LABORATORIO DI INGEGNERIA DEI SISTEMI SOFTWARE

Introduction

All'interno dello Sprint1 è stato analizzato il core-business dell'applicazione, ovvero le funzionalità offerte dal *ColdStorageService*, usando la ServiceAccessGUI come prompt per utilizzare i servizi. In questa fase, ci siamo occupati della logica del *ColdStorageService* usando un contesto semplificativo in cui vi fosse un'interazione per volta.
In questo Sprint gestiremo invece la presenza di più utilizzatori simultanei, analizzando quindi il concetto di **ticket** e costruendo la versione finale della **ServiceAccessGUI**.

Requirements

Descrizione requirements a questa pagina

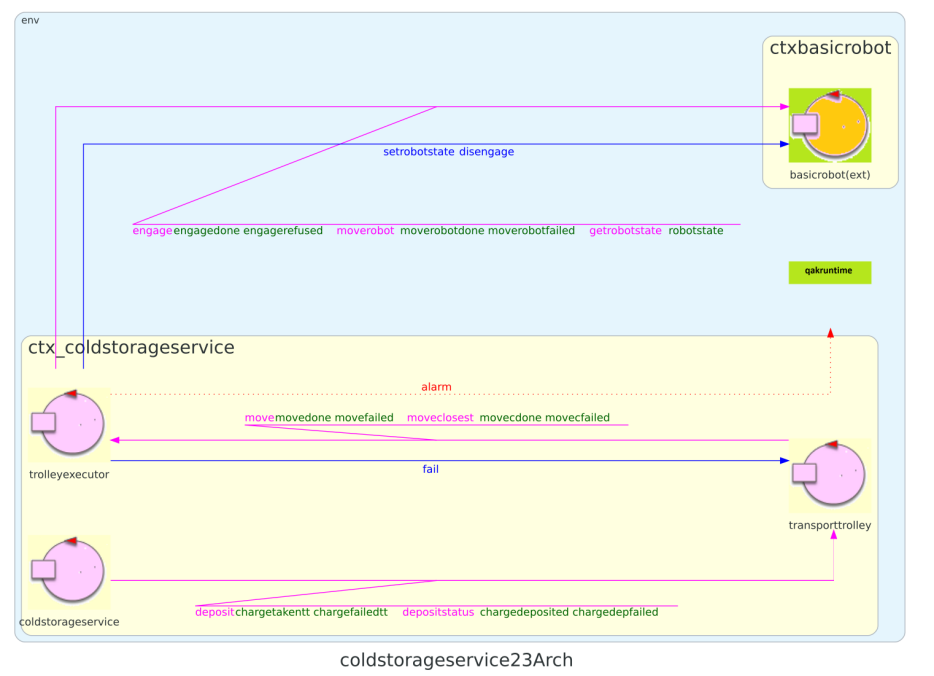
Goals dello Sprint2:

1. Definire cos'è il **ticket** e come gestirlo
2. Implementare la **ServiceAccessGUI** con il concetto di più utilizzatori contemporanei in un sistema distribuito

> Visto quanto realizzato nell'analisi della ServiceAccessGUI relativa allo *Sprint1*, abbiamo già pronta l'applicazione web di partenza, realizzata con il framework **SpringBoot**.

Requirement analysis

Si riporta il modello risultante dall'analisi in Sprint1, sul quale si baserà la seguente analisi.



coldstorageservice23Arch

Inoltre, viene riportato nei seguenti documenti la versione del *TemaFinale23* analizzata contestualmente a questo Sprint, ed in particolare ai due macro-argomenti:

- ServiceAccessGUI
- gestione del ticket

ServiceAccessGUI

La *ServiceAccessGUI* è l'interfaccia grafica a disposizione dell'utente per accedere ai servizi forniti dal *ColdStorageService*.
Oltre a quanto già realizzato nella ServiceAccessGUI relativa allo *Sprint1*, si aggiungono i requisiti di:

- gestire il ticket, fornendolo all'utente e mostrando un campo di inserimento
- mostrare il peso corrente all'interno della ColdRoom

Interazioni

Di seguito, vengono riportate le interazioni presenti e già implementate nello *Sprint1*.
Abbiamo tre attori:

- FridgeTruck: utente umano
- ServiceAccessGUI: interfaccia grafica (applicazione web realizzata relativamente allo *Sprint1*)
- coldstorageservice: attore *qak*

Mittente	Destinatario	Tipologia interazione/messaggio	Identificatore messaggio	Payload	Descrizione
FridgeTruck	ServiceAccessGUI	Interazione umana			Inserimento quantitativo di cibo da depositare in chilogrammi e innesco richiesta di deposito.
ServiceAccessGUI	coldstorageservice	Request	storerequest	storerequest(FW)	Richiesta di **storage** di **FW** chili di cibo.
coldstorageservice	ServiceAccessGUI	Reply	loadaccepted	loadaccepted()	La richiesta di deposito è stata accettata.
coldstorageservice	ServiceAccessGUI	Reply	loadrejected	loadrejected()	La richiesta è stata rifiutata.
FridgeTruck	ServiceAccessGUI	Interazione umana			Innesco richiesta di stato del carico.
ServiceAccessGUI	coldstorageservice	Request	chargestatus	chargestatus()	Viene richiesto lo stato del carico depositato.
coldstorageservice	ServiceAccessGUI	Reply	chargetaken	chargetaken()	Il deposito è stato preso in carico dal transport trolley, il camion deve spostarsi da INDOOR.
coldstorageservice	ServiceAccessGUI	Reply	chargefailed	chargefailed()	Ci sono stati problemi durante la presa in carico del deposito.

Dai requisiti non emerge nessun nuovo tipo di interazione tra i componenti già individuati. Verranno discusse modifiche alle interazioni già presenti per integrare le nuove funzionalità in fase di analisi del problema.

Ora, come visto nei requisiti, a seguito di una richiesta accettata (*loadaccepted*), il *ColdStorageService* deve **fornire** il **ticket number**.
Di conseguenza, dovrà essere presente nella *SAG* un campo per l'**inserimento del ticket**, da cui poi innescare la richiesta di deposito verso il *coldstorageservice*, specificando appunto il *ticket number*.

Inoltre, si aggiunge il requisito di visualizzare il peso corrente del materiale immagazzinato nella Cold Room.

L'applicazione deve mostrare visualizzazioni separate per ogni utente (operiamo in un contesto distribuito). Quindi, un'utente dovrà visualizzare l'accettazione (o rifiuto) della richiesta e l'eventuale presa in carico solo relativamente alla sua richiesta di deposito.

Ticket

Termine	Descrizione
TICKETTIME	Quantità di secondi che esprime la durata della validità di un ticket.
Ticket	Rappresenta una prenotazione da parte dell'operatore di un camion refrigerato che identifica univocamente una azione di deposito in attesa di *charge taken*, è identificato da un **numero univoco**.

Il **ticket** è quindi l'elemento che il servizio *ColdStorageService* dovrà inizialmente fornire all'utilizzatore, per poi controllarne la validità in fase di inserimento.

Parlando con il committente si è scoperto non essere necessario che il *"numero"* del ticket sia propriamente un numero: intendiamolo quindi come un **codice**.

Problem analysis

ServiceAccessGUI

Contesto distribuito

La *ServiceAccessGUI* dovrà notificare l'utente della presa in carico del proprio deposito. La separazione delle risposte in base alle varie istanze di *SAG* presenti è già stata realizzata grazie alla semantica request-reply definita in precedenza. Quindi, non abbiamo problemi di memorizzazione degli indirizzi a cui inviare i messaggi.

Visualizzazione stato ColdRoom

La *SAG* dovrà essere aggiornata di pari passo con il materiale immagazzinato nella ColdRoom e si suppone dovrà averlo disponibile sin dall'inizializzazione.

L'informazione, in pancia all'attore *coldstorageservice*, sarà fornita tramite l'emissione di un'evento da parte dell'attore stesso, ogni qualvolta che il carico immagazzinato sarà modificato. In questo modo, tutte le istanze di ServiceAccessGUI (ed in generale un qualsiasi componente interessato all'informazione) si sottoscriverà all'attore *coldstorageservice* in vece di osservatore.

KEY-POINT: l'attore *coldstorageservice* sarà una **risorsa osservabile**

Per risolvere invece il problema dell'*inizializzazione*, introduciamo una nuova interazione **request-reply**, innescata dalla *SAG* verso il *CSS* all'avvio.

Mittente	Destinatario	Tipologia interazione/messaggio	Identificatore messaggio	Payload	Descrizione
ServiceAccessGUI	coldstorageservice	Request	initcoldroom	initcoldroom()	Richiesta del valore attuale della ColdRoom.
coldstorageservice	ServiceAccessGUI	Reply	coldroom	coldroom(ACTUAL,TEMP)	Valore di kg nella ColdRoom effettivi e totali (contando anche quelli in coda).

COMMITTENTE: abbiamo pensato di fornire alla *SAG* le informazioni relative sia al totale attualmente immagazzinato nella ColdRoom, sia a quanto è riservato ma non ancora depositato.
Essendo entrambe le info in pancia al *CSS*, non ci sarebbe nessun overhead aggiuntivo per questa (a nostro parere utile) informazione, ma meglio discuterne con il committente.

Visualizzazione del ticket

Il ticket verrà restituito dal *CSS* e la *SAG* dovrà fornirlo all'utente. Per farlo, abbiamo pensato a due vie:

- mostarlo in chiaro nell'applicazione web (all'interno della pagina)
- fornirlo in maniera esterna all'applicazione

Mentre il primo modo semplifica la vita all'utente che dovrà fare solamente un copia-incolla all'interno della pagina, il secondo permette invece di far fronte a problemi come la chiusura dell'applicazione.
Infatti, se una volta ottenuto il ticket (e fosse visualizzabile solo nella pagina) l'utente chiudesse per errore la pagina web, perderebbe la possibilità di inserirlo in quanto non se lo ricorderebbe (e dovrebbe effettuare una nuova richiesta di deposito).
Al contrario, fornendo un via d'accesso al codice esterna all'app web (come un file *.pdf* scaricabile dall'app web), questo problema si eviterebbe.

KEY-POINT: fornire il *ticket* all'utente in modo **esterno** all'applicazione web *SAG*

Ticket

Dall'analisi dei requisiti vediamo che, oltre al **codice**, un'altro elemento inerente alla gestione del **ticket** è il **TICKETTIME**.
Dobbiamo pensare quindi come gestire la scadenza del ticket.

Validità temporale del ticket

Trattiamo il tempo di validità del ticket come un parametro configurabile alla definizione del *ColdStorageService*; quindi ogni ticket all'interno della stessa istanza del servizio avrà la stessa durata temporale di validità.
Sappiamo poi che è il *ColdStorageService* a controllare la conformità del ticket (correttezza del codice e validità temporale).
Decidiamo quindi di utilizzare la **data di emissione** come attributo del ticket, così che il *CSS* possa controllarne la validità calcolando il tempo attuale meno quello di emissione e vedere se è maggiore o meno del **TICKETTIME**.

KEY-POINT: il TICKETTIME è un parametro configurabile del servizio *ColdStorageService*, non del ticket in sè.

KEY-POINT: ogni ticket avrà associata la propria data di emissione.

Codice univoco

Per quanto riguarda il **codice** abbiamo bisogno che sia univoco, così da non avere collisioni tra diverse richieste di deposito.
Dobbiamo quindi pensare a:

- che tipo di codice deve essere
- come generarlo e garantire che non collida con altri
- come memorizzarlo ed effettuare il controllo di validità in fase di inserimento dalla *SAG*

Formato del ticket

Come detto nei requisiti, il codice del ticket sarà alfanumerico, possiamo utilizzarlo quindi per inserirci delle informazioni.
In particolare, il ticket avrà le proprietà riportate in tabella.

Descrizione	Formattazione
T = ticket	t
Peso in kg (senza il punto per la separazione del decimale) del carico depositato	Numero intero positivo
N = numero del ticket	n
Numero incrementale corrispondente ai ticket emessi dall'avvio del servizio *CSS*	Numero intero positivo, incrementato di volta in volta
2 caratteri generati casualmente per dare aleatorietà al codice	2 caratteri dell'alfabeto (minuscoli)

KEY-POINT: il *"ticket code"* avrà il formato **t<pesoinkg>n<numeroincrementale><2caratterirandom>**

Generazione

La generazione del codice deve avere i requisiti di univocità e di imprevedibilità. Nonostante non si stiano trattando dati sensibili è bene che per utente malevolo sia computazionalmente impossibile prevedere quale sia il prossimo codice che verrà generato; se così non fosse, l'utente potrebbe calcolare il codice ed immetterlo nella *SAG* prima che l'utente originale arrivi alla *INDOOR*, depositando al posto suo.
Conseguentemente a questi requisiti, dovremo usare uno **Pseudo Random Number Generator** crittograficamente sicuro (la sua implementazione sarà meglio trattata in fase di progettazione).

KEY-POINT (Requisito di sicurezza): per la generazione del codice del ticket verrà usato un PRNG crittograficamente sicuro.

Memorizzazione e controllo

Per questioni di sicurezza il **ticket** deve essere memorizzato in modo tale da non poter essere scoperto o dedotto dall'esterno. Per ottenere questo risultato possiamo usare una funzione **hash**: in questo modo memorizzeremo solamente l'impronta generata dalla suddetta funzione, risparmiando molto in termini di memoria occupata.
Per effettuare il controllo, al momento dell'inserimento del codice il *CSS* ricalcolerà l'impronta tramite la stessa funzione hash per vedere se trova corrispondenza.

Funzione hash
Anche qui necessitiamo di una funzione crittograficamente sicura, che sia efficiente, che non permetta il calcolo a ritroso e che soprattutto sia resistente alle collisioni (l'implementazione verrà discussa in fase di progettazione).

Una volta effettuata la validazione di un ticket non deve essere possibile riverificare lo stesso ticket.

Si predispone un meccanismo per invalidare il ticket ma mantenerlo nello storico di quelli generati.

Associazione ticket-richiesta

Quando il *CSS* emette il ticket deve esserci un'associazione 1 ad 1 con la richiesta di deposito, così che, quando l'utente inserisce il codice, sia possibile risalire alla richiesta e alla quantità di kg **FW** da depositare.
In questo modo non sarà necessario che la *SAG* invii due volte l'informazione relativa al quantità di cibo da depositare.

KEY-POINT: il *CSS* dovrà memorizzare l'associazione fra una determinata richiesta di deposito (con il relativo quantitativo FW) e il ticket emesso in risposta ad essa.

Questa associazione sarà utile per poter richiamare il valore del peso di una richiesta una volta depositata, allo scopo di aggiornare il valore del totale depositato nella ColdRoom.

TicketManager

Con il fine di rispettare il principio di singola responsabilità abbiamo deciso di creare il nuovo componente (attore) **ticketmanager**, che sarà incaricato di:

- generare un nuovo ticket
- validare un ticket dato il suo codice
- mantenere uno storico dei ticket generati

Di seguito si riporta il modello di interazione tra *ServiceAccessGUI*, *ColdStorageService* e il nuovo *TicketManager*.
I nuovi messaggi derivanti da questa analisi sono evidenziati con il colore rosso.

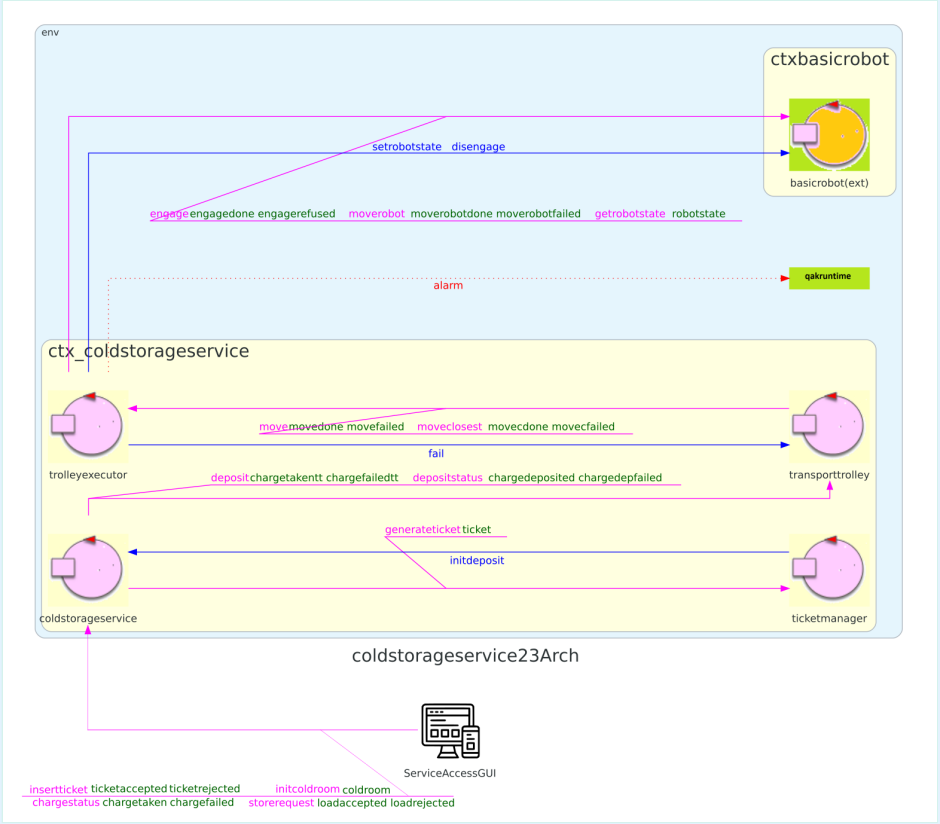
Mittente	Destinatario	Tipologia interazione/messaggio	Identificatore messaggio	Payload	Descrizione
ServiceAccessGUI	coldstorageservice	Request	storerequest	storerequest(FW)	Richiesta di **storage** di **FW** chili di cibo.
coldstorageservice	ServiceAccessGUI	Reply	loadaccepted	loadaccepted()	La richiesta d deposito è stata accettata.
coldstorageservice	ServiceAccessGUI	Reply	loadrejected	loadrejected()	La richiesta è stata rifiutata
ServiceAccessGUI	ticketmanager	Request	insertticket	insertticket(TICKET)	Viene inviato ticket inserito

Mittente	Destinatario	Tipologia interazione/messaggio	Identificatore messaggio	Payload	Descrizione
					al ticketmanage
ticketmanager	ServiceAccessGUI	Reply	ticketaccepted	ticketaccepted()	Il ticket è stat validato e accettato dal sistema.
ticketmanager	ServiceAccessGUI	Reply	ticketrejected	ticketrejected()	Il ticket è stat validato ma rifiutato dal sistema.
coldstorageservice	ticketmanager	Request	generateticket	generateticket(FW)	Richiesta per la generazione di un ticket.
ticketmanager	coldstorageservice	Reply	ticket	ticket(TICKET)	Il ticket è stat generato.
ticketmanager	coldstorageservice	Dispatch	initdeposit	initdeposit(TICKET)	Inizia una azione di deposito dop la validazione di un ticket.
ServiceAccessGUI	coldstorageservice	Request	chargestatus	chargestatus()	Viene richiesto lo stato del carico depositato.
coldstorageservice	ServiceAccessGUI	Reply	chargetaken	chargetaken()	Il deposito è stato preso i carico dal transport trolley, il camion deve spostarsi da INDOOR.
coldstorageservice	ServiceAccessGUI	Reply	chargefailed	chargefailed()	Ci sono stati problemi durante la presa in carico del deposito.
ServiceAccessGUI	coldstorageservice	Request	initcoldroom	initcoldroom()	Richiesta del valore attuale della ColdRoom.
coldstorageservice	ServiceAccessGUI	Reply	coldroom	coldroom(ACTUAL,TEMP)	Valore di kg nella ColdRoom effettivi e tota (contando anche quelli coda).

KEY-POINT: il messaggio **insertticket** viene inviato a coldstorageservice che delegherà la gestione al ticketmanager. Abbiamo così un **unico punto d'accesso** al sistema.

Architettura logica

Si riporta di seguito il modello di architettura logica sulla quale si baserà la successiva fase di progettazione.



Test plans

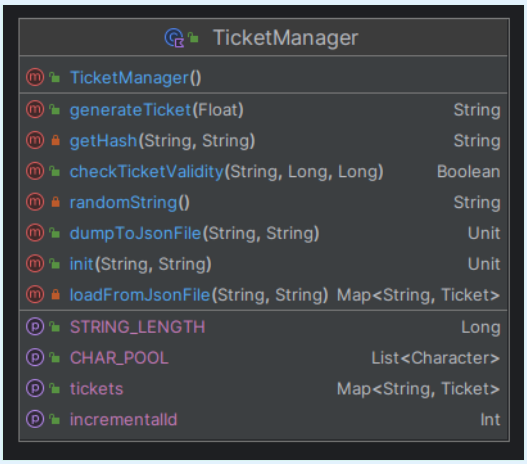
Per quanto riguarda i piani di test da implementare, rimanendo ad un **livello logico**, bisogna effettuare controllare che ogni istanza di *SAG* riceva le risposte dal *CSS* solo ed esclusivamente relativamente alla sua sessione. In particolare:

- ad ogni richiesta di deposito deve arrivare la risposta (accettata, rifiutata o un errore)
- in caso di richiesta accettata deve essere essere fornito il *ticket code*
- se il *ticket code* viene inserito correttamente, deve essere accettato, altrimenti rifiutato
- in caso di ticket accettato, alla richiesta di stato del carico deve arrivare il messaggio *chargetaken*

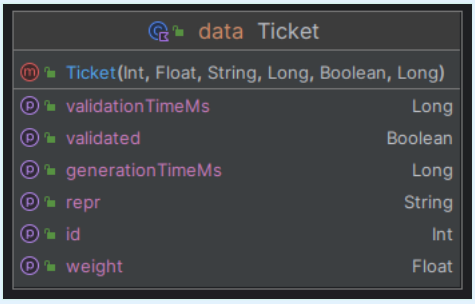
Project

TicketManager

L'attore **ticketmanager** mantiene un'istanza della classe `TicketManager`, questa classe implementa metodi per la generazione di un nuovo ticket e gestisce lo storico dei ticket generati, esponendo metodi per salvare e caricare memoria persistente le informazioni.
Si riporta di seguito l'UML della classe in questione.



La classe `TicketManager` mantiene i ticket come istanze della data class `Ticket`, si riportano di seguito le specifiche delle proprietà della classe.



Identificazione ticket

A ogni ticket viene associato un id intero incrementale che lo identifica univocamente, tuttavia questo valore è a solo uso interno del TicketManager, il sistema tratterà i ticket tramite la loro rappresentazione, ovvero il codice che viene scambiato con gli utenti del sistema.

Per generare i caratteri random che fanno parte della rappresentazione del ticket si fa utilizzo dei metodi offerti dalla classe `java.security.SecureRandom` che garantisce la sicurezza tramite l'utilizzo di uno PRNG.

```
private fun randomString(): String {
    val sr: SecureRandom = SecureRandom.getInstance("SHA1PRNG")
    return sr.ints(STRING_LENGTH, 0, CHAR_POOL.size)
        .asSequence()
        .map(CHAR_POOL::get)
        .joinToString("")
}
```

Gestione tempo di creazione

A ogni ticket viene associato un timestamp campionato nel momento in cui viene restituito il valore dal metodo che lo genera, viene salvato come valore unix timestamp `long` in millisecondi.

```
val generationTimeMs: Long
```

ColdStorageService

Delega al ticketmanager

Come specificato in precedenza, il messaggio *insertticket* può essere inviato a *coldstorageservice* ma verrà comunque gestito dal *ticketmanager*, questa funzionalità è stata implementata utilizzando la keyword `delegate` messa a disposizione dal framework qak.

Nello stato iniziale dell'attore *coldstorageservice* è riportata la seguente istruzione:

```
delegate "insertticket" to ticketmanager
```

Associazione ticket-richiesta

È stata implementata l'associazione tra ticket e richiesta utilizzando una `MutableMap`, dichiarata come segue:

```
val weightTicketMap = mutableMapOf<String, Float>()
```

Utilizzando questa mappa il ColdStorageService sarà in grado di comunicare al TransportTrolley il peso relativo alla ticket che sta gestendo, riottenendolo alla risposta del TransportTrolley per finalizzare il deposito aggiungendo il valore alla variabile `actualCurrentColdRoom`, che mantiene il valore del peso di cibo effettivamente contenuto nella ColdRoom.

```
val FW = payloadArg(0).toFloat()
```

```
actualCurrentColdRoom += FW
```

ServiceAccessGUI

Abbiamo già parlato e analizzato buona parte della *SAG* nello Sprint precedente.
E' stato anche introdotto il protocollo **Coap**, utilizzato per le interazioni tra la *SAG* e il servizio *ColdStorageService*.
Relativamente a ciò, riportiamo nel *key-point* seguente un fatto fondamentale:

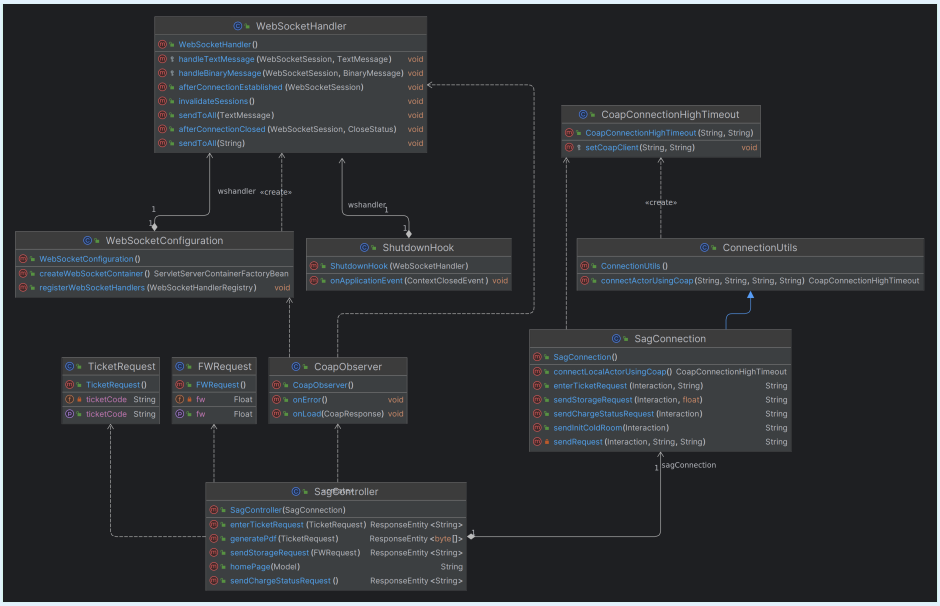
> **KEY-POINT:** gli attori *qak* sono risorse **CoapObservable**, overro risorse osservabilei tramite la classe CoapObserver definita nella libreria `unibo.basicomm23`.

Ricezione evento di aggiornamento ColdRoom

Essendo che la *SAG*, progetto in Spring, non appartiene al mondo di attori qak, non supporta direttamente la ricezione di un *event* da parte dell'attore qak *coldstorageservice*.
Possiamo però ovviare al problema grazie alla primitiva **updateResuource**, disponibile in qak. Questa ha lo scopo di inviare un "event" a tutte quelle risorse che stanno osservando l'attore *qak* di interesse.

Lato applicazione Spring, entra quindi in gioco la nuova entità *CoapObserver*, che ha lo scopo di porsi come osservatore dell'attore *coldstorageservice*. Sono necessarie inoltre le *WebSocket* per far giungere l'informazione dall'oggetto java alla pagina html di presentazione.

Di seguito, viene riportato il nuovo diagramma UML delle classi del progetto *Spring*.



Vediamo esserci il nuovo componente **CoapObserver**: osservatore che, ad ogni aggiornamento emesso dal ColdStorageService, avvisa tutte le entità registrate ad esso tramite *WebSocket*.

Per l'aggiornamento automatico della pagina da parte del server utilizziamo le **WebSocket**. Abbiamo infatti un file ws_utils.js che definisce la connessione e gestisce i messaggi in arrivo in maniera opportuna.

- **WebSocketConfiguration**: implementa la classe *WebSocketConfigurer* di `org.springframework.web.socket.config.annotation`.
- **WebSocketHandler**: memorizza le sessioni registrate e alle quali inviare i messaggi.

Visualizzazione del ticket

Abbiamo parlato in analisi del problema di fornire il codice all'utente in maniera esterna all'applicazione.
Pensando in termini di portabilità e semplicità d'uso, abbiamo optato per la generazione di un file **pdf**, contenente il ticket.
La scelta è stata mossa dal fatto che un file pdf è leggibile facilmente su tutti i dispositivi (smartphone, pc, tablet, ...) senza l'ausilio di applicazione esterne o non pre-installate. Inoltre, è facilmente scaricabile dalla pagina web.

> **KEY-POINT:** portabilità e semplicità d'uso tramite un file *.pdf* contente il ticket code.

Qui il server risponde alla richiesta all'endpoint `/generatePdf` con il contenuto di un file pdf, questo contenuto binario è convertito in un `Blob` per poi essere salvato come pdf, a questo scopo si fa utilizzo della libreria `FileSaver.js`

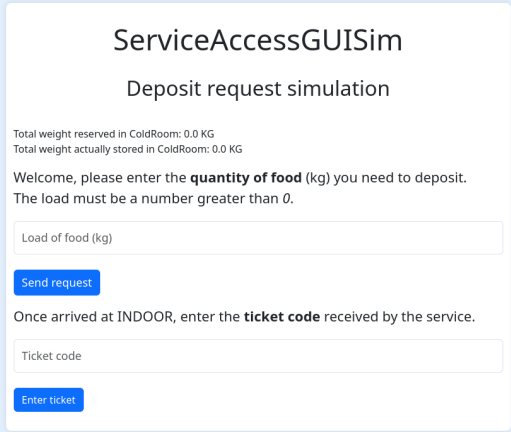
```
function generatePdf(inputValue) {

    //data structure to send to server
    const requestBodyMap = {
        ticketCode: inputValue
    }

    //request to server
    const relativeEndpoint = '/generatePdf';
    fetch(
        relativeEndpoint,
        {
            method: 'POST',
            headers: {
                'Accept': 'application/pdf',
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(requestBodyMap)
        }
    ).then(response => {
        return response.arrayBuffer();
    }).then(data => {
        var blob = new Blob([data], {type: 'application/pdf'});
        saveAs(blob, 'ticket.pdf');
    }).catch(error => {
            console.log(error);
    });
}
```

Front-end

Strettamente correlato alla sezione precedente, mostriamo la nuova grafica della *ServiceAccessGUI*.



Vediamo esserci il nuovo campo per l'inserimento del ticket. L'idea è che questo campo sia sempre disponibile all'interno della pagina, anche senza che la richiesta di deposito sia stata inviata.
In questo modo, diamo sempre la possibilità all'utente di inserire il ticket e questo si concilia perfettamente con il concetto della risorsa esterna contenente il ticket, come già spiegato in analisi del problema.

Testing

I test pianificati nella sezione precedente sono stati implementati tramite un'unità di test JUnit all'interno del progetto *qak*.

Deployment

Per il Deployment si fa riferimento alla sezione dello Sprint1.
Una più attenta analisi verrà rimandata in un documento successivo.

GIT repo: https://github.com/RiccardBarbieri/ColdStorageService

Riccardo Barbieri - riccardo.barbieri11@studio.unibo.it

Leonardo Ruberto - leonardo.ruberto@studio.unibo.it