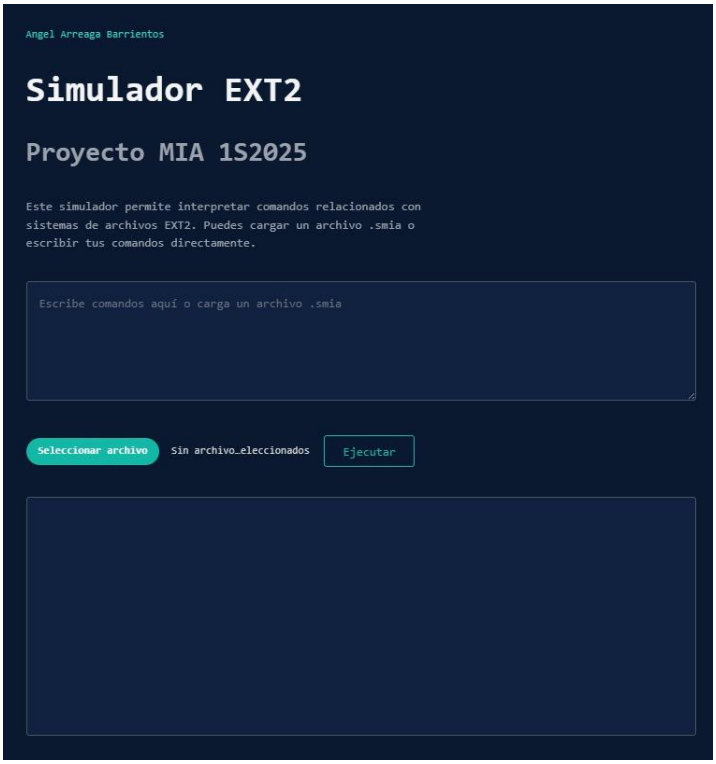
Manual Técnico

1. Arquitectura del Sistema

Frontend:

El frontend de la aplicación está desarrollado con un framework moderno como React, Angular o Vue. Su principal objetivo es proporcionar una interfaz amigable para que los usuarios puedan ingresar comandos directamente o cargarlos desde un archivo de script.



Backend:

El backend está desarrollado en Go (Golang) y se encarga de simular un sistema de archivos EXT2 sobre archivos binarios `.mia`. Este expone APIs RESTful que procesan los comandos y devuelven resultados al frontend. El backend se comunica con estructuras internas que simulan discos, particiones, inodos, bloques, etc.

2. Estructuras de Datos Implementadas

MBR (Master Boot Record): Contiene información general del disco y hasta 4 particiones primarias o extendidas. Se guarda en el primer sector del disco.

```go
type MBR struct {
    Mbr_size           int32        // Tamaño del MBR en bytes
    Mbr_creation_date  float32      // Fecha y hora de creación del MBR
    Mbr_disk_signature int32        // Firma del disco
    Mbr_disk_fit       [1]byte      // Tipo de ajuste
    Mbr_partitions     [4]PARTITION // Particiones del MBR
}

// SerializeMBR escribe la estructura MBR al inicio de un archivo binario
func (mbr *MBR) SerializeMBR(path string) error {
    file, err := os.OpenFile(path, os.O_WRONLY|os.O_CREATE, 0644)
    if err != nil {
        return err
    }
    defer file.Close()

    // Serializar la estructura MBR directamente en el archivo
    err = binary.Write(file, binary.LittleEndian, mbr)
    if err != nil {
        return err
    }

    return nil
}
```

Partition: Representa una división lógica del disco. Puede ser primaria o extendida. Las extendidas pueden contener particiones lógicas (EBR).

```go
type PARTITION struct {
    Part_status      [1]byte  // Estado de la partición
    Part_type        [1]byte  // Tipo de partición
    Part_fit         [1]byte  // Ajuste de la partición
    Part_start       int32    // Byte de inicio de la partición
    Part_size        int32    // Tamaño de la partición
    Part_name        [16]byte // Nombre de la partición
    Part_correlative int32    // Correlativo de la partición
    Part_id          [4]byte  // ID de la partición

}
```

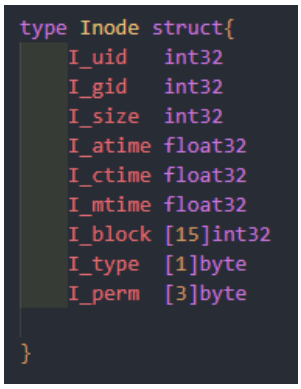
EBR (Extended Boot Record): Estructura enlazada que permite simular múltiples particiones lógicas dentro de una extendida.

```go
type EBR struct {
    Part_mount [1]byte
    Part_fit   [1]byte
    Part_start int32
    Part_s     int32
    Part_next  int32
    Part_name  [16]byte

}

// Serialize guarda el EBR en el archivo en la posición indicada
func (ebr *EBR) Serialize(path string, offset int64) error {
    file, err := os.OpenFile(path, os.O_WRONLY|os.O_CREATE, 0644)
    if err != nil {
        return err
    }
    defer file.Close()

    _, err = file.Seek(offset, 0)
    if err != nil {
        return err
    }

    return binary.Write(file, binary.LittleEndian, ebr)
}
```

SuperBloque: Contiene metadatos del sistema de archivos como número de inodos, bloques, y sus ubicaciones.

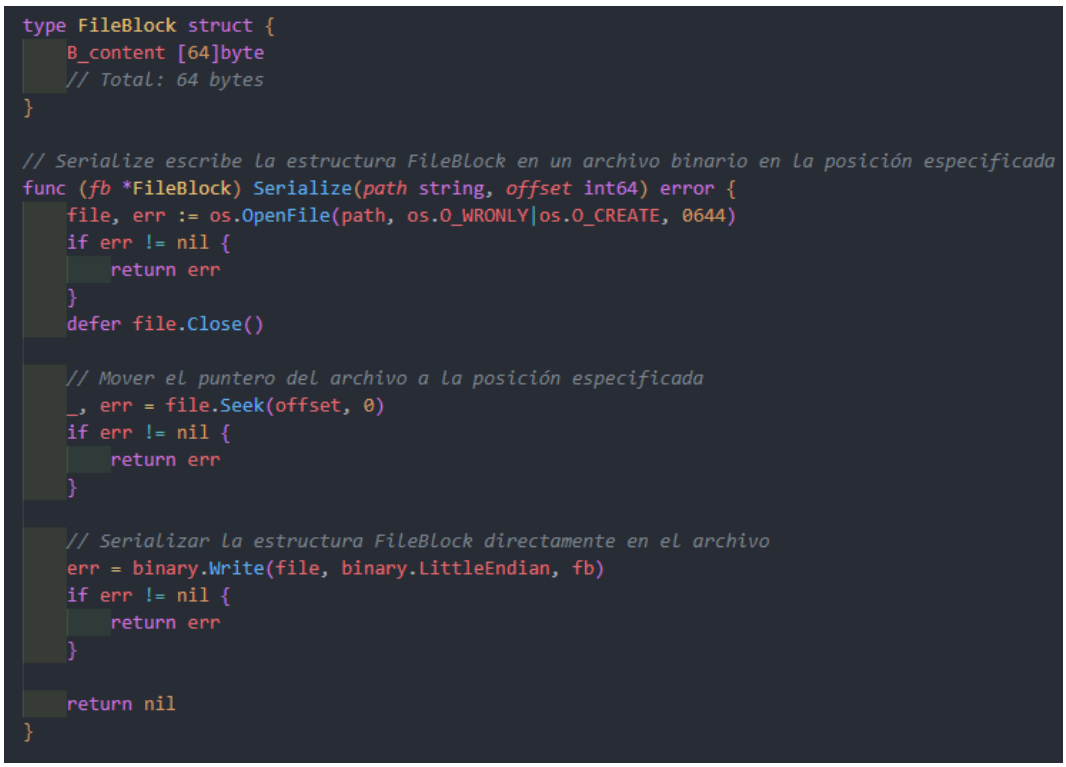
```go
type SuperBlock struct {
    S_filesystem_type   int32
    S_inodes_count      int32
    S_blocks_count      int32
    S_free_inodes_count int32
    S_free_blocks_count int32
    S_mtime             float32
    S_umtime            float32
    S_mnt_count         int32
    S_magic             int32
    S_inode_size        int32
    S_block_size        int32
    S_first_ino         int32
    S_first_blo         int32
    S_bm_inode_start    int32
    S_bm_block_start    int32
    S_inode_start       int32
    S_block_start       int32

}
```

Inodo: Representa cada archivo o carpeta. Contiene UID, GID, tamaño, tiempos de acceso/modificación, tipo, permisos y apuntadores a bloques.



```go
type Inode struct{
    I_uid   int32
    I_gid   int32
    I_size  int32
    I_atime float32
    I_ctime float32
    I_mtime float32
    I_block [15]int32
    I_type  [1]byte
    I_perm  [3]byte

}
```

Bloques: Unidades mínimas de almacenamiento de 64 bytes. Pueden ser bloques de carpetas, archivos o de apuntadores.



```go
type FileBlock struct {
    B_content [64]byte
    // Total: 64 bytes
}

// Serialize escribe la estructura FileBlock en un archivo binario en la posición especificada
func (fb *FileBlock) Serialize(path string, offset int64) error {
    file, err := os.OpenFile(path, os.O_WRONLY|os.O_CREATE, 0644)
    if err != nil {
        return err
    }
    defer file.Close()

    // Mover el puntero del archivo a la posición especificada
    _, err = file.Seek(offset, 0)
    if err != nil {
        return err
    }

    // Serializar la estructura FileBlock directamente en el archivo
    err = binary.Write(file, binary.LittleEndian, fb)
    if err != nil {
        return err
    }

    return nil
}
```

Bitmap: Estructura de bits para indicar disponibilidad de inodos y bloques (0 libre, 1 ocupado).

```go
// CreateBitMaps crea los Bitmaps de inodos y bloques en el archivo especificado
func (sb *SuperBlock) CreateBitMaps(path string) error {
    // Escribir Bitmaps
    file, err := os.OpenFile(path, os.O_WRONLY|os.O_CREATE, 0644)
    if err != nil {
        return err
    }
    defer file.Close()

    // Bitmap de inodos
    // Mover el puntero del archivo a la posición especificada
    _, err = file.Seek(int64(sb.S_bm_inode_start), 0)
    if err != nil {
        return err
    }

    // Crear un buffer de n '0'
    buffer := make([]byte, sb.S_free_inodes_count)
    for i := range buffer {
        buffer[i] = '0'
    }

    // Escribir el buffer en el archivo
    err = binary.Write(file, binary.LittleEndian, buffer)
    if err != nil {
        return err
    }

    // Bitmap de bloques
    // Mover el puntero del archivo a la posición especificada
    _, err = file.Seek(int64(sb.S_bm_block_start), 0)
    if err != nil {
        return err
    }

    // Crear un buffer de n 'O'
    buffer = make([]byte, sb.S_free_blocks_count)
    for i := range buffer {
        buffer[i] = 'O'
    }
```

3. Descripción de Comandos Implementados

Administración de Discos:

- MKDISK: Crea un archivo binario `.mia` que simula un disco físico con tamaño fijo inicializado con ceros.

```go
// MKDISK estructura que representa el comando mkdisk con sus parámetros
type MKDISK struct{
    size int
    unit string
    fit  string
    path string

}

func ParseMkdisk(tokens []string) (*MKDISK, error){
    cmd := &MKDISK{}
    args := strings.Join(tokens, " ")
    re := regexp.MustCompile(`-size=\d+|-unit=[kKmM]|-fit=[bBfFwW]{2}|-path="[^"]+"|-path=[^\s]+`)
    matches := re.FindAllString(args, -1)

    for _, match := range matches{
        kv := strings.SplitN(match, "=", 2)
        if len(kv) != 2{
            return nil, fmt.Errorf("formato de parámetro inválido: %s", match)

        }

        key, value := strings.ToLower(kv[0]), kv[1]
        if strings.HasPrefix(value, "\"") && strings.HasSuffix(value, "\""){
            value = strings.Trim(value, "\"")

        }
```

- RMDISK: Elimina un disco simulado (.mia) dado su path.

```go
type RMDISK struct {
    path string
}

// ParseRmdisk analiza el comando rmdisk y ejecuta la eliminación del archivo
func ParseRmdisk(tokens []string) (*RMDISK, error) {
    cmd := &RMDISK{}
    args := strings.Join(tokens, " ")

    re := regexp.MustCompile(`-path="[^"]+"|-path=[^\s]+`)
    matches := re.FindAllString(args, -1)

    for _, match := range matches {
        kv := strings.SplitN(match, "=", 2)
        if len(kv) != 2 {
            return nil, fmt.Errorf("formato de parámetro inválido: %s", match)
        }
        key, value := strings.ToLower(kv[0]), kv[1]

        if strings.HasPrefix(value, "\"") && strings.HasSuffix(value, "\"") {
            value = strings.Trim(value, "\"")
        }
```

- FDISK: Administra particiones dentro de un disco. Permite crear, eliminar y modificar particiones primarias, extendidas o lógicas.

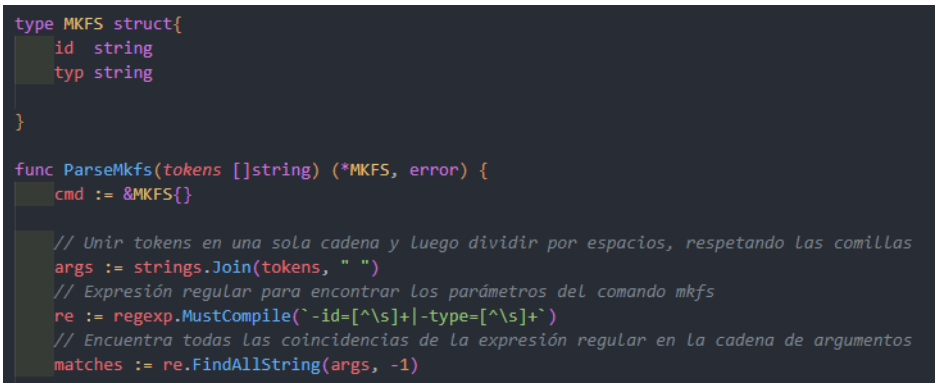
- MOUNT: Monta una partición primaria o lógica en RAM y genera un ID único para su identificación.

```go
// MOUNT representa el comando mount
type MOUNT struct {
    path string
    name string
}

// ParseMount parsea los tokens y devuelve una instancia del comando mount
func ParseMount(tokens []string) (*MOUNT, error) {
    cmd := &MOUNT{}
    args := strings.Join(tokens, " ")
    re := regexp.MustCompile(`-path="[^"]+"|-path=[^\s]+|-name="[^"]+"|-name=[^\s]+`)
    matches := re.FindAllString(args, -1)

    for _, match := range matches {
        kv := strings.SplitN(match, "=", 2)
        if len(kv) != 2 {
            return nil, fmt.Errorf("formato de parámetro inválido: %s", match)
        }
        key, value := strings.ToLower(kv[0]), kv[1]

        if strings.HasPrefix(value, "\"") && strings.HasSuffix(value, "\"") {
            value = strings.Trim(value, "\"")
        }
```

- MOUNTED: Lista todas las particiones montadas actualmente en el sistema.

Administración del Sistema de Archivos:

- MKFS: Formatea una partición montada con EXT2. Crea estructuras internas como superbloque, inodos, bloques y el archivo `users.txt`.



```go
type MKFS struct{
    id  string
    typ string

}

func ParseMkfs(tokens []string) (*MKFS, error) {
    cmd := &MKFS{}

    // Unir tokens en una sola cadena y luego dividir por espacios, respetando las comillas
    args := strings.Join(tokens, " ")
    // Expresión regular para encontrar los parámetros del comando mkfs
    re := regexp.MustCompile(`-id=[^\s]+|-type=[^\s]+`)
    // Encuentra todas las coincidencias de la expresión regular en la cadena de argumentos
    matches := re.FindAllString(args, -1)
```

- CAT: Muestra el contenido de uno o varios archivos si el usuario tiene permisos de lectura.

Administración de Usuarios y Grupos:

- LOGIN: Inicia sesión en una partición montada. Solo un usuario puede estar activo a la vez.

- LOGOUT: Cierra la sesión actual del usuario.

- MKGRP: Crea un nuevo grupo de usuarios. Solo puede ser ejecutado por el usuario root.

- RMGRP: Elimina un grupo de usuarios existente. Requiere permisos de root.

- MKUSR: Crea un nuevo usuario dentro de un grupo existente. Solo el root puede ejecutarlo.

- RMUSR: Elimina un usuario existente. Solo el root puede ejecutarlo.

- CHGRP: Cambia el grupo al que pertenece un usuario. Solo el root puede realizarlo.

Archivos y Carpetas:

- MKFILE: Crea un archivo con tamaño específico y contenido opcional. Verifica permisos del usuario sobre la carpeta padre.

- MKDIR: Crea carpetas. Puede crear jerarquía completa si se usa el parámetro `-p`.

Script:

- Archivos .smia: Permiten cargar una serie de comandos y comentarios para ejecución masiva.

Reportes (REP):

- REP: Genera reportes visuales y de texto sobre las estructuras del sistema de archivos con Graphviz.

```go
func ReportMBR(mbr *structures.MBR, diskPath string, reportPath string) error {
    err := utils.CreateParentDirs(reportPath)
    if err != nil {
        return err
    }

    dotFileName, outputImage := utils.GetFileNames(reportPath)

    dotContent := fmt.Sprintf(`digraph G {
        node [shape=plaintext]
        tabla [label=<
            <table border="0" cellborder="1" cellspacing="0">
                <tr><td colspan="2"> REPORTE MBR </td></tr>
                <tr><td>mbr_tamano</td><td>%d</td></tr>
                <tr><td>mrb_fecha_creacion</td><td>%s</td></tr>
                <tr><td>mbr_disk_signature</td><td>%d</td></tr>
            `, mbr.Mbr_size, time.Unix(int64(mbr.Mbr_creation_date), 0), mbr.Mbr_disk_signature)

    for i, part := range mbr.Mbr_partitions {
        if part.Part_size == -1 {
            continue
        }

        partName := strings.TrimRight(string(part.Part_name[:]), "\x00")
        partStatus := rune(part.Part_status[0])
        partType := rune(part.Part_type[0])
        partFit := rune(part.Part_fit[0])
```

4. Notas Finales

- El sistema simula EXT2 sobre archivos `.mia`, lo que permite experimentar sin hardware real.

- La sesión activa es obligatoria para ejecutar la mayoría de comandos después de `mkfs`.

- El usuario `root` tiene permisos absolutos (777).