# WebSocket Guide

DevOpsCoreAutomation

*Real-time push updates with Gorilla WebSocket*
*Hub pattern + Gin integration + Scheduler broadcasting*

# 1. New File: internal/websocket/hub.go

The Hub is the central registry of all active WebSocket connections. It runs in its own goroutine and handles register, unregister, and broadcast.

***internal/websocket/hub.go***

```
package websocket

import "sync"

type Hub struct {
    clients    map[*Client]bool
    broadcast  chan []byte
    register   chan *Client
    unregister chan *Client
    mu         sync.RWMutex
}

func NewHub() *Hub {
    return &Hub{
        clients:    make(map[*Client]bool),
        broadcast:  make(chan []byte, 256),
        register:   make(chan *Client),
        unregister: make(chan *Client),
    }
}

func (h *Hub) Run() {
    for {
        select {
        case client := <-h.register:
            h.mu.Lock()
            h.clients[client] = true
            h.mu.Unlock()

        case client := <-h.unregister:
            h.mu.Lock()
            if _, ok := h.clients[client]; ok {
                delete(h.clients, client)
                close(client.send)
            }
            h.mu.Unlock()

        case message := <-h.broadcast:
            h.mu.RLock()
            for client := range h.clients {
                select {
                case client.send <- message:
                default:
                    close(client.send)
                    delete(h.clients, client)
                }
            }
            h.mu.RUnlock()
        }
    }
}

func (h *Hub) Broadcast(message []byte) {
    h.broadcast <- message
}
```

## 2. New File: internal/websocket/client.go

Each connected browser gets a Client. Two goroutines run per client: ReadPump (listens for pings/messages from client) and WritePump (sends messages + periodic pings to detect dead connections).

***internal/websocket/client.go***

```
package websocket

import (
    "time"
    "github.com/gorilla/websocket"
)

const (
    writeWait      = 10 * time.Second
    pongWait       = 60 * time.Second
    pingPeriod     = (pongWait * 9) / 10
    maxMessageSize = 512
)

type Client struct {
    hub  *Hub
    conn *websocket.Conn
    send chan []byte
}

func NewClient(hub *Hub, conn *websocket.Conn) *Client {
    return &Client{
        hub:  hub,
        conn: conn,
        send: make(chan []byte, 256),
    }
}

func (c *Client) ReadPump() {
    defer func() {
        c.hub.unregister <- c
        c.conn.Close()
    }()
    c.conn.SetReadLimit(maxMessageSize)
    c.conn.SetReadDeadline(time.Now().Add(pongWait))
    c.conn.SetPongHandler(func(string) error {
        c.conn.SetReadDeadline(time.Now().Add(pongWait))
        return nil
    })
    for {
        _, _, err := c.conn.ReadMessage()
        if err != nil {
            break
        }
    }
}

func (c *Client) WritePump() {
    ticker := time.NewTicker(pingPeriod)
    defer func() {
        ticker.Stop()
        c.conn.Close()
    }()
    for {
        select {
        case message, ok := <-c.send:
            c.conn.SetWriteDeadline(
                time.Now().Add(writeWait),
            )
```

```
            if !ok {
                c.conn.WriteMessage(
                    websocket.CloseMessage, []byte{},
                )
                return
            }
            err := c.conn.WriteMessage(
                websocket.TextMessage, message,
            )
            if err != nil {
                return
            }

        case <-ticker.C:
            c.conn.SetWriteDeadline(
                time.Now().Add(writeWait),
            )
            err := c.conn.WriteMessage(
                websocket.PingMessage, nil,
            )
            if err != nil {
                return
            }
        }
    }
}
```

## 3. New File: internal/websocket/handler.go

The Gin handler that upgrades HTTP to WebSocket. It validates the JWT cookie before accepting the connection, matching your existing auth flow.

***internal/websocket/handler.go***

```
package websocket

import (
    "net/http"

    auth "github.com/Flafl/DevOpsCore/internal/Auth"
    "github.com/gin-gonic/gin"
    "github.com/gorilla/websocket"
)

var upgrader = websocket.Upgrader{
    ReadBufferSize:  1024,
    WriteBufferSize: 1024,
    CheckOrigin: func(r *http.Request) bool {
        return true
    },
}

func ServeWS(
    hub *Hub, jwtManager *auth.JWTManager,
) gin.HandlerFunc {
    return func(c *gin.Context) {
        token, err := c.Cookie("access_token")
        if err != nil || token == "" {
            c.JSON(http.StatusUnauthorized,
                gin.H{"error": "not authenticated"})
            return
        }

        _, err = jwtManager.ValidateToken(token)
        if err != nil {
            c.JSON(http.StatusUnauthorized,
                gin.H{"error": "invalid token"})
            return
        }

        conn, err := upgrader.Upgrade(
            c.Writer, c.Request, nil,
        )
        if err != nil {
            return
        }

        client := NewClient(hub, conn)
        hub.register <- client

        go client.WritePump()
        go client.ReadPump()
    }
}
```

# 4. Modify: internal/router/router.go

Add the hub parameter and the /ws route. The WebSocket route sits outside the auth middleware groups because handler.go checks the JWT cookie itself.

***internal/router/router.go (showing changed parts)***

```
package router

import (
    auth "github.com/Flafl/DevOpsCore/internal/Auth"
    "github.com/Flafl/DevOpsCore/internal/handlers"
    "github.com/Flafl/DevOpsCore/internal/middleware"
    ws "github.com/Flafl/DevOpsCore/internal/websocket"
    "github.com/gin-gonic/gin"
)

func Setup(
    r *gin.Engine,
    jwtManager *auth.JWTManager,
    hub *ws.Hub,                    // NEW parameter
    powerH *handlers.PowerHandler,
    descH *handlers.DescriptionHandler,
    healthH *handlers.HealthHandler,
    portH *handlers.PortHandler,
    backupH *handlers.BackupHandler,
    userH *handlers.UserhHandler,
    authH *handlers.AuthHandler,
    pageH *handlers.PageHandler,
) {
    // WebSocket endpoint (auth inside handler)
    r.GET("/ws", ws.ServeWS(hub, jwtManager))

    // ... rest of existing routes unchanged ...
    r.GET("/login", pageH.Login)
    r.POST("/api/auth/login", authH.Login)
    // ... etc ...
}
```

# 5. Modify: internal/scheduler/scheduler.go

Add a hub field to the Scheduler. After each scan job finishes, broadcast a JSON message so all connected browsers know to refresh.

## 5a. Updated struct + constructor

```
import (
    // ... existing imports ...
    ws "github.com/Flafl/DevOpsCore/internal/websocket"
)

type Scheduler struct {
    cfg        *config.Config
    hub        *ws.Hub               // NEW field
    powerRepo  repository.PowerRepository
    descRepo   repository.DescriptionRepository
    healthRepo repository.HealthRepository
    portRepo   repository.PortProtectionRepository
    backupRepo repository.BackupRepository
}

func New(
    cfg *config.Config,
    hub *ws.Hub,                     // NEW parameter
    pr repository.PowerRepository,
    dr repository.DescriptionRepository,
    hr repository.HealthRepository,
    pp repository.PortProtectionRepository,
    br repository.BackupRepository,
) *Scheduler {
    return &Scheduler{
        cfg:        cfg,
        hub:        hub,             // NEW
        powerRepo:  pr,
        descRepo:   dr,
        healthRepo: hr,
        portRepo:   pp,
        backupRepo: br,
    }
}
```

## 5b. Notify helper + usage

```
// Add this helper at the bottom of the file:
func (s *Scheduler) notify(eventType string) {
    msg, _ := json.Marshal(map[string]string{
        "type": eventType,
    })
    s.hub.Broadcast(msg)
}

// Then add ONE line at the end of each scan function,
// right before the "done" log:

func (s *Scheduler) runPowerScan() {
    // ... existing logic ...
    s.notify("power_update")
    log.Println("[job] power-scan: done")
}

func (s *Scheduler) runDescScan() {
    // ... existing logic ...
    s.notify("desc_update")
    log.Println("[job] desc-scan: done")
}

func (s *Scheduler) runHealthScan() {
    // ... existing logic ...
    s.notify("health_update")
    log.Println("[job] health-scan: done")
}

func (s *Scheduler) runPortScan() {
    // ... existing logic ...
    s.notify("port_update")
    log.Println("[job] port-scan: done")
}

func (s *Scheduler) runBackup() {
    // ... existing logic ...
    s.notify("backup_update")
    log.Println("[job] backup: done")
}
```

## 6. Modify: cmd/api/main.go

Create the Hub, start it in a goroutine, and pass it to both the scheduler and the router.

***cmd/api/main.go (showing changed parts)***

```
import (
    // ... existing imports ...
    ws "github.com/Flafl/DevOpsCore/internal/websocket"
)

func main() {
    cfg := config.Load()
    database := db.Connect(cfg)

    // ... existing repo creation ...

    jwtManager := auth.NewJWTManager(auth.JWTconfig{
        // ... existing config ...
    })

    // === NEW: Create and start WebSocket hub ===
    hub := ws.NewHub()
    go hub.Run()

    // Pass hub to scheduler (NEW: hub parameter)
    sched := scheduler.New(
        cfg, hub,
        powerRepo, descRepo, healthRepo,
        portRepo, backupRepo,
    )
    sched.Start()

    server := gin.Default()
    // ... existing static file setup ...
    // ... existing handler creation ...

    // Pass hub to router (NEW: hub parameter)
    router.Setup(
        server, jwtManager, hub,
        powerH, descH, healthH, portH,
        backupH, userH, authH, pageH,
    )

    // ... existing graceful shutdown ...
}
```

## 7. Client-Side JavaScript

Add this to your dashboard template (templates/excess/dashboard.html or templates/layout/base.html for global coverage). It auto-connects and auto-reconnects on disconnect.

***Add to dashboard.html or base.html***

```
<script>
(function() {
    var ws;

    function connect() {
        ws = new WebSocket(
            "ws://" + window.location.host + "/ws"
        );

        ws.onmessage = function(event) {
            var msg = JSON.parse(event.data);

            if (msg.type === "power_update") {
                fetchPowerData();
            }
            if (msg.type === "health_update") {
                fetchHealthData();
            }
            if (msg.type === "port_update") {
                fetchPortData();
            }
            if (msg.type === "backup_update") {
                fetchBackupData();
            }
            if (msg.type === "desc_update") {
                fetchDescData();
            }
        };

        ws.onclose = function() {
            console.log("WS closed, reconnecting...");
            setTimeout(connect, 3000);
        };

        ws.onerror = function() {
            ws.close();
        };
    }

    connect();
})();
</script>
```

Replace fetchPowerData(), fetchHealthData(), etc. with whatever JavaScript functions you already use to reload data on your dashboard.

## 8. Install the Dependency

Run this in your project root:

```
go get github.com/gorilla/websocket
```

## 9. Message Types Reference

| Type | Sent After | Client Action |
|---|---|---|
| power_update | runPowerScan() | Refresh power table |
| health_update | runHealthScan() | Refresh health charts |
| port_update | runPortScan() | Refresh port alerts |
| desc_update | runDescScan() | Refresh descriptions |
| backup_update | runBackup() | Refresh backup list |

## 10. Complete File Summary

New files to create:

| File | Purpose |
|---|---|
| internal/websocket/hub.go | Hub: register, unregister, broadcast |
| internal/websocket/client.go | Client: ReadPump + WritePump |
| internal/websocket/handler.go | Gin handler: JWT check + upgrade |

Existing files to modify:

| File | Change |
|---|---|
| internal/router/router.go | Add hub param + /ws route |
| internal/scheduler/scheduler.go | Add hub field + notify() calls |
| cmd/api/main.go | Create hub, pass to scheduler + router |
| templates (dashboard/base) | Add JS WebSocket listener |

## 11. How It All Flows Together

1. Application starts: main.go creates the Hub and calls go hub.Run()

2. Browser loads dashboard, JavaScript opens WebSocket to /ws

3. handler.go validates JWT cookie, upgrades connection, registers Client in Hub

4. Client ReadPump + WritePump goroutines start

5. Scheduler runs a scan (e.g. runPowerScan), writes results to DB

6. Scheduler calls s.notify("power_update") which sends to hub.Broadcast()

7. Hub forwards the message to ALL connected Clients via their send channels

8. WritePump picks up the message and writes it to the WebSocket connection

9. Browser receives the message, JavaScript calls fetchPowerData() to reload

10. If browser disconnects, ReadPump detects it and unregisters the Client