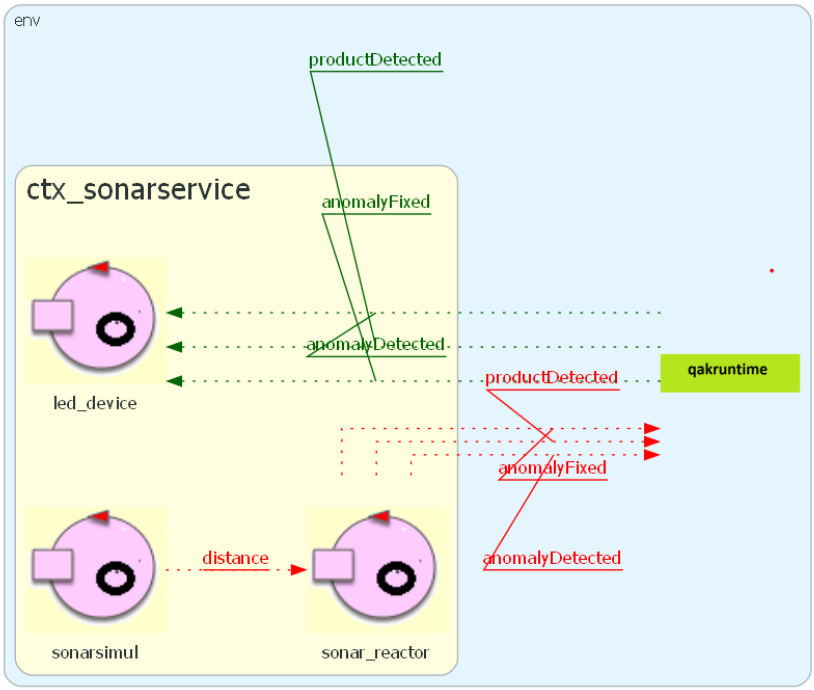
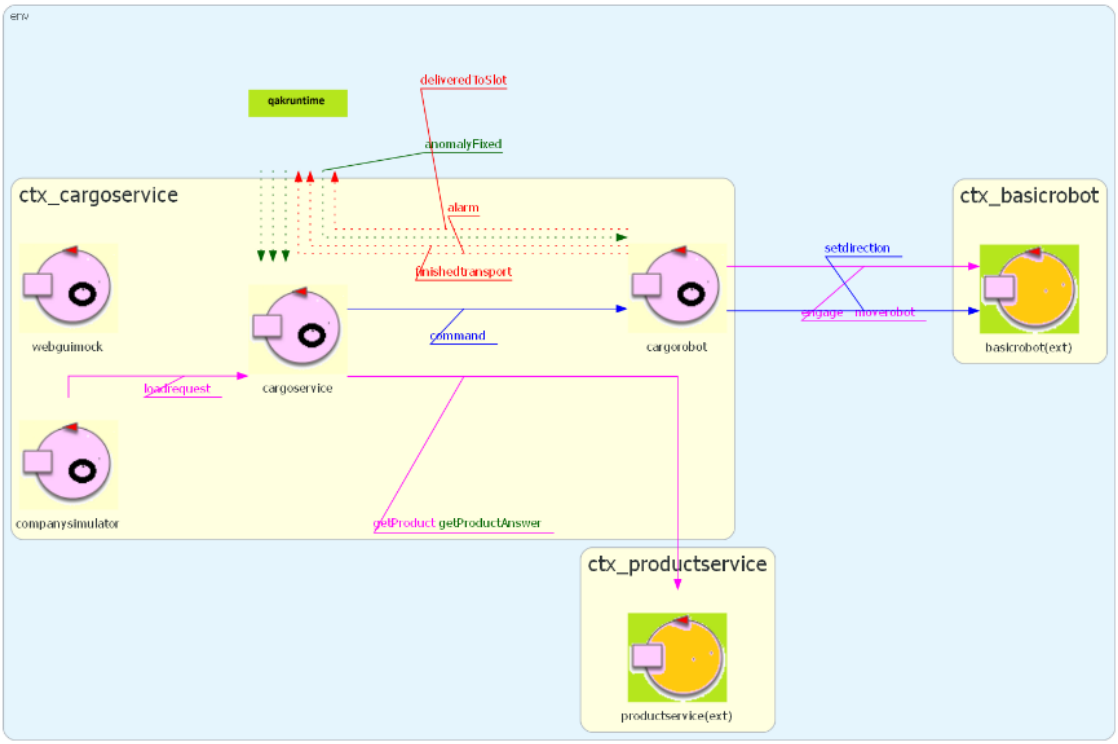
## SPRINT 3:

## Punto di Partenza:

Nello sprint2 il team ha implementato il componente sonarservice, sostituendolo al mock object corrispondente creato durante lo sprint1, ottenendo la seguente architettura:



Si vuole fornire per comprensione un vocabolario riassuntivo di termini definiti nei precedenti sprint:

## Vocabolario:

| Termine | Significato attribuito |
|---|---|
| Container | Contenitore in cui viene inserito il prodotto |
| loadrequest /richiesta di carico | Richiesta mandata dalla compagnia, specificando un PID |
| prodotto | Nel sistema è l'oggetto trasportato dal robot, la cui presenza può innescare diversi eventi |
| prodotto registrato | Prodotto conosciuto da ProductService a cui è associato un PID e un peso(Weight) |
| Microservizio | Componente software progettato per svolgere una specifica funzione del sistema. Ogni microservizio comunica con gli altri tramite messaggi, rendendo il sistema flessibile e scalabile. |
| GUI (Graphical User Interface) /WebGUI | Interfaccia grafica utente accessibile via web, che consente di visualizzare in tempo reale lo stato della stiva e interagire in modo intuitivo con il sistema. |
| Bounded Context | Il "bounded context" (**contesto limitato**) è un concetto fondamentale nel Domain-Driven Design (DDD) e si riferisce a un ambito applicativo ben definito e autonomo all'interno del quale vengono definite entità, regole e logiche di business in modo univoco e chiaro. All'interno di un bounded context, il significato di ogni entità o concetto è inequivocabile e specifico per quel contesto, evitando ambiguità e conflitti con altri contesti. |
| IOPort | Punto fisico (porta) attraverso il quale i contenitori dei prodotti entrano o escono dalla nave. È il punto in cui il sonar rileva la presenza di un prodotto. |
| Sonar | Sensore a ultrasuoni che misura la distanza tra sé e un oggetto. Nel nostro sistema serve per rilevare se un contenitore è presente all'IOPort. |
| DDRobot | è un robot che utilizza due motori indipendenti per muovere le ruote o i cingoli. È il supporto fisico che viene comandato da cargorobot. |
| PID (Product Identifier) | Numero intero univoco assegnato a ciascun prodotto registrato, usato per tracciarne l'identità all'interno del sistema. |
| Slot | Spazio fisico nella stiva della nave dove può essere posizionato un contenitore. Esistono 4 slot disponibili; uno è sempre occupato (slot5). |

| | |
|---|---|
| Cargorobot | Robot mobile autonomo (a guida differenziale) incaricato di trasportare i contenitori dall'IOPort fino allo slot assegnato e poi tornare alla posizione HOME. |
| Stiva | Area rettangolare della nave in cui i contenitori vengono caricati. Contiene gli slot e l'IOPort. |
| ProductService | Microservizio che gestisce la registrazione dei prodotti. Verifica i dati e assegna un PID univoco |
| CargoService | Microservizio che riceve richieste di carico, controlla i vincoli, assegna gli slot e coordina il caricamento tramite cargorobot. |
| SonarService | Microservizio che rileva la presenza di un contenitore all'IOPort tramite i dati forniti dal sonar. |
| DFREE | Distanza soglia usata dal sonar: se la distanza misurata è maggiore di DFREE per 3 secondi, si ipotizza un malfunzionamento del sensore. |
| MaxLoad | Peso massimo complessivo che la nave può sopportare. Il sistema rifiuta richieste che farebbero superare questo limite. |
| Worker | Persona che colloca fisicamente i contenitori sull'IOPort dopo che sono stati registrati. |
| Sistema logico di riferimento | Rappresentazione concettuale dell'intero sistema, con attori, componenti e interazioni, usata come base per l'architettura e la progettazione tecnica. |
| attore | Entità che svolge un ruolo attivo nel sistema, eseguendo azioni e comunicando con gli altri attori attraverso messaggi |
| Linguaggio QAK | Linguaggio modellistico usato per descrivere e simulare il comportamento dei componenti del sistema come "attori" |
| POJO | Plain Old Java Object: un oggetto di una classe in java |
| Anomalia | Nel documento è inteso come un comportamento inatteso di un componente hardware, tale da compromettere il normale funzionamento del sistema. |
| Refactoring | Modifica del codice per integrare nuove funzionalità |
| Framework | Un framework è una struttura predefinita o un insieme di codice già pronto all'uso che fornisce una base solida per lo sviluppo di applicazioni software |
| CoAP | Protocollo di comunicazione updater-observer |
| Mqtt | Protocollo di comunicazione publisher-subscriber |
| TCP | Tipo di connessione tramite rete (sicura) |

| | |
|---|---|
| UDP | Tipo di connessione tramite rete (best-effort) |
| Http | Protocollo di rete utilizzato dalle connessioni di rete internet (client-server) |
| Rest | |

## Goal dello Sprint3:

- Enunciazione esplicita dei requisiti della webgui
- Analisi del problema
  - Refactoring
  - WebGUI
- Definizione dell'architettura logica con modello eseguibile
- Progetto e realizzazione della webgui
- Deployment
- Tempo impiegato dal team

## Enunciazione esplicita dei requisiti della webgui:

Il sistema deve fornire un'interfaccia grafica accessibile via web (**webgui**) che consenta di monitorare lo stato della stiva (**hold**) in tempo reale. In particolare:

### RF1. Visualizzazione dello stato della stiva

- L'interfaccia deve mostrare lo stato corrente di ciascun **slot** (occupato/libero).
- Deve essere mostrato anche il **peso complessivo** del carico presente nella stiva.

### RF2. Aggiornamento dinamico

- La WebGUI deve aggiornarsi automaticamente al variare dello stato della stiva, senza necessità di ricaricare manualmente la pagina.
- L'aggiornamento avviene tramite notifiche push, ottenute osservando la risorsa CoAP hold e inoltrate ai client via WebSocket.

### RF3. Accessibilità via Web

- La WebGUI deve essere consultabile da un browser attraverso un endpoint HTTP dedicato.
- Deve essere disponibile almeno una pagina HTML responsiva che consenta all'utente di monitorare lo stato della stiva in maniera chiara e intuitiva.

## Analisi del problema:

L'obiettivo principale di questo sprint è la realizzazione della **WebGUI**, che consente agli utenti di monitorare in tempo reale lo stato della stiva.
Per raggiungere questo risultato è stato necessario **centralizzare le informazioni** relative agli slot e al peso complessivo del carico in un unico punto di riferimento.

Durante lo **Sprint 1**, la responsabilità della gestione dello stato della stiva era stata affidata direttamente al **CargoService**. Tuttavia, con l'introduzione della WebGUI, mantenere questa soluzione avrebbe reso il sistema eccessivamente complesso, poco mantenibile e difficile da estendere. CargoService, infatti, ha come funzione principale quella di **coordinare le operazioni**, non di gestire i dati.

Per questo motivo è stato introdotto, tramite refactoring, un nuovo componente dedicato: **Hold**, incaricato di mantenere e fornire lo stato aggiornato della stiva.

Inoltre, per rendere il progetto **più modulare** in prospettiva aziendale, è stato aggiunto un secondo componente: **CompanyRequestReceiver**. Questo attore si occupa della ricezione delle richieste provenienti dalla GUI della compagnia e del loro inoltro al CargoService, fungendo da punto di ingresso e da primo livello di filtraggio/interpretazione delle richieste.

## REFACTORING:

### Il modello Hold: Ruolo e responsabilità

Questo componente agisce come punto di riferimento centralizzato per lo stato della stiva. La sua responsabilità principale è mantenere aggiornate le seguenti informazioni:

- il **peso complessivo** degli elementi presenti nella stiva
- lo **stato (occupato/libero)** di ogni slot.

Questo design riflette il principio di separazione delle responsabilità, semplificando gli altri microservizi e rendendo le informazioni sulla stiva facilmente accessibili, migliorando la robustezza e la scalabilità del sistema.

### Interazione tra Hold e Cargoservice:

Con il **refactoring**, il *CargoService* non gestisce più i dati della stiva, ma delega a *hold* la responsabilità di valutare le richieste di carico. Questa collaborazione si basa su un preciso scambio di messaggi:

1. **Richiesta di valutazione:** *CargoService* invia a *Hold* una richiesta per verificare se è possibile caricare un prodotto, utilizzando il messaggio *checkIfFits.*

| ***Request checkIfFits : checkIfFits(PID, Weight)*** |
|---|

2. **Risposta di Hold:** *Hold* valuta la richiesta in base al peso del prodotto e alla disponibilità degli slot.

o La richiesta viene rifiutata se il peso supera il valore di MaxLoad o se la stiva risulta piena (assenza di slot disponibili).
o In caso contrario, viene assegnato uno slot al prodotto.

| | |
|---|---|
| ***Reply accepted: accepted(JsonString) for checkIfFits*** | La richiesta di carico viene accettata e viene mandata un oggetto Json in formato stringa, che contiene le informazioni riguardanti il prodotto e lo slot in cui deve essere posizionato |
| ***Reply refused: refused(Reason) for checkIfFits*** | La richiesta di carico viene rifiutata. |

### Hold e la WebGUI:

*Hold* è fondamentale per il funzionamento in tempo reale della WebGUI. Per garantire che l'interfaccia utente sia sempre aggiornata, Hold invia periodicamente il proprio stato alla WebGUI.

Questa comunicazione avviene tramite un messaggio di tipo Dispatch, che contiene un oggetto JSON che descrive lo stato attuale della stiva (slot occupati/liberi e peso totale).

| |
|---|
| ***Dispatch update : update(HoldJsonString)*** |

Mentre la Hold rimane in attesa di nuove richieste, manda il messaggio di update.

Nota: si era pensato di mandare un aggiornamento solo ad ogni consegna avvenuta, ma la natura del canale di comunicazione, fallace, rischierebbe di far perdere messaggi e di rendere inaffidabile l'interfaccia, pertanto, si è optato fare più aggiornamenti.

In questo modo, la WebGUI riceve le informazioni e aggiorna l'interfaccia in modo dinamico, senza bisogno di ricaricare la pagina.

### Comunicazione con la WebGui tramite CoAP:

Per la comunicazione tra Hold e la WebGUI, si è scelto **CoAP** (Constrained Application Protocol), uno dei protocolli più usati nell'Internet of Things. CoAP è basato su **REST** e utilizza il protocollo **UDP**, che, pur avendo una **alta probabilità di perdita dei messaggi**, consente una trasmissione dei dati rapida e adatta per aggiornamenti dinamici come quelli richiesti dalla WebGUI.

CoAP è un protocollo **request-response**:

- Il client invia richieste **GET** per osservare una risorsa.

- Il server risponde con notifiche che contengono il nuovo stato della risorsa, aggiornando così l'interfaccia senza la necessità di una comunicazione continua.

Questo lo rende ideale per scenari dove è fondamentale l'aggiornamento dinamico e tempestivo dei dati.

Difatti la Hold non manda il dispatch alla WebGui tramite TCP, ma lo fa tramite comunicazione CoAP:

```
Dispatch update : update(HoldJsonString)
```

Ovvero:

```
[#
                hold = HoldData()
                val HoldJsonString = hold.holdToJson()
        #]
        updateResource [#HoldJsonString#]
```

L'endpoint che aggiorna la risorsa diventa il server, mentre quello che la osserva il client.

Ogni volta che viene aggiornata la risorsa, la WebGUI riceve una notifica in automatico, senza dover inviare un dispatch.

### Il formato per il trasferimento di informazioni:

Per rendere più facile il trasferimento dei dati, si è optato per l'utilizzo del formato JSON (JavaScript Object Notation).

La *Hold* manda oggetti in formato JSON sotto forma di stringa. Tale formato è stato scelto come standard per lo scambio di dati perché è leggero e basato su testo, caratteristiche particolarmente adatte per l'architettura a microservizi.

Le principali caratteristiche sono:

- **Leggibilità**: la sua sintassi a coppie "chiave-valore" è intuitiva e facile da leggere sia per gli sviluppatori che per le macchine, semplificando il debug e la manutenzione.
- **Indipendenza dal linguaggio**: è un formato universale supportato nativamente dalla maggior parte dei linguaggi di programmazione moderni. Questo permette ai nostri microservizi, anche se scritti in linguaggi diversi, di comunicare senza problemi.
- **Efficienza**: La sua struttura concisa, che non richiede tag di chiusura come altri formati, si traduce in messaggi più piccoli e leggeri. Questo riduce la quantità di dati d1a trasmettere sulla rete, migliorando la velocità e l'efficienza della comunicazione tra i componenti.

**Modello dei dati HoldData.java:**

Il cuore del microservizio hold è la classe **HoldData.java**. Progettata come un **POJO** (Plain Old Java Object), questa classe incapsula lo stato della stiva e implementa la logica di business necessaria per gestirne i dati. In questo modo, l'entità hold funge da fonte centralizzata e coerente per tutti i componenti che necessitano di informazioni sul carico.

La classe HoldData gestisce due attributi principali:

- **s**: un'istanza della classe Slots che gestisce lo stato di ogni slot fisico della stiva.
- **Cur_HoldWeight**: un intero che rappresenta il peso totale del carico attualmente a bordo, aggiornato dinamicamente.

I metodi implementati in questa classe sono pensati per supportare le interazioni con il CargoService e la WebGUI:

| | |
|---|---|
| ***canLoad(int slotID, int productWeight)*** | esegue la **verifica dei vincoli** richiesta dal CargoService. Restituisce true solo se il peso del prodotto non supera il MaxLoad totale della nave e se lo slot specificato è valido, garantendo l'integrità del sistema |
| ***getAvailableSlot()*** | Fornisce il primo **Slot** disponibile per un nuovo carico |
| ***registerProductInSlot(Slot slot, int productWeight)*** | Aggiorna lo stato dello slot una volta che un prodotto è stato registrato e aggiorna il peso totale del carico (Cur_HoldWeight). |
| ***checkResultToJson(Slot slot, int pid, int weight):*** | Converte il risultato di una verifica di carico in un **oggetto JSON**. Questo metodo crea un messaggio standardizzato che Hold invia al CargoService per comunicare i dettagli dello slot assegnato. |
| ***String holdToJson()*** | Genera un **oggetto JSON completo** che rappresenta lo stato attuale dell'intera stiva. Questo formato è ottimizzato per la visualizzazione sulla WebGUI, includendo il peso totale del carico e lo stato di ogni slot, e facilita l'aggiornamento dinamico dell'interfaccia. |

## CompanyRequestReceiver:

Come introdotto nello **Sprint 0**, il sistema deve essere in grado di ricevere le richieste di carico (*load request*) provenienti dalla compagnia.
In quella fase, le richieste giungevano direttamente al **CargoService**, che si limitava a gestirne l'elaborazione. Con l'arrivo della WebGUI, però, si è reso necessario introdurre un attore intermedio capace di gestire in modo più flessibile le richieste, alleggerendo CargoService e rendendo il sistema più estendibile.

Per questo motivo, durante il refactoring è stato introdotto l'attore **CompanyRequestReceiver**, con il compito di fungere da **punto di ingresso ufficiale** delle richieste provenienti dalla compagnia.

Grazie a questa scelta architetturale, il sistema ha guadagnato nuove funzionalità che prima non erano presenti:

- attesa e ricezione delle richieste inviate dalla compagnia;
- verifica preliminare della disponibilità del sistema (accettazione immediata o rifiuto);
- inoltro delle richieste accettate al CargoService e gestione della risposta;
- invio di un riscontro immediato alla GUI;

In questo modo, CargoService rimane focalizzato sul proprio ruolo di coordinatore delle operazioni, mentre CompanyRequestReceiver gestisce l'interazione con l'esterno e fornisce un primo livello di filtraggio e interpretazione delle richieste.

## WEBGUI:

Con l'introduzione della WebGUI si è presentata l'esigenza di rendere accessibile in tempo reale lo stato della stiva anche a un utente esterno, in modo chiaro e immediato.
In particolare, la WebGUI deve permettere di:

- visualizzare lo stato corrente degli slot (occupati/liberi);
- monitorare il peso complessivo del carico presente nella stiva;
- ricevere aggiornamenti dinamici senza richiedere un intervento manuale (refresh).

In precedenza, questa funzionalità era solo simulata tramite l'attore *webgui_mock*, con CargoService incaricato di fornire i dati. Tale soluzione non era più sostenibile: accresceva la complessità del CargoService, che già aveva il compito principale di coordinare le operazioni.

Per questo motivo, con il refactoring si è reso necessario delegare la responsabilità della comunicazione con la WebGUI al nuovo attore Hold, che diventa il punto di riferimento unico per lo stato della stiva.

Sebbene non sia stato richiesto dai requisiti del committente, si è deciso di introdurre una **componente aggiuntiva** per facilitare il debug e rendere più agevole la fruizione del sistema,

ovvero, un'interfaccia interattiva che consente all'utente di **inviare richieste di carico** al sistema.

Questa componente deve permettere all'utente di:

- inserire il valore del PID del prodotto che vuole richiedere
- inviare la richiesta al sistema
- visualizzare lo stato della richiesta

## Progettazione del WebGUIModel:

**WebGUIModel** rappresenta il comportamento della WebGUI del sistema senza utilizzare interfacce grafiche reali. In questa fase, la GUI è simulata tramite la stampa a video di messaggi, aggiornamenti e stati, permettendo di verificare la logica di interazione con gli altri componenti del sistema.

### Ciclo di vita

1. **Avvio del sistema**
    - L'attore webgui si avvia e inizia ad osservare il componente hold come risorsa CoAP
    - L'attore webgui si mette in attesa degli update provenienti da cargoservice
2. **Ricezione dell'aggiornamento da cargoservice:**
    - Ogni volta che cargoservice aggiorna le sue risorse attraverso il Dispatch 'update', il messaggio viene ricevuto e filtrato in automatico dall'attore webgui, che interpreta il formato JSON.
    - Il risultato dell'elaborazione viene propagato sotto forma di evento filteredupdate(Update) a tutti gli attori interni, così da mantenere aggiornata la simulazione della GUI.
3. **Ricezione dell'aggiornamento dalle pagine:**
    - holdshowpage riceve le informazioni filtrate e aggiorna la sua rappresentazione (in formato di stringa in questo caso)

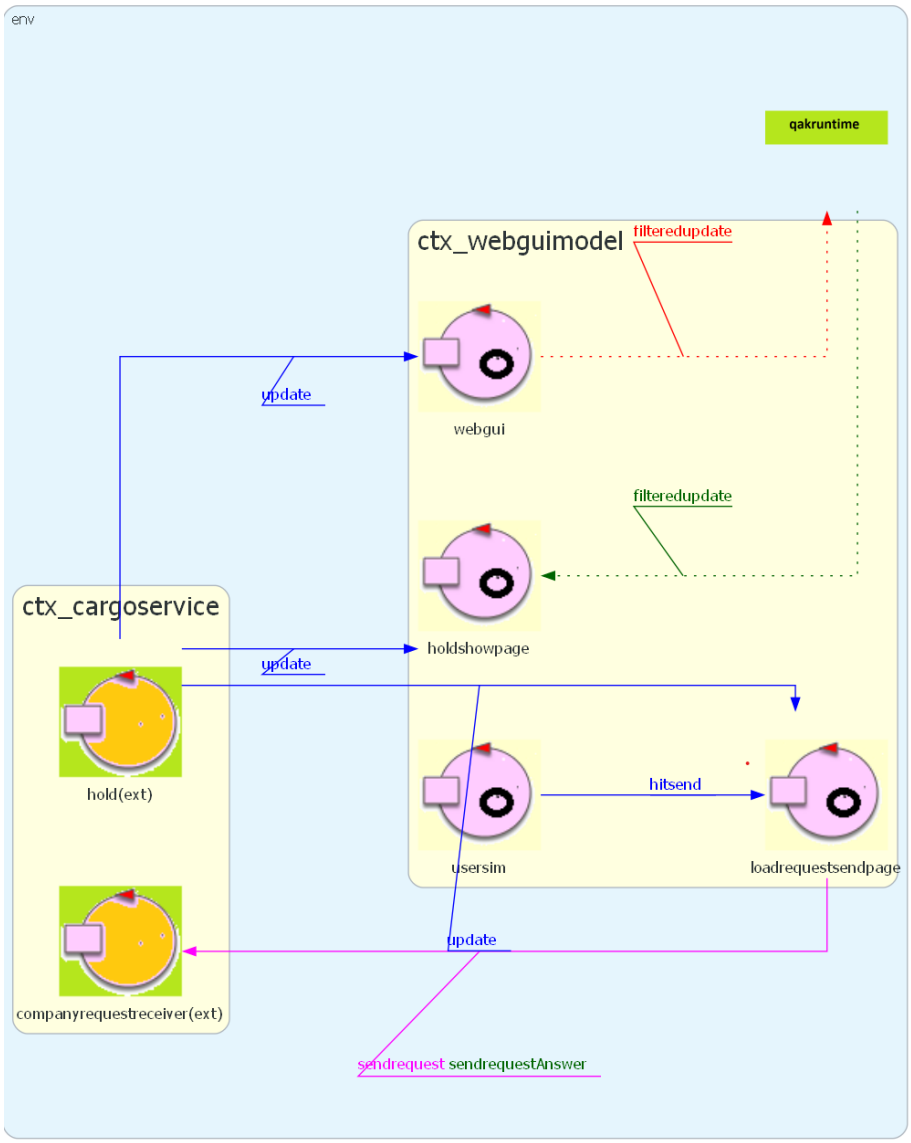
### Funzionalità:

#### Interazione utente simulata

- L’attore usersim rappresenta un utente che, a intervalli regolari, genera richieste di carico (hitsend(PID)).
- L’attore loadrequestsendpage intercetta il comando e invia la richiesta al companyrequestreceiver di cargoservice tramite sendrequest(PID).

### Gestione delle risposte

- cargoservice risponde immediatamente con sendrequestAnswer(Answ), che viene mostrato a video da loadrequestsendpage.
- L’utente (simulato) può così vedere se la richiesta è stata presa in carico o se il sistema era occupato.



# Progettazione della WebGUI:

## La WebGUI(Hold):

è il componente responsabile della presentazione dello stato della stiva (hold), che include il monitoraggio degli slot e del peso totale trasportato. In precedenza, un attore webgui_mock era utilizzato per emulare questa funzionalità, con **cargoservice** responsabile della comunicazione. Dopo il refactoring, si è deciso che gli aggiornamenti arrivino direttamente dall'attore **Hold** descritto prima.

La WebGui funziona come segue:

Possiamo considerarlo un sistema **passivo**, dato che riceve soltanto informazioni.

L'interfaccia deve essere user-friendly ed intuitiva, deve quindi mostrare gli slot in un modo che renda facilmente comprensibile all'utente se sono occupati oppure no.

## La WebGUI(Request_Receiver):

Componente interattiva in cui l'utente può inserire una richiesta di carico.

Negli sprint precedenti le richieste di carico venivano inviate da un simulatore, companysimulator.

Dallo sprint0 si sa che la compagnia possiede una webgui attraverso cui mandare le richieste di carico. Si è voluta quindi dare un'interfaccia che permettesse all'utente di inserire un numero che corrisponde al PID e cliccare *SEND*, facendo arrivare la richiesta al cargoservice.

# Implementazione della WebGUI:

La webgui è stata progettata utilizzando SpringBoot, un framework che semplifica lo sviluppo di applicazioni web e microservizi in Java. Offre una serie di funzionalità integrate, come il supporto per la configurazione automatica, la gestione delle dipendenze e l'integrazione con vari sistemi di backend, il che permette di sviluppare rapidamente un'applicazione scalabile e facilmente mantenibile.

Le componenti che permettono la comunicazione della WebGUI con il cargoservice sono:

- **WsHandler.java**: gestore delle connessioni con il client browser.
- **CoapToWS.java**: il client CoAP che ottiene i messaggi.
- **CallerService.java:** manda i messaggi provenienti dalla WebGUI(http) attraverso un canale TCP, per farli giungere a cargoservice.

## WSHandler.java:

è il componente di comunicazione WebSocket che si occupa delle connessioni con i client browser.

Gestisce le connessioni WebSocket in entrata, permettendo l'invio di messaggi a tutte le sessioni connesse.

**responsabilità:**

- **tiene traccia delle sessioni connesse**: ogni volta che un nuovo client si connette, viene aggiunto alla lista delle sessioni, e quando una sessione si disconnette, viene rimossa.
- **fornisce il metodo *sendToAll:*** il metodo *sendToAll* è utilizzato per inviare i messaggi ricevuti da CoAP a tutte le sessioni WebSocket attive, assicurando che tutte le WebGUI connesse ricevano gli aggiornamenti in tempo reale.

**WebSocketConfig.java:**

Registra il WSHandler su un endpoint specifico, in questo caso "/status-updates"

## CoapToWS.java:

Fa da ponte tra il protocollo CoAP e WebSocket.

Corrisponde all'ObserveResource in qak.

Si sottoscrive come osservatore dello stato di hold accessibile. Ogni volta che hold viene aggiornata, la WebSocket riceverà un messaggio.

**responsabilità:**

- **monitora il cambiamento dello stato della stiva tramite CoAP (observeResource):** quando il client riceve una risposta dal server CoAP, il contenuto viene parsato (convertito in un oggetto JSON) e inviato a tutte le sessioni WebSocket attive tramite il componente WSHandler.

La comunicazione tra CoAP e WebSocket consente alla WebGUI di ricevere aggiornamenti in tempo reale senza necessità di ricaricare la pagina.

**Parsing**:

I messaggi che arrivano alla WebSocket tramite Coap sono in forma JSON.

La classe HoldResponseParser.java fornisce un metodo **parseHoldState(String message)** che interpreta i messaggi ricevuti e li restituisce in formati interpretabili.

## Interfaccia web:

Per il monitoraggio della stiva è stata sviluppata un'interfaccia **web responsiva** basata su HTML, CSS e JavaScript.
La pagina fornisce una rappresentazione chiara e immediata dello stato del sistema, mostrando:

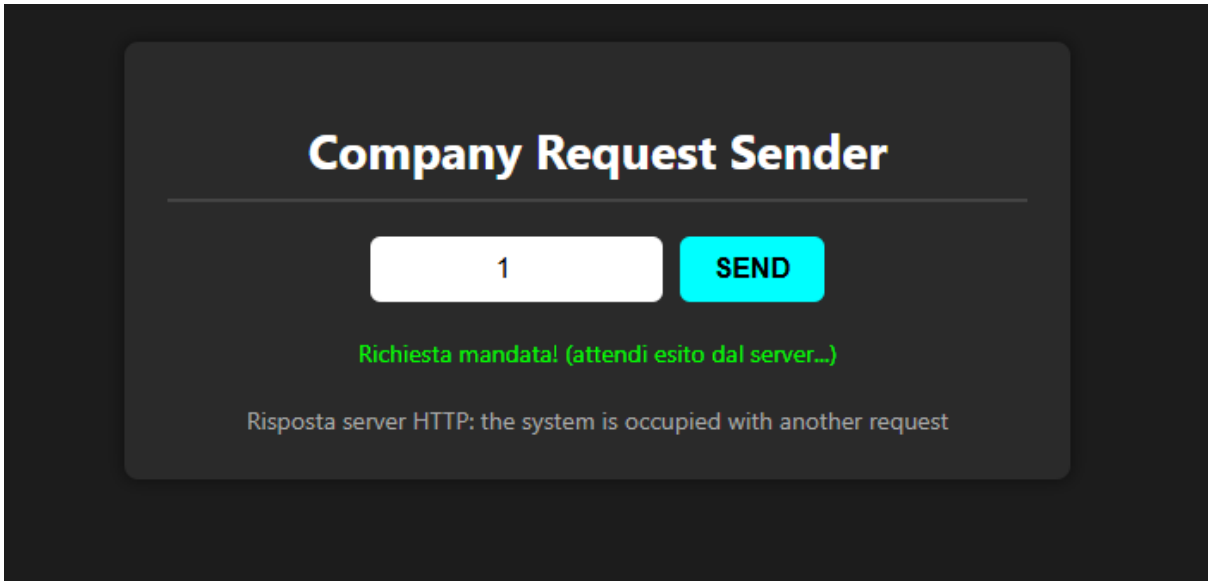
- il **peso totale del carico a bordo**, aggiornato in tempo reale;
- lo **stato dei quattro slot del deposito**, distinti come liberi o occupati;
- un **collegamento automatico con l'endpoint /status-updates**, che permette di ricevere notifiche push sugli eventi;
- un **meccanismo di refresh dinamico** dell'interfaccia al verificarsi di variazioni, senza dover ricaricare la pagina.

Al primo accesso, il browser si connette alla websocket, in modo da poter iniziare a ricevere gli aggiornamenti da parte del server.


## Progettazione per l'invio delle richieste:

### RequestSender:

è stata aggiunta un'interfaccia grafica anche per le richieste.




L'utente può inserire solo valori numerici e cliccare invio.

L'interfaccia segnala se la richiesta è stata mandata correttamente al sistema cargoservice o se quest'ultimo è già occupato e pertanto non può prenderne in carico un'altra.

## CallerService.java:

Quando l'utente clicca *SEND*, cerca in realtà di mandare la richiesta di carico al sistema cargoservice.

Pertanto, è necessario un punto di accesso che permetta alla WebGUI a mandare richieste tramite TCP.

CallerService gestisce quindi le richieste HTTP che provengono dalla WebGUI.

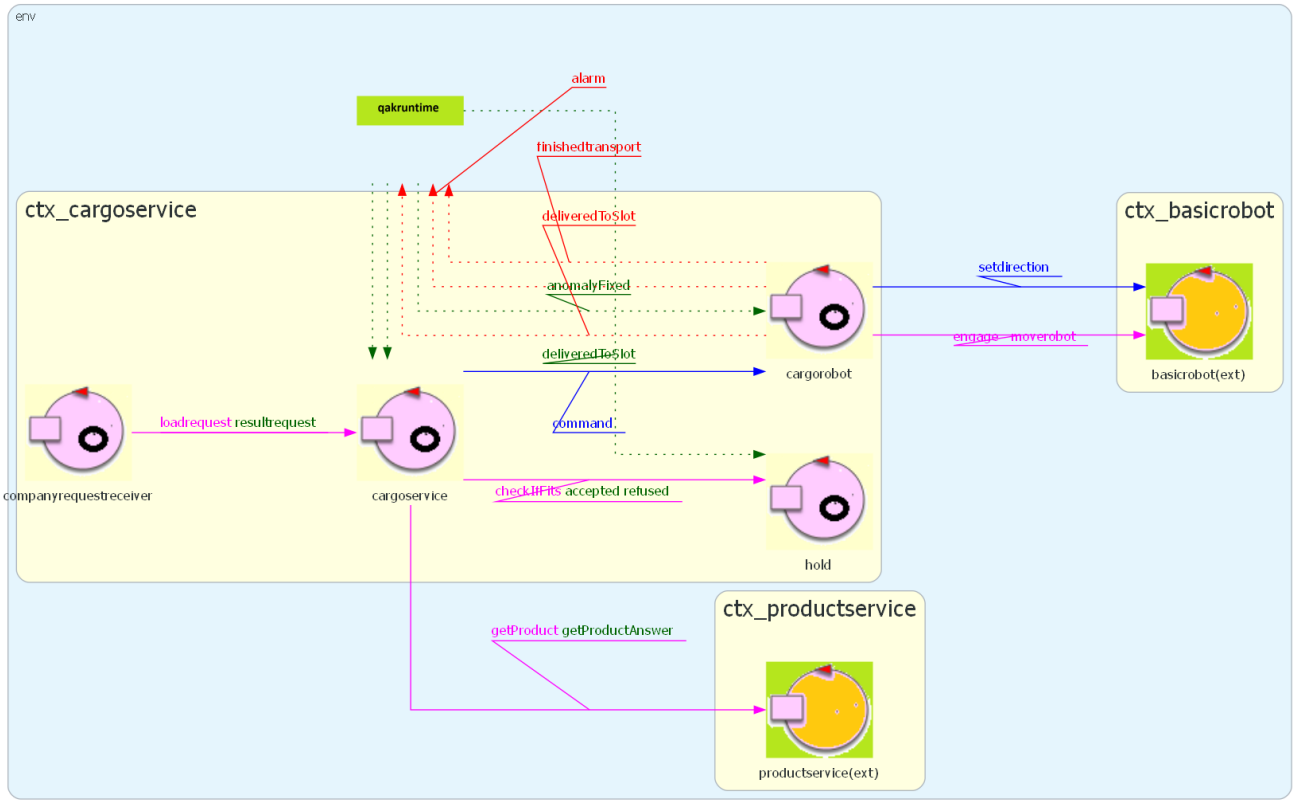
Quando arriva una richiesta dall'interfaccia (l'utente ha cliccato il pulsante *SEND*), il controller HTTP invoca un metodo che invia una richiesta al sistema *cargoservice* tramite una connessione TCP.

Una volta giunta al sistema, l'attore CompanyRequestReceiver se ne occupa.

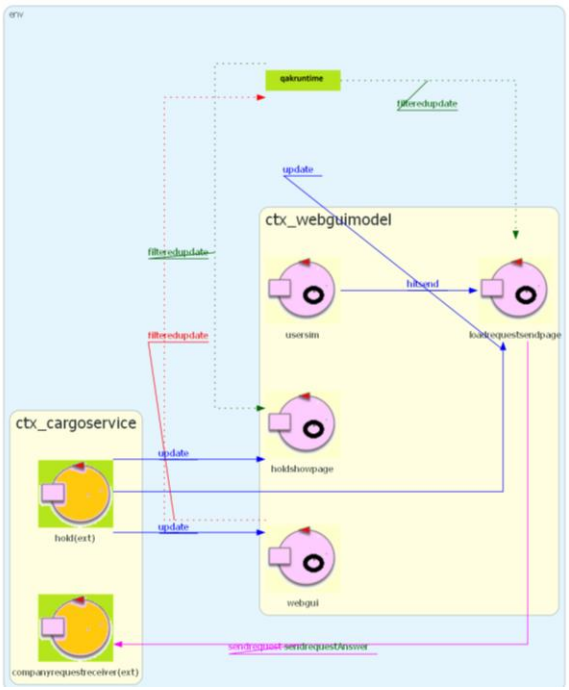
## Definizione dell’architettura logica con modello eseguibile:

Al termine dell’analisi e della progettazione si è giunti al nuovo modello eseguibile:

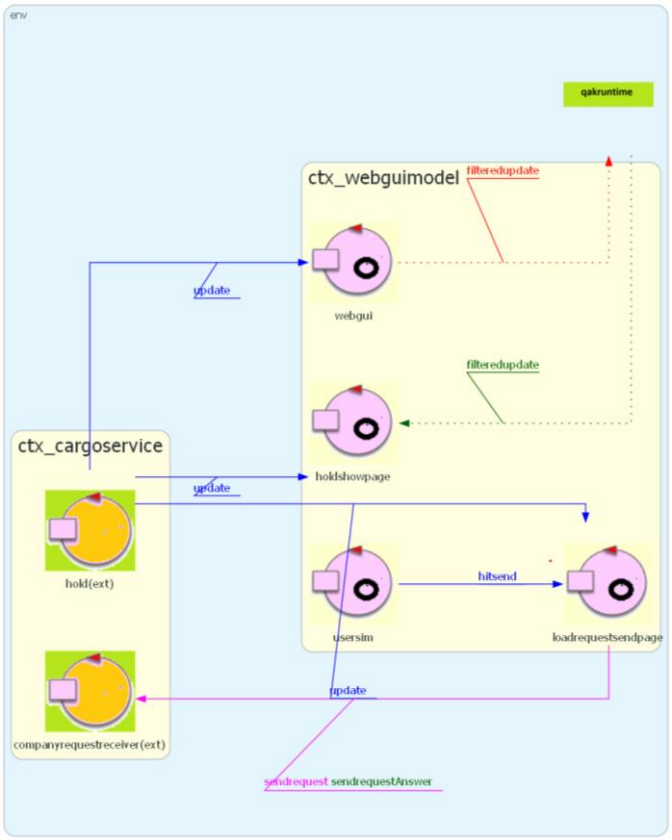
### cargoservice:




### sonarservice:



### webguimodel:




## Deployment:

I modelli qak sviluppati sono reperibili sulla seguente repository github, nella cartella sprint3:

- cargoservice: componente principale
- sonarservice: microservizio per raspberry device con connessi i dispositivi sonar e led
- webguimodel: prototipo privo di interfacce della webgui
- webgui: componente con interfacce

## Istruzioni:

### cargoservice e webgui :

questi due servizi sono stati entrambi dockerizzati per renderli più fruibili su più piattaforme.

Si presenta quindi la seguente guida per il loro funzionamento:

0) fare il download del progetto dalla repository git:
   **https://github.com/SvevaNullBologna/CargoService.git**

1) aprire il progetto cargoservice nella cartella sprint3/cargoservicefordocker e il progetto webgui nella cartella sprint3/webguifordocker su Eclipse

**Creazione dell'immagine docker di cargoservice:**

2) dopo aver fatto click sul nome del progetto sul package explorer, pigiare i tasti ctrl + alt + T per aprire il terminale Eclipse sulla directory principale del progetto
3) eseguire il comando **gradlew distTar**. Verrà generato un file .tar nella directory cargoservice/build/distributions
4) Sempre rimanendo nella directory principale *cargoservice* , eseguire il comando **docker build -t cargoservice:1.0 .**

**Creazione dell'immagine docker di webgui:**

5) dopo aver fatto click sul nome del progetto sul package explorer, pigiare i tasti ctrl + alt + T per aprire il terminale Eclipse sulla directory principale del progetto
6) eseguire il comando **gradlew distTar**. Verrà generato un file .tar nella directory webgui/build/distributions
7) Sempre rimanendo nella directory principale *webgui* , eseguire il comando **docker build -t webgui:1.0**

**Caricamento container completo:**

8) Da terminale, spostarsi nella cartella del progetto CargoService/sprint3
9) Eseguire il comando: **docker compose -f services.yaml up**
10) Visualizzare su Docker le immagini (programma consigliato: DockerDesktop)
11) Far partire in caso di problemi quelle non attive (si consiglia di avviare cargoservice e webgui per ultimi in quest'ordine).

**Riempimento del database mongodb:**

è obbligatoria l'installazione di Node.js.

è obbligatorio aver avviato il servizio docker mongodb.

12) Aprire il terminale postarsi nella directory Cargoservice/sprint3

13*) Opzionale: modificare il file mongosetup per aggiungere o eliminare prodotti

13) Eseguire da cmd il comando **node mongosetup.js**

**sonarservice:**

sonarservice è l'unico servizio non posto su container Docker, poiché è ideato per funzionare direttamente sul dispositivo raspberry.

Per poter utilizzare il servizio, è consigliato seguire questa guida:

0)fare il download del progetto dalla repository git:
**https://github.com/SvevaNullBologna/CargoService.git**
1) aprire il progetto sonarservice nella cartella **sprint3/sonarserviceforraspberry** su Eclipse

2) dopo aver fatto click sul nome del progetto sul package explorer, pigiare i tasti ctrl + alt + T per aprire il terminale Eclipse sulla directory principale del progetto

3) eseguire il comando **gradlew distZip**

4) spostarsi sulla cartella sonarservice/build/distributions. Al suo interno sarà presente la zip generata.

5) trasferire il file zip sul raspberry (metodo consigliato: programma fileZilla)

6)sul terminale del raspberry (metodo consigliato per connessioni headless: PuTTY) portarsi sulla directory dove è presente il file zip

7)eseguire il comando **unzip sonarservice-1.0.zip**

8)spostarsi sulla cartella generata sonarservice-1.0/bin ed eseguire il comando **./sonarservice-1.0**

Attenzione: è consigliato aver fatto già partire il server mosquittoalone su Docker

## Tempo impiegato dal team:

Il team ha impiegato più tempo rispetto a quanto inizialmente previsto, principalmente a causa di problemi tecnici: malfunzionamenti di Eclipse, regole del firewall del dispositivo di lavoro troppo restrittive e, infine, la sostituzione forzata del dispositivo stesso a seguito di un guasto.
Tali problematiche, tuttavia, non possono essere considerate come tempo effettivo di lavoro, che risulta comunque superiore rispetto alle stime iniziali.

Come previsto, la parte più impegnativa è stata la progettazione del microservizio core, ovvero *cargoservice*. Il lavoro sul *sonarservice* ha richiesto effettivamente meno tempo, in linea con le aspettative, mentre la *webgui* ha comportato un impegno maggiore del previsto a causa di attività di refactoring non considerate in fase di pianificazione.

## Il team:

### Silvia Angela Sveva Carollo