# Sprint3

## Introduzione

L’obiettivo in questo sprint è introdurre un'interfaccia grafica web (WebGUI) che consenta all’utente finale di monitorare lo stato della hold, fornendo una visione chiara e aggiornata del sistema, senza interferire con la logica applicativa.

In questo sprint viene realizzata una base architetturale della WebGUI, sulla quale sarà possibile costruire e completare le funzionalità di integrazione.

### Requisiti

Il committente ha come requisito quello di poter avere un interfaccia web consultabile che permetta la rappresentazione in tempo reale dello stato della Hold.

## Analisi del problema

### Modellazione

Dai requisiti emerge che la GUI deve essere:

- Accessibile via web
- Indipendente dal sistema operativo
- Orientata alla sola visualizzazione dello stato

Per tali motivi, la WebGUI non viene modellata come un attore, ma come una applicazione web separata, in grado di comunicare con il sistema esistente e accessibile semplicemente tramite browser, indipendente dal sistema sottostante.

### Estrarre le informazioni sullo stato della hold e visualizzazione

L'attore cargoservice, in esecuzione nel contesto ctx_cargo, contesto del corebusiness del sistema aggiorna costantemente lo stato della hold. Per consentire alla GUI di visualizzare lo stato della hold, si possono esaminare molteplici modalità, come ad esempio:

- Implementare un'interfaccia RESTful nell'attore cargoservice per consentire alla GUI di effettuare richieste HTTP e ottenere lo stato della hold in formato JSON.
- Utilizzare un sistema di messaggistica (ad esempio MQTT o CoAP) per inviare aggiornamenti in tempo reale da cargoservice alla GUI ogni volta che lo stato della hold cambia.
- Implementare un meccanismo di polling nella GUI che richieda periodicamente lo stato della hold all'attore cargoservice.
- Prevedere l' aggiornamento di una risorsa CoAP da parte di cargoservice e recuperarne lo stato con webgui inizialmente e ad ogni cambiamento.

Si suggerisce di adottare l'ultima soluzione, in quanto coerente con le tecnologie già adottate nel sistema e con i paradigmi di comunicazione utilizzati, inoltre in fase di progettazione dell'attore qak cargoservice è già stato previsto l'aggiornamento di una risorsa CoAP. Tramite questa scelta ci sarà permesso di rimanere

comunque nel concetto di sistema distribuito e non solo, sarà necessario (e lo vedremo in fase di deisgn) un sistema che permetta di recuperare lo stato attuale della hold per poi aggiornarlo in maniera incrementale in base agli avvenimenti. In questo modo WebGui non avrà la necessità di ricevere comunicazioni contenenti interamente tutti i parametri di hold ma solo quelli modificati.

## System Build

Per realizzare un applicativo web come da requisiti suggeriamo l'utilizzo della tecnologia springboot che permette di realizzare una semplice logica per la gestione di pagine web dinamiche. Si pone un secondo problema strutturale, come possiamo gestire lo stato della hold se gli aggiornamenti che provengono dal cargoservice sono incrementali? Non vogliamo che la webgui sia un microservizio dipendente dal cargoservice, bensì deve entrare in funzione in un qualunque momento che può essere distinto dall'avvio del resto del sistema. Abbiamo pensato di risolvere questo problema implementando un primo messaggio che informa su tutto lo stato completo della Hold e qual'ora quel messaggio sia gia stato inviato (Flag memorizzata su webgui) i successivi messaggi vengono ignorati. Dunque qual'ora una seconda webgui venisse aperta su un browser differente, riceverebbe sia gli aggiornamenti incrementali che un messaggio informativo iniziale contente lo stato attuale della hold, consentendo a due webgui separate di poter essere "sincronizzate" con il resto del sistema.

Questo comportamento è spiegato dal seguente attore "hold_observer" che intercetta gli eventi del cargoservice(vive nel suo contesto o un qualsiasi contesto noto al cargoservice in modo da ricevere le sue "emit"). Contiene una modellazione della hold semplificata che viene modificata conseguentemente agli eventi emessi dal cargoservice e si occupa :

```
QActor hold_observer context ctx_cargo{

    [#
        var Taken_slot=arrayListOf("false","false","false","false","true")
        val MAX_LOAD=500
        var CURRENT_LOAD=0
        var Led = "Spento"
        var Sonar = "DFREE"
    #]

    State start initial{
        println("$name | start")
        delay 2000
    }
    Goto updateState

    State observe_hold{
        println("$name | Observing hold")
    }


    Transition t0
    whenEvent slot_changed-> change_slot_status
    whenEvent led_changed -> change_led_status
    whenEvent sonar_changed -> change_sonar_status
    whenEvent current_weight -> change_weight
```

```
    whenRequest get_hold_state -> updateState


    State change_slot_status{
        onMsg(slot_changed: slot_changed(ID,status)){
            [#
            val Slot =  payloadArg(0).toInt()
            if (Taken_slot[Slot]=="false") {
                    Taken_slot[Slot]="true"
                }
                else{
                    Taken_slot[Slot]="false"
                }
            #]
            println(" ho ricevuto un cambio dello slot $Slot che ora
vale")
        }
    }
    Goto updateState


    State change_led_status{
            [#
            if (Led=="Acceso") {
                Led="Spento"
                }
                else{
                    Led="Acceso"
                }
            #]
            println(" ho ricevuto un cambio dello stato del led che ora e'
$Led")
    }
    Goto updateState


    State change_sonar_status{
                onMsg(sonar_changed: sonar_changed(status)){
                [#
                Sonar = payloadArg(0)
                #]
                println(" ho ricevuto un cambio dello stato del sonar che
ora vale $Sonar")
    }
    }
    Goto updateState


    State change_weight{
        onMsg(current_weight: current_weight(weight)){
            [#
            CURRENT_LOAD =  payloadArg(0).toInt()
            #]
            println("ho ricevuto un cambio del peso della stiva che ora
```

```
pesa $CURRENT_LOAD")
        }
    }
    Goto updateState

    State updateState{
        [#
            val Slot1 = Taken_slot[0]
            val Slot2 = Taken_slot[1]
            val Slot3 = Taken_slot[2]
            val Slot4 = Taken_slot[3]
            val Robot_state = "ok"
        #
        ]

updateResource[#"hold_state(slots:1=$Slot1;2=$Slot2;3=$Slot3;4=$Slot4,maxl
oad:1000,weight:$CURRENT_LOAD,sonar:$Sonar,led:$Led,robot:$Robot_state)"#]
        }
    Goto observe_hold
}
```

Questo attore serve per aggiornare le nuove webgui sullo stato attuale dell'hold. I cambiamenti incrementali erano già previsti nello sprint1 per cui sono a carico dell'attore cargoservice.

### Dati mostrati

All'interno dell'applicativo web sono visualizzati:

- Tutti e **5 gli Slot** e il loro stato (Occupato / Libero)
- Lo stato di **LED, SONAR, l'operatività del ROBOT** ed eventuli **interruzioni** che avvengono durante l'esecuzione
- Il **peso attuale** rispetto a MAXLOAD

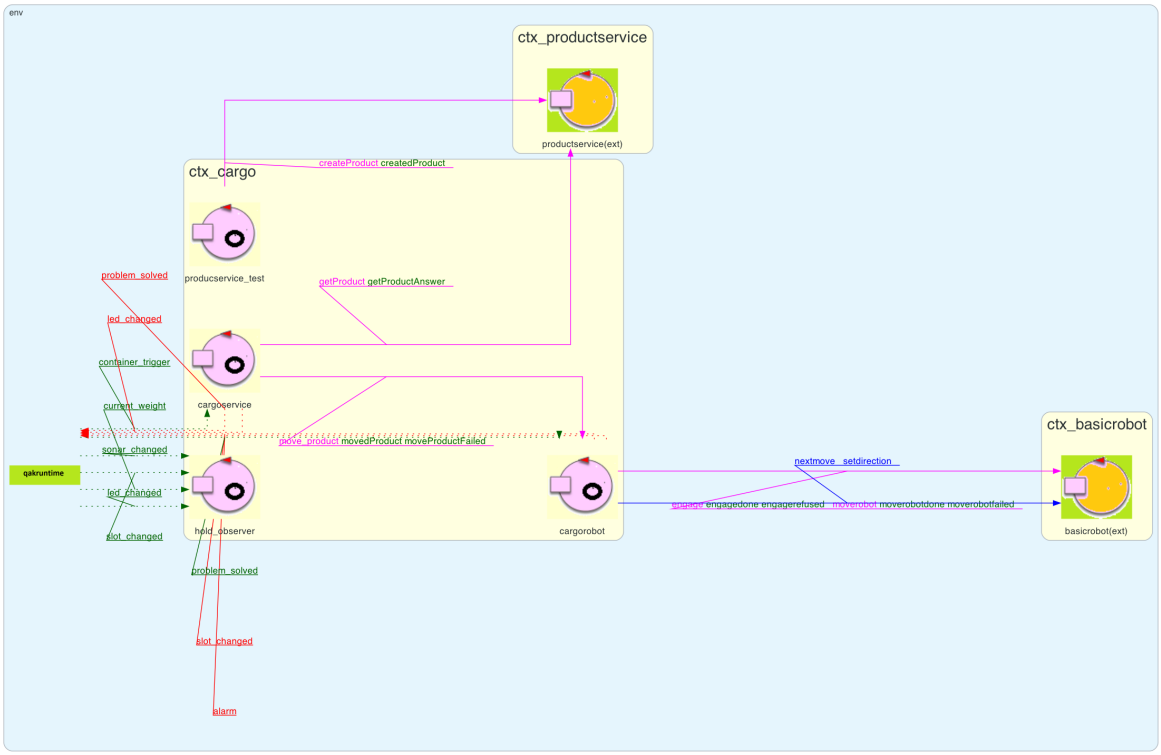
### Architettura Logica di SpringBoot

La WebGUI segue un'architettura SpringBoot con chiare separazioni di responsabilità tra i seguenti layer:

1. Modello dati (`HoldState`): Rappresentazione immutabile dello stato della stiva, stato del sensore, led e metriche sui pesi.
2. Servizio (`HoldStateService`): Coordinatore centrale che mantiene lo stato e orchestra il broadcast ai client connessi.
3. Origine eventi (`CoapObserverService`): Osserva le risorse CoAP dal backend qak usando Eclipse Californium. Estrae eventi strutturati eseguendo del parsing sui payload. Grazie al metodo `parseAndDispatch()` la GUI e le specifiche CoAP sono disaccoppiate, consentendo facilmente simulazioni e testing.
4. Comunicazione (`WebSockerHandlerDemo`, `WebSocketConfig`): Gestione connessioni websocket e propagazione in tempo reale dello stato. Mantiene una lista delle sessioni attive e manda in broadcast lo stato serializzato in JSON a tutti i client connessi (quando lo stato cambia).
5. REST API (`SimulateController`): Fornisce endpoint HTTP per i test ed integrazioni esterne senza richiedere una connessione CoAP in backend.

6. User Interface (`HIControllerDemo`, `index.html`): Interfaccia grafica aggiornata in tempo reale.

## System Architecture




sprint1Arch

## Piano di Test

La WebGUI dispone di una suite di test che copre tutti i componenti principali:

- Data model (`HoldState`): Inizializzazione, update di slot occupati, tracciamento dei pesi.
- CoAP parsing (`CoapObserverService`): Parsing del payload per cambiamenti degli slot, stato del sonar, stato del LED, allarmi, eventi sui pesi.
- State Management (`HoldStateService`): Propagazione dello stato e meccanismi di broadcast.
- REST API (`SimulateController`): Simulazione di eventi per il testing senza una vera connessione CoAP.
- WebSocket Handler: Gestione connessioni e instradamento dei messaggi.
- Flusso end-to-end: Scenario multi-eventi che valida il flusso completo (parsing -> update -> broadcast).