Universidad De San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
LABORATORIO SISTEMAS OPERATIVOS 2
Sección “A”

“MANUAL TECNICO”

Luis Angel Barrera Velásquez

Carné: 202010223

Índice

Introducción

Este manual técnico tiene como objetivo principal detallar el funcionamiento del código fuente del Proyecto 1. Está diseñado para ser de utilidad a cualquier persona con conocimientos en programación que desee llevar a cabo futuras actualizaciones o corregir errores durante la ejecución del programa. Es importante destacar que, para lograr una actualización efectiva de este proyecto, se recomienda encarecidamente la lectura de este artículo. Esto permitirá comprender el flujo del programa, tanto para modificaciones en el backend como en el frontend, que se describirán más adelante.

El proyecto se compone esencialmente de un backend desarrollado en Go, encargado de proporcionar información sobre el estado de la memoria RAM. Esta información se transmite al frontend a través de un endpoint que se consulta cada 5 segundos. El frontend, desarrollado en React, se encarga de presentar esta información de manera agradable para el usuario. A continuación, se detallará con más precisión su funcionamiento.

Backend

Lenguaje Go

El Lenguaje Go, al igual que C y C++, es un lenguaje compilado y concurrente, o en otras palabras: soporta canales de comunicación basados en el lenguaje CSP. Sin embargo, dentro de las características de Go aparece su concurrencia. En Go es diferente a los criterios de programación basados en bloqueos como pthreads. Los creadores de Go, además, se inspiraron en la versatilidad y las cualidades de otros lenguajes como Python, C++ y Java (entre otros), para conseguir un lenguaje con las siguientes características, algunas únicas, y otras compartidas con otros lenguajes compilados.

Explicación de código fuente

Importaciones

```
import (
    "encoding/json"
    "fmt"
    "net/http"
    "os/exec"
    "strconv"
    "strings"

    "github.com/gorilla/handlers"
    "github.com/gorilla/mux"
)
```

encoding/json
Este paquete se utiliza para poder manejar de mejor manera la codificación y decodificar json que contienen los datos necesarios para la lógica del programa y encapsular la información de manera correcta al frontend.

fmt
Este paquete fue necesario nada mas para poder mostrar mensajes en consola y fueron de utilidad para la realización de pruebas durante el desarrollo de las funciones.

net/http
Este paquete es necesario para cuando se tiene que trabajar Go en alguna aplicación web ya que funciona para manejar solicitudes http y https.

os/exec
Este paquete es necesario para la ejecucion de comandos externos al programa en este caso fue necesario para ejecutar comandos de Linux mint y ver la información necesaria de RAM y procesos.

strconv
Este paquete es necesario para el manejo de strings, es decir para conversiones a strings.

Strings
Este paquete a diferencia de strconv es de utilidad para el manejo de cadenas mas no para convertir.

github.com/gorilla/handlers
Es una biblioteca que es de utilizada gorilla mux y sirve para el manejo de aplicaciones web de go.

github.com/gorilla/mux
Es un paquete de utilizada en aplicaciones web de go que sirve para el enrutamiento de rutas de solicitudes https.

Structs utilizados

```go
type InfoRAM struct {
    Total    uint64 `json:"total"`
    Usada    uint64 `json:"usada"`
    Activa   uint64 `json:"activa"`
    Inactiva uint64 `json:"inactiva"`
    Libre    uint64 `json:"libre"`
    Bufers   uint64 `json:"bufers"`
}

type InfoProceso struct {
    PID      int    `json:"pid"`
    Usuario  string `json:"usuario"`
    Comando  string `json:"comando"`
    OOMScore int    `json:"oom_score"`
}
```

InfoRAM
Este struct guarda la información de la memoria RAM del sistema y que encapsula los valores para regresarlos al frontend las cadenas `json: xxxxx` son para especificar el nombre de la llave del json que se retornara.

InfoProceso

Este struct es utilizado para guardar la información de cada proceso que después se enviaran varios en un arreglo al frontend y ser mostrados de manera mas agradable al usuario.

Función ObtenerInfoRAM

```go
func ObtenerInfoRAM() (InfoRAM, error) {
    out, err := exec.Command("vmstat", "-s", "-S", "M").Output()
    if err != nil {
        return InfoRAM{}, err
    }

    lines := strings.Split(string(out), "\n")
    info := InfoRAM{}

    for _, line := range lines {
        fields := strings.Fields(line)
        if len(fields) < 2 {
            continue
        }

        value, _ := strconv.ParseUint(fields[0], 10, 64)
        switch {
        case strings.Contains(line, "M memoria total"):
            info.Total = value
        case strings.Contains(line, "M memoria usada"):
            info.Usada = value
        case strings.Contains(line, "M memoria activa"):
            info.Activa = value
        case strings.Contains(line, "M memoria inactiva"):
            info.Inactiva = value
        case strings.Contains(line, "M memoria libre"):
            info.Libre = value
```

Esta función básicamente ejecuta el comando vmstat -s -S M que lo que hace este comando es obtener la información de la memoria virtual y este después en la función es procesada la información que devuelve el comando línea a línea y es guardada en el struct infoRAM.

Función ObtenerListaProcesos

```go
func ObtenerListaProcesos() ([]InfoProceso, error) {
    out, err := exec.Command("ps", "aux").Output()
    if err != nil {
        return nil, err
    }

    lineas := strings.Split(string(out), "\n")
    procesos := []InfoProceso{}
    for _, linea := range lineas[1:] {
        campos := strings.Fields(linea)
        if len(campos) >= 11 {
            pid, _ := strconv.Atoi(campos[1])
            usuario := campos[0]
            comando := campos[10]
            oomScore, err := obtenerOOMScore(campos[1])
            if err != nil {
                fmt.Println("Error al obtener el OOM score:", err)
            }
            procesos = append(procesos, InfoProceso{
                PID:      pid,
                Usuario:  usuario,
                Comando:  comando,
                OOMScore: oomScore,
            })
        }
    }
```

Esta función ejecuta el comando ps aux donde ps es el comando principal y aux separando la a significa todos los usuarios, la u significa formato largo o detallado es decir devuelve a detalle toda la información de los procesos.

Función ObtenerInfoSistemaHandler

```go
func ObtenerInfoSistemaHandler(w http.ResponseWriter, r *http.Request) {
    infoRAM, err := ObtenerInfoRAM()
    if err != nil {
        http.Error(w, "Error al obtener la información de RAM", http.StatusInternalServerError)
        return
    }

    listaProcesos, err := ObtenerListaProcesos()
    if err != nil {
        http.Error(w, "Error al obtener la lista de procesos", http.StatusInternalServerError)
        return
    }

    infoSistema := struct {
        RAM      InfoRAM       `json:"ram"`
        Procesos []InfoProceso `json:"procesos"`
    }{
        RAM:      infoRAM,
        Procesos: listaProcesos,
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(infoSistema)
}
```

Esta función básicamente sirve para llamar a las demás funciones y estructurar un único json de respuesta a la petición que hace el frontend.

Función obtenerOOMScore

```go
func obtenerOOMScore(pid string) (int, error) {
    out, err := exec.Command("cat", fmt.Sprintf("/proc/%s/oom_score", pid)).Output()
    if err != nil {
        return 0, err
    }

    oomScore, err := strconv.Atoi(strings.TrimSpace(string(out)))
    if err != nil {
        return 0, err
    }

    return oomScore, nil
}
```

Esta función es bastante sencilla recibe como parámetro el pid y retorna el valor del OOM score consultando el directorio /proc de Linux

Función main

```go
func main() {
    enrutador := mux.NewRouter()

    enrutador.HandleFunc("/info-sistema", ObtenerInfoSistemaHandler).Methods("GET")

    origenesPermitidos := handlers.AllowedOrigins([]string{"*"})
    metodosPermitidos := handlers.AllowedMethods([]string{"GET", "OPTIONS"})
    headersPermitidos := handlers.AllowedHeaders([]string{"Content-Type"})

    fmt.Println("El servidor está ejecutándose en :5000")
    http.ListenAndServe(":5000", handlers.CORS(origenesPermitidos, metodosPermitidos, headersPermitidos)(
}
```

Función principal del programa que hace levantar la api en el puerto 5000 de manera local y en esta función también es donde se realiza en enrutamiento del endpoint info-sistema que es el endpoint único que consulta el frontend.

Frontend

React

Es una biblioteca de JavaScript desarrollada por Facebook que se utiliza para crear interfaces de usuario interactivas y dinámicas en aplicaciones web. Se basa en la creación de componentes reutilizables que representan partes específicas de la interfaz de usuario y permite una actualización eficiente de la interfaz en respuesta a cambios en los datos o el estado de la aplicación. React se integra fácilmente con JavaScript y se utiliza ampliamente en el desarrollo web moderno para construir aplicaciones rápidas y eficientes.

Explicación de código fuente

Funcionalidad de actualizar cada 5 segundos

```
useEffect(() => {
    const interval = setInterval(() => {
      logs();
    }, 5000);

    return () => clearInterval(interval);
  }, []);
```

Esta funcion utiliza un hook llamado useEffect que hace que la pagina se actualice cada 5 segundos este tiempo esta definido en ms en este caso 5000 ms por lo que si se quiere aumentar se puede hacer sin ningún problema tomando en cuenta que son ms y no segundos directamente esta funcion llama a la funcion logs cada 5 segundos.

Funcion logs

```
async function logs() {
  const response = await fetch('http://localhost:5000/info-sistema');
  const data = await response.json();

  setramTotal(data.ram.total)
  setramUsada(data.ram.usada)
  setramActiva(data.ram.activa)
  setramInactiva(data.ram.inactiva)
  setramLibre(data.ram.libre)
  setramBufCache(data.ram.bufers)


  const ramUsada = data.ram.usada;
  const ramTotal = data.ram.total;
  const porcentajeRam = ((ramUsada / ramTotal) * 100).toFixed(1);
  setRam(porcentajeRam);

  const procesosConId = data.procesos.map((proceso, index) => ({
    ...proceso,
    id: index + 1,
  }));

  setProcesos(procesosConId);
}
```

Esta función básicamente lo que hace es ejecutar un fetch que va a consultar al backend anteriormente descrito consultando el localhost en el puerto 5000 y haciendo un get con /info-sistema que devuelve un json anteriormente explicado y guardando los datos en variables que se utilizaran para el despliegue de información en el caso de la RAM en componentes visuales agradables para el usuario y los procesos en una tabla de React.

Conclusiones

- El backend está desarrollado en Go y se beneficia de una estructura bien organizada, con funciones claramente definidas y paquetes específicos que facilitan el manejo de tareas como la obtención de información del sistema y la gestión de solicitudes HTTP. El uso de paquetes estándar de Go y bibliotecas externas como Gorilla Mux demuestra una elección inteligente de herramientas para el desarrollo de aplicaciones web.
- El frontend se basa en React, lo que facilita la creación de una interfaz de usuario dinámica y reactiva. La utilización de useEffect para actualizar automáticamente los datos cada 5 segundos proporciona una experiencia de usuario en tiempo real, lo que es especialmente útil para mostrar información de la RAM y los procesos del sistema de forma continua.

Referencias

- Yellavula, N. (2017). Building RESTful Web services with Go: Learn how to build powerful RESTful APIs with Golang that scale gracefully. Packt Publishing Ltd.
- Ardiansyah, H., & Fatwanto, A. (2022). Comparison of Memory usage between REST API in Javascript and Golang. *MATRIK: Jurnal Manajemen, Teknik Informatika dan Rekayasa Komputer*, *22*(1), 229-240.
- Thakkar, M., & Thakkar, M. (2020). Introducing react. js. *Building React Apps with Server-Side Rendering: Use React, Redux, and Next to Build Full Server-Side Rendering Applications*, 41-91.