MANUAL TÉCNICO

Servidor Golang

Puerto: 5000

Librerías:

github.com/gorilla/mux

Permite enrutar distintas direcciones de la API, estableciendo los métodos http por los cuales serán consumidos los endpoints.

```
router := mux.NewRouter().StrictSlash(true)
router.HandleFunc("/getRam", getRam).Methods("GET")
router.HandleFunc("/getProcs", getProcs).Methods("GET")
router.HandleFunc("/killProc", killProc).Methods("DELETE")
```

github.com/gorilla/handlers

Establece configuraciones de acceso necesarias para la API como los encabezados , métodos http permitidos , origen de las peticiones como también las CORS.

```
headers := handlers.AllowedHeaders([]string{"X-Requested-With", "Content-Type", "Authorization", "application/json"})
methods := handlers.AllowedMethods([]string{"GET", "POST", "PUT", "DELETE"})
origins := handlers.AllowedOrigins([]string{"*"})
```

net/http

Es necesario para poder levantar la API, en este caso funciona con el primer parametro que es el número de puerto, y en el segundo establecemos las cors.

```
http.ListenAndServe(":5000",handlers.CORS(headers, methods, origins)(router))
```

io/ioutil

Permite la lectura de los archivos modificados por los módulos de kernel y también la lectura de las peticiones.

encoding/json

Útil para convertir objetos a json y viceversa.

os

Permite abrir los archivos modificados por los módulos de kernel y también obtener un proceso del sistema y poder detenerlo.

strconv

Permite la conversión de cadenas de caracteres a enteros y de enteros a cadenas de caracteres.

Endpoints:

getRam(w http.ResponseWriter, r *http.Request)

No recibe ningún tipo de objeto en el request, es de tipo GET, lee el archivo /proc/serverRam y devuelve el porcentaje de Ram utilizado en un objeto json con el porcentaje de ram.

```
func getRam(w http.ResponseWriter, r *http.Request){

	file_data, err := os.Open("/proc/serverRam")
	if err != nil {
		fmt.Println("Error al leer modulo ram")
	}
	defer file_data.Close()
	byteValue, _ := ioutil.ReadAll(file_data)
	var result map[string]interface{}
	json.Unmarshal([]byte(byteValue), &result)
	fmt.Println(result["ram"])
	json.NewEncoder(w).Encode(result)
}
```

getProcs(w http.ResponseWriter, r *http.Request)

No recibe ningún tipo de objeto en el request, es de tipo GET, lee el archivo /proc/serverProcs2 y devuelve un arreglo en el que se encuentra la lista de procesos del sistema en formato string.

```
func getProcs(w http.ResponseWriter, r *http.Request){
	file_data, err := ioutil.ReadFile("/proc/serverProcs2")
	if err != nil {
		fmt.Println("Error al leer modulo procs")
	}
	fmt.Fprintf(w, string(file_data))
}
```

killProc(w http.ResponseWriter, r *http.Request)

Recibe el pid del proceso que se quiere detener, es de tipo DELETE y detiene el proceso.

```
func killProc(w http.ResponseWriter, r *http.Request){
	var p Proceso
	reqbody, errpid := ioutil.ReadAll(r.Body)
	if errpid != nil {
		fmt.Fprintf(w, "Error en leer proceso")
	}
	json.Unmarshal(reqbody, &p)
	proc, err := os.FindProcess(p.Pid)
	if err != nil {
		fmt.Println("Error al eliminar proceso", p.Pid)
	}
	proc.Kill()
	fmt.Fprintf(w, "Proceso " + strconv.Itoa(p.Pid) + " eliminado.")
}
```

MANUAL DE USUARIO

SERVIDOR API GOLANG

Este servidor nos permite la interacción con el sistema a monitorear, funciona en el puerto 5000 y puede ser levantado con el comando “go run app.go”.

El servidor cuenta con los siguientes endpoints:

getRam

Es de tipo GET y nos devuelve el porcentaje de ram utilizado en formato json. Ejemplo {“ram”:58}.

getProcs

Es de tipo GET y nos devuelve el listado de procesos del sistema por medio de un string.

killProc

Es de tipo DELETE y recibe un objeto de tipo json de la siguiente manera {“pid” : 513} la cual nos permite detener el proceso con id 513Módulos de Kernel

serverRAM

- Este módulo de kernel el cual revisa la cantidad de RAM en el sistema, tomando en cuenta la total, la disponible y la libre, esto mediante el uso del struct sysinfo el cual es interno al kernel.
    - Sysinfo es el struct en C del kernel que representa al sistema como tal.
        - Sysinfo.totalram es la cantidad de RAM disponible en el sistema en razón de paginas
        - Se define una función K la cual transforma el valor de páginas a bytes.

serverProcs2

- Este módulo de kernel el cual revisa cada proceso en el sistema mediante la función interna “for_each_process” con el struct “task_struct”. Task_struct es interno del kernel de Linux y contiene toda la información necesaria sobre el proceso.
    - task_struct es el struct en C del kernel el cual representa a los procesos en ejecución, de este se sacan los datos:
        - pid -> que es el identificador, numérico, del proceso.
        - Comm -> es el nombre que recibe el proceso.
        - Cred->uid.val -> es el identificador, numérico, del usuario dueño del proceso.
        - State -> es una representación numérica del estado del proceso, basado en:
            - 0 -> Ejecutándose
            - 1 -> Interrumpible
            - 2 -> no interrumpible
            - 4 -> zombi
            - 8 -> detenido
        - bytesUsed -> que representa task->mm->total_vm << PAGE_SHIFT lo cual es la cantidad de bytes usados de RAM (task->mm->total_vm retorna la cantidad de páginas utilizadas en el sistema)

Sitio Web