Sprint 3

Sommario

Obiettivi

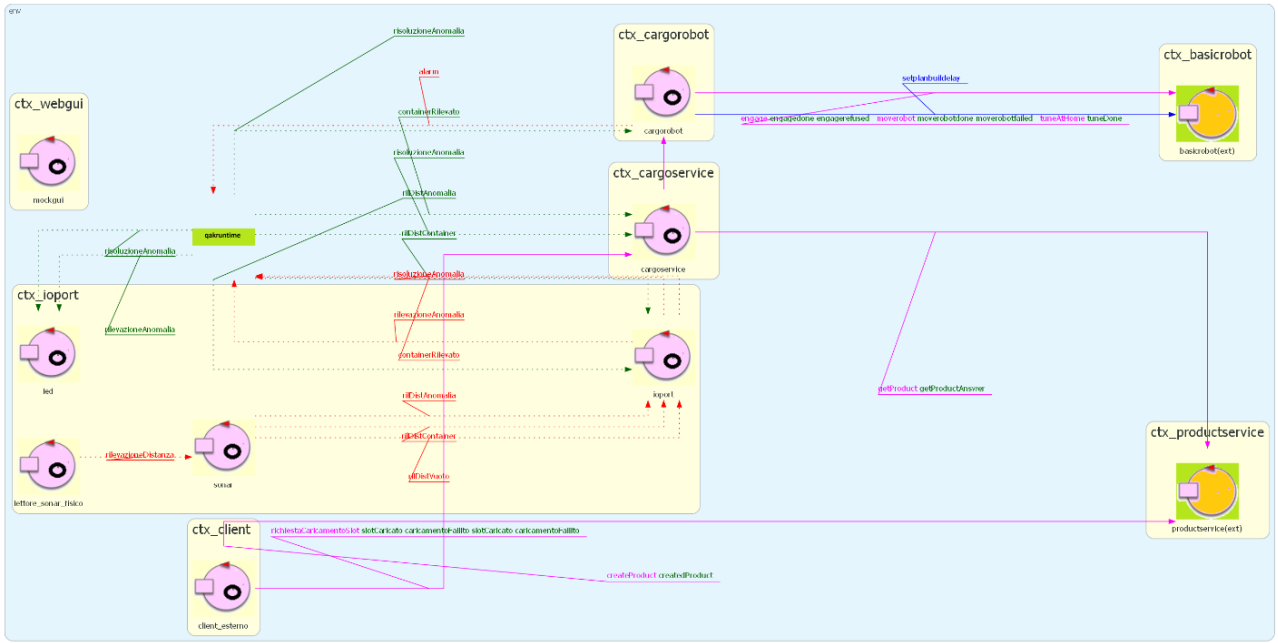
Sottoinsieme di requisiti considerati

In questo ultimo sprint l'unico requisito mancante che verrà preso in considerazione è l'unica richiesta del cliente inerente a un'interfaccia grafica web:

- È necessario che il cargoservice mostri lo stato della stiva tramite una web-gui aggiornata dinamicamente.

Si sottolinea il fatto che già il cliente ha legato questa funzionalità a cargoservice (punto 4 della parte dei requisiti relativi a cargoservice)

Architettura dello sprint 2



Analisi del Problema

La richiesta del cliente in questo ultimo requisito minimale è abbastanza semplice e chiara nelle funzionalità che il sistema deve raggiungere: serve che l'utente finale possa accedere a una web GUI la cui sola funzionalità richiesta è quella di mostrare lo stato della stiva. Si nota che, fino ad ora, l'unico componente software dell'intero sistema a conoscenza dello stato della stiva è *cargoservice*, che quindi evidentemente dovrà permettere in qualche modo di fare fluire questa informazione fino alla GUI.

Nonostante la richiesta sia semplice, essa porta a dover risolvere alcuni problemi e a dover fare una scelta consapevole tra le varie e valide possibilità

Si notano in primo luogo i seguenti problemi e punti di scelta:

- Cosa si intende precisamente per "stato della stiva"? Qual è il modo più conveniente per "condividerlo"?

- Come farà *cargoservice* a comunicare lo stato della stiva alla GUI? È necessario qualche intermediario?
- Che protocolli è possibile utilizzare in maniera semplice per raggiungere gli obiettivi di comunicazione dei dati?
- Dato che lo stato della stiva in *cargoservice* è per ora stato salvato come un POJO, sarà necessario un altro formato per poterlo utilizzare nella web-gui? Nel caso quale?
- Quali tecnologie permettono di realizzare la web-gui nel miglior modo possibile considerando vincoli, costi e risultato?
- Come fare affinché la gui venga aggiornata in tempo reale a fronte di un cambiamento nello stato della stiva?
- La web-gui potrebbe essere espansa con ulteriori funzionalità in futuro (es. funzionalità di input, altri dati da visualizzare…)? Nel caso quale potrebbe essere la migliore soluzione per rendere questo pezzo del sistema facilmente espandibile?

Scelte forzate: da requisiti, la richiesta di realizzazione di una gui adoperando tecnologie web implica automaticamente la necessità di prevedere nell'architettura un **server web** e che l'implementazione di tutte le funzionalità inerenti all'interfaccia web svolte direttamente sulla gui sia effettuata tramite linguaggio **JavaScript**. Il primo punto è obbligatorio in quanto, se si vuole recuperare una pagina web sulla rete, è indispensabile un server web; il secondo potrebbe avere delle alternative, ma al giorno d'oggi JavaScript è lo standard di fatto per quanto riguarda la programmazione web lato client, e per questo non ci si sforza neanche a valutare altre possibilità implementative (che spesso risultano anche obsolete). La scelta della modalità di utilizzo di JavaScript verrà analizzata successivamente (vanilla, utilizzo di un framework, librerie particolari…)

Stato della stiva: Negli sprint precedenti, per "stato della stiva" si intendeva in sostanza l'associazione slot-prodotto caricato. È bene soffermarsi a ragionare se ciò che viene indicato in questo requisito come stato della stiva sia solamente quello appena descritto o possa riferirsi anche ad altro. In questo caso, la locuzione indica i dati da mostrare nella gui; sicuramente tra questi vi sarà, per l'appunto, l'associazione slot-prodotto appena citata. Si nota anche come possa essere utile mostrare il carico massimo accettabile dalla stiva in quanto questo potrebbe essere una buona indicazione per l'utente finale ai fini del controllo della stiva (ciò si rende necessario in quanto il parametro precedentemente denominato MAX_LOAD è stato realizzato negli sprint precedenti come un parametro impostabile in fase di deployment e quindi potenzialmente suscettibile a cambiamenti, anche se minimi)

Altri dati che il cliente potrebbe voler visualizzare potrebbero essere lo stato di caricamento del container o lo stato di allarme; tali informazioni sono però logicamente legate allo stato di lavoro del cargorobot o agli eventi scatenati dall'IOPort, perciò non sembra ricadano precisamente sotto la definizione di "stato della stiva". In conseguenza di ciò, data la loro presunta inessenzialità per il cliente, per ora non verranno considerate, ma l'aggiunta di queste informazioni potrà avvenire in versioni future a fronte della richiesta del cliente.

A prescindere dalle informazioni inserite come "stato della stiva", quest'ultimo non sembra avere la forma di un messaggio o di una comunicazione generica, ma, piuttosto, è modellabile, in maniera logicamente più coerente, come una **risorsa** accessibile anche dalle entità che comporranno la gui.

Cargoservice, stato della stiva e QAK: Il cliente lega la funzionalità di visualizzazione della stiva a *cargoservice*. Si ricorda che, dagli sprint precedenti, *cargoservice* è stato sviluppato mediante il linguaggio QAK. Tale linguaggio (come è possibile notare in modo immediato) **non offre la possibilità di attivare un server web integrato** con il DSL. La richiesta d'implementazione di questa funzionalità aggiuntiva al team di sviluppo di QAK comporterebbe troppo tempo rispetto alla consegna di questo progetto (potrebbe comunque essere una funzionalità utile per progetti futuri). Detto ciò, si figura la necessità di un **server web indipendente da QAK** (e quindi da *cargoservice*): i dati sullo stato della stiva**,** quindi, **dovranno essere comunicati da *cargoservice*** in modo tale che possano essere recepiti da un componente esterno all'attore stesso.

Osservando come lo stato della stiva è stato modellato e implementato in *cargoservice,* si può notare come le informazioni principali siano completamente reperibili dall'oggetto POJO di tipo SlotMap. Questo rappresenta l'associazione slot-prodotto, vale a dire le informazioni principali che sarà necessario mostrare; quindi, in primo luogo, sarà sufficiente comunicare queste informazioni alla gui per ottenere il risultato. Guardando i POJO, si osserva come sia Slot sia Product abbiano dei metodi per serializzare l'oggetto in una stringa JSON: ciò rende semplice estendere questa rappresentazione anche per SlotMap, poiché esso è poco più che un'aggregazione di oggetti di questi due tipi. La scelta di rappresentare SlotMap come stringa JSON ha molteplici vantaggi:

- Semplicità realizzativa (come appena spiegato)
- Il fatto che sia una stringa permette generalmente di essere compatibile con qualunque protocollo di comunicazione che comunica tramite stringhe
- È un formato molto diffuso e estremamente comodo da manipolare una volta recepito dalla gui tramite JavaScript

Possono esistere altre soluzioni sia di tipo stringa (es. XML) sia in altri formati più complessi, ma si ritiene come più semplice ed efficace quella di **serializzare i dati in formato JSON** e utilizzare tale formato per le comunicazioni con entità esterne al QAK.

In questo modo è anche possibile aggiungere MAX_LOAD alla stringa json in maniera relativamente semplice.

Protocollo per la comunicazione dei dati al di fuori di Cargoservice: Dal punto precedente si evince che sarà necessaria una comunicazione tra attori QAK e mondo esterno, dove con "mondo esterno" si indicano tutti i componenti esterni a QAK che realizzeranno la web gui. Per tale comunicazione, innanzitutto, è conveniente sfruttare un protocollo standard: una soluzione ad hoc realizzata direttamente a basso livello risulterebbe inutilmente laboriosa per le funzionalità da implementare e nettamente meno scalabile. Perciò, come prime possibilità, si analizzano i protocolli offerti e integrati direttamente dal linguaggio QAK, cioè MQTT e CoAP. Come già evidenziato precedentemente, lo stato della stiva può essere visto come una risorsa accessibile dalla gui, perciò sembrerebbe poter essere modellabile in maniera più conveniente attraverso il protocollo CoAP, il quale è un protocollo orientato alle risorse basato su un modello RESTful. MQTT, di contro, è più orientato a messaggi di tipo publish/subscribe, il che implica non solo l'invio esplicito di un messaggio ogni qual volta lo stato della stiva viene aggiornato, ma anche quando viene recuperato da un qualunque cliente; e, inoltre, necessita di un server broker aggiuntivo che aumenta la complessità del sistema e la velocità di comunicazione. Una terza possibilità (non fornita da QAK) è quella di utilizzare una soluzione RESTful basata su HTTP, come, ad esempio, una REST API che offre tutti i vantaggi di modellazione orientata alle risorse. Altre possibilità non incluse nativamente in QAK non vengono analizzate nello specifico in quanto non si ritengono rilevanti.

In questo momento, le soluzioni RESTful si ritengono più adatte allo scopo con una netta propensione verso l'utilizzo di CoAP, poiché già gestito nativamente da QAK, fatto che porterebbe a minori costi e tempi di sviluppo.

Aggiornamento automatico della web-gui: Uno degli obiettivi principali dello *sprint 3* è la realizzazione di una Web-GUI che mostri in tempo reale lo stato del sistema, in particolare la situazione della stiva (occupazione degli slot). Per garantire un aggiornamento dinamico e coerente dell'interfaccia, è necessario analizzare le modalità con cui la gui riceverà le informazioni dai servizi QAK, in particolare dal **cargoservice.**

In prima analisi si individuano due strategie possibili per la comunicazione:

1. Aggiornamento **tramite protocollo** CoAP.
2. Aggiornamento **tramite meccanismo push**, ad esempio mediante **WebSocket**.

L'approccio basato su ***CoAP*** ha il vantaggio di essere direttamente compatibile con il meccanismo con cui cargoservice renderà disponibile la risorsa: esso, infatti, supporta il meccanismo di *observe*, che consente a un client di iscriversi a una risorsa e ricevere notifiche automatiche in caso di variazioni. Tale soluzione mantiene un'architettura omogenea con il resto del sistema, sfruttando un protocollo leggero e riducendo l'overhead rispetto all'utilizzo di WebSocket: si potrebbe, difatti, pensare di effettuare la subscribe alla risorsa direttamente con il client finale (la gui); tuttavia, sfortunatamente, l'interazione diretta tra CoAP e un'applicazione web lato client non è supportata per varie ragioni tecniche. Quest'ultima considerazione porta la soluzione che utilizza il solo CoAP a non essere applicabile a una web-gui, perché sarebbe indispensabile un intermediario (tale soluzione potrebbe essere tenuta in considerazione in caso di sviluppo futuro di una gui non web).

L'alternativa è l'utilizzo di ***WebSocket***, protocollo pienamente supportato dai browser e particolarmente adatto per la visualizzazione in tempo reale. In questo caso, il cargoservice (o un microservizio dedicato) mantiene una connessione persistente con la GUI, inviando eventi ogni volta che cambia lo stato della stiva o si verificano anomalie. Scegliendo di operare in questo modo, si offre una maggiore immediatezza e compatibilità con gli strumenti web, al prezzo di un carico leggermente superiore lato server; ma, in tal caso, è possibile prevedere un server web che mantenga la subscribe a CoAP e, una volta ricevuti gli aggiornamenti, li comunichi alla gui tramite WebSocket.

Non si prende in considerazione la possibilità di realizzare gli aggiornamenti tramite polling: esso risulta un meccanismo nettamente meno efficiente e comunque non compatibile con CoAP, cosa che necessiterebbe l'impiego di maggiore tempo per lo sviluppo di una soluzione basata su HTTP.

L'approccio con WebSocket può essere anche declinato per lavorare con http REST API anziché CoAP, ma questa soluzione si ritiene meno preferibile per i motivi indicati nel punto precedente.

Eventuali estensioni future:

In fase di analisi, è opportuno valutare anche la eventualità di estensioni future del sistema, e in particolare nel caso in cui si voglia consentire all'interfaccia non solo di visualizzare lo stato della stiva, ma anche di **interagire attivamente con il sistema:** ad esempio, la GUI potrebbe inviare richieste di caricamento al *cargoservice*, operazioni verso il *productservice* oppure simulare eventi utili per fini di test e monitoraggio.

In questo contesto si apre la questione relativa alla **tecnologia di comunicazione** più adatta per garantire compatibilità, semplicità di integrazione e coerenza con l'architettura QAK esistente. Poiché il linguaggio QAK e il relativo runtime sono basati su **Java/Kotlin**, una prima soluzione naturale sarebbe quella di utilizzare un **middleware Java-based**, ad esempio un microservizio basato su Spring, in grado di fungere da ponte tra la GUI e gli attori QAK, sfruttando quello che è anche stato definito a lezione "principio di IronMan". Tale approccio garantisce una semplice integrazione, dovuta anche alle librerie basiccomm, e semplifica l'invio di messaggi direttamente nel contesto QAK.

Un'alternativa più flessibile è l'adozione di protocolli di messaggistica come **MQTT**. In questo scenario, i servizi QAK pubblicano e sottoscrivono messaggi su specifici topic MQTT, mentre la GUI – tramite un client MQTT – può interagire con essi senza dipendere da un layer Java intermedio. Tale soluzione renderebbe il sistema più indipendente dal linguaggio e favorirebbe l'estensione verso altri componenti scritti in linguaggi diversi (Python, Node.js, ecc.), semplificando anche l'integrazione con eventuali dashboard esterne o sistemi di monitoraggio.

In prospettiva, quindi, l'adozione di un **adapter** tra QAK e protocolli più diffusi (HTTP, MQTT o WebSocket) rappresenta la scelta più scalabile: nel breve termine si può mantenere un'integrazione Java nativa per la semplicità di sviluppo, ma prevedendo sin da ora le interfacce in modo disaccoppiato, così da poter in futuro sostituire la comunicazione diretta con soluzioni più leggere e cross-platform, senza modificare la logica interna del cargoservice.

Si sottolinea, però, che anche in questo caso il cliente non ha richiesto esplicitamente tali funzionalità di richiesta tramite web-gui; evidentemente possiede già delle interfacce per effettuare tali azioni. Si mantiene, dunque, questa possibilità aperta solo a fronte di nuovi requisiti futuri da parte del cliente.

Scelta del server web e dell'architettura web: Dai punti precedenti dell'analisi è emerso come la soluzione migliore per realizzare una web-gui che mostri lo stato della stiva e che si possa aggiornare in tempo reale sia quello di sfruttare il protocollo CoAP offerto da QAK attraverso un server web che ovviamente fornisca la web-gui agli utenti che ne necessitano e mantenga la subscribe alla risorsa che rappresenta lo stato della stiva girandone gli aggiornamenti tramite WebSocket ai client JavaScript.

Questi sono i dettagli che sono stati definiti in fase di analisi, ma vi è ancora spazio di scelta per quel che riguarda i seguenti punti:

- Linguaggio lato server
    - java (eventualmente con Spring): offre in più rispetto agli altri la possibilità di sfruttare la libreria commutils per comunicare con QAK (cosa che può anche avvenire eventualmente con MQTT)
    - python
    - node.js
    - Altri...

- Framework JavaScript lato client
    - Js vanilla
    - Js con librerie come Jquery
    - React
    - Altri framework come Angular o Vue...
- Architettura server web
    - Un unico server che offre sia le pagine web sia la WebSocket: più compattezza nello svolgimento della stessa funzionalità, si mantiene una buona separazione delle responsabilità (l'unico server si occupa di tutto ciò che riguarda la realizzazione e funzionamento di questa web-gui)
    - Due server: uno per le pagine web e uno che interagisce con CoAP e offre la WebSocket: più flessibilità e maggiore separazione delle responsabilità a fronte di una maggiore complessità (ad esempio gestione delle CORS policy e comunicazione tra i due server)

Le scelte riguardanti questi due punti verranno prese in fase di progettazione e, per quanto riguarda l'ultimo punto, in fase di deployment. Si suggerisce di propendere solamente verso tecnologie già conosciute dal team di sviluppo, così da evitare tempi e costi per la formazione.

Analisi requisiti aggiuntivi: Si nota come possa essere utile considerare in questo sprint almeno la funzionalità di invio di richieste di carico attraverso la gui in quanto non si figura come un'operazione complicata e potrebbe anche diminuire il tempo di test e fornire un ambiente più simile a quello reale. Fino a questo punto non era ancora stato definito chi dovesse effettuare la richiesta per caricare un container di uno specifico prodotto. Come già emerso negli sprint precedenti, il committente non ha mai fornito indicazioni precise in merito e la questione era rimasta in sospeso. Si è deciso di assegnare questo ruolo all'utente, il quale può effettuare tale azione attraverso la GUI.

L'obiettivo di questo requisito aggiuntivo è permettere all'utente di **inviare una richiesta di caricamento di un prodotto direttamente dalla Web-GUI**, facendo sì che tale richiesta raggiunga il **cargoservice**, responsabile dell'effettivo caricamento nella stiva.
Per realizzare questa comunicazione, sono state analizzate diverse possibilità tecnologiche, valutandone vantaggi e svantaggi.

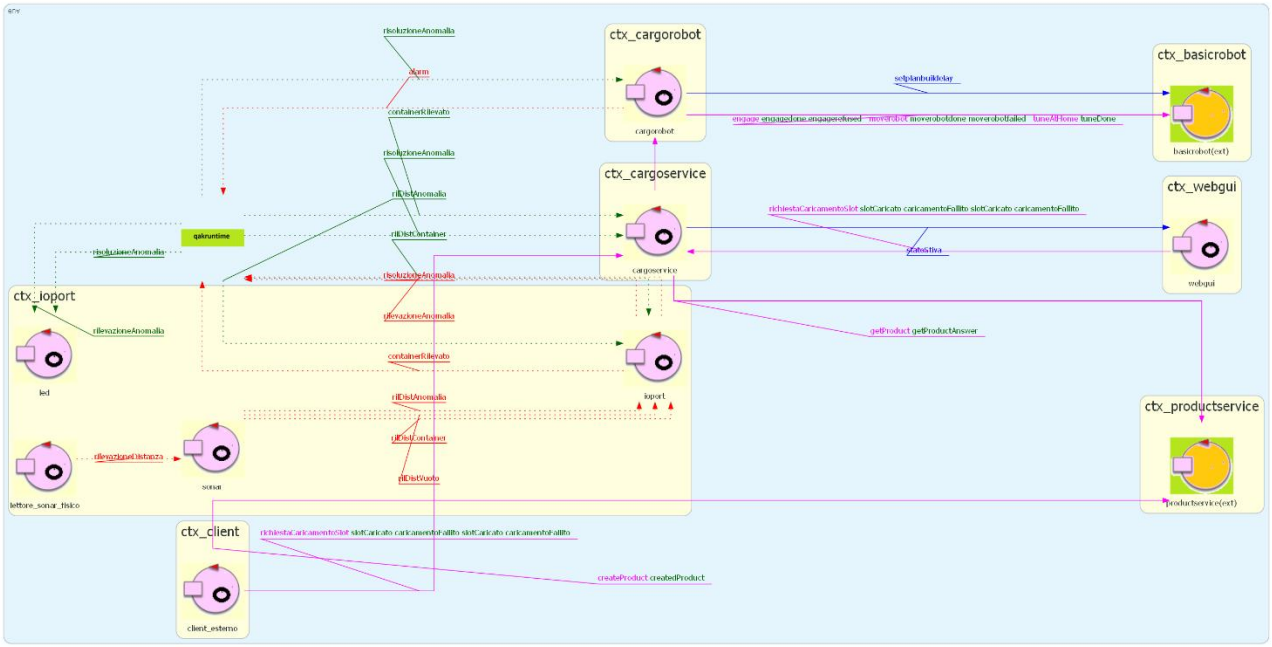
1. **Comunicazione diretta tramite messaggi QAK**
    a. *Vantaggi*: integrazione totale con il modello ad attori, semplicità logica nella gestione dei messaggi e nessun componente intermedio aggiuntivo.
    b. *Svantaggi*: non è possibile inviare messaggi QAK direttamente da un browser, poiché questo richiederebbe l'utilizzo del protocollo mqtt o una semplice comunicazione tcp tramite socket, non compatibili con JavaScript per browser.

2. **Utilizzo del protocollo MQTT**
    a. *Vantaggi*: protocollo leggero, asincrono e scalabile, peienamente supportato da QAK. Permette una comunicazione publish/subscribe efficiente e può essere usato anche per l'invio di comandi.
    b. *Svantaggi*: richiede la presenza di un **broker MQTT** aggiuntivo (es. Mosquitto) e un client MQTT nel browser, il quale però non è pienamente supportato nativamente e comporterebbe complessità di configurazione, sicurezza.

3. **API http tramite server intermedio**
    a. *Vantaggi*: sfrutta **HTTP**, protocollo standard e nativamente accessibile dal browser; non richiede componenti aggiuntivi né modifiche al QAK runtime. Il server intermedio traduce la richiesta web in un messaggio QAK (o sfrutta mqtt), permettendo così la comunicazione indiretta ma trasparente tra GUI e cargoservice. L'interazione http mappa anche bene la natura del messaggio in quanto http funziona sempre tramite interazione richiesta/risposta.
    b. *Svantaggi*: maggiore latenza rispetto alla comunicazione diretta e necessità di mantenere il server intermedio attivo.

Tra le varie opzioni, l'ultima risulta la più adatta al contesto, in quanto si integra naturalmente con l'infrastruttura già esistente: infatti, **il server Java Spring** utilizzato come ponte tra CoAP e WebSocket può essere esteso con minime modifiche per gestire anche le richieste http provenienti dalla GUI, mantenendo coerenza architetturale e semplicità di sviluppo.

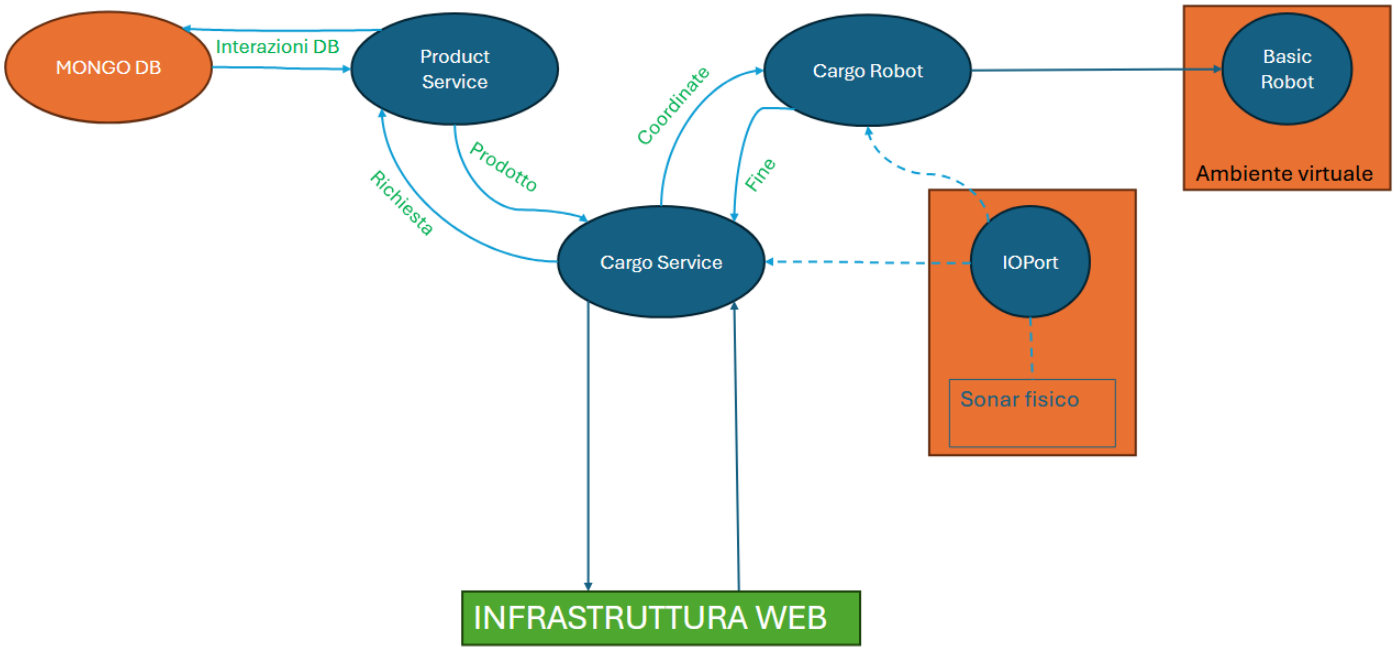
Architettura logica

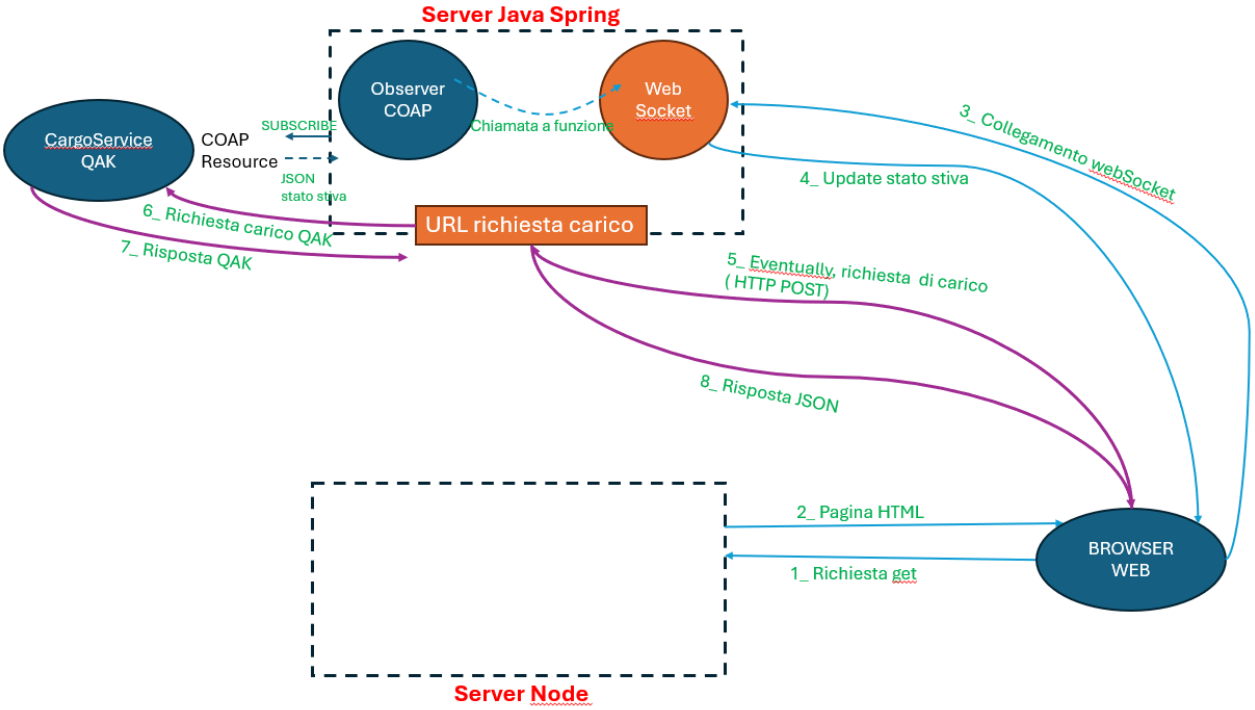


Si noti come webgui è al momento modellato come attore QAK. Probabilmente non verrà implementato come tale (QAK non implementa nativamente un server web, come già segnalato in precedenza), ma Il suo ruolo logico rimane comunque corretto.

Progettazione

Infrastruttura del sistema





Nella prima immagine è raffigurata l'infrastruttura del sistema a microservizi, e come essi sono collegati fra loro. Il comportamento degli attori presenti nell'immagine è il seguente:

- **Product Service**: Permette di registrare i prodotti associando ad essi un PID univoco e permette di recuperare le informazioni relative a tali prodotti.
- **Cargo Service:** Gestisce le richieste di carico di container, recupera i prodotti dal productservice e li associa agli slot dove dovranno essere caricati e mediante il cargorobot gestisce il caricamento concreto dei container.
- **Cargo Robot:** Quando una richiesta di caricamento viene accettata deve muoversi dentro la stiva raggiungendo prima l'IOPort e successivamente lo slot corretto in cui caricare il container
- **Basic Robot:** servizio che permette di interagire con **VirtualRobot** a un livello di astrazione più alto. Nello specifico offre funzionalità di spostamento del robot ad alto livello
- **IOPort:** Attore che interagisce con il mondo esterno e manda informazioni relative alla presenza di un container e alla rilevazione di anomalie.

Il riquadro verde con cui interagisce il CargoService rappresenta l'infrastruttura WEB contenente anche la logica della WebGUI. Questa parte è descritta nel particolare nella seconda immagine, dove viene fatto uno "zoom" su quali sono le componenti e le interazioni che entrano in gioco in questa infrastruttura.

L'infrastruttura web progettata prevede la presenza di due componenti server distinti: **Server Node** e **Server Java Spring,** i quali consentono di visualizzare lo stato della stiva in tempo reale.
Il **CargoService QAK**, che detiene le informazioni aggiornate sulla stiva, espone tale stato come **risorsa CoAP** (come stringa in formato JSON). Un componente **Observer CoAP**, implementato nel server Java Spring, si sottoscrive (observe) a questa risorsa, ricevendo automaticamente notifiche ogni volta che si verificano una variazione dello stato della stiva.
Sempre il server Java Spring gestisce una **WebSocket** che funge da **ponte** tra il mondo QAK e i client web: ogni aggiornamento ricevuto dal CargoService, infatti, viene immediatamente inoltrato alla pagina web sul browser connesso tramite il canale WebSocket, garantendo l'aggiornamento in tempo reale della GUI.

Parallelamente, un **server Node.js** si occupa di fornire la **pagina HTML** della GUI: l'interazione è quella tipica http in cui il browser invia una richiesta GET al server Node, che risponde con la pagina web; una volta caricata, la GUI apre automaticamente la connessione WebSocket verso il server Java Spring, dal quale poi riceve in tempo reale gli aggiornamenti relativi allo stato della stiva.

L'utilizzo di **Spring** come livello intermedio tra il sistema QAK (CargoService) e la web-GUI offre un **ponte naturale tra due mondi diversi**:

- quello **reactive e message-based** di QAK, che comunica tramite **CoAP** e messaggi asincroni;
- e quello **web tradizionale**, basato su **HTTP/WebSocket**, con cui i browser interagiscono.

Questa soluzione permette di far fronte al problema esposto in analisi per il quale la web gui non può accedere direttamente alla risorsa CoAP per via dei limiti tecnologici già analizzati.

Spring, grazie alla sua struttura modulare e all'integrazione nativa con protocolli standard, funge da **adapter** che, incorporando le funzionalità core dell'applicazione (cioè la logica e i dati forniti dal CargoService), le rende accessibili al mondo web, **senza dover modificare il codice QAK**. (Come discusso più volte a lezione facciamo riferimento al "principio di Ironman").

In fase di progettazione, quindi, si è deciso di adottare questa infrastruttura per le motivazioni enunciate ed esplorate finora. Per avere una visione chiara delle scelte prese, riassumiamo le motivazioni nel seguente paragrafo:

L'architettura a due livelli (Node.js per la parte web, Java Spring come adapter tra CoAP e WebSocket) assicura una chiara separazione delle responsabilità: il mondo QAK continua a operare tramite CoAP, mentre la GUI utilizza tecnologie standard del web (HTTP e WebSocket). In aggiunta a ciò, la presenza del livello intermedio in Java facilita eventuali estensioni future, come l'invio di comandi verso il CargoService.

Da ora in poi si farà riferimento al server Java Spring come backend e a quello Node come frontend

Progettazione backend

Il backend è stato realizzato in **Java** con l'utilizzo del framework **Spring** che funge da componente intermedio tra il CargoService QAK e la Web-GUI. Questa scelta deriva dai vantaggi legati alla compatibilità con l'ambiente QAK: in questo modo è possibile riutilizzare gli oggetti POJO realizzati in precedenza e sfruttare le librerie di interazione con QAK (commutils principalmente).

Per quel che riguarda il framework Spring, esso è uno dei principali framework Java e permette di facilitare molte fasi dello sviluppo, nello specifico permette di implementare con facilità i componenti che realizzano WebSocket e il CoAP observer tramite apposite librerie e le funzionalità del framework. Grazie a questo approccio, il server si sottoscrive alla risorsa CoAP del CargoService e, ogni volta che questa cambia, propaga in tempo reale l'aggiornamento ai client web connessi.

Questa soluzione, dunque, garantisce un'elevata **indipendenza** tra logica applicativa e presentazione, una **minore complessità** rispetto ad alternative e una maggiore **scalabilità futura**, poiché il server Spring potrà facilmente essere esteso per gestire comandi in ingresso dalla GUI. La scelta è ricaduta su questa soluzione anche in quanto il team di sviluppo conosce già questa tecnologia.

Progettazione frontend

Come già riportato nella sezione precedente la scelta riguardante i linguaggi da utilizzare per la realizzazione della gui è obbligata a quelli che di fatto sono gli standard del web (HTML, CSS e JavaScript), ma tra i vari framework JavaScript disponibili si è deciso di sviluppare il frontend utilizzando **React**, per via della volontà di creare un'interfaccia **reattiva, modulare e facilmente aggiornabile in tempo reale**. React consente di rappresentare la stiva come una struttura di componenti indipendenti, ciascuno dei quali si aggiorna automaticamente al variare dei dati ricevuti tramite la connessione **WebSocket** con il backend Spring. Così facendo, ogni cambiamento dello stato della stiva viene immediatamente riflesso nella GUI senza necessità di ricaricare la pagina o effettuare polling periodico. Inoltre React rende più efficiente l'aggiornamento dei componenti da modificare, ricaricando solamente le porzioni necessarie e non tutti i componenti.

L'adozione di React offre, inoltre, vantaggi in termini di **manutenibilità e scalabilità:** permette, infatti, di aggiungere facilmente nuove sezioni o funzionalità senza modificare la logica esistente. In aggiunta, l'utilizzo di un linguaggio standard come **JavaScript** e di un framework largamente diffuso garantisce infine massima compatibilità con i browser moderni.

Anche in questo caso la scelta è stata influenzata dal fatto che il team di sviluppo ha già competenze nell'utilizzo di questo framework. Nel paragrafo successivo abbiamo discusso quali sono le componenti React e le loro funzionalità.

Componenti React

- **Stato stiva**: visualizzato al centro della pagina, riporta lo stato della stiva: al suo interno, infatti, vi sono cinque rettangoli che rappresentano gli slot. Quando uno slot viene occupato, all'interno del rettangolo corrispondente sono riportati il PID del prodotto e il peso del container, e, in più, cambia colorazione. Come si può osservare dall'immagine, lo slot 5, che da requisiti rimane sempre occupato, appare con la scritta UNAVAILABLE.

- **Richiesta Carico**: visualizzato alla sinistra del componente precedente, serve all'utente per indicare il prodotto del container che vuole sia caricato nella stiva. Si è deciso che, per fare ciò, l'utente debba inserire nell'area di testo apposita l'identificativo del prodotto (PID) desiderato, in quanto unica informazione che discrimina ogni prodotto dagli altri. In aggiunta, viene anche riportato lo stato della connessione WebSocket ("disconnesso" o "connesso") con il server Java Spring.

- **Statistiche in tempo reale**: visualizzato alla destra del componente "Stato stiva", riporta quattro informazioni in tempo reale suddivise in due riquadri, uno superiore e uno inferiore. Nel riquadro superiore viene indicato il peso totale di tutti i container presenti nella stiva in quel momento; viene anche rappresentata la percentuale sul peso massimo sostenibile dalla stiva. Nel riquadro inferiore, invece, sono riportati: il numero di slot occupati, il peso medio per slot, e il prodotto più pesante.
  Nota di particolare interesse è come viene trattata la **MaxLoad**: non è un valore fissato nel codice, ma viene **inviata dinamicamente dal server** insieme allo stato della stiva grazie ad un **oggetto JSON** che contiene due campi: MaxLoad e SlotMap; la componente React estrae automaticamente il valore di `maxLoad` e lo salva nello **state** del componente (`this.state.maxLoad`). In questo modo la GUI si aggiorna in tempo reale con il valore corrente del peso massimo della stiva, senza doverlo modificare manualmente nel codice.

Progettazione requisiti aggintivi

Una volta analizzato il requisito aggiuntivo relativo alla possibilità di **inviare una richiesta di caricamento di un prodotto direttamente dalla Web-GUI,** in questo paragrafo ci siamo occupati della relativa progettazione.

Dal punto di vista progettuale, la soluzione adottata consiste nell'utilizzare il **server web Java già presente** come intermediario tra la Web-GUI e il sistema QAK. Questo server, essendo sviluppato in **Java** può sfruttare la libreria **commutils** compatibile con QAK e permette di comunicare con tali attori in modo agevole. L'alternativa sarebbe sfruttare mqtt (integrato anch'esso con QAK), ma ciò necessiterebbe dell'introduzione di un broker aggiuntivo non necessario. Tale broker porterebbe a un maggiore overhead nelle communicazioni e nella presenza di un componente in più da gestire.
Nello specifico la Web-GUI invierà una **richiesta HTTP POST** a un endpoint esposto dal server (`/api/cargoservice/richiesta-carico`), contenente nel corpo JSON l'ID del prodotto da caricare. Il server riceve questa richiesta e, tramite CommUtils, inoltra un messaggio al cargoservice per attivare il caricamento.

L'utilizzo di una **richiesta POST**, anziché una GET, è motivato dal fatto che l'operazione comporta **una modifica dello stato del sistema** (richiesta di carico di un nuovo prodotto), e non una semplice lettura di dati. POST, infatti, consente di inviare in modo strutturato dati nel corpo della richiesta (come l'ID del prodotto) e rispetta le convenzioni REST secondo cui GET è un'operazione ripetibile e priva di effetti collaterali (cioè non modifica lo stato del sistema), mentre **POST è destinato a operazioni che alterano lo stato applicativo**.

In tutto ciò l'implementazione all'interno del server backend java è coerente con la suddivisione delle responsabilità in quanto anche in questo caso tale funzionalità rappresenta la realizzazione di un intermediario tra gui e mondo QAK.

Rappresentazione risorsa stiva

Come già introdotto in analisi, è vantaggioso rappresentare la risorsa CoAP "stiva" come una stringa in formato JSON (i vantaggi sono descritti nella fase di analisi). Di seguito viene riportato un esempio della struttura dati specifica utilizzata: a ogni slot esistente e utilizzabile (si ricorda che lo slot 5 è stato omesso in quanto non significativo) nella stiva è associato il prodotto che lo occupa (di cui vengono riportati nome, PID e peso) o null in caso quello slot sia libero. Da notare che l'omissione dello slot 5 non inficia l'estendibilità ad un suo futuro utilizzo: in caso lo slot 5 diventasse disponibile in futuro, basterà aggiungerlo al file di configurazione degli slot in cargoservice e automaticamente sarà considerato come parte funzionale della stiva

```
{
    "slotMap":{
        "Slot1": {"productId": 1, "name": "prod1", "weight": 20},
        "Slot2": {"productId": 2, "name": "prod2", "weight": 15},
        "Slot3": null,
        "Slot4": null
    },
    "maxLoad": 100
}
```

Modifiche a cargoservice

Finora non sono state trattate le modifiche fatte a cargoservice. Queste consistono solo nell'esposizione della risorsa rappresentante la stiva (associazione slot-prodotto) tramite la

funzionalità di QAK *updateResource*. In questo modo si modella lo stato della stiva come una risorsa QAK legata all'attore cargoservice, il quale produce l'aggiornamento di una risorsa CoAP. Nello specifico, l'aggiornamento della risorsa può avvenire in tre momenti principali: inizializzazione, per rendere disponibile la risorsa inizialmente contenente gli slot disponibili; aggiornamento, vale a dire ogni qualvolta un prodotto viene assegnato a uno slot; e in fase di reset della stiva (è un caso utilizzato per ora solamente per i test, cosicché la gui possa rappresentare uno stato consistente a seguito di un reset).

Piano di Test

Il collaudo della Web-GUI è stato effettuato principalmente tramite **test visivi**, poiché l'obiettivo dell'interfaccia è la corretta rappresentazione grafica dell'aggiornamento in tempo reale dello stato della stiva. In questa fase sono state simulate diverse situazioni operative del sistema — come caricamento di container, slot liberi o pieni e segnalazione di anomalie — verificando che la GUI reagisse correttamente, aggiornando la visualizzazione senza ritardi o incoerenze. Gli stessi test hanno inoltre permesso di confermare il corretto funzionamento della comunicazione tra backend e frontend, cioè la ricezione degli aggiornamenti via WebSocket e la corretta sincronizzazione con lo stato effettivo mantenuto dal CargoService.

Si noti come, dato che le funzionalità di invio di richieste dalla gui non erano state commissionate, le richieste a cargoservice, al fine di vedere gli effetti sulla gui, verranno effettuate tramite la libreria commutils in Java, sfruttando il codice utilizzato nei test del cargoservice sviluppati nello sprint 1.

Deployment

L'architettura della web-gui può essere vista come composta da due microservizi distinti: il primo (server java Spring) che permette di recuperare gli aggiornamenti della risorsa CoAP e riportarli su WebSocket; il secondo (server node) che implementa il server web con il fine di fornire le pagine web. Questi due servizi, anche se molto legati fra loro, verranno rilasciati come due microservizi separati di cui verrà fatto il deployment su docker; così facendo, si fornirà maggiore flessibilità e modularità all'intero sistema.

Ciò nonostante, i due microservizi verranno anche forniti come un unico docker-compose che comprenderà i container di entrambe le immagini.

Per caricare le immagini dei due servizi su docker:

1. Creare l'immagine docker (una solva volta oppure di nuovo quando cambia il progetto). Entrare nella cartella del progetto di cui si vuole fare l'immagine e fare:
   a. ***./gradlew distTar***
   b. ***docker build -t <nome>:1.0 .***

   dove <nome> è il nome del progetto, ovvero quello dopo System nella prima riga del qak.
2. Entrare nelle varie cartelle dove ci sono i file yaml e lanciare il comando docker-compose giusto

   docker-compose -f nomefile.yaml up

Ripartizione del lavoro

Lo sviluppo dello **sprint 3** è stato organizzato attraverso una chiara suddivisione delle attività tra i tre membri del gruppo, al fine di garantire un avanzamento parallelo e coordinato delle diverse componenti del progetto. Ognuno dei membri del team si è occupato di una macroarea, anche se ogni punto è stato ampiamente discusso fra tutti i membri:

- **Brighi Valerio: Implementazione del frontend:** Si è concentrato sulla realizzazione dell'interfaccia grafica utilizzando **React**, framework JavaScript scelto per garantire modularità, reattività e aggiornamento dinamico dei dati.
  Ha curato la realizzazione della pagina web e della componente JavaScript che gestisce la connessione WebSocket verso il server Spring, aggiornando dinamicamente l'interfaccia grafica a fronte delle variazioni dello stato della stiva.

- **Zoccadelli Lorenzo: Implementazione del backend e analisi del problema:**
  Si è occupato della schematizzazione dell'infrastruttura web e ha reso possibile la comprensione dell'intera architettura pensata agli altri memebri del gruppo.
  Si è occupato della realizzazione del server Java Spring che funge da ponte tra il CargoService QAK e la GUI web. In questa parte sono stati gestiti il collegamento CoAP con il CargoService, la sottoscrizione (observe) alla risorsa che rappresenta lo stato della stiva, e la propagazione in tempo reale degli aggiornamenti ai client tramite WebSocket; inoltre, state definite le strutture dati e i formati JSON scambiati con il frontend.

- **Guiducci Daniele: Analisi del problema, Progettazione, supporto per il frontend:**
  Il membro ha svolto la fase preliminare di analisi dei requisiti (concentrandosi sulla parte di aggiornamento automatico della web-gui) e di definizione dell'architettura complessiva del sistema. In particolare, si è concentrato nella progettazione e spiegazione dell'infrastruttura web, valutando le alternative tecnologiche (CoAP, MQTT, WebSocket) e definendo la struttura a due livelli — con server Spring come ponte verso QAK e server Node.js per la distribuzione della GUI — che garantisce separazione delle responsabilità e scalabilità del sistema. Inoltre, ha offerto supporto al membro incaricato all'implementazione del frontend nella scrittura di alcune componenti React.

Questa distribuzione del lavoro ha permesso di coprire in modo completo le fasi di analisi, progettazione, sviluppo e integrazione, mantenendo una chiara responsabilità di ogni membro del gruppo. Come già scritto nei precedenti sprint, alcune parti sono state fatte totalmente insieme: ad esempio, l'analisi generale del problema, la revisione finale del documento, i test non automatici della verifica del corretto funzionamento della GUI e la verifica finale del funzionamento dell'intero sistema.

Conclusioni

Lo sviluppo dello **Sprint 3** ha permesso di completare il sistema realizzando con successo la **Web-GUI per la visualizzazione dello stato della stiva**, integrata con l'architettura QAK esistente. Attraverso la progettazione di un'infrastruttura a due livelli — composta da un server **Java Spring** come adapter tra il CargoService e il mondo web, e da un server **Node.js** per la distribuzione della GUI — è stato possibile

garantire un aggiornamento **in tempo reale**, una **chiara separazione delle responsabilità** e la **scalabilità** dell'intero sistema.

L'utilizzo di **React** per il frontend ha reso l'interfaccia reattiva, moderna e facilmente estendibile, mentre l'integrazione con **CoAP** e **WebSocket** ha assicurato una comunicazione efficiente e coerente con l'architettura QAK.

Nel complesso, il risultato finale soddisfa pienamente i requisiti richiesti dal cliente, mantenendo al contempo un'elevata flessibilità per eventuali **estensioni future**, come l'aggiunta di funzionalità interattive o l'integrazione con altri servizi.