo y sus directorios padres si no existen, retornando errores del sistema en caso de fallo. Usa os.MkdirAll para crear la estructura de directorios necesaria (con permisos 0777).

Función OpenFile

Parámetros

- file *os.File: Archivo binario abierto donde se escribirá.
- data interface{}: Objeto a serializar (debe ser compatible con binary.Write).
- position int64: Posición exacta (en bytes) donde se escribirá.

Funcionamiento: Escribe un objeto en formato binario (Little-Endian) en una posición exacta del archivo, manejando errores de posicionamiento o serialización

Función: ReadObject

Parámetros:

- file: Archivo binario abierto (debe permitir lectura)
- data: Puntero a variable donde almacenar los datos leídos
- position: Posición en bytes desde donde leer (offset desde inicio)

Funcionamiento: Lee datos binarios (Little-Endian) desde una posición exacta en el archivo. Primero posiciona el puntero en el offset especificado (Seek), luego deserializa los datos directamente en la variable proporcionada. Retorna error si falla el posicionamiento o la lectura (posición inválida o tipo de dato incompatible).

Función: ReadFromFile

Parámetros:

- file: Archivo binario abierto en modo lectura
- offset: Posición exacta en bytes donde comenzar a leer

- data: Puntero a la variable donde almacenar los datos (debe coincidir con el formato binario)

Funcionamiento: Lee datos en formato binario (Little-Endian) desde una posición específica del archivo, manejando errores con mensajes descriptivos. Primero posiciona el puntero en el offset indicado, luego deserializa los datos directamente en la variable proporcionada.

Función: CreateParentDirs

Parámetros:

- path: Ruta completa del archivo/directorio

Funcionamiento: Crea recursivamente todos los directorios padres necesarios para una ruta específica, usando permisos estándar (0777). No afecta directorios existentes.

CARPETA: Scanner

Archivo: scanner.go:

Función: Scan

Parámetros:

- input: Texto de entrada con múltiples líneas (incluye comandos y comentarios)

Funcionalidad: Procesa un texto multilínea identificando comandos (líneas sin #) y comentarios (líneas con # al inicio). Para cada comando válido Separa el comando y parámetros (getCommandAndParams) y luego Ejecuta análisis (AnalyzeCommand)

Función: getCommandAndParams

Parámetros:

- input: Cadena de texto con el comando y parámetros

Funcionalidad: Extrae y normaliza el comando (primera palabra en minúsculas) y parámetros (resto del texto) de una cadena de entrada, ignorando espacios vacíos.

Función: AnalyzeCommand

Parámetros:

- command: Nombre del comando (case-insensitive)
- params: Argumentos del comando como string

Funcionalidad: Ejecuta acciones específicas según el comando recibido, delegando a módulos especializados (discos, sistema de archivos, usuarios, etc.).

Subcarpeta: CommandsDisk

Archivo: mkdisk.go:

Función: MkDisk

Parámetros:

- params: Argumentos del comando como string

Funcionalidad: Analiza los parámetros ingresados por el usuario para crear un disco virtual con parámetros configurables (size, fit, unit, path) y realiza validaciones básicas.

Llama a DiskManager.Mkdisk con los parámetros validados.

Archivo: fdisk.go:

Función: FDisk

Parámetros:

- input: Argumentos del comando como string

Funcionalidad: Analiza los parámetros ingresados por el usuario para crear una partición en un disco virtual con parámetros configurables (size, fit, unit, path, name, type) y realiza validaciones básicas.

Llama a DiskManager.Fdisk con los parámetros validados.

Archivo: mount.go:

Función: Mount

Parámetros

- params: Cadena de texto con los argumentos del comando.

Funcionalidad: Extrae los argumentos mediante expresiones regulares. Si las validaciones son exitosas, llama a DiskManager.Mount con los parámetros procesados.

Archivo: mounted.go:

Función: Mounted

Parámetros

- params: Cadena de texto que debe estar vacía (no acepta parámetros).

Funcionalidad: Verifica que no se hayan proporcionado parámetros (strings.TrimSpace(params) == ""). Si se detectan parámetros, devuelve un error

Llama a DiskManager.PrintMountedPartitions() para mostrar las particiones montadas.

Archivo: rmdisk.go:

Función: RmDisk

Parámetros

- **input**: Cadena de texto que debe contener el parámetro -path con la ruta del disco a eliminar.

Funcionalidad: Analiza la estructura del path y si es correcta: Llama a DiskManager.RmDisk(path) para eliminar el disco.

Subcarpeta: FileSystem

Archivo: cat.go:

Función: Cat

Parámetros

- params: Cadena de texto con los archivos a mostrar

Funcionalidad: Extrae valores usando regex y valida que los flags existan (muestra error si no), Llama a FileSystem.Cat(files) para mostrar el contenido.

Archivo: mkfs.go:

Función: Mkfs

Parámetros

- **params**: Cadena de texto con los parámetros del comando

Funcionalidad: Extrae valores usando regex y valida que los flags existan (muestra error si no). Llama a FileSystem.Mkfs(id, type, fs) con los parámetros normalizados.

Subcarpeta: Users

Archivo: chgrp.go:

Función: Chgrp

Parámetros

- **params**: Cadena de texto con los parámetros del comando (-user, -grp)

Funcionalidad: Extrae valores usando regex y valida que los flags existan (muestra error si no). Llama a UsersManager.Chgrp(user, group) con los parámetros validados.

Archivo: login.go:

Función: Login

Parámetros

- **params**: Cadena de texto con credenciales (-user, -pass, -id)

Funcionalidad: Extrae valores usando regex y valida que los flags existan (muestra error si no). Llama a UsersManager.Login(user, pass, id) con los parámetros validados.

Archivo: logout.go:

Función: LogOut

Parámetros

- **params**: No acepta parámetros (debe ser string vacío).

Funcionalidad: Llama a UsersManager.Logout() para cerrar sesión.

Archivo: mkgrp.go:

Función: Mkgrp

Parámetros

- **params**: Cadena de texto que debe contener el parámetro -name con el nombre del grupo a crear.

Funcionalidad: Si las validaciones son exitosas, llama a UsersManager.Mkgrp(name) para crear el grupo.

Archivo: mkusr.go:

Función: Mkusr

Parámetros

- **params**: Cadena de texto que debe contener los parámetros -name -grp -pass para crear el usuario

Funcionalidad: Si las validaciones son exitosas. Llama a UsersManager.Mkusr(user, pass, group) si todo es correcto.

Función: Rmgrp

Parámetros

- **params**: Cadena de texto que debe contener el parámetro -name para eliminar el grupo

Funcionalidad: Si las validaciones son exitosas. Llama a UsersManager.Rmgrp(name) si todo es correcto.

Función: Rmusr

Parámetros

- **params**: Cadena de texto que debe contener el parámetro -name para eliminar el usuario

Funcionalidad: Si las validaciones son exitosas. Llama a UsersManager.Rmusr(name) si todo es correcto.

CARPETA: Structs

Subcarpeta: disk

Archivo: mbr.go:

Estructura: MBR (Master Boot Record)

La estructura MBR (Master Boot Record) representa el registro de arranque maestro de un disco, que contiene información crítica sobre su tamaño, metadatos y particiones.

Descripción de los campos

1. **Size (int64)**
   - Descripción: Representa el tamaño total del disco en bytes.
   - Tipo: int64 (entero de 64 bits), lo que permite almacenar tamaños de disco grandes (hasta 2^{63}-1 bytes).
2. **CreationDate ([16]byte)**
   - Descripción: Almacena la fecha y hora de creación del disco en formato binario.
   - Tipo: Arreglo fijo de 16 bytes ([16]byte), que puede contener una cadena codificada (por ejemplo, "YYYY-MM-DD HH:MM").
3. **Signature (int32)**
   - Descripción: Un número entero aleatorio que actúa como identificador único del MBR.
   - Tipo: int32 (entero de 32 bits), con un rango de -2^{31} a 2^{31}-1.
4. **Fit (byte)**
   - Descripción: Indica el tipo de ajuste utilizado para la gestión de particiones en el disco.

- Tipo: byte (8 bits), que puede representar un carácter ASCII como 'B' (Best Fit), 'F' (First Fit) o 'W' (Worst Fit).

5. **Partitions ([4]Partition)**

   - Descripción: Un arreglo fijo de 4 particiones, donde cada elemento es una estructura de tipo Partition.
   - Tipo: [4]Partition, un arreglo de tamaño estático que refleja el límite tradicional del MBR (máximo 4 particiones primarias).

Archivo: partition.go:

Estructura: Partition

La estructura Partition representa una partición individual dentro del esquema del Master Boot Record (MBR). Define las propiedades y metadatos de una partición en un disco, como su estado, tipo, ubicación y nombre.

Descripción de los campos

1. **Status (byte)**

   - **Descripción**: Indica si la partición está montada o no.
   - **Tipo**: byte (8 bits), con valores '0' (no montada) o '1' (montada).

2. **Type (byte)**

   - **Descripción**: Define el tipo de partición.
   - **Tipo**: byte, con valores posibles: 'P' (Primaria), 'E' (Extendida) o 'L' (Lógica).

3. **Fit (byte)**

   - **Descripción**: Indica el algoritmo de ajuste utilizado para asignar espacio a la partición.
   - **Tipo**: byte, con valores 'B' (Best Fit), 'F' (First Fit) o 'W' (Worst Fit).

4. **Start (int32)**

   - **Descripción**: Especifica el byte inicial de la partición en el disco.
   - **Tipo**: int32 (entero de 32 bits), con un rango de 0 a 2^{31}-1.

5. **Size (int32)**
    - **Descripción**: Representa el tamaño total de la partición en bytes.
    - **Tipo**: int32, con un límite de 2^{31}-1 bytes (aproximadamente 2 GB).
6. **Name ([16]byte)**
    - **Descripción**: Almacena el nombre asignado a la partición.
    - **Tipo**: Arreglo fijo de 16 bytes ([16]byte), que puede contener una cadena ASCII (por ejemplo, "Particion1").
7. **Correlative (int32)**
    - **Descripción**: Un contador que indica el orden de montaje o un valor especial.
    - **Tipo**: int32, con valor por defecto -1 (sin montar), incrementándose al montar la partición.
8. **Id ([4]byte)**
    - **Descripción**: Identificador único de la partición cuando está montada.
    - **Tipo**: Arreglo fijo de 4 bytes ([4]byte), que puede contener una cadena corta (por ejemplo, "431A").

Archivo: ebr.go:

Estructura: EBR(Extended Boot Record)

La estructura EBR (Extended Boot Record) se utiliza para describir particiones lógicas dentro de una partición extendida en el esquema del Master Boot Record (MBR). Cada EBR contiene información sobre una partición lógica y un enlace al siguiente EBR, formando una lista enlazada.

Descripción de los campos

1. **PartMount (byte)**
    - **Descripción**: Indica si la partición lógica está montada en el sistema.
    - **Tipo**: byte (8 bits), con valores '0' (no montada) o '1' (montada).
2. **PartFit (byte)**

- **Descripción**: Define el algoritmo de ajuste usado para asignar espacio a la partición lógica.
- **Tipo**: byte, con valores 'B' (Best Fit), 'F' (First Fit) o 'W' (Worst Fit).

3. **PartStart (int32)**
   - **Descripción**: Especifica el byte inicial de la partición lógica en el disco.
   - **Tipo**: int32 (entero de 32 bits), con un rango de 0 a 2^{31}-1.
4. **PartSize (int32)**
   - **Descripción**: Representa el tamaño total de la partición lógica en bytes.
   - **Tipo**: int32, con un límite de 2^{31}-1 bytes (aproximadamente 2 GB).
5. **PartNext (int32)**
   - **Descripción**: Apunta al byte donde se encuentra el siguiente EBR en la lista enlazada.
   - **Tipo**: int32, con valor -1 si no hay un siguiente EBR (es decir, es la última partición lógica).
6. **PartName ([16]byte)**
   - **Descripción**: Almacena el nombre asignado a la partición lógica.
   - **Tipo**: Arreglo fijo de 16 bytes ([16]byte), que puede contener una cadena ASCII (por ejemplo, "Logica1").

Archivo: mountedpartition.go:

Estructura MountedPartition

La estructura MountedPartition representa una partición montada en el sistema, almacenando información esencial para su identificación y manejo en tiempo de ejecución.

Descripción de los campos

1. **Path (string)**
   - **Descripción**: Almacena la ruta absoluta del archivo que representa el disco donde reside la partición.
   - **Tipo**: string, una cadena de longitud variable

2. **Name (string)**

   - **Descripción**: Contiene el nombre asignado a la partición montada.
   - **Tipo**: string, permitiendo nombres de longitud variable (por ejemplo, "Particion1").

3. **ID (string)**

   - **Descripción**: Un identificador único asignado a la partición montada.
   - **Tipo**: string, que puede ser una cadena generada (por ejemplo, "431A").

4. **Status (byte)**

   - **Descripción**: Indica el estado actual de la partición en el sistema.
   - **Tipo**: byte (8 bits), con valores 0 (no montada) o 1 (montada).

5. **Start (int32)**

   - **Descripción**: Especifica la posición de inicio de la partición en bytes, relativa al inicio del archivo del disco.
   - **Tipo**: int32 (entero de 32 bits), con un rango de 0 a 2^{31}-1.

Subcarpeta: ext2

Archivo: inode.go:

Estructura Inode

La estructura Inode representa un inodo, una estructura de datos fundamental en muchos sistemas de archivos que almacena metadatos sobre un archivo o directorio, como su propietario, tamaño, fechas y ubicación de los datos en el disco.

Descripción de los campos

1. **I_uid (int32)**

   - **Descripción**: Identificador único del usuario propietario del archivo o directorio.
   - **Tipo**: int32 (entero de 32 bits), con un rango de 0 a 2^{31}-1.

2. **I_gid (int32)**

   - **Descripción**: Identificador único del grupo propietario del archivo o directorio.
   - **Tipo**: int32, similar a I_uid.

3. **I_size (int32)**

   - **Descripción**: Tamaño del archivo en bytes.
   - **Tipo**: int32, con un límite de 2^{31}-1 bytes (aproximadamente 2 GB).

4. **I_atime ([16]byte)**

   - **Descripción**: Fecha y hora del último acceso al archivo o directorio.
   - **Tipo**: Arreglo fijo de 16 bytes ([16]byte), que puede contener una cadena como "YYYY-MM-DD HH:MM".

5. **I_ctime ([16]byte)**

   - **Descripción**: Fecha y hora de creación del inodo.
   - **Tipo**: Arreglo fijo de 16 bytes, similar a I_atime.

6. **I_mtime ([16]byte)**

   - **Descripción**: Fecha y hora de la última modificación del contenido del archivo o directorio.
   - **Tipo**: Arreglo fijo de 16 bytes, como los campos anteriores.

7. **I_block ([15]int32)**

   - **Descripción**: Arreglo de punteros a los bloques de datos que contienen el contenido del archivo o directorio.
   - **Tipo**: Arreglo fijo de 15 enteros de 32 bits ([15]int32), cada uno representando una dirección de bloque en el disco.

8. **I_type ([1]byte)**

   - **Descripción**: Tipo del inodo, indicando si es un archivo o directorio.
   - **Tipo**: Arreglo fijo de 1 byte ([1]byte), con valores como '0' (archivo) o '1' (directorio).

9. **I_perm ([3]byte)**

   - **Descripción**: Permisos del inodo en formato octal (por ejemplo, "755").
   - **Tipo**: Arreglo fijo de 3 bytes ([3]byte), representando permisos para propietario, grupo y otros (lectura, escritura, ejecución).

Archivo: superblock.go:

Estructura Superblock

La estructura Superblock representa el superbloque de un sistema de archivos, una estructura crítica que almacena metadatos globales sobre la partición, como el número de inodos y bloques, su disponibilidad y las ubicaciones de las estructuras de datos principales.

Descripción de los campos

1. **S_filesystem_type (int32)**
   - **Descripción**: Identificador numérico del tipo de sistema de archivos.
   - **Tipo**: int32 (entero de 32 bits).
2. **S_inodes_count (int32)**
   - **Descripción**: Número total de inodos en el sistema de archivos.
   - **Tipo**: int32.
3. **S_blocks_count (int32)**
   - **Descripción**: Número total de bloques en el sistema de archivos.
   - **Tipo**: int32.
4. **S_free_blocks_count (int32)**
   - **Descripción**: Número de bloques libres disponibles.
   - **Tipo**: int32.
5. **S_free_inodes_count (int32)**
   - **Descripción**: Número de inodos libres disponibles.
   - **Tipo**: int32.
6. **S_mtime ([16]byte)**
   - **Descripción**: Fecha y hora del último montaje del sistema de archivos.
   - **Tipo**: Arreglo fijo de 16 bytes ([16]byte), como "YYYY-MM-DD HH:MM".
7. **S_umtime ([16]byte)**

- **Descripción**: Fecha y hora del último desmontaje del sistema de archivos.
- **Tipo**: Arreglo fijo de 16 bytes.

8. **S_mnt_count (int32)**

   - **Descripción**: Contador de cuántas veces se ha montado el sistema de archivos.
   - **Tipo**: int32.

9. **S_magic (int32)**

   - **Descripción**: Valor mágico que identifica al sistema de archivos.
   - **Tipo**: int32, fijado en 0xEF53 (un estándar en sistemas como ext2).

10. **S_inode_size (int32)**

    - **Descripción**: Tamaño en bytes de cada inodo.
    - **Tipo**: int32.

11. **S_block_size (int32)**

    - **Descripción**: Tamaño en bytes de cada bloque de datos.
    - **Tipo**: int32.

12. **S_fist_ino (int32)**

    - **Descripción**: Dirección del primer inodo libre en la tabla de inodos.
    - **Tipo**: int32.

13. **S_first_blo (int32)**

    - **Descripción**: Dirección del primer bloque libre en la tabla de bloques.
    - **Tipo**: int32.

14. **S_bm_inode_start (int32)**

    - **Descripción**: Byte de inicio del bitmap de inodos.
    - **Tipo**: int32.

15. **S_bm_block_start (int32)**

    - **Descripción**: Byte de inicio del bitmap de bloques.
    - **Tipo**: int32.

16. **S_inode_start (int32)**

    - **Descripción**: Byte de inicio de la tabla de inodos.
    - **Tipo**: int32.

17. **S_block_start (int32)**

    - **Descripción**: Byte de inicio de la tabla de bloques.
    - **Tipo**: int32.

Archivo: fileblock.go:

Estructura Fileblock

La estructura Fileblock representa un bloque de datos utilizado para almacenar el contenido de un archivo en un sistema de archivos. Es una unidad básica de almacenamiento que se referencia desde los punteros de un inodo (como I_block en la estructura Inode).

Descripción de los campos

1. **B_content ([64]byte)**

    - **Descripción**: Almacena el contenido real de un archivo en este bloque.
    - **Tipo**: Arreglo fijo de 64 bytes ([64]byte), que puede contener datos binarios o texto.

Archivo: content.go:

Estructura Content

La estructura Content representa una entrada dentro de un bloque de directorio, utilizada para almacenar información sobre un archivo o subdirectorio, como su nombre y la referencia al inodo correspondiente.

Descripción de los campos

1. **B_name ([12]byte)**

    - **Descripción**: Almacena el nombre del archivo o directorio.
    - **Tipo**: Arreglo fijo de 12 bytes ([12]byte), que puede contener una cadena ASCII de hasta 12 caracteres.

2. **B_inodo (int32)**

    - **Descripción**: Puntero al inodo que describe los metadatos y datos del archivo o directorio.
    - **Tipo**: int32 (entero de 32 bits), con un rango de 0 a 2^{31}-1, o -1 si no hay inodo asociado.

Archivo: folderblock.go:

Estructura Folderblock

La estructura Folderblock representa un bloque de datos utilizado para almacenar las entradas de un directorio en un sistema de archivos. Cada instancia contiene un arreglo fijo de entradas Content, que describen archivos o subdirectorios dentro del directorio.

Descripción de los campos

1. **B_content ([4]Content)**

    - **Descripción**: Un arreglo fijo de 4 entradas de tipo Content, donde cada entrada almacena el nombre y el puntero al inodo de un archivo o subdirectorio.
    - **Tipo**: Arreglo de 4 elementos ([4]Content), con cada Content ocupando 16 bytes (12 para B_name y 4 para B_inodo).

FRONTEND:

Archivo: Index.js:

Este archivo configura un sistema de enrutamiento en una aplicación Vue.js utilizando vue-router.

1. **Importación de módulos**: Se importan createRouter y createWebHistory de vue-router, así el componente MenuPrincipal que se utilizará como vista.

Las rutas definidas son:

   - /: Carga el componente MenuPrincipal.

2. **Creación del enrutador**:

   - Se crea una instancia del enrutador utilizando createRouter, configurando el historial con createWebHistory() y pasando el arreglo de rutas.

Archivo: Main.js:

Este archivo configura y monta una aplicación Vue.js.

1. **Importación de módulos**:

   - Se importa createApp de Vue para crear una nueva instancia de la aplicación.
   - Se importa el componente raíz App desde App.vue.
   - Se importa el enrutador configurado previamente desde ./router.

2. **Creación de la aplicación**:

   - Se crea una instancia de la aplicación Vue utilizando createApp(App), lo que permite definir el componente raíz de la aplicación.

3. **Uso del enrutador**:

   - Se utiliza el método app.use(router) para registrar el enrutador en la aplicación. Esto permite que la aplicación maneje las rutas definidas en el enrutador y renderice los componentes correspondientes según la URL.

4. **Montaje de la aplicación**:

    - Se monta la aplicación en el elemento del DOM con el ID #app usando app.mount('#app'). Esto renderiza el componente raíz en la página, iniciando la aplicación.

Archivo: App.vue:

Este archivo establece la estructura básica de la aplicación Vue.js, específicamente en la sección de la plantilla.

- La plantilla contiene un solo elemento: <router-view/>. Este componente especial de vue-router actúa como un marcador de posición donde se renderizarán los componentes correspondientes a las rutas definidas en el enrutador.
- Cuando la URL de la aplicación cambia, router-view se actualizará automáticamente para mostrar el componente que coincide con la ruta actual.

Subcarpeta: components

Archivo: MenuPrincipal.vue:

La página principal donde inicia la aplicación, contiene objetos como botones para ejecutar, abrir un archivo, y limpiar la consola, así como dos editores de texto de Codemirror para simular el código

- **data**:
    - **codigoEntrada**: Contiene el código ingresado por el usuario.
    - **salidaCodigo**: Guarda el resultado devuelto por el backend después de ejecutar el código.
    - **nombreArchivo**: Guarda el nombre del archivo actualmente abierto o creado.
    - **extensions**: Define la configuración del editor para la entrada de código, incluyendo temas y estilos.
    - **outputExtensions**: Configura el editor para mostrar la salida de manera solo lectura, con un estilo diferente.

Métodos:

- **ejecutar**:

- Realiza una llamada al backend para ejecutar el código ingresado por medio de axios con una solicitud GET a la ruta: http://localhost:7777/Scannear y guarda la salida en salidaCodigo.

- **abrirArchivo**:
  - Permite al usuario cargar un archivo de código, lee su contenido y lo coloca en codigoEntrada.