

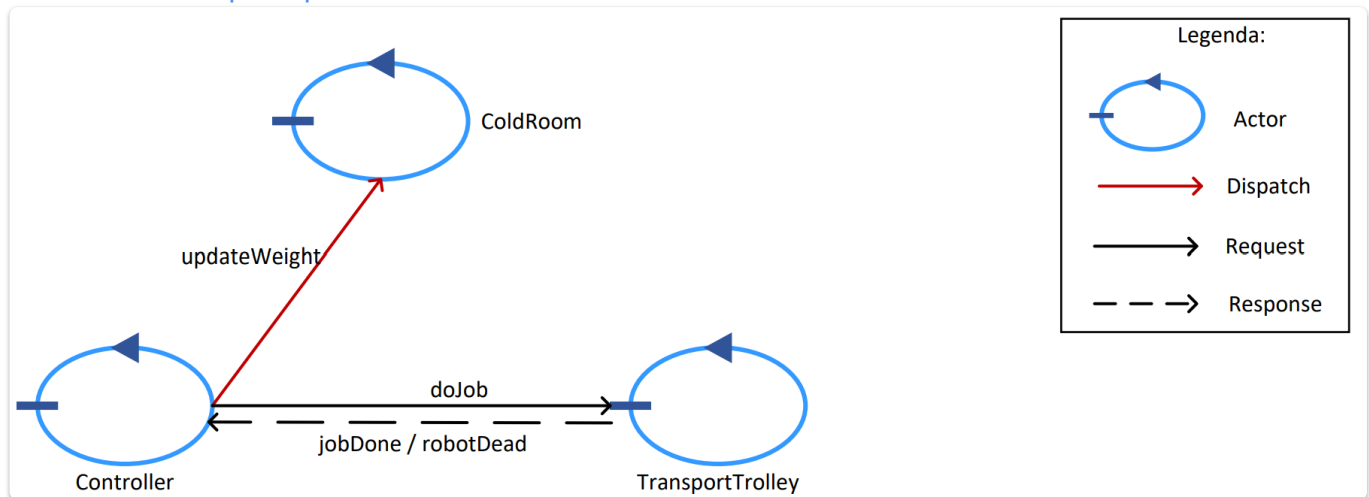
Goal Sprint 1.1

TicketHandler + ServiceAccessGUI

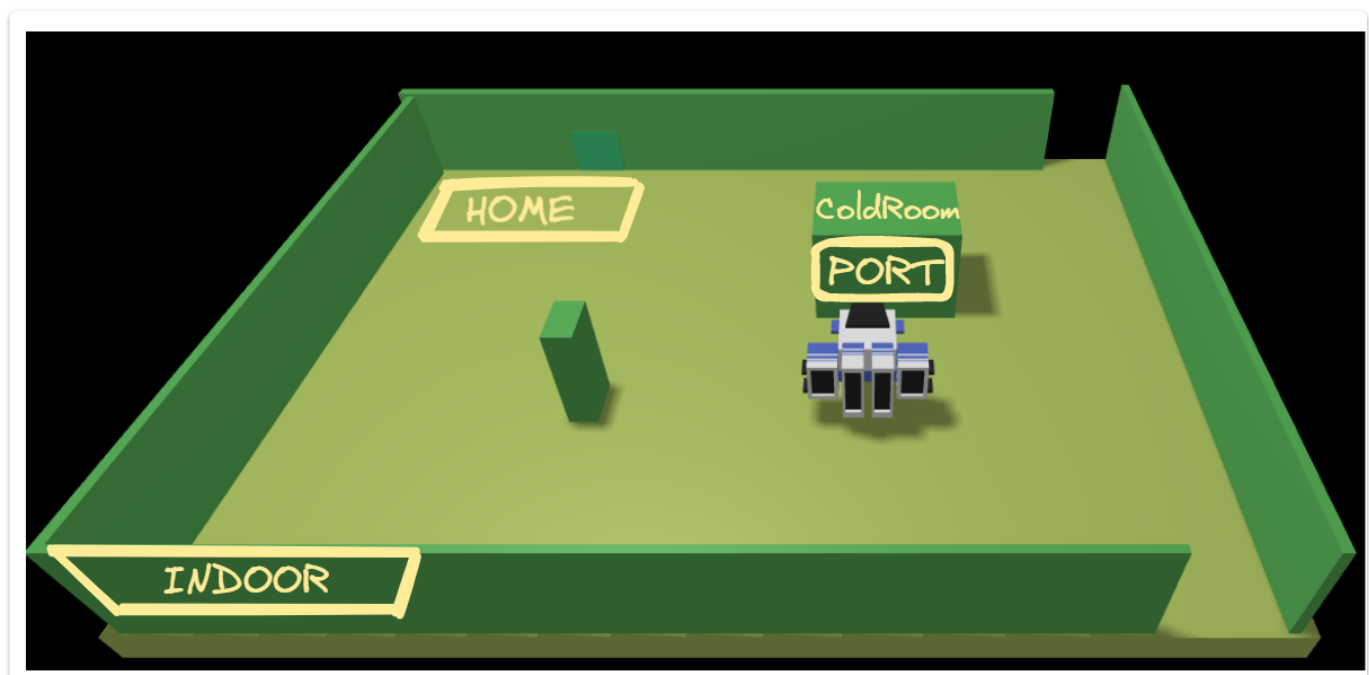
[Descrizione >](#)

In questa seconda parte del primo sprint verrà trattata la logica dei ticket, i componenti e la sicurezza associati.

Modello dello [sprint precedente](#).



Requisiti



[Requisiti](#)

Analisi dei Requisiti

[requisiti sprint 0](#)

Analisi del Problema

Compiti di TicketHandler

TicketHandler si occuperà di:

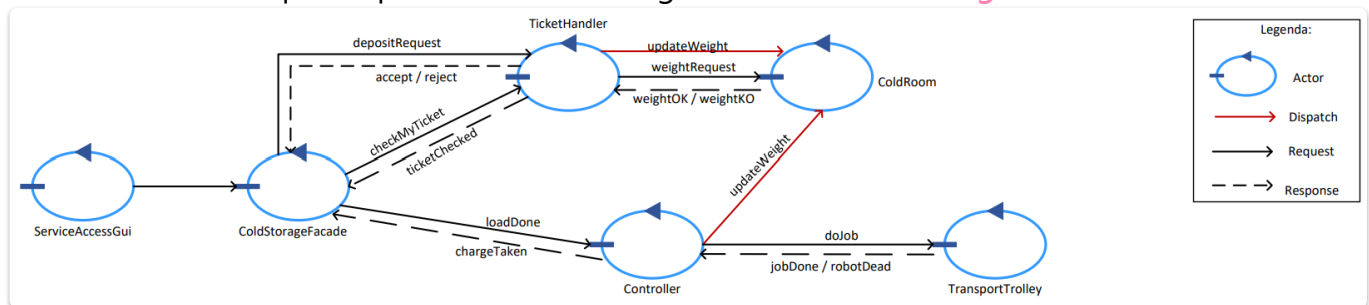
1. verificare se è possibile generare il Ticket richiesto;
2. generare i Ticket;
3. verificare se il Ticket ricevuto è scaduto o meno.

motivazioni >

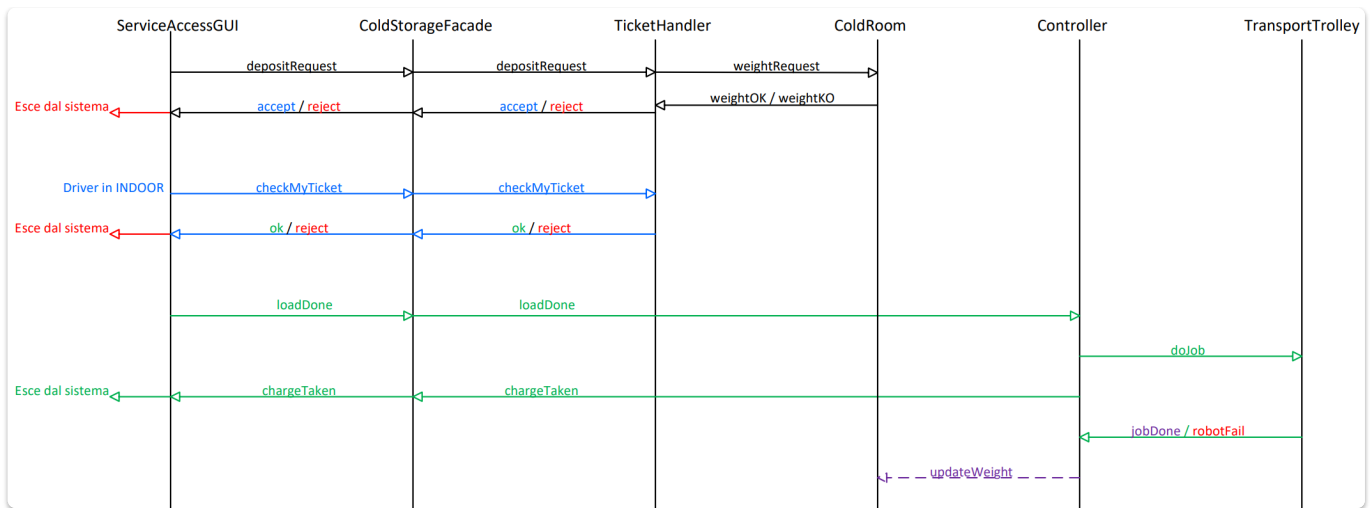
1. Principio di singola responsabilità: Il TicketHandler ha la responsabilità di gestire i Ticket, di conseguenza è corretto che sia quest'ultimo ad occuparsi sia di generare i Ticket richiesti sia di verificarne la validità.
2. Motivi disicurezza: si preferisce assegnare la verifica al TicketHandler, avendo lui tutte le informazioni del driver necessarie per generare e verificare i Ticket stessi (ad esempio l'istante di emissione o l'id del driver associato al ticket).

La separazione di TicketHandler e Controller porta l'utente a dover potenzialmente interagire con due entità diverse del sistema. Decidiamo di introdurre un componente intermedio per nascondere questa complessità dal lato dell'utente secondo il modello del **patter facade**.

Il sistema sarà dunque ampliato secondo la seguente **Architettura logica**:



Protocollo di richiesta e generazione del ticket



1. Inizia con una request/response dal driver verso TicketHandler

```

Request depositRequest : depositRequest(PESO)
Reply accept : accept(TICKET)
Reply reject : reject(NO_PARAM)
  
```

☐ Reject potrebbe essere restituito anche per problemi del sistema, tipo il robot che non va.

1. TicketHandler chiede a ColdRoom se c'è abbastanza spazio;

```

Request weightrequest : weightrequest(PESO)
Reply weightOK : weightOK( NO_PARAM )
Reply weightKO : weightKO( NO_PARAM )
  
```

3. Se c'è abbastanza spazio, ColdRoom aggiorna i propri attributi;
4. Se TicketHandler riceve True genera il ticket e lo invia a ServiceAccessGui, altrimenti rejected;
5. Arrivato in INDOOR, il driver, invia il Ticket a TicketHandler. Il TicketHandler verifica il **TICKETTIME** e restituisce Ok / Rejected;

```

Request checkmyticket : checkmyticket(TICKET)
Reply ticketchecked : ticketchecked(BOOL)
  
```

6. Se la richiesta viene approvata ServiceAccessGUI invia "loaddone" al Controller per notificare che il FridgeTruck è pronto, col peso da scaricare. Il Controller risponderà con "chargetaken".

```
Request loaddone : loaddone(PESO)
Reply   chargetaken : chargetaken(NO_PARAM)
```

Quando viene inviato il "charge taken"?

"chargetaken" viene inviato dal Controller subito dopo la "doJob" associata alla richiesta.

motivazioni >

1. Supponiamo che quest'ultimo scarichi la merce in una piattaforma dedicata, dalla quale il DDR robot preleverà il cibo e lo scaricherà in ColdRoom in uno o più volte a seconda della quantità di materiale dichiarata.
2. Al driver non interessa sapere se il TransportTrolley ha avuto problematiche durante il trasporto del materiale, quindi il "charge taken" può essere inviato prima che il TransportTrolley comunichi al Controller se il carico/scarico in ColdRoom è terminato.

Ricevuta la "chargetaken" il driver può uscire dal sistema considerando la transizione conclusa con successo.

Problema del peso ipotetico

Un driver potrebbe inviare la richiesta di un Ticket prima che un secondo driver, a cui è stato generato un Ticket in precedenza, abbiano scaricato.

Rischio di emettere un ticket per un peso non realmente disponibile nel momento di scarico.

Per risolvere il problema definiamo due pesi diversi:

1. Peso effettivo : quantità (peso) di cibo contenuto in ColdRoom nell'istante attuale.
Aggiornato dopo l'azione del TransportTrolley.
2. Peso promesso : quantità di peso richiesta dai driver tramite Ticket non ancora scaricato, incrementato dopo l'emissione di un ticket e decrementato dopo l'azione del Transport Trolley o a seguito della scadenza della validità di un Ticket.

Useremo la somma dei due pesi per validare o meno una richiesta di emissione ticket.

```
QActor coldroom context ctxcoldstoragearea {
    [#
        var PesoEffettivo = 0
        var PesoPromesso = 0
        var MAXW = 50
    #]
```

```

State updateWeight {
    onMsg ( updateWeight : updateWeight(P_EFF, P_PRO) ) {
        [# PesoEffettivo += P_EFF
        PesoPromesso -= P_PRO
        #]
    }
}

State checkweight {
    onMsg(weightrequest : weightrequest(PESO)){
        [# var PesoRichiesto = PESO #]

        if [# PesoEffettivo + PesoPromesso + PesoRichiesto <= MAXW
#]    {

            [# PesoPromesso += PesoRichiesto #]
            replyTo weightrequest with weightOK : weightOK(
NO_PARAM)

        } else {
            replyTo weightrequest with weightKO : weightKO(
NO_PARAM )
        }
    }
}

```

Problema del peso fantasma

A seguito della scadenza di un Ticket, il Transport Trolley non si farà carico della richiesta e il peso promesso del ticket rimarrà considerato il Cold Room.

Gestione dei Ticket scaduti

L'eliminazione dei ticket scaduti viene fatta per necessità.

All'arrivo di una richiesta di emissione del Ticket, se lo spazio calcolato non fosse sufficiente si verifica il TICKETTIME associato ai Ticket generati e non ancora scaricati.

In presenza di Ticket scaduti allora il TicketHandler procederà ad aggiornare il peso. In questo modo risolviamo anche il [problema del peso fantasma](#).

Quando e da chi vengono aggiornati i pesi in ColdRoom?

1. Terminata l'azione del Transport Trolley, Controller aggiorna i due pesi tramite dispatch. Viene passata la quantità da decrementare dal peso promesso e la quantità da incrementare al peso effettivo (i due valori possono essere diversi a causa del problema del [Driver Distratto](#)).
2. Caso particolare: i pesi sono aggiornati da TicketHandler tramite dispatch "updateWeight" nella [gestione dei ticket scaduti](#).

Sicurezza dei Ticket

Dall'analisi della sicurezza sono apparse le seguenti vulnerabilità:

1. Bisogna assicurarsi che chi richiede il ticket sia l'unico a poterlo usare.
2. Ulteriori dati potrebbero essere visibili ad un utente malevolo (Peso scaricato, ecc...).
3. Un ticket non deve essere riutilizzabile da un qualsiasi utente.
4. Possibile DoS di un utente che richiede troppi ticket e occupa tutto il peso disponibile.
5. Possibile modifica malevola del Ticket dopo l'emissione.

Dei punti definiti, parlando col committente, dovremmo rispettare solo 1 e 3.

soluzioni possibili >

1. Devo assicurarmi che la risposta con il ticket generato venga inviata solo a chi ha fatto la richiesta iniziale e non sia visibile anche agli altri utenti collegati.
2. Avendo già l'elenco dei ticket emessi in TicketHandler per controllare i ticket scaduti posso imporre che ogni ticket che ricevo debba essere dentro quella lista e rimuoverlo appena lo ricevo, in questo modo un ticket non può essere presentato più di una volta.

Gestione dei parametri di sistema

TICKETTIME è un parametro variabile al lancio del sistema. Definiamo un file di configurazione con i valori da caricare al lancio (AppConfig.json):

```
{  
  "TicketTime": "600"  
}
```

```
QActor tickethandler context ctxcoldstoragearea {  
  [# var TICKETTIME = GetTicketTimeFromConfigFile(); #]
```

```
}
```

...

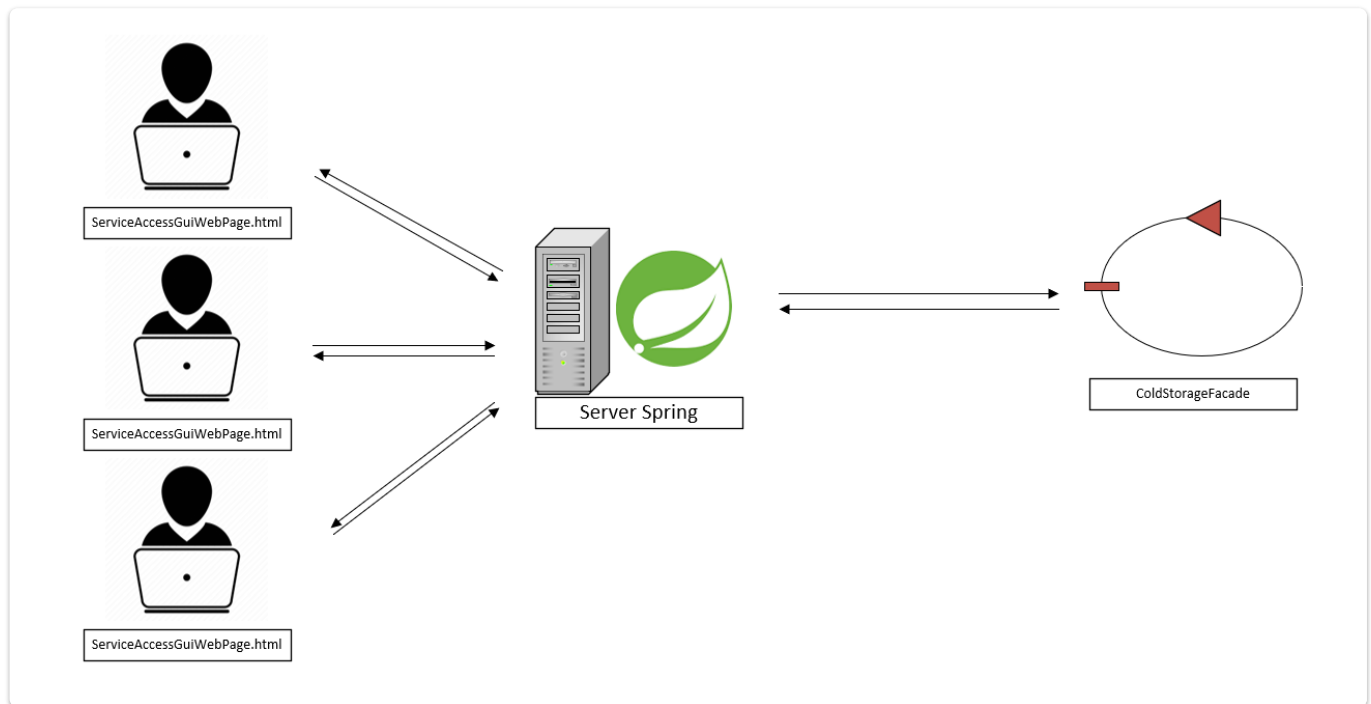
ServiceAccessGUI

Progettare le GUI come attori non è ottimale, dobbiamo progettarela come un componente alieno al sistema che si interfacci con esso.

Per fare ciò ci appoggiamo alla tecnologia di SPRING che permette l'interazione tramite web e la gestione di molti utenti collegati contemporaneamente.

Nello schema iniziale il server Spring prenderà quindi il posto dell'attore ServiceAccessGUI, l'interazione con l'utente avverrà tramite pagine html statiche fornite dal server ad ogni utente che si collega.

NOTA: In questa fase il server spring verrà lanciato localmente al resto del sistema, in futuro potrebbe non essere così. Come gestiamo la conoscenza dell'indirizzo degli attori?



WEB PAGE design

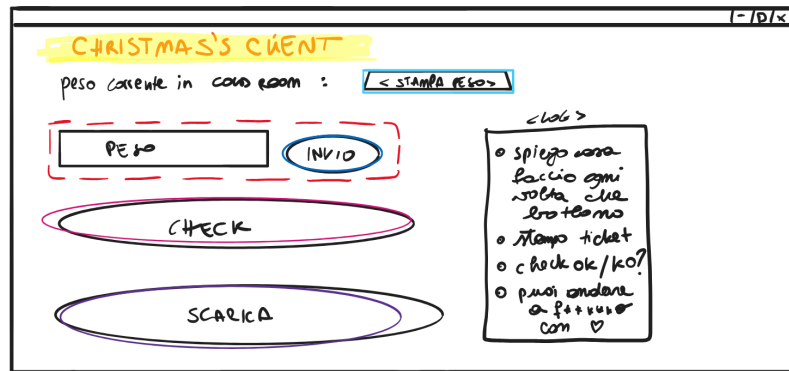
Cosa è il
TICKET ?

stringa con
i seguenti elementi :

- 1) numero
- 2) generation time (quale formato?)
- 3) peso (in kg)

esempio

3 ; 13092023: 12:25 ; 14



o simile
o simile
dopo
richiesta
ticket

o simile
dopo il
check

o aggiornato ogni
volta che una
cosa fa
deposit request o
check my ticket
(se ci riesce E
NON PESA)

Aggiornamento peso in ServiceAccessGUI

La soluzione migliore sarebbe metterlo in ascolto dei cambiamenti a ColdRoom, ColdRoom diventa observable come da analisi preliminari.

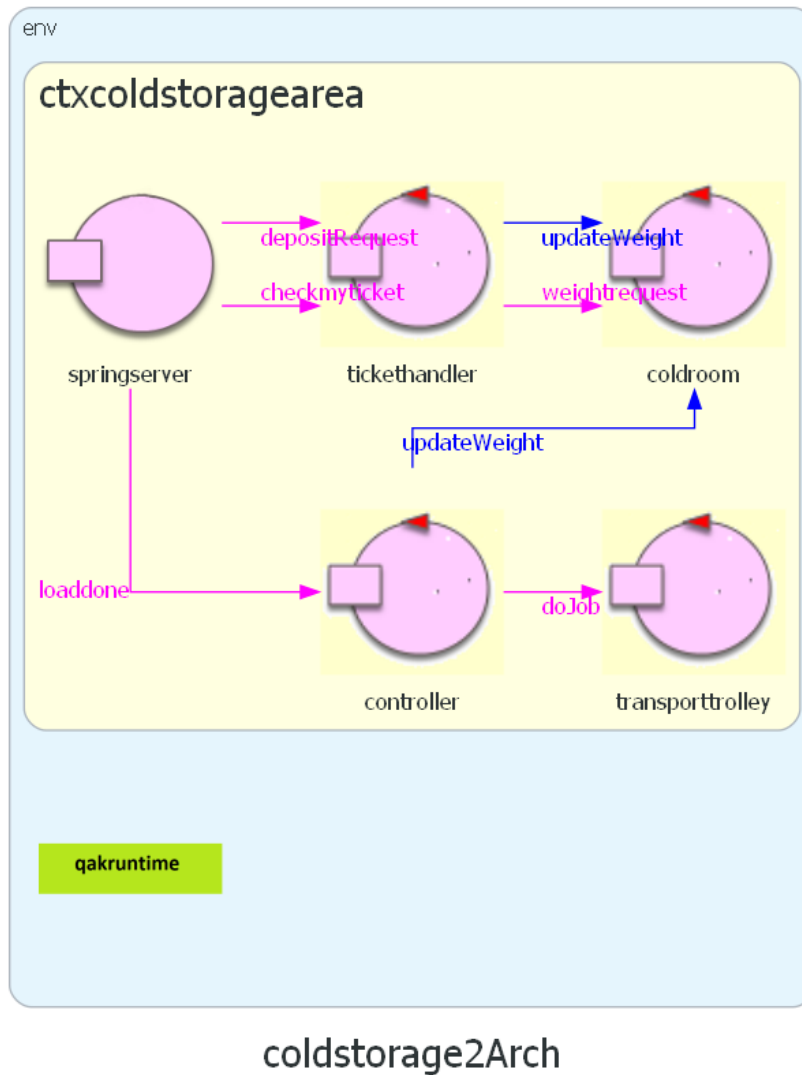
In alternativa Req/Resp di deposit weight fa una richiesta per sapere il peso in coldRoom.

In entrambi i casi usiamo la somma tra peso effettivo e peso promesso.

PROBLEMA: Usando pagine html statiche, anche mantenendo aggiornato il peso corrente nel server spring l'utente deve ricaricare la pagina per visualizzare il nuovo peso.

Si tratta di un problema di poco conto che non giustifica un cambiamento verso pagine html dinamiche e non verrà trattato.

Architettura logica dopo l'analisi del problema



Testing

Durante la fase di testing dovranno essere verificati i seguenti casi:

1. Test del processo in condizioni normali
2. Test con ticket scaduto

3. Test con ticket ripetuto
4. Test con peso superiore al disponibile
5. Controllare che quando il controller aggiorna la ColdRoom il cambiamento di peso risulti nella service access gui.
Ciascuno dei test deve essere superato con più utenti collegati contemporaneamente da uno stesso browser o da browser diversi.

Progettazione

Ticket

Ticket conterrà TIME, PESO e SEQ. La stringa sarà composta da questi 3 valori separati da "_" ed inizierà con "T":

```
int TIME
int PESO
int SEQ

Ticket = "T"+"_"+TIME+"_"+PESO+"_"+SEQ           #esempio di ticket:
T_1697643071_15_0
```

Domanda >

Ci sono ottimi motivi per cui abbiamo deciso di farlo così, se vuole possiamo nascondere il tutto al cliente nell'html.

Definizione messaggi e contesti

```
System coldstorage2

//-----

Request doJob : doJob(KG)
Reply jobdone : jobdone(NO_PARAM)
Reply robotDead : robotDead(NO_PARAM)

//-----

Request depositRequest : depositRequest(PESO)
Reply accept : accept(TICKET)
```

```
Reply reject : reject(NO_PARAM)
```

```
Request weightrequest : weightrequest(PESO)
```

```
Reply weightOK : weightOK( NO_PARAM )
```

```
Reply weightKO : weightKO( NO_PARAM )
```

```
Request checkmyticket : checkmyticket(TICKET)
```

```
Reply ticketchecked : ticketchecked(BOOL)
```

```
Request loaddone : loaddone(PESO)
```

```
Reply chargetaken : chargetaken(NO_PARAM)
```

```
Request getweight : getweight(NO_PARAM)
```

```
Reply currentweight : currentweight(PESO_EFF,PESO_PRO)
```

```
//-----
```

```
Context ctxcoldstoragearea ip [host="localhost" port=8040]
```

```
//-----
```

Controller

Rispetto allo sprint 1 non abbiamo più bisogno della mockRequest e gestiamo il [problema del peso ipotetico](#).

```
QActor controller context ctxcoldstoragearea {  
  
    [# var P_EFF = 0  
      var P_DIC = 0  
    #]  
  
    State s0 initial { printCurrentMessage } Goto work  
  
    State work{  
        println("controller - in attesa") color green  
    } Transition t0 whenRequest loaddone -> startjob  
  
    State startjob {
```

```

        onMsg(loaddone : loaddone(P_EFF, P_DIC) ){
            [# P_EFF = payloadArg(0).toInt()
              P_DIC = payloadArg(1).toInt()
              #]
        }
        replyTo loaddone with chargetaken : chargetaken( NO_PARAM )
        request transporttrolley -m doJob : doJob($P_EFF)
    } Transition endjob whenReply robotDead -> handlerobotdead
      whenReply jobdone -> jobdone

    State jobdone{
        forward coldroom -m updateWeight : updateWeight($P_EFF, $P_PROM)
    } Transition repeat -> work

```

ColdRoom

Rispetto allo sprint precedente ColdRoom deve verificare se è presente abbastanza spazio e rispondere di conseguenza.

UpdateWeight inoltre deve essere aggiornato per gestire il [problema del peso ipotetico](#).

Il peso promesso viene sottratto, se va aumentato fornire **P_PRO negativo**.

```

QActor coldroom context ctxcoldstoragearea {
    [#
        var PesoEffettivo = 0
        var PesoPromesso = 0
        var MAXW = 50
    #]

    State s0 initial {
        printCurrentMessage
    } Goto work

    State work{
        printCurrentMessage
    } Transition update whenMsg updateWeight -> updateWeight
      whenRequest weightrequest -> checkweight
      whenRequest getweight -> returnweight

    State checkweight {
        onMsg(weightrequest : weightrequest(PESO)){

```

```

        [# var PesoRichiesto = payloadArg(0).toInt() #]
        if [# PesoEffettivo + PesoPromesso + PesoRichiesto <= MAXW
#]      {
            [# PesoPromesso += PesoRichiesto #]
            replyTo weightrequest with weightOK : weightOK(
NO_PARAM)
        } else {
            replyTo weightrequest with weightKO : weightKO(
NO_PARAM )
        }
    }
} Goto work

State returnweight{
    onMsg(getweight : getweight(NO_PARAM)){
        replyTo getweight with currentweight :
currentweight($PesoEffettivo, $PesoPromesso)
    }
} Goto work

State updateWeight {
    onMsg ( updateWeight : updateWeight(P_EFF, P_PRO) ) {
        [# PesoEffettivo += payloadArg(0).toInt()
        PesoPromesso -= payloadArg(1).toInt()
        #]
    }
} Goto work
}

```

TicketHandler

```

QActor tickethandler context ctxcoldstoragearea {

    [#
        var TICKETTIME = DomainSystemConfig.getTicketTime();

        var Token = "_"
        var Ticket = ""
        var Peso = 0
        var Sequenza = 0
    #]
}

```

```

        var Accepted = false

        var Tickets = mutableSetOf<String>()
#]

State s0 initial{
    printCurrentMessage
} Goto work

State work {
    printCurrentMessage
} Transition t0 whenRequest depositRequest -> checkforweight
                                whenRequest checkmyticket -> checktheticket

State checkforweight {
    onMsg(depositRequest : depositRequest(PESO)){
        [# Peso = payloadArg(0).toInt() #]
        request coldroom -m weightrequest : weightrequest($Peso)
    }
} Transition t1 whenReply weightKO -> checkdeadlines
                                whenReply weightOK -> returnticket

//prima di rifiutare la richiesta controlliamo se ci sono ticket scaduti
State checkdeadlines{
    [# var SpazioLiberato = 0
        Accepted = false
        var Now = java.util.Date().getTime()/1000

        val it = Tickets.iterator()
    while (it.hasNext()) {
        var CurrentTicket = it.next()
        var TicketTokens = CurrentTicket.split(Token)
        var StartTime = TicketTokens.get(1).toInt()

        if( Now > StartTime + TICKETTIME){ //scaduto
            var PesoTicket =
TicketTokens.get(2).toInt()

            SpazioLiberato += PesoTicket
            it.remove()
        }
    }
}

```

```

    }

    if (SpazioLiberato >= Peso){ //c'è abbastanza spazio adesso
        SpazioLiberato -= Peso
        Accepted = true
    }

    #]
    forward coldroom -m updateWeight : updateWeight(0, $SpazioLiberato)
} Goto returnticket if [# Accepted #] else reject

State reject {
    replyTo depositRequest with reject : reject( reject )
} Goto work

State returnticket {
    [#
        var Now = java.util.Date().getTime()/1000
        Ticket = "T"+"_"+TIME+"_"+PESO+"_"+SEQ
        Sequenza++

        Tickets.add(Ticket)

    #]
    replyTo depositRequest with accept : accept( $Ticket )
} Goto work

State checktheticket {
    onMsg(checkmyticket : checkmyticket(TICKET)){
        [#
            var Ticket = payloadArg(0)
            var Ticketvalid = false;

            if(Tickets.contains(Ticket)){
                var StartTime =
Ticket.split(Token).get(1).toInt()

                var Now = java.util.Date().getTime()/1000
                if( Now < StartTime + TICKETTIME){
                    Tickets.remove(Ticket)
                    Ticketvalid = true
                }
            }
        ]#
    }
}

```

```

        }

        #]
        replyTo checkmyticket with ticketchecked :
ticketchecked($Ticketvalid)
        }
    } Goto work
}

```

Parametrizzazione valori

```

object DomainSystemConfig {
    private var TicketTime : Long = 0;

    init {
        try {
            val config = File("AppConfig.json").readText(Charsets.UTF_8)
            val jsonObject = JSONObject( config );

            TicketTime= jsonObject.getLong("TicketTime")
        } catch (e : Exception) {
            println(e)
        }
    }

    fun getTicketTime() : Long {
        return TicketTime;
    }
}

```

Spring Server

Il server si collegherà agli attori tramite socket

```

public class MessageSender {
    String COLDSTORAGESERVICEIPADDRESS = "127.0.0.1";
    int COLDSTORAGESERVICEPORT = 8040;

    Socket client;
    BufferedReader reader;
    BufferedWriter writer;
}

```



```

public String sendMessage(String msg){
    System.out.print(msg);
    String response = "";
    try{
        this.connectToColdStorageService();
        writer.write(msg);
        writer.flush();
        response = reader.readLine();
    } catch (IOException e){
        e.printStackTrace();
    }
    return response;
}

private void connectToColdStorageService() throws IOException {
    client = new Socket(COLDSTORAGESERVICEIPADDRESS, COLDSTORAGESERVICEPORT);
    writer = new BufferedWriter(new
OutputStreamWriter(client.getOutputStream()));
    reader = new BufferedReader(new
InputStreamReader(client.getInputStream()));
}
}

```

Handler dei bottoni:

```

@RestController
@RequestMapping("/api")
public class ApiController {

    private MessageSender sender = new MessageSender();

    @PostMapping("/weightreq")
    public String weightreq(){
        String msg =
"msg(getweight,request,roberto,coldroom,getweight(NO_PARAM),1)\n";
        return sender.sendMessage(msg);
    }

    @PostMapping("/depositreq")

```

```

    public String depositreq(@RequestParam String fw){
        String msg =
"msg(depositRequest,request,_,tickethandler,depositRequest("+fw+"),1)\n";
        return sender.sendMessage(msg);
    }

    @PostMapping("/checkreq")
    public String checkreq(@RequestParam(name = "ticket") String ticket){
        String msg =
"msg(checkmyticket,request,_,tickethandler,checkmyticket("+ticket+"),1)\n";
        return sender.sendMessage(msg);
    }

    @PostMapping("/loadreq")
    public String loadreq(@RequestParam(name = "weight") String weight){
        String msg = "msg(loaddone,request,_,controller,loaddone("+weight+"),1)\n";
        return sender.sendMessage(msg);
    }
}

```

HTML page

[ServiceAccessGuiWebPage](#)

Deployment