Delivery Checker

Simone Del Gatto

Abstract

Scopo del progetto è la realizzazione di un sistema che supporti il lavoro dei magazzinieri di un'azienda di logistica in cui all'interno del magazzino e in particolare per questo progetto il task principale da considerare è il **controllo dei prodotti in ingresso da parte dei magazzinieri**.

I prodotti arrivano al magazzino organizzati in spedizioni. Ogni spedizione si divide in scatole e ogni scatola contiene poi tipologie di prodotti diversi in diverse quantità. Le informazioni relative a come la spedizione sia composta vengono indicate al momento in cui i prodotti vengono spediti e i magazzinieri sono tenuti al corrente solo delle spedizioni che possono essere lavorate perchè già arrivate in magazzino. Per verificare la spedizione i prodotti devono essere contati all'interno delle scatole e devono corrispondere con quanto dichiarto dal mittente. Eventuali anomalie dovranno essere segnalate tramite nota testuale. I magazzinieri collaborano per la lavorazione delle spedizioni talvolta controllando spedizioni differenti, altre volte in maniera cooperativa sulla stessa, con la possibilità di lavorare sul conteggio delle unità dello stesso prodotto nella stessa scatola.

Il sistema da realizzare deve supportare l'**editing in modalità cooperativa e distribuita dello stato della spedizione** all'arrivo, facendo si che sia comparabile con a quanto è stato dichiarato dal mittente della spedizione.

Indice

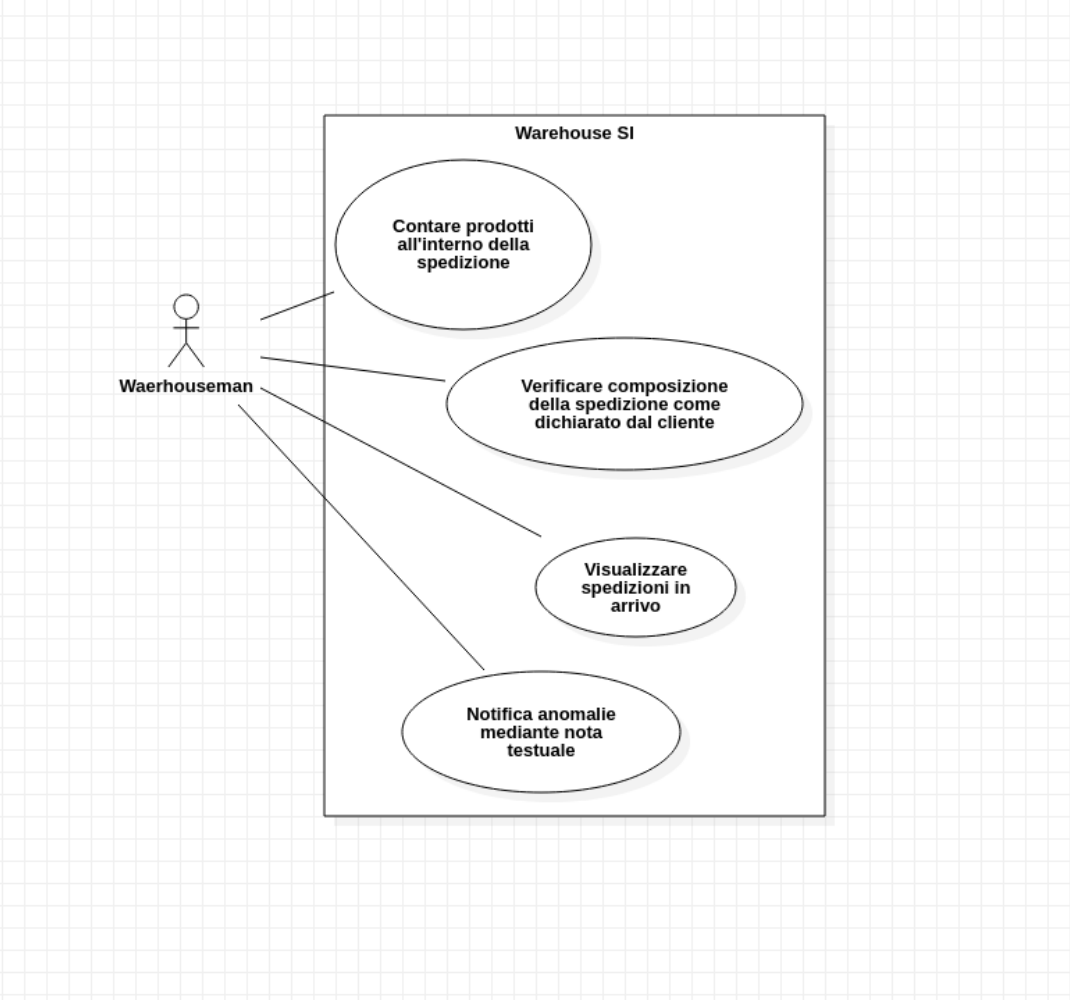
Analisi dei requisiti

In questa sezione verrà descritto il dominio applicativo, le principali funzionalità del sistema da realizzare, chi saranno i suoi utenti e in linea di massima quali sono i prodotti attesi al termine della fase di sviluppo.

Descrizione del problema

Il sistema informativo che si vuole realizzare ha come scopo quello di supportare le attività del magazzino di un'azienda di logistica che si occupa di stoccare prodotti nel proprio magazzino per conto di fornitori terzi, e distribuirli poi successivamente in base alle loro direttive.

Con il presente progetto si intende supportare gli utenti nella **verifica dei prodotti in arrivo, del loro stato e della loro distribuzione fisica all'interno della spedizione**



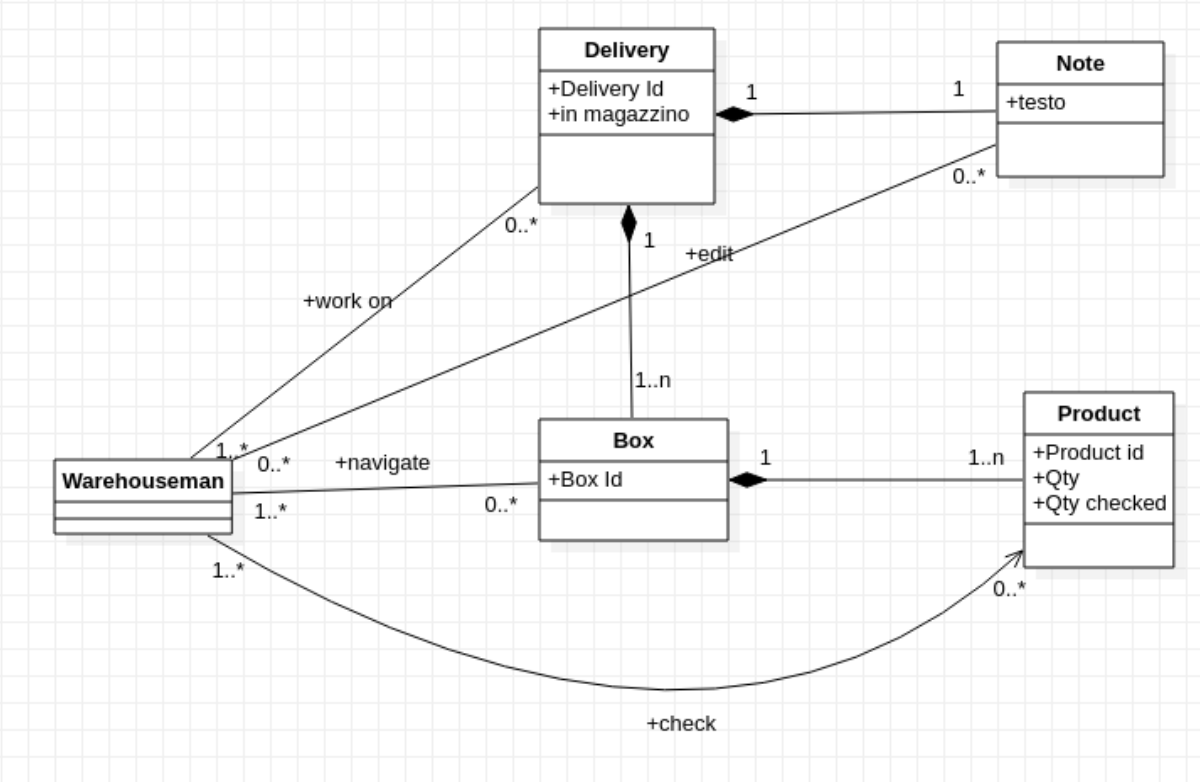
Funzionalità del sistema

I prodotti arrivano al magazzino organizzati in spedizioni. Ogni spedizione si divide in scatole e ogni scatola contiene poi tipologie di prodotti diversi in diverse quantità. Ogni spedizione ha un proprio identificativo così come ogni scatola e ogni prodotto in modo da essere univocamente identificati. Come sia composta effettivamente la spedizione viene deciso nel task 1 e non riguarda le attività del magazzino che viene solo informato delle spedizioni che possono essere lavorate.

Durante la giornata lavorativa vengono verificate diverse spedizioni e ogni magazziniere partecipa alla lavorazione di una o più spedizioni. Sulla stessa spedizione e anche sulla stessa scatola possono lavorare anche più magazzinieri contemporaneamente. Per ogni spedizione e per ogni scatola viene controllato che

la quantità di prodotti di una certa tipologia sia effettivamente presente così come dichiarato dal cliente. In caso di smarrimento o rottura della merce questo deve essere opportunamente.notificato.

Sfruttando le possibilità tecnologiche ogni magazziniere dovrà essere anche a conoscenza di chi eventualmente insieme a lui sta lavorando alla stessa spedizione e su quali spedizioni stanno invece lavorando gli altri magazzinieri.



Dominio applicativo

Scopo del progetto è la realizzazione di una sistema software che supporti le attività descritte in precedenza e che in questa fase prototipale consenta ai magazzinieri di interfacciarsi con il sistema mediante l'utilizzo di uno smartphone/palmare in dotazione ad ognuno. I dispositivi saranno connessi con il sistema tramite internet, e potranno comunicare tra loro attraverso un servizio apposito che gestisce le informazioni relative al magazzino.

Requisiti funzionali

1. **Accesso tramite alias**: non è previsto un layer di autenticazione, ogni magazziniere tuttavia dovrà inserire all'inizio di ogni sessione un proprio *alias* con cui verrà identificato all'interno del sistema.
2. **Scelta della spedizione su cui lavorare**: dovrà essere possibile accedere tramite interfaccia grafica alla lista di spedizioni che è possibile verificare, sceglierne una e iniziare la verifica. Dovrà essere possibile anche nella stessa sessione cambiare spedizione.
3. **Navigazione delle scatole**: il sistema dovrà consentire all'utente tramite interfaccia grafica di navigare il contenuto delle scatole della spedizione prescelta.
4. **Editing della quantità di un prodotto (*)**: deve essere possibile editare la quantità di una tipologia di prodotto all'interno di una scatola. Solo un magazziniere per volta potrà eseguire questa operazione.
5. **Count della quantità di un prodotto (*)**: si potrà aggiungere una unità di un prodotto alla quantità già verificata. Questa operazione deve poter essere eseguita dai magazzinieri anche in parallelo ma non in concomitanza dell'editing della quantità stessa.
6. **Editing nota testuale (*)**: il sistema dovrà permettere tramite interfaccia grafica di editare una nota testuale per notificatre eventuali anomalie nella spedizione. Questa operazione può essere eseguita solamente da un magazziniere per volta.
7. **Visualizzazione dell'attività degli altri magazzinieri (*)**: il sistema dovrà provvedere a rendere noto ad ognuno quali sono gli altri utenti che nello stesso momento stanno utilizzando il sistema, su quali spedizioni stanno lavorando, soprattutto nel caso stiano verificando la stessa spedizione del magazziniere attuale. Eventuali aggiornamenti allo stato della spedizione (conteggio dei prodotti, editing della nota testuale) dovranno essere resi visibili dal sistema e tenuti aggiornati. Deve essere anche possibile visualizzare quali risorse gli altri magazzinieri stanno editando se stanno lavorando sulla stessa spedizione.
8. **Gestione accesso concorrente alle risorse (*)**: il sistema dovrà orchestrare l'accesso degli utenti alle risorse così come previsto dai task precedentemente descritti
9. **Tolleranza al partizionamento (*)**: in caso di mancanza di connessione del dispositivo, il sistema dovrà permettere di:
    - continuare a visualizzare la spedizione corrente
    - aggingere una singola quantità di prodotto con aggiornamento di tutti gli altri client al momento della riconnessione. La riconnessione è intesa come procedura automatica che non deve essere avviata dall'utente ma che viene tentata a fronte di una disconnessione.
10. Lo stato e i dati relativi alle spedizioni devono essere **memorizzate in maniera persistente su una base di dati (*)**

Le funzionalità contrassegnate con (*) saranno denominate *core*: sono tutte quelle che riguardano l'interazione con risorse la cui modellazione rappresenta un aspetto centrale per il dominio applicativo descritto in fase di analisi.

Requisiti non funzionali

- **Affidabilità**: l'affidabilità del codice prodotto dovrà essere garantita dall'utilizzo di un linguaggio tipato per lo sviluppo almeno delle funzionalità *core*. Una suite di test automatici dovrà consentire di monitorare almeno le funzionalità 4 6 7 8 9.
- **Estendibilità e multiprogettualità**: Tutti gli artefatti prodotti dovranno essere distinti ma gestiti in un unico progetto. Le funzionalità *core*, dovanno essere raggruppate, progettate e gestite in modo

che sia semplice riutilizzarle. Ogni applicazione sarà sviluppata come a sè stante ma con possibili dipendenze con il codice sviluppato all'interno del progetto stesso. Il progetto deve essere predisposto per futuri sviluppi ed estensioni.

- **Building automation**:
    - eventuali dipendenze esterne dovranno essere gestite in maniera automatica
    - i test automatici dovranno essere lanciati ed eseguiti *con un solo comando* a prescindere dalla complessità dei servizi aggiuntivi necessari per il loro corretto svolgimento (ad esempio lancio e popolamento di un db di test)
    - easy building and deployment: il sistema dovrà poter essere buildato e lanciato *con un solo comando*

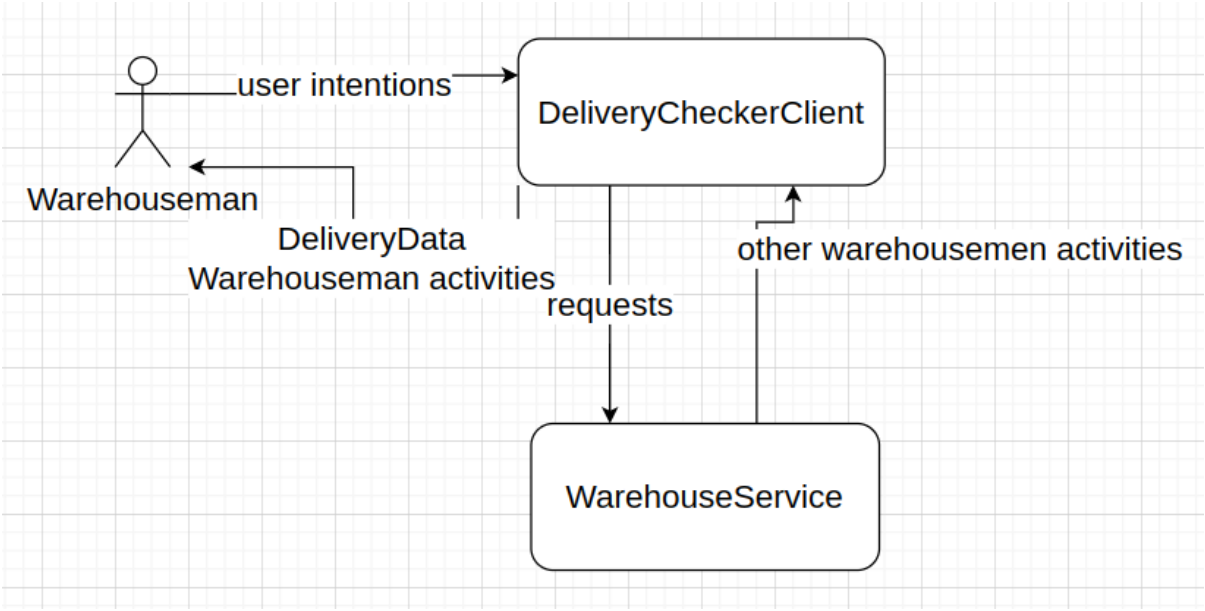
La clausola *con un solo comando* è stata indicata per mantenere basso il livello di difficoltà per accedere alle procedure di building automation.

- Sviluppo delle funzionalità *core* seguendo un processo di **test driven development**.
- **Accesso tramite dispositivo mobile** come unico requisito di usabilità
- **Utilizzo del sistema tramite Browser**: le applicazioni utilizzate dagli utenti saranno accessibili come web application tramite browser
- **Novanta ore di sviluppo** come previsto dal regolamento

Design

In questa sezione si descriverà l'architettura del sistema da realizzare per soddisfare i requisiti descritti, i componenti principali in termini di resposabilità e comportamento, e l'interazione che sussiste tra loro

Architettura



Architettura del sistema

È stato deciso di suddividere il sistema in tre componenti principali, ognuno dei quali con le proprie resposabilità.

- **DeliveryChecker**: ogni magazziniere disporrà, di un proprio palmare come da specifica dei requisiti, da cui attraverso il browser potrà utilizzare la web application DeliveryChecker che realizza le funzionalità previste per la verifica delle spedizioni. L'applicazione metterà a disposizione un'interfaccia grafica con cui il magazziniere potrà interagire con le spedizioni e modificarne lo stato, inoltre potrà visualizzare le attività degli altri utenti, su quali spedizioni stanno lavorando e soprattutto se stanno collaborando alla spedizione su cui sta lavorando il magazziniere stesso.
- **WarehouseService**: è il servizio che gestisce le informazioni relative al magazzino. Principalmente si occupa di:
    - svolgere una funzione di **web server** occupandosi rendendo disponibili le applicazioni DeliveryChecker.
    - offre un **interfaccia per l'accesso e la modifica delle risorse** in particolare per quanto riguarda la risorsa spedizioni, con la sua suddivisione in scatole e prodotti. Deve inoltre amministrare l'accesso concorrente alle risorse stesse.
    - notificare ai client interessati l'eventuale modifica delle risorse.
    - si occupa della gestione del database.
- Un **database** per la memorizzazione persistente dei dati.

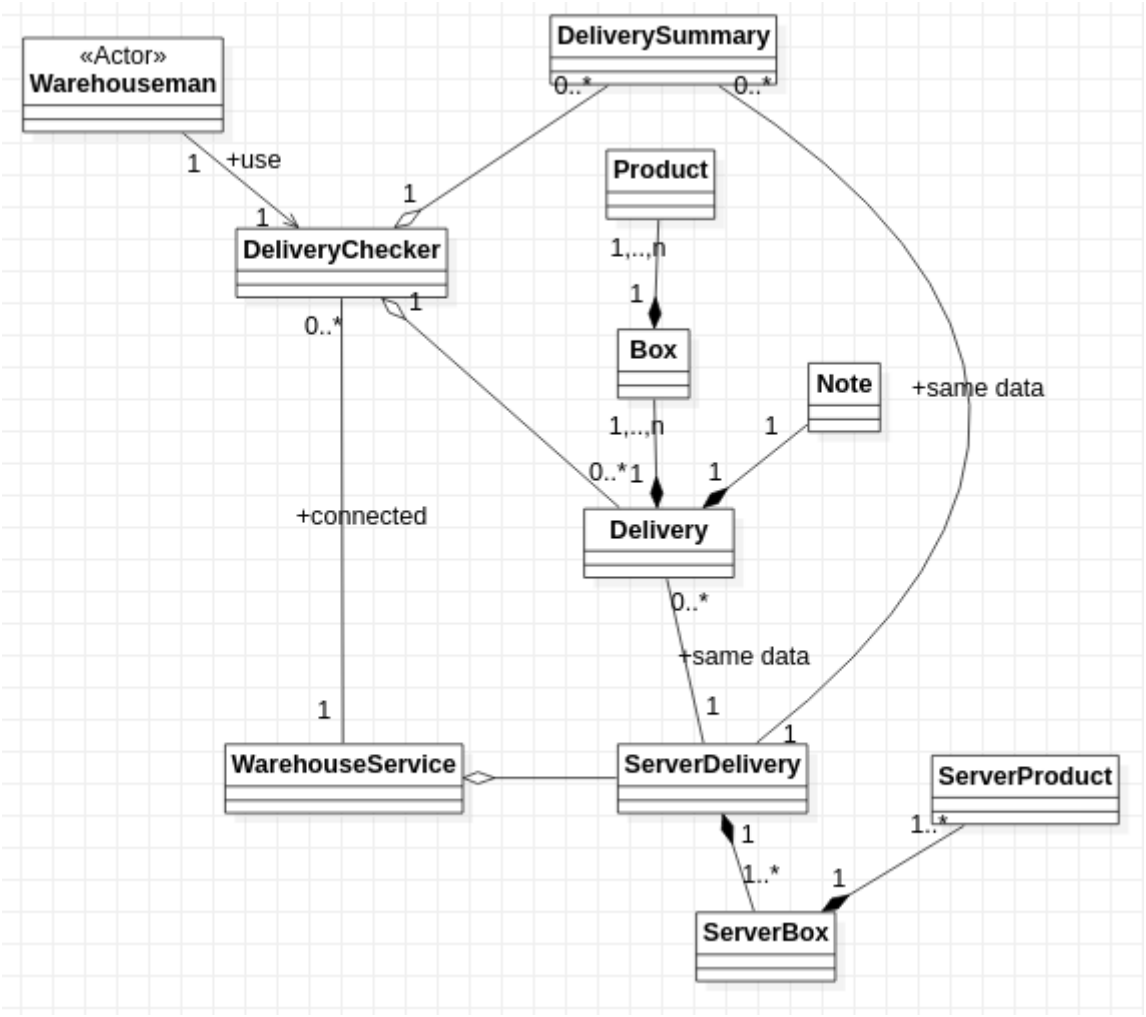
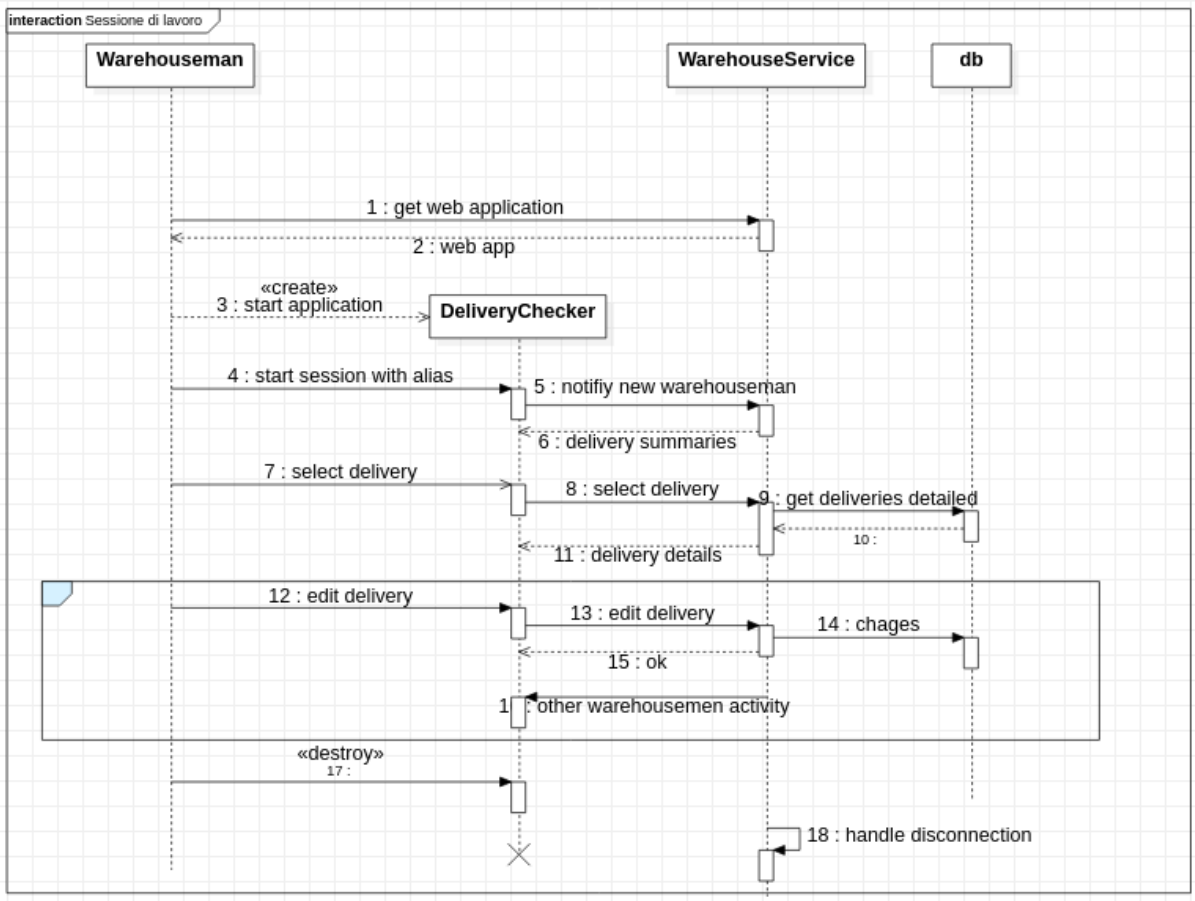


Diagramma delle classi, approfondimento

Le applicazioni DeliveryChecker avranno in gestione una spedizione corrente (Delivery), composta da scatole (Box), prodotti (Product) e nota testuale (Note) che hanno una corrispondenza con una delle spedizioni che è possibile lavorare gestite da WarehouseService (DeliveryServer). La applicazioni DeliverChecker potranno anche vedere una panoramica di tutte le spedizioni presenti in WarehouseService con un numero ridotto di dati rispetto al dettaglio della spedizione corrente (DeliverySummary), per permettere all'utente di scegliere una spedizione su cui lavorare e per mostrare le attività degli altri magazzinieri.

Una volta definiti i componenti è stato progettata la loro modalità di interazione. Vista la natura **distribuita** del sistema, tutte le comunicazioni tra le sue parti sono state modellate come **asincrone**, cioè tali per cui chi attende la risposta da un elemento precedentemente contattato lo fa senza bloccare il suo flusso di esecuzione, con la consapevolezza che la risposta gli sarà data in futuro.

Di seguito il flusso di utilizzo per un singolo magazziniere e le corrispondenti interazioni intersistema.



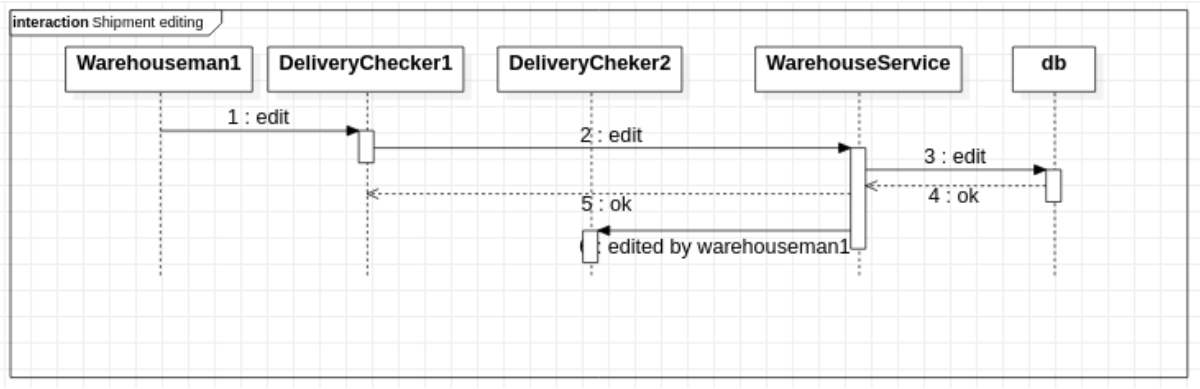
Sessione di lavoro, design

1. All'inizio di una *sessione di lavoro*[1] il magazziniere, utilizzando il proprio dispositivo, accede alla web application DeliveryChecker conoscendo il suo indirizzo di locazione.
2. Inserisce l'alias con cui verrà identificato per tutta la sessione di lavoro. DeliveryChecker invia l'alias a WarehouseService che ne tiene traccia e inizia effettivamente così la sessione.
3. Vengono mostrate successivamente tutte le spedizioni che è possibile lavorare, inviate da WarehouseService a DeliveryChecker in risposta alla notifica dell'alias da parte dell'utente. Insieme alle spedizioni il magazziniere in questa fase conosce anche su quali spedizioni stanno lavorando gli altri magazzinieri.
4. Una volta trovata la spedizione da verificare dovrà essere possibile selezionarla e iniziare la lavorazione. DelveryChecker notificherà a WarehouseService l'intenzione dell'utente, e WarehouseService spedirà tutti i dati realativi alla spedizione selezionata all'applicazione che li renderà disponibili al magazziniere (scatole, prodotti, nota testuale, presenza e attività degli altri magazzinieri)
5. A questo punto il magazziniere può editare la spedizione verificando le quantità dei prodotti e notifca di anomalie mediante nota testuale associata alla spedizione
6. Il qualsiasi momento il magazziniere può decidere di cambiare spedizioni selezionandola a partire dalla lista di quelle disponibili. DeliveryChecker provvederà a notificare WarehouseService e l'attività riprenderà come dal punto 4.
7. In qualsiasi momento il magazziniere può chiudere il browser o la pagina su cui si trova l'applicazione e WarehouseService interpreterà una eventuale disconnessione come l'intenzione di chiudere la sessione.

[1]Con *sessione di lavoro* per un magazziniere si intendono tutte quelle interazioni che vanno dall'accesso a DeliveryChecker fino alla chiusura dell'applicazione data o alla chiusura del browser o dal cambio della

pagina.

Tutte le volte che un magazziniere seleziona una spedizione WarehouseService si occuperà di notificare l'evento alle applicazioni DeliveryChecker connesse i modo che possa essere visualizzato anche dagli altri utenti.



Editing di una risorsa, design

Come da analisi dei requisiti ci sono varie operazioni che possono essere fatte per editare lo stato sulle spedizioni.

Sono previste 3 operazioni sulla risorsa spedizione: editing della nota testuale per notificare le anomalie, editing della quantità presente di una tipologia di prodotto, aggiunta di una unità alla quantità presente in un prodotto. In generale per tutte e tre necessitano delle interazioni descritte dal diagramma di sequenza soprastante:

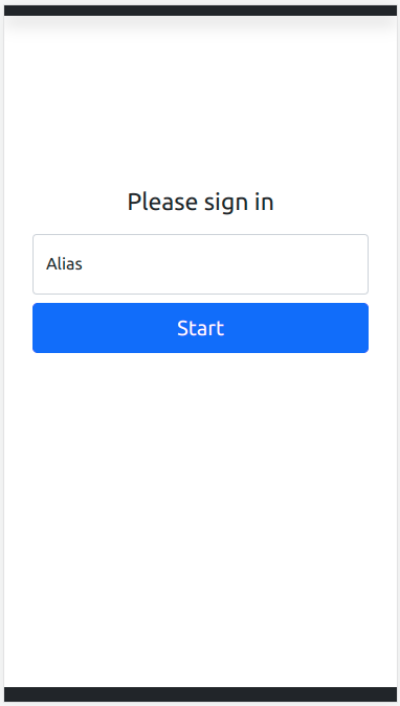
1. il magazziniere esprime la necessità di voler editare la nota/la quantità tramite l'interfaccia grafica di DeliveryChecker, inserendo i dati.
2. DeliveryChecker invierà la richiesta insieme ai data a WarehouseService
3. WarehouseService si occuperà non solo di memorizzare nel db, ma anche di notificare le nuove informazioni alle altre applicazioni attive (come *DeliveryChecker2* nella figura)

Nella sezione successiva si approfondirà ulteriormente quest'ultimo aspetto producendo una soluzione più raffinata e opportuna.

DeliveryChecker

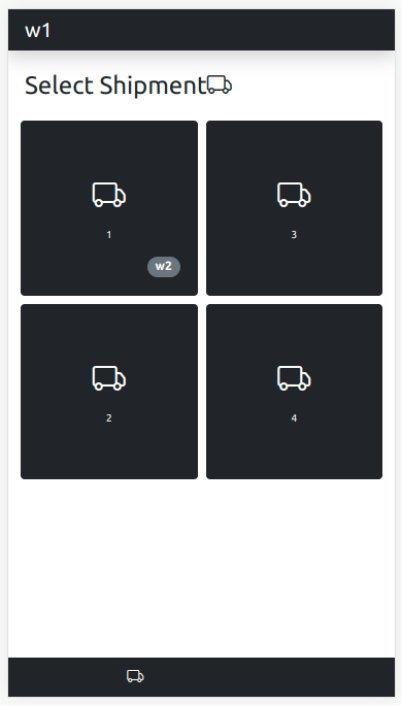
DeliveryChecker è l'applicazione web a cui ogni magazziniere può accedere attraverso il proprio browser conoscendone a priori l'indirizzo. L'applicazione metterà a disposizione un'interfaccia grafica con cui il magazziniere potrà interagire con le spedizioni e modificarne lo stato, inoltre potrà visualizzare le attività degli altri utenti, su quali spedizioni stanno lavorando e soprattutto se stanno collaborando alla spedizione su cui sta lavorando il magazziniere stesso. Per svolgere opportunamente le funzioni previste dovrà connettersi con WarehouseService attraverso il quale potrà sia modificare le spedizioni, sia ricevere aggiornamenti.

Di seguito descritte le schermate messe disposizione da DeliveryChecker.



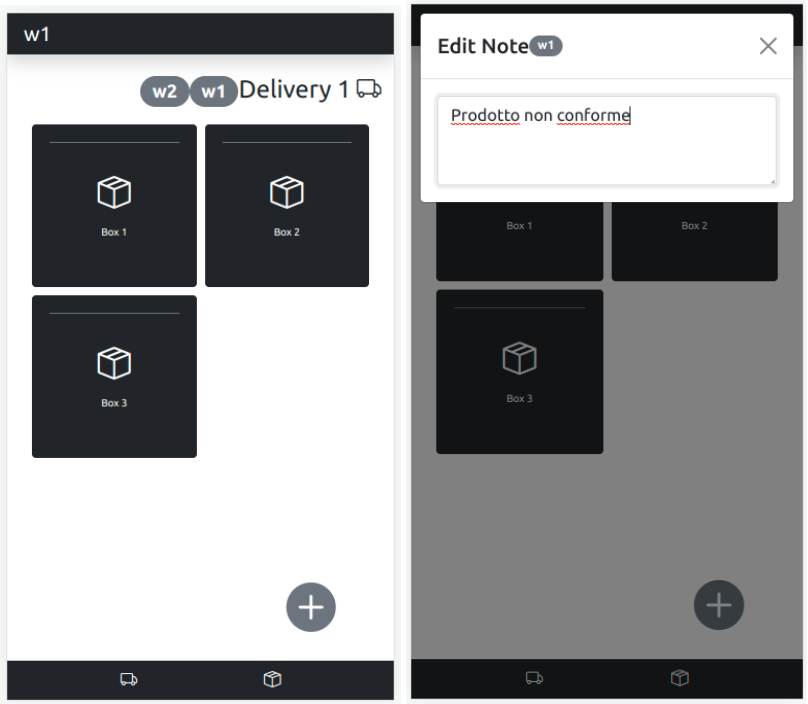
Schermata 1

Nella prima schermata il magazziniere potrà inserire il proprio alias e premendo start inizierà la sessione di lavoro.



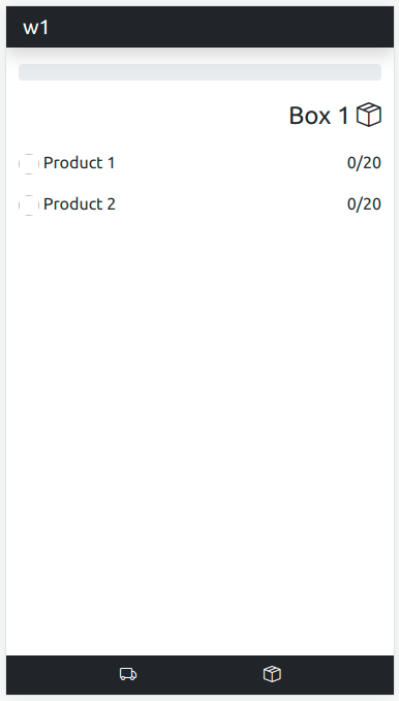
Schermata 2

Dalla prima schermata si passerà alla seconda in cui compare la lista di tutte le spedizioni con associati anche gli alias degli eventuali magazzinieri che ci stanno lavorando. Cliccando su uno dei bottoni DeliveryChecker recupererà da WarehouseService le informazioni relative alla spedizione e si passerà alla schermata successiva.



Schermata 3 e schermata 4

Da questa schermata sarà possibile visualizzare le scatole e scegliere una da editare. Sono visibili tutti i magazzinieri che stanno lavorando sulla spedizione. Tramite il bottone **!** sarà possbile acceder a una modale per editare la nota. In questa schermata verrà introdotta anche il navbar in basso che sarà presente anche nelle altre schermate: tramite il bottone a sinistra con l'icona del camion sarà possible tornare alla schermata delle spedizioni, con il tasto a destra con la scatola si tornerà alla schermata 3. Cliccando sul bottone di una scatola si arriverà alla schermata 5.



Schermata 5

Nella schermata 5 si possono visualizzare tutti i prodotti di una scatola, in particolare il loro codice e la quantità di ogni prodotto effettivamente trovata rispetto a quanto dichiarato dal cliente. Cliccando su una riga della lista rappresentante i prodotti si arriva alla schermata 6.

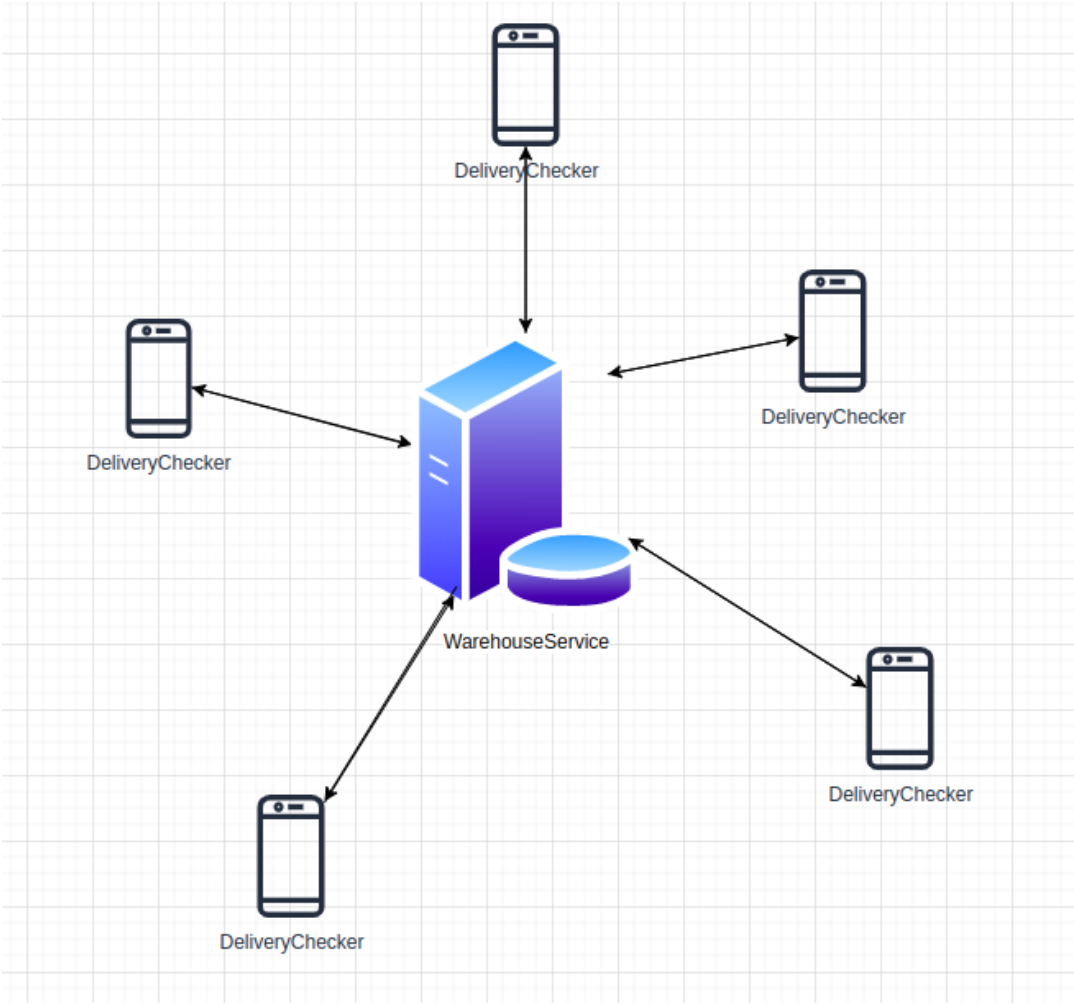
Schermata 6

La schermata 6 consente di fare eseguire le operazioni di editing di un prodotto, sia editando il campo numerico, sia aggiungendo una singola quantità con il campo **+**

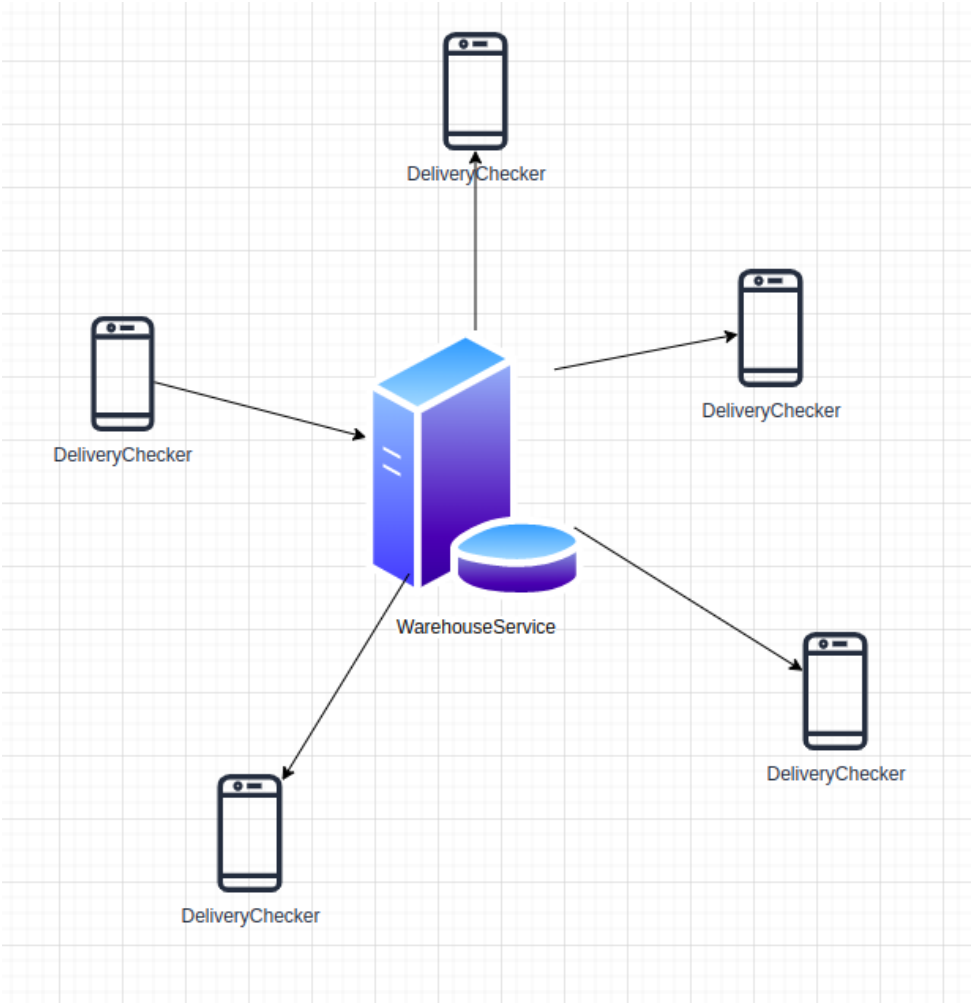
WarehouseService

È il servizio che gestisce le informazioni relative al magazzino.

Principalmente svolge una funzione di **web server** occupandosi rendendo disponibili l'applicazione DeliveryChecker. Ogni applicazione sarà resa disponibile tramite un indirizzo URL del tipo `warehouse_service_address/nome_applicazione`. Nel caso del progetto corrente l'applicazione DeliveryChecker sarà resa disponibile all'indirizzo `warehouse-service-address/delivery-checker`.



Tutte le applicazioni sono servite da WarehouseService



Broadcast di un messaggio inviato da un client

WarehouseService **offre un'interfaccia che esprime le possibili azioni che possono essere effettuate collegandosi al servizio**:

- connessione di un magazziniere al servizio
- disconnessione di un magazziniere dal servizio
- selezione di una spedizione
- editing della quantità di un prodotto
- editing di una nota testuale

A fronte di tutte le operazioni descritte il sistema notifica anche tutti i client connessi, eccetto chi ha inoltrato la richiesta, con un operazione di broadcast. Nel design proposto saranno i client stessi a verificare poi se l'informazione ricevuta sia utile e vada processata oppure no.

In ultimo si occupa della gestione del database.

Design di dettaglio

Modello di concorrenza

Uno degli aspetti del sistema che necessitano di maggior attenzione è l'accesso concorrente alle risorse stesse. Nell'implementazione attuale del sistema con risorse si intendono:

- quantità presente verificata di un prodotto
- nota testuale

Le operazioni di scrittura e lettura non possono essere semplicisticamente modellate come atomiche in quanto coinvolgono una serie di sottotask non atomici e una serie di comunicazioni asincrone. Con *editing* di una risorsa si intende anche il tempo necessario all'utente per decidere e scrivere nell'interfaccia grafica il dato da inserire. Di seguito sono stati descritti quindi alcuni scenari di esecuzione in cui potrebbe esserci perdita di informazione o inconsistenza dei dati.

1. *mag1* comincia la sessione di lavoro usando l'app *delchecker1* nel momento in cui *mag2* usando *delchecker2* sta selezionando *sped1*. Uno dei possibili scenari di esecuzione potrebbe essere il seguente:

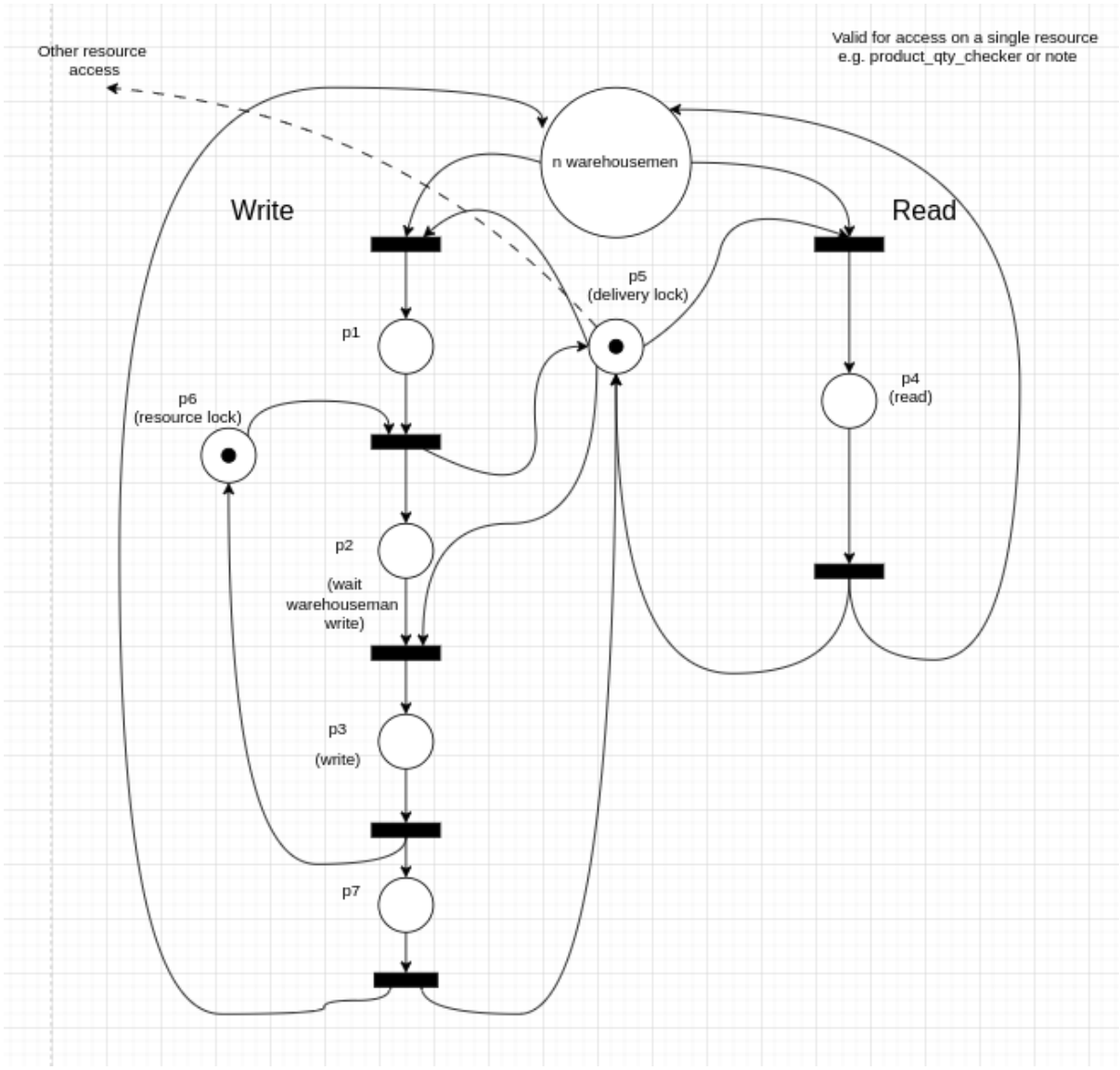
    1. WarehouseService legge raccoglie le informazioni per *delchecker1* leggendo dal database e facendo un merge con i dati in suo possesso relativi alla distribuizione dei vari magazzinieri sulle spedizioni.
    2. WarehouseService aggiunge *mag2* a *sped1*
    3. WarehouseService invia notifica a *delchecker1* che *mag2* è stato aggiunto a *sped1*. *delchecker1* scarta l'informazione perchè non è nelle condizioni di comprenderla (non possiede dati di nessuna spedizione)
    4. WarehouseService invia a *delchecker1* le informazioni raccolte al punto 1 ma ***delcheker1* perde l'informazione che *mag2* ha selezionato *sped1*** l

2. *mag1* edita la quantità di un prodotto usando *delchecker1* nel momento stesso un cui anche *mag2* usando *delchecker2* compie la stessa operazione. Si tratta di una classico problema di accesso a una sezione critica in un sistema concorrente. Il comportamento più opportuno sarebbe **impedire l'editing** di una risorsa nel momento in cui anche un altro utente la sta editando.

3. *mag1* edita la quantità di un prodotto usando *delchecker1* nel momento stesso un cui *mag2* con *delchecker2* sta selezionando la stessa spedizione su cui sta lavorando *mag1*.

    1. WarehouseService legge raccoglie le informazioni per *delchecker1* leggendo dal database e facendo un merge con i dati in suo possesso relativi a quali magazzinieri stanno lavorando sulla stessa spedizione.
    2. WarehouseService edita la quantità del prodotto così come richiesto da *mag1*
    3. WarehouseService invia notifica a *delchecker2* che il prodotto è stato editato. *delchecker2* scarta l'informazione perchè non è nelle condizioni di aggiornare informazioni (non possiede dati di nessuna spedizione)
    4. WarehouseService invia a *delchecker2* i dati raccolti al punto 1 ma ***delcheker2* perde l'informazione sull'editing del prodotto**

A fronte di queste possibili interazioni è stato definito un opportuno modello di concorrenza, per l'accesso **a una singola risorsa** (ad esempio il testo della nota testuale).



Modello di concorrenza per una singola risorsa

```
//pseudocodice

//write
takeLockOnDelivery()
takeLockOnResource() //e.g. takeLockOnDeliveryNote() or takeLockOnProduct()
releaseLockOnDelivery()

... warehouseman type the new content ...

while(warehouse man want type value on the same resource)
    takeLockOnDelivery()

    writeOnDb()
    sendMessageToOtherClients()

    releaseLockOnDelivery()
```

```
takeLockOnDelivery()
releaseLockOnResource()
releaseLockOnDelivery()
```

```
//read
takeLockOnDelivery()
readFromDb()
sendInfo()
releaseLockOnDelivery()
```

Di fatto per accedere a una risorsa in scrittura sarà necessario per prima cosa prendere il lock sulla spedizione e poi sulla risorsa prescelta. Fatto ciò il magazziniere potrà scrivere un nuovo valore richiedendo ancora il lock sulla spedizione e poi rilasciandolo. Nel momento in cui sarà soddisfatto del valore previsto potrà rilasciare il lock anche sulla risorsa stessa, previa acquisizione della spedizione con successivo rilascio. In questo flusso sembra esserci un passaggio di acquisizione del lock sulla risorsa non necessario (si poteva pensare di prendere il lock ad esempio sulla spedizione per tutto il tempo necessario). Tuttavia questo è importante per impedire il verificarsi una condizione di *starvation* per coloro che necessitano di un accesso in lettura, e per tutti quelli che richiedono di effettuare una scrittura di una risorsa diversa ma inerente alla spedizione stessa (deve essere possibile richiedere la modifica della quantità di un prodotto lasciando la possibilità ad un'altro di scrivere sulla nota testuale). Il tempo necessario a un magazziniere per decidere il valore da scrivere è ovviamente di un ordine temporale ben diverso rispetto al tempo di computazione del sistema stesso, e il modello cerca di tenerne conto appunto aggiungendo un passaggio aggiuntivo che possa permette ai lettori e anche agli scrittori di un'altra risorsa di poter svolgere le operazioni richieste.

Per la lettura invece è sufficiente prendere il lock sulla spedizione e poi una volta ottenuti i dati rilasciarlo.

Alla luce dello studio fatto sulla concorrenza lo schema precedente mostra una versione approssimativa del protocollo di comunicazione che è stato quindi esteso e completato, come anche l'interfacca del servizio WarehouseSevice e di conseguenza anche l'applicazione DeliveryChecker.

Protocollo di comunicazione

Come modello di interazioni tra le varie parti del sistema è stato definito un **protocollo a scambio di messaggi su un canale di comunicazione asincrono**. Ad ogni richiesta da parte di DeliveryChecker corrisponderà l'invio asincrono di un messaggio di una certa tipologia con il possibile conseguente inoltro da parte di WarehouseService delle azioni effettuate anche agli altri client. Ogni tipologia di messaggio descrive un'azione che si intede intraprendere se inviata da DeliveryChecker a WarehouseService, oppure a un'azione che è già stata eseguita se il messaggio viene inviato da WarehouseService a DeliveryChecker. Ogni messaggio è corredato da informazioni specifiche utili all'esecuizione o alla notifica dell'azione.

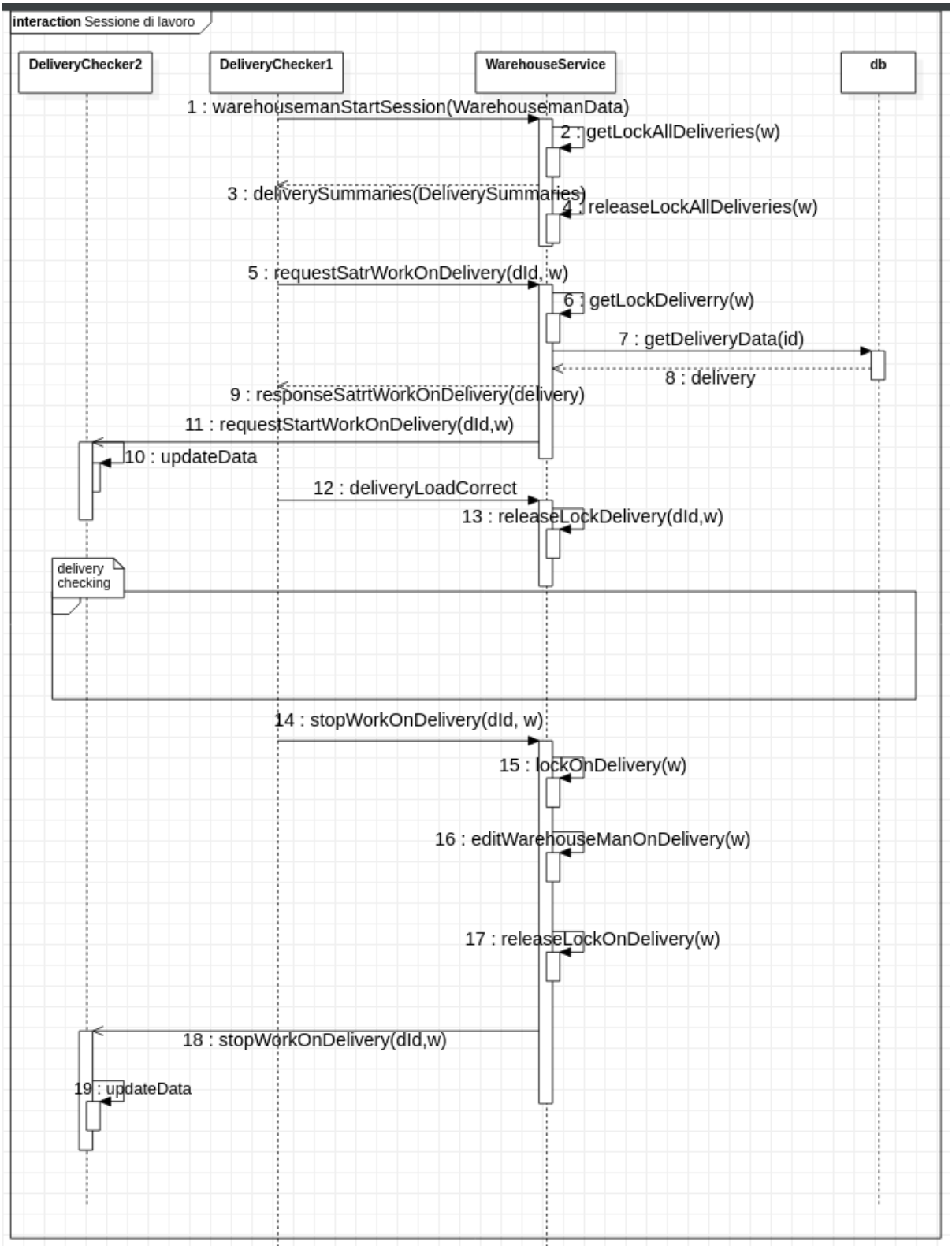
- warehouseManStartSession(WarehousemanData): indica l'inizio di una sessione di lavoro da parte di un magazziniere.
- deliverySummary(summaries: DeliverySummary[]): indica l'ok dell'inizio di una sessione.
- requestStartWorkOnDelivery(alias: WarehousemanData, id: number): indica l'intenzione di voler iniziare a lavorare su una certa spedizione.
- responseStartWorkOnDelivery(delivery: DeliveryData): indica che è possibile cominciare a lavorare su una spedizione.
- stopWorkOnDelivery(alias: WarehousemanData, id: number): indica l'intenzione di non voler più lavorare su una spedizione.
- deliveryLoadCorrectly(id: number): indica che tutti i dati di una spedizione sono stati inviati correttamente.
- addProductOk(deliveryId: number, boxId: number, productId: number, w: WarehousemanData): indica che si vuole aggiungere una unità alla quantità verificata di un prodotto.
- getLockOnProduct(deliveryId: number, boxId: number, productId: number, w: WarehousemanData): indica l'intezione di voler prendere il lock sulla quantità varificata di un prodotto.
- editProductOk(deliveryId: number, boxId: number, productId: number, qty: number, w: WarehousemanData): indica l'intenzione di voler editare la quantità verificata di un prodotto.
- releaseLockOnProduct(deliveryId: number, boxId: number, productId: number, w: WarehousemanData): indica l'intenzione di voler rilasciare il lock sulla quantità verificata di un prodotto.
- getLockOnDeliveryNote(deliveryId: number, w: WarehousemanData): indica l'intenzione di prendere il lock sulla nota di una spedizione
- editDeliveryNote: (deliveryId: number, w: WarehousemanData, deliveryNote: string): indica l'intenzione di voler editare la nota della spedizione
- releaseDeliveryNote: (deliveryId: number, w: WarehousemanData): indica l'intenzione di voler rilasciare il lock sulla nota della spedizione

```
WarehousemanData {
    alias: string
}

ProductData {
    box: number,
    id: number,
    qty: number,
    ok: number
    wQtyLocker: WarehousemanData | undefined
```

```
}

BoxData {
    id: number
    products: ProductData[]
    warehousemen: WarehousemanData[]
}

DeliveryData {
    boxes: BoxData[]
    customerId: number
    id: number,
    warehousemen: WarehousemanData[],
    note?: string
    noteLocker?: WarehousemanData
}

DeliverySummary {
    id: number;
    warehousemen: WarehousemanData[]
}
```

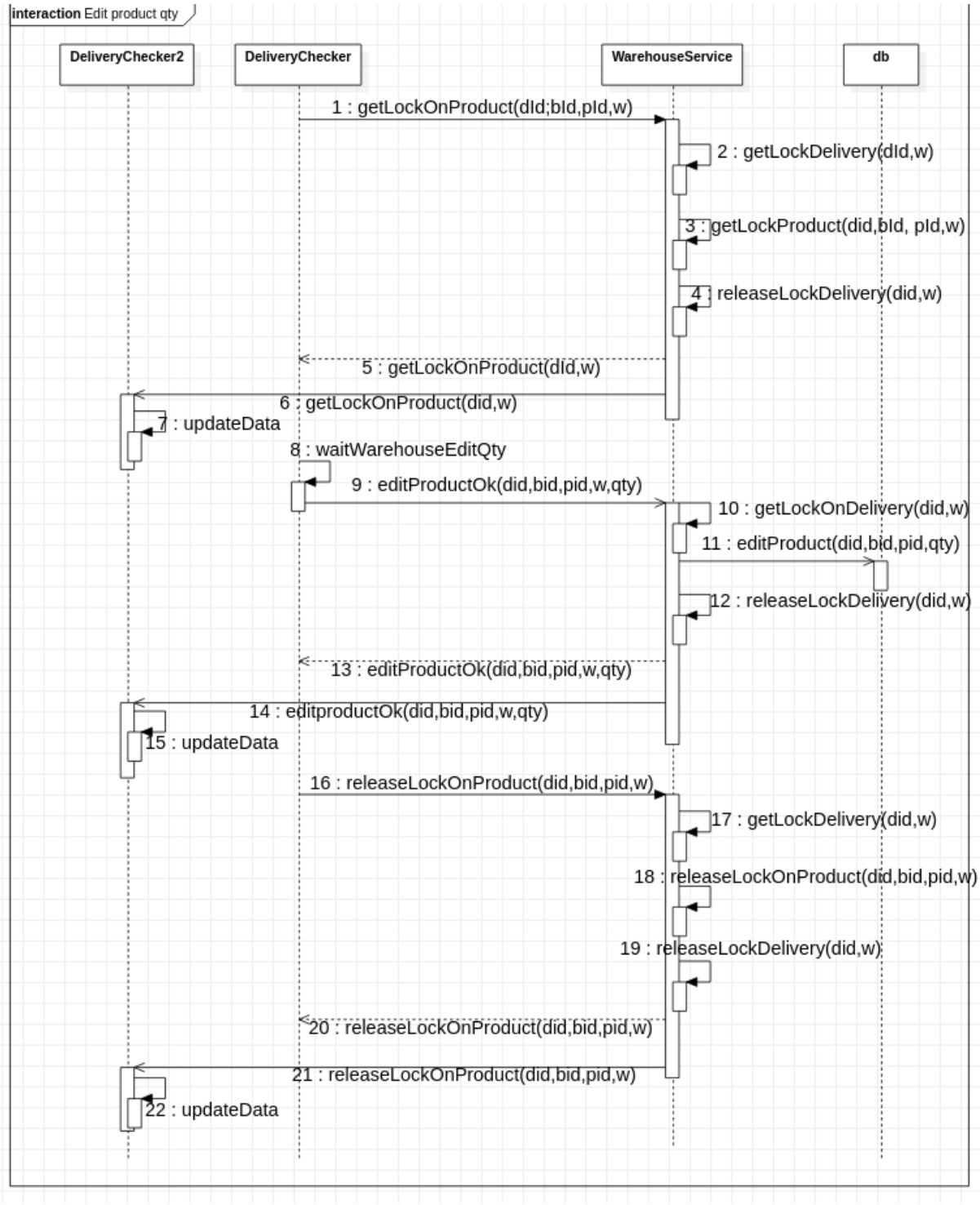
Comportamento complessivo del sistema

Dopo lo studio sul modello di concorrenza e la definizione di un protocollo di comunicazione più preciso è stato modellato il comportamento complessivo del sistema come descritto di seguito. Il flusso operativo e il flusso comunicativo che sono stati progettati propongono **più fasi** rispetto a una soluzione che non gesticsce in alcun modo le eventuali corse critiche: è stata quindi **privilegiata la proprietà di consistenza piuttosto che l'availability**.



Comportamento complessivo del sistema

1. Inizio di una sessione: DeliveryChecker invia a WarehousService il messaggio *warehouseManStartSession* a WarehouseService, richiedendo il lock su tutte le spedizioni utilizzabili. Una volta ottenuto WarehouseService spedisce una lista di DeliverySummary a DeliveryChecker. Viene rilasciato il lock sulle spedizioni.
2. Selezione della spedizione: DeliveryChecker invia a WarehousService il messaggio *requestStartWorkOnDelivery* a WarehouseService, prendendo il lock sulla spedizione richiesta. WarehouseService invia a DeliveryChecker il messaggio *responseStartWorkOnDelivery* con tutti i dati della spedizione richiesta, inoltrando il messaggio *requestStartWorkOnDelivery* a tutti gli altri client che aggiornano così sia la loro lista delle spedizioni che eventualmente la spedizione da loro selezionata. DeliveryChecker risponde una volta caricati i dati a WarehouseService con il messaggio *deliveryLoadCorrectly* e rilascia il lock sulla spedizione.
3. A questo punto il tramite DeliveryChecker il magazziniere può editare le risorse e partecipare alla verifica della spedizione.
4. Per cambiare la spedizione corrente su cui lavorare DeliveryChecker invierà a WarehouseService il messaggio *stopWorkOnDelivery*. Verrà preso il lock sulla spedizione, WarehouseService modificherà la lista di magazzinieri che lavorano sulla spedizione, verrà rilasciato il lock sulla risorsa e inviato, sempre da WarehouseService, il messaggio *stopWorkOnDelivery* anche agli altri client. A seguire DeliveryChecker rieseguirà la procedura descritta al punto 2.



Editing di una risorsa, comportamento del sistema

Per quanto riguarda l'editing della nota testuale, sia della quantità presente di un prodotto invece si eseguirà il seguente flusso operativo (è stato preso come riferimento l'editing della quantità di un prodotto).

1. DeliveryChecker invia a WarehouseService il messaggio *getLockOnProduct*: viene preso il lock sulla quantità del prodotto:
    - viene preso il lock sulla spedizione.
    - viene preso il lock sulla quantità del prodotto.
    - viene rilasciato il lock sulla spedzione.

Poi il messaggio *getLockOnProduct* viene inviato a tutti gli altri client, e in questo caso anche al client mittente con il significato di messaggio di acknolegment. Per migliorare l'interazione utente-applicazione, gli applicativi che ricevono questo messaggio impediranno di richiedere il lock sulla quantità del prodotto in questione.

2. Il magazziniere edita la quantità del prodotto utilizzando l'interfaccia grafica di DeliveryChecker utilizzando il tempo necessario.

3. DeliveryChecker invia la quantità decisa dall'utente con il messaggio *editProductOk*:

    - viene preso il lock sulla spedizione.
    - editata la quantità del prodotto come richiesto scrivendo anche sul databse.
    - viene inoltrato a tutti i client il messaggio editProductOk, anche allo stesso mittente con il significato di acknolegment. Tutti i client processano la modifica.
    - rilasciato il lock sulla spedizione.

Il punto 3 può essere ripetutto tante volte quanto necessario.

4. DeliveryChecker invia a WarehouseService il messaggio *releaseLockOnProduct*:

    - viene preso il lock sulla spedizione.
    - rilasciato il lock sul prodotto.
    - rilasciato il lock sulla spedizione.

Per quanto riguarda l'aggiunta di una singola quantità di un prodotto invece saranno eseguiti i seguenti step:

1. DeliveryChecker registra l'aggiunta del prodotto in locale.
2. DeliveryChecker invia WarehoseService il messaggio *addProductOk*.
3. Su WarehouseService:
    - viene preso il lock sulla spedizione.
    - viene preso il lock sulla quantità.
    - viene editata la quantità scrivendo anche sul db.
    - viene rilasciata la quantità.
    - viene rilasciata la spedizione.
    - tutti i client eccetto il mittente ricevono la modifica tramite messaggio *addProductOk*.

Anche questa operazione risulta disabilitata su DeliveryChecker in caso di editing della quantità del prodotto da parte di un altro magazziniere.

Nel caso di disconnessione da parte di un'applicazione DeliveryChecker invece il comportamento del sistema sarà il seguente:

1. WarehouseService toglierà il lock da tutte le risorse locked dall'applicazione, prendendo e rilasciando prima il lock sulle spedizioni.
2. WarehouseService eventualmente rimuoverà il magazziniere associato dalla spedizione utilizzata, prendendo prima il lock sulla spedizione e poi rilasciandolo.
3. Invierà il messaggio *stopWorkOnDelivery* con riferimento all'eventuale spedizione su cui stava lavorando il magazziniere tramite l'app e gli altri client provvederanno fare quanto fatto da WarehouseService nel punto 1 e 2.

Nel caso sia il server a disconnettersi dall'applicazione sarà impedito al magazziniere di:

- cambiare spedizione su cui lavorare.
- editare la quantità di un prodotto.
- editare la nota testuale.

Sarebbe necessario prendere il lock sulle risorse prima di fare queste operazioni e la disconnessione non consente di gestire opportunamente la concorrenza. Resterà invece possibile aggiungere una unità alla quantità verificata di un prodotto che sarà eseguita in locale e a fronte di una riconnessione sarà notificata anche a WarehouseService. Ciò non sarà effettuato se l'applicazione viene chiusa e poi riaperta prima della riconnessione (ad esempio non sarà effettuato se il magazziniere ricaricherà la pagina prima della riconnessione).

Tecnologie

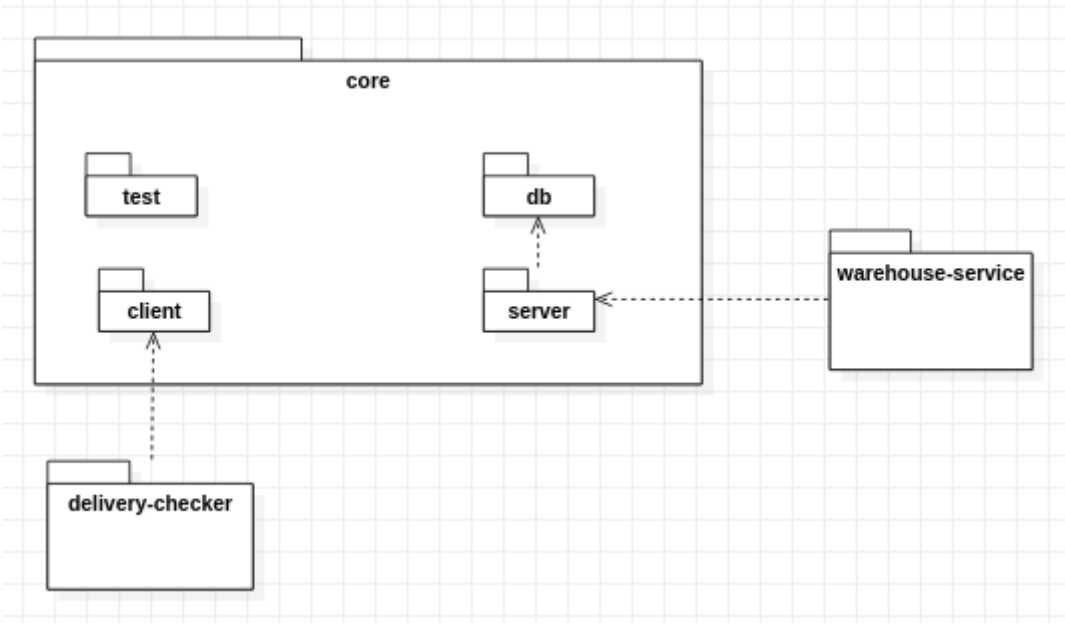
A fronte del design del sistema sono state individuate le tecnologie necessarie per il suo sviluppo.

- per realizzare le comunicazioni tra WarehouseService e le applicazioni DeliveryChecker è stata scelta la tecnologia WebSocket nella sua implementazione realizzata dalla libreria socket.io: permette di realizzare un canale affidabile con comunicazione asincrona a scambio di messaggi adattandosi al modello ideato.
- Typescript, scelto come linguaggio tipato per lo sviluppo delle funzionalità core
- Javascript sia per l'interfaccia di DeliveryChecker sia di WarehouseServer. È una tecnologia con cui si ha molta confidenza ed è molto indicata per implementare velocemente interfacce grafiche. L'ambiente nodejs mette a disposizione anche una suite importante per lo sviluppo server quindi è una tecnologia versatile anche su questo fronte.
- docker e docker-compose come tecnologie per la building automation
- npm per la gestione dell'assetto multiprogettuale con la tecnologia degli npm workspaces e dei task realtivi al progetto (build, test, lancio dei servizi in fase di test)
- Per lo sviluppo dei test ci si è affidati alla libreria javascript Jest per lo stile particolarmente predisposto a una tipologia di test rivolta alla verifica del comportamento (viene infatti spesso utilizzata in ambito web per il test delle interfacce grafiche), per la semplicità di utilizzo e di configurazione anche con Typescript, la chiarezza della documentazione e per il largo utizzo nella comunità javascript (con il benefit di poter accedere quindi alle molte risorse in merito presenti in rete)
- MongoDB per la realizzazione del database e Mongoose per interagire con il db stesso.

Sviluppo

In questa sezione saranno descritti nel dettaglio i componenti principali del sistema e la loro implementazione nei vari moduli. In seguito sarà spiegata la progettazione e l'implementazione della suite di test che ha guidato lo sviluppo del progetto, e delle procedure di automazione utili per il deployment e lo sviluppo stesso.

Moduli



Dipendenze tra i moduli del sistema

Il progetto avrà la seguente struttura:

- core
  - src
    - *client*
    - db
    - *server*
    - test
- *warehouse-service*
- *delivery-checker*

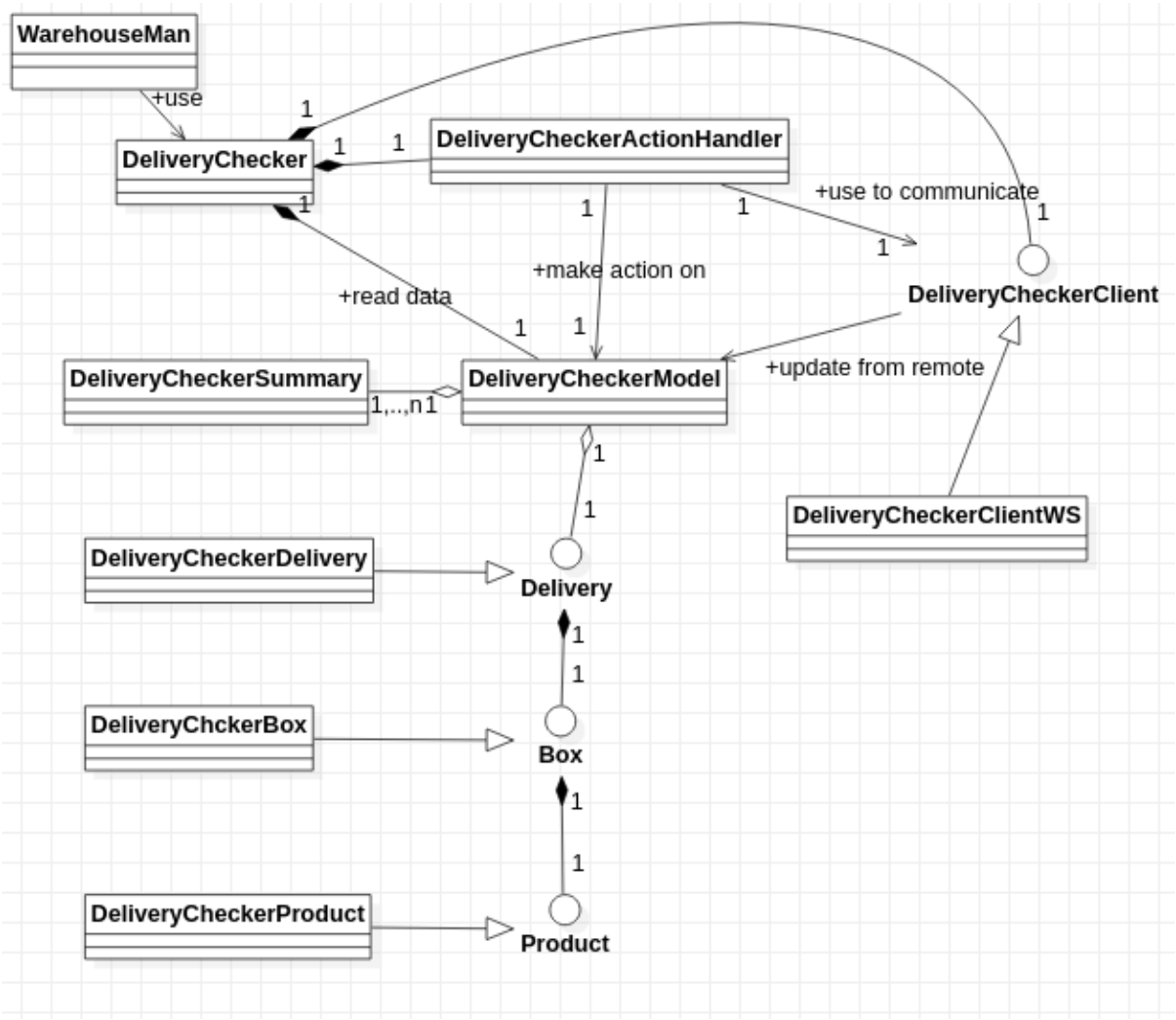
L'implementazione di WarehouseService si trova nel modulo *warehouse-service*, mentre quella di DeliveryChecker nella cartella *delivery-checker*. Come da figura entrambi dipendono dal modulo *core*.

CORE

All'interno del modulo core sono state raggruppate tutte le parti di codice necessarie a implementare quelle funzionalità defintie nell'analisi dei requisiti appunto come *core*, cioè centrali nell'implemetazione corrente del progetto, utili per successivi i sviluppi come punto di partenza per nuove applicazioni, riutilizzabili in più contetesti e che necessitano di avere un alto livello di affidabilità, il che comporta la necessità di essere testati e soprattutto sviluppati con un linguaggio tipato come *Typescript* (il resto del progetto infatti è stato sviluppato in javascript). E stato ritenuto opportuno mettere insieme queste parti di

codice anche perchè si ipotizza che in un futuro si potrebbe effettuare un'operazione di reingegnerizzazione che tenga conto anche di un'analisi più ampia del dominio applicativo. Quindi si potrebbe ricavare un vantaggio dall'avere tutto il codice riguardante la logica applicativa del core business nello stesso punto.

Nella cartella *client* è contenuto il codice per le funzionalità *core* inerenti ai client, in particolare DeliveryChecker, così come la cartella *server* contiene il codice che implementa le funzionalità *core* per WarehouseService.



CORE, DeliveryChecker

DeliveryCheckerActionHandler nel file *DeliveryCheckerActionHandler.ts* è la classe a cui dovrà fare riferimento principalmente DeliveryChecker per prendere atto delle intenzioni del magazziniere e comunicarle a WarehouseService. La sua interfaccia propone una serie di metodi che ricalcano i requisiti funzionali descritti in precedenza. Al suo interno contine il riferimento anche ai dati del magazziniere che sta utilizzando l'applicazione, in paticolare il suo alias. Per comunicare con WarehouseService si avvale dell'utilizzo di una implementazione di *DeliveryCheckerClient*, che permette appunto la comunicazione con il servizio di magazzino. I dati necessari alla comunicazione vengono attini da *DeliveryCheckerModel*, in particolare i dati inerenti alla spedizione corrente.

```
/**
 * Collect warehouseman intentions and communicate them to
 * warehouse service.
```

```
 */
export class DeliveryCheckerWarehousemanActionHandler {

    constructor(w: WarehousemanData, r: DeliveryCheckerClientWS, m:
DeliveryCheckerModel) {
        ...
    }


    /**
     * Start working on a delivery available
     * @param id
     */
    startWorkingOnDelivery(id: number) {
        ...
    }

    /**
     * Get lock on product on the current delivery
     * @param boxId
     * @param productId
     */
    getLockOnProduct(boxId: number, productId: number) {
        ...
    }

    /**
     * Get lock on delivery note
     */
    getLockOnDeliveryNote() {
       ...

    }

    /**
     * Edit product ok
     * @param boxId
     * @param productId
     * @param qty
     */
    editProductOk(boxId: number, productId: number, qty: number): void {
        ...
    }

    /**
     * Edit delivery note
     * @param note
     */
    editDeliveryNote(note: string): void {
        ...
    }

    /**
     * Add one unit ok
```

```
     * @param boxId
     * @param productId
     */
    addProductOk(boxId: number, productId: number) {
        ...
    }

    /**
     * Release lock on product
     * @param boxId
     * @param productId
     */
    releaseLockOnProduct(boxId: number, productId: number) {
        ...
    }

    /**
     * Release lock on delivery note
     */
    releaselockOnDeliveryNote() {
        ...
    }
}
```

DeliveryCheckerClient nel file *DeliveryCheckerClient.ts* definisce l'interfaccia per relazionarsi con WarehouseService permettendo di effettuare le comunicazioni descritte in fase di design. La sua implementazione grazie al protocollo WebSocket la si può trovare nella classe *DeliveryCheckerClientWS* nel file *DeliveryCheckerClientWS.ts*.

```
interface DeliveryCheckerClient {
    /**
     * start to use warehouse service
     * @param warehouseman
     */
    start(warehouseman: WarehousemanData): void

    /**
     * Communicate to warehouse service that a warehouseman start working
on a specific delivery
     * @param id
     * @param w
     */
    startWorkingOnDelivery(id: number, w: WarehousemanData): void

    /**
     * Communicate to warehouse service that a warehouseman stop working on
delivery
     * @param deliveryId
     * @param w
     */
    stopWorkingOnDelivery(deliveryId: number, w: WarehousemanData): void
```

```
    /**
     * Communicate to warehouse service that a warehouseman get lock on a
product checked qty
     * @param deliveryId
     * @param boxId
     * @param productId
     * @param w
     */
    getLockOnProduct(deliveryId: number, boxId: number, productId: number,
w: WarehousemanData): void

    /**
     * Communicate to warehouse service that a warehouseman want get lock
on delivery note
     * @param deliveryId
     * @param w
     */
    getLockOnDeliveryNote(deliveryId: number, w: WarehousemanData): void

    /**
     * Communicate to warehouse service that a warehouseman want add a unit
on product ok/checked qty
     * @param deliveryId
     * @param boxId
     * @param productId
     * @param w
     */
    addProductOk(deliveryId: number, boxId: number, productId: number, w:
WarehousemanData): void

    /**
     * Communicate to warehouse service that a warehouseman want release
lock on product checked qty
     * @param deliveryId
     * @param boxId
     * @param productId
     * @param w
     */
    releaseLockOnProduct(deliveryId: number, boxId: number, productId:
number, w: WarehousemanData): void

    /**
     * Communicate to warehouse service that a warehouseman want release
lock on a delivery note
     * @param id
     * @param w
     */
    releaseLockOnNote(id: number, w: WarehousemanData): void

    /**
     * Communicate to warehouse service that a warehouseman want release
lock on  a delivery notes
     * @param deliveryId
```

```
     * @param boxId
     * @param productId
     * @param qty
     * @param w
     */
    editProductOk(deliveryId: number, boxId: number, productId: number,
qty: number, w: WarehousemanData): void

    /**
     * Communicate to warehouse service that a warehouseman want edit a
delivery notes with this data
     * @param deliveryId
     * @param note
     * @param w
     */
    editDeliveryNote(deliveryId: number, note: string, w:
WarehousemanData): void

    /**
     * set wat happens if there is a disconnection with the server
     * @param fn
     */
    setOnDisconnection(fn: () => void): void

    /**
     * check if is possible to communicate with the service
     */
    isReady() : boolean,

    /**
     * Close the communication with the service and eventually set the
function to execute after disconnection
     * @param fn
     */
    close(fn?: () => void): void
```

DeliveryCheckerClientWS conserva un riferimento al *DeliveryCheckerModel* per poter notificare i cambiamenti che riceve da remoto.

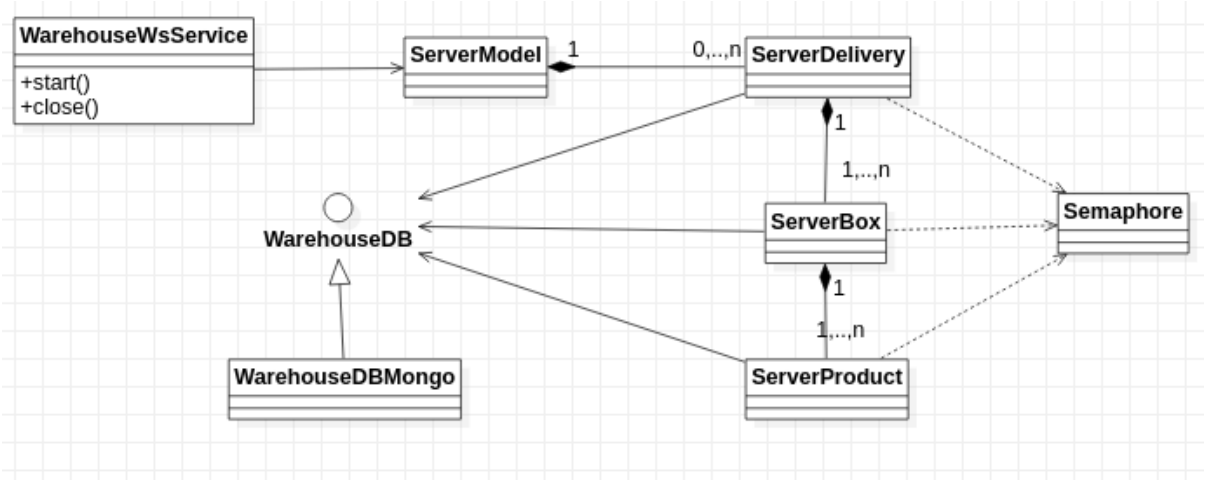
È da notificare la procedura eseguita da *DeliveryChekcerClientWS* a fronte di una disconnessione temporanea da WarehouseService. Questa è stata pensata per aumentare la tolleranza al partizionamento del client, e per permettere all'utente almeno di aggiungere una unità alla quantità di un prodotto anche in caso di non connessione. Per fare ciò viene sfruttata la feature della libreria socket.io di riconnessione automatica e del lifecycle previsto dal WebSocket client di questa libreria: a fronte di una disconnessione il client prova a riconnettersi al server, e nel caso riesca, invia in ordine i messaggi che non ha potuto inviare in mancanza di connessione. La procedura di riconnessione come richiesto nei requisiti è automatica, ed è possibile effettuare un cambiamento in locale della quantità di un prodotto aggiungendo una unità con la garanzia che in futuro ci sarà un cambiamento anche remoto in qualisiasi situazione anche a fronte di una temporanea disconnessione.

DeliveryCheckerModel invece rappresenta un'interfaccia agli elementi del dominio applicativo con cui può interagire il magazziniere: la spedizione corrente (*Delivery*), i riassunti delle spedizioni che possono essere selezionate, le scatole (*Box*), i prodotti (*Prodotti*). Gli oggetti di queste classi presentano un'interfaccia che consente sostanzialmente di reperire e aggiornare i dati utili a descrivere l'oggetto stesso (ad esempio nel prodotto troveremo un codice che lo identifica, la quantità confermata del prodotto, ...). Ogni oggetto è stata concepito come un *osservabile*, e ad ogni cambiamento ne notifica l'avvenimento. Un eventuale osservatore interessato può richiedere i dati dell'oggetto in questione una volta ricevuta la notifica. *DeliveryCheckerModel* è esso stesso un osservabile che notifica il caricamento di nuove spedizioni disponibli e dell'arrivo dei dati della spedizione corrente. I dati della spedizione corrente saranno ovvimente caricati solo nel momento stesso in viene selezionata la spedizione, e vengono trasmessi da WarehouseServer fino DeliveryChecker.

```
class DeliveryCheckerModel {

    /**
    * add summary of a new available delivery
    * @param s
    */
    addSummary(s: DeliverySummary): void {
        ...
    }

    /**
     * set the available delivery summaries
     * @param s
     */
    setSummaries(s: DeliverySummary[]): void {
        ...
    }

    /**
     * set in the summary that a warehouseman start work on a delivery.
     * Add also this information if the delivery is the current delivery
     * @param w
     * @param id
     */
    warehousemanStartWorkOnDelivery(w: WarehousemanData, id: number) {
        ...
    }

    /**
     * Warehouseman stop working on delivery
     * @param w
     * @param id
     */
    warehousemanStopWorkOnDelivery(w: WarehousemanData, id: number) {
        ...
    }

    /**
     * get the available delivery summaries
```

```
     * @returns
     */
    getDeliverySummaries(): DeliverySummary[] {
        ...
    }

    /**
     * Set the current delivery data
     * @param d
     */
    loadCurrentDelivery(d: DeliveryData) {
        ...
    }

    /**
     * Get current delivery data
     * @returns
     */
    getCurrentDeliveryData(): DeliveryData | undefined {
        ...
    }

    /**
    * Get current delivery
    * @returns
    */
    getCurrentDelivery(): Delivery | undefined {
        ...
    }

    /**
     * Add one unit of product ok
     */
    addProductOk(deliveryId: number, boxId: number, productId: number) {
        ...
    }

    /**
     * Release all lock on the current delivery
     */
    releaseAllLock(): void {
        ...
    }


    /**
     * Listen update from model
     * @param f
     */
    listenUpdate(f: () => void): void {
     ...
    }

    /**
```

```
        * Stop listen update from model
        * @param f
        */
       offListen(f: () => void): void {
           ...
       }
   }
```



CORE, WarehouseService

Nella cartella *server* si trova la classe *WarehouseWSService* che realizza l'interfaccia WebSocket di WarehouseService e *ServerModel* che mette a diposizione gli oggetti che fanno parte del dominio applicativo che implementano la gestione della concorrenza e l'interfacciamento con il database.

Di fatto ogni procedura è così eseguita:

- *WareshouseWSService* riceve un messaggio tramite protocollo WebSocket da un'applicazione DeliveryChecker.
- *WarehouseService* a seconda del messaggio ricevuto richiede a *ServerModel* di eseguire un'azione e lo fa sfruttando i dati inoltrati nel messaggio.
- *ServerModel* esegue l'operazione e restituisce il risultato.
- *WarehouseWSService* comunica il risultato.

Per implementare il comportamento di broadcasting dei messaggio come definito nella sezione di desig sono state sfruttate le funzionalità della libreria socket-io `socket.broadcast.emit(message)` e `io.emit(message)`, in cui `socket` è il riferimento a una socket, mentre `io` è il riferimento al WebSocket server. Il primo metodo consente di inviare il messaggio a tutte le socket connesse eccetto quella di riferimento, e questo è usato ad esempio per gestire il messaggio `addProduct`. Il secondo invece è utilizzato in tutti i casi in cui è necessario fare il braodcast di un messaggio.

Per gestire l'accesso concorrente alle risorse è stato sviluppato un *Semaphore*, che permette a un qualcuno di un certo tipo T di richiedere il lock di una risorsa ed eventualemente di rilasciarla. Per sviluppare il *Semaphore* è stata sfruttata la meccanica delle Promise messa a disposizione da Typescript, presentata in maniera più elegante grazie alla sintassi async await. L'interfaccia di *Semaphore* mette a disposizione un metodo asyncrono, `async lock(who: T): Promise<void>`, che restituisce una Promise che viene risolta nel momento in cui è possibile prendere il lock per chi lo ha richiesto. Il semaforo segue una logica FIFO nella concessione dei permessi, utilizzando un meccanismo di ticketing.

Come si può vedere dal listato di codice sottostante la procedura per prendere il lock è così implementata:

- Se nessuno ha il lock sulla risorsa chi lo ha richiesto viene segnato come locker
- Se invece qualcun'altro ha il lock sulla risorsa viene creata una nova Promise come risultato che sarà risolta solamente quando un emitter di eventi interno al semaforo dichiarerà che la risorsa è stato rilasciata, e che è il turno di chi ha fatto la richiesta, cioè colui che ha come ticket il numero pubblicato dall'emettitore. L'emitter pubblica in realtà una sequenza progressiva di numeri, e anche i ticket sono stati implementati come una squenza successiva di cui si tiene traccia del prossimo numero da distribuire grazie a un counter interno al semaforo. La Promise viene stoccata nella lista delle richieste e restituita come risultato.

```
    async lock(who: T): Promise<void> {
        if (this.locker != undefined) {
            let t = this.ticket
            let p = new Promise<void>((res, rej) => {
                this.e.addListener('next', (ct: number) => {
                    if (t == ct) {
                        this.locker = who
                        res()
                    }
                })
            })
            this.requests.push({ who: who, promise: p })
            this.ticket++
            return p
        } else {
            this.locker = who
        }
    }
```

Per rilasciare la risorsa invece l'interfaccia propone un metodo `release(): boolean` che restituisce `true` nel caso ci siano ancora altre richieste di lock in attesa, `false` altrimenti. La procedura lascia libero il programmatore e permettere il rilascio della risorsa anche se non richiesta dalla stessa entità che ne detiene il lock: in caso sia necessario un controllo è possibile farlo esternamente utilizzando il metodo proposto dall'interfaccia `getLocker(): T | undefined` che dà come risultato il detentore della risorsa.

La procedura funziona così:

- Se non ci sono altre richieste rimuovi il locker e restituisci `false`
- Altrimenti notifica il prossimo che deve prendere il lock richiedendo la propagazione di un evento `next` corredato dal prossimo numero di ticket da chiamare all'event emitter interno al semaforo.

```
    /**
     * Release the resource. Could be necessary from outside check who have
the lock
     * @returns
     */
    release(): boolean {
        if (this.requests.length == 0) {
```

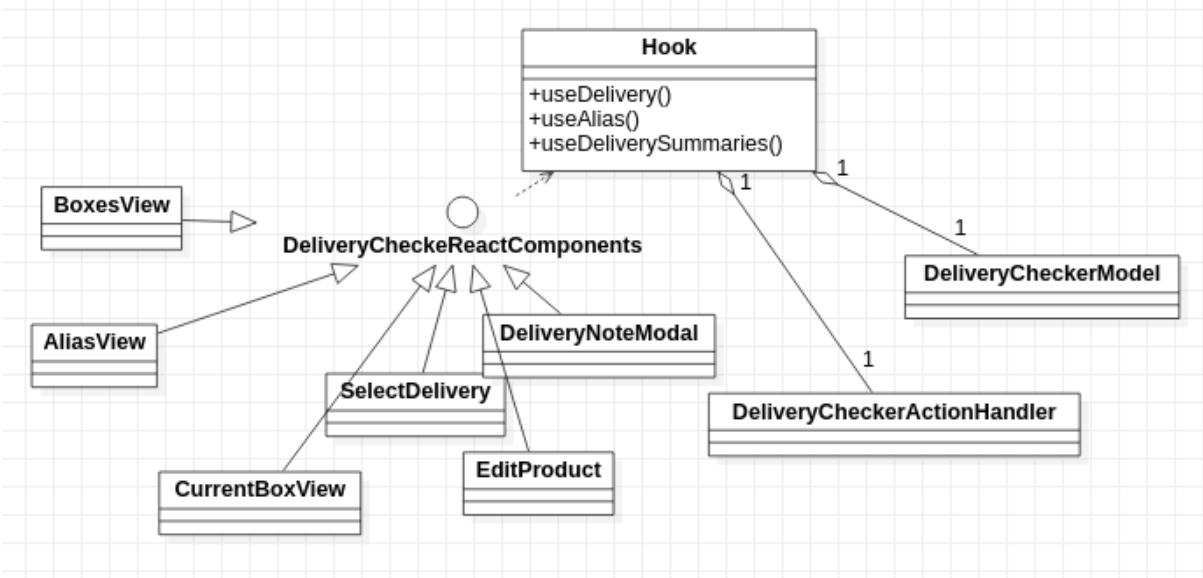
```
            this.locker = undefined
            return false
        } else {
            this.e.emit('next', this.counter)
            this.counter++
            this.requests.pop()
            return true
        }
    }
```

Di seguito un esempio dell'utilizzo di *Semaphore*:

```
//da DeliveryServer
...
    /**
    * lock delivery
    */
    async lock(w: WarehousemanData): Promise<void> {
        return await this.semaphore.lock(w)
    }
    ...
    /**
    * release delivery
    */
    release(): void {
        this.semaphore.release()
    }

//da ServerModel

    async getLockOnProduct(deliveryId: number, boxId: number, productId:
number, w: WarehousemanData): Promise<boolean> {
        let del = this.getDeliveryFromCurrents(deliveryId)
        await del.lock(w) //get lock on delivery
        let res = del.getBox(boxId)?.getProduct(productId)?.getLock(w)
        del.release() //release lock on delivery
        return res != undefined ? res : false
    }
...
```

DeliveryChecker

DeliveryChecker è stata sviluppata come una single page application in ReactJS. Tutta l'applicazione è scritta in javascript. Se la logica applicativa principale si trova nel modulo *core*, l'applicazione realizzata implementa l'interfaccia grafica per l'interazione con l'utente e la presentazione dei dati.



DeliveryChecker app

Le schermate realizzate sono analoghe a quelle descritte in fase di design con l'aggiunta di qualche interazione e widget grafico (come la barra di completamento), per rendere più fluida l'esperienza utente.

Per lo sviluppo è stato largamente utilizzato il meccanismo degli hook e la libreria React Router per lo switch tra le interfacce.

Ogni React Component di DeliveryChecker utilizza gli hook messi a disposizione dal modulo `./hook/hook.js` che incapsula le funzionalità importate dal modulo *core*, e mette a disposizione i dati di *DeliveryCheckerModel* ai componenti.

Di seguito una lista che associa a ogni componente l'interfaccia realizzata:

- AliasView - interfaccia 1
- SelectDelivery - interfaccia 2
- DeliveryNoteModal - interfaccia 4
- BoxesView - interfaccia 3
- CurrentBoxView - interfaccia 5
- EditProduct - interfaccia 6

WarehouseService

WarehouseService è stato implementato come un javascript web server utilizzando la libreria ExpressJS.

La sua funzionalità di web server viene implementata nello script `app.js` con la definizione delle *route* che indicizzano le applicazioni del magazzino (per ora solo DeliveryChecker) le cui build si trovano nella cartella `./public` di warehouse-service. Informazioni aggiuntive per quanto riguarda la build della applicazioni sono date nella sezione sull'automazione.

```
app.use('/delivery-checker', express.static(path.join(__dirname, 'public',
'delivery-checker')));
app.get('/delivery-checker/*', function (req, res) {
  res.sendFile(path.join(__dirname, 'public', 'delivery-checker',
'index.html'));
});
```

Al lancio del server viene poi avviata anche l'interfaccia WebSocket implementata nel modulo *core* e viene stabilta la connessione con il database. Per maggiori dettagli si faccia riferimento al file `./warehouse-service/bin/www`.

Test Driven Development

Nella prima fase dello sviluppo è stata definita una suite di test e la loro risoluzione ha guidato lo sviluppo del codice e delle funzionalità.

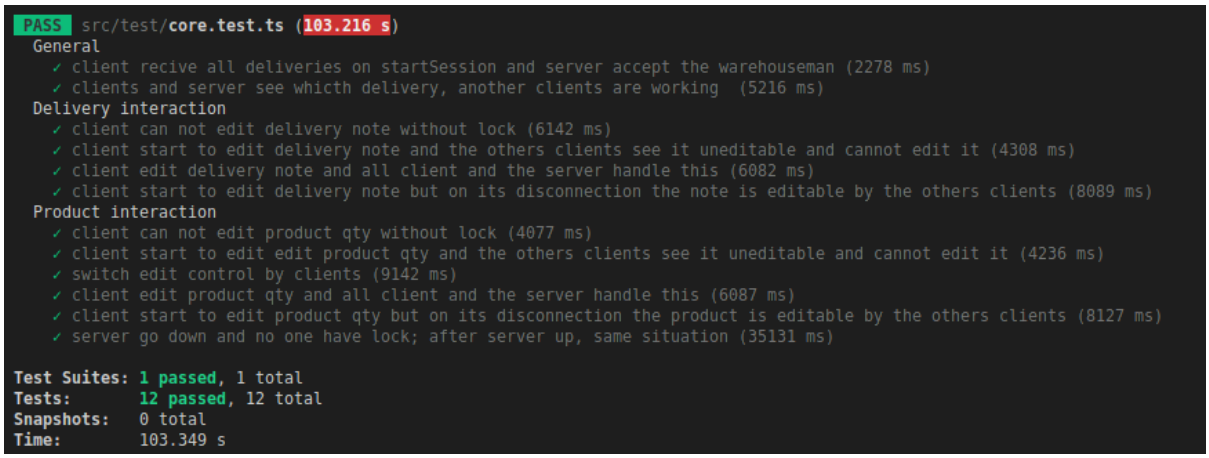
Il focus principale dei test è *validare il comportamento del sistema*, non il risultato dei singoli metodi delle classi o dei moduli.

In particolare è stato testato il codice che implementa le funzionalità *core* di cui è richiesto un alto livello di affidabilità.

La suite di test è stata organizzata in gruppi di test (i numeri tra parentesi associano i test ai requisiti funzionali espressi nella sezione analisi descritti tramite elenco puntato)

1. **General**: la prima parte delle suite di test, che permette di testare le funzionalità base del sistema. È stata pensata e utilizzata soprattutto per cercare di guidare lo sviluppo del sistema nelle sue prime fasi. Le funzionalità testate sono:
    - Inizio di una sessione con accesso tramite alias (1)
    - Selezione della selezione su cui lavorare (2)
    - Monitoraggio della scelta della spedizione degli altri magazzinieri (su quale spedizione gli altri utenti stanno lavorando?) (7)
2. **Delivery interaction**: il secondo gruppo di test è stato utilizzato per testare il comportamento del sistema in fase di editing della spedizione, in particolare per quanto riguarda l'editing della nota testuale. In ordine cronologico i requisiti espressi da questi test sono stati in realtà soddisfatti per ultimi. Le funzionalità testate in questo gruppo sono:
    - Editing della nota testuale (6)
    - Gestione dell'accesso concorrente alla risorsa nota (8)
    - Verifica del comportamento in caso di disconnessione del client, declinato nel caso di editing della nota testuale (9)
    - Verificare che sia possibile scoprire chi sta editando la nota testuale (7)
3. **Product Interaction**: l'ultimo gruppo di test permette di testare l'editing della quantità del prodotto verificata. È stato particolarmente importante nella seconda fase di sviluppo anche per implementare il modello di concorrenza definito in fase di design. Le funzionalità testate in questo gruppo sono:
    - Editing della quantità di un prodotto (4)
    - Gestione dell'accesso concorrente alla risorsa quantità del prodotto (8)
    - Verifica del comportamento in caso di disconnessione del client, declinato nel caso di editing della quantità di un prodotto (9)
    - Verificare che sia possibile scoprire chi sta editando la quantità del prodotto (7)

La nomenclatura dei test è stata scelta in modo che siano facilemente ricollegabili ai requisiti funzionali descritti in fase di analisi.



Screenshoot esecuzione suite di test

L'esecuzione dei test segue il seguente schema per cui è stata attivata un'opportuna procedura automatica come si vedrà in seguito.

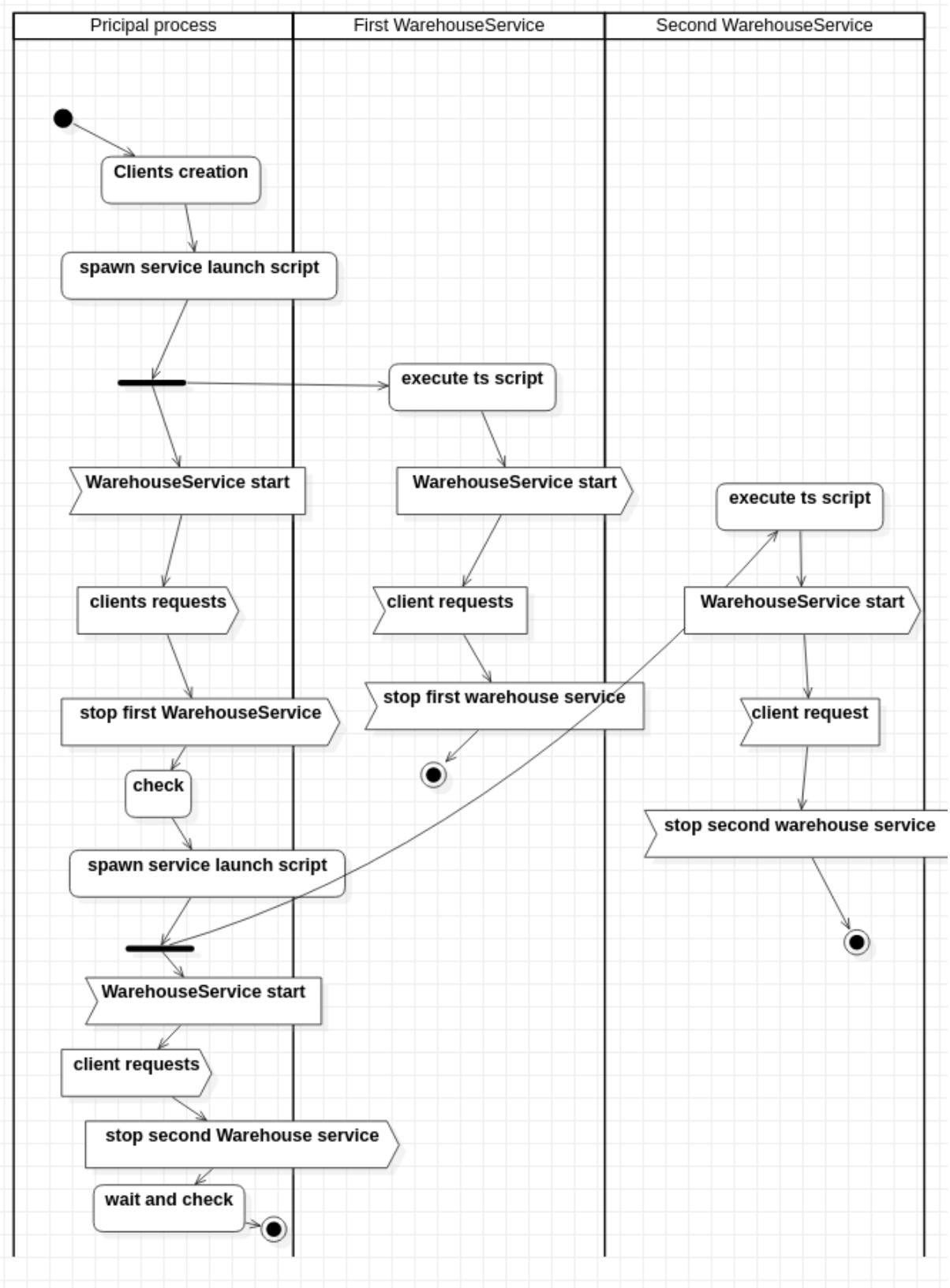
- lancio di un database di test
- per ogni test:
  - lancio di un server di test
  - lancio dei client
  - test
  - chiusura del server
  - chiusura dei client

Per lo sviluppo dei test ci si è affidati alla libreria javascript Jest Per i seguenti motivi:

- per lo stile particolarmente predisposto a una tipologia di test rivolta alla verifica del comportamento (viene infatti spesso utilizzata in ambito web per il test delle interfacce grafiche)
- per la semplicità di utilizzo e di configurazione anche con Typescript
- la chiarezza della documentazione
- per il largo utizzo nella comunità javascript (con il benefit di poter accedere quindi alle molte risorse in merito presenti in rete)

Testare un sistema distribuito soprattutto su un motore single thread come javascript presenta la difficoltà di non riuscire appunto fisicamente a distribuire i flussi operativi. Se per la maggior parte dei test questa problematica resta presente, nel dodiciesimo test `server go down and no one have lock; after server up, same situation` è stato fatto qualche passo in avanti per cercare di creare in uno scenario di esecuzione più simile a quello reale.

Sostanzialmente anzichè eseguire i client e il WarehouseService tutti sullo stesso processo è stato creato un processo separato per WarehouseService, che una volta lanciato viene stoppato dopo un po' di tempo per simulare un errore del servizio. Per simulare la ripresa del servizio viene poi successivamente lanciato un altro processo che avvia WarehouseService. Per lanciare in questo test WarhouseService è stato necessario creare uno script bash, che si occupa appunto del lancio, e definire un apposito script typescript `launchTemporalServer.ts`. I client sono invece eseguiti sullo stesso processo che esegue i test e che lancia i processi che eseguono WarehouseService.



Test utilizzando i processi

Altra complessità legata al test del sistema è l'asincronia di procedure e comunicazioni che fanno si che diventi difficile coordinare azioni in sequenza attendendo l'esecuzione delle azioni che hanno un termine non preciso nel tempo. Per fare ciò non sempre sono sufficienti le funzionalità messe a disposizione dal linguaggio (come ad esempio il meccanismo delle promise con la corrispondente callback then). Sono stati adottati diversi approcci.

Il primo, è quello di attendere un tempo ragionevole prima di eseguire l'azione successiva ed è l'approccio utilizzato nella maggioranza dei test. È un approccio semplice ma ha anche il vantaggio di permettere di

testare implicitamente che il sistema impieghi un massimo quantitativo di tempo per eseguire un'azione.

Un alreo approccio possibile poteva essere quello di utilizzare la propagazione di un evento per dare il via all'esecuzione di azioni successive. Questo approccio, più robusto soprattuto in assenza di vincoli temporali, non è stato però utilizzato perchè avrebbe reso necessaria la modifica di funzioni e metodi e quindi un'ulteriore attività di sviluppo che non è stato ritenuto necessario sostenere.

Automazione

Molta cura e attenzione è stata data alla definizione di pipline di automazione e bulding automation. Per definire l'attuale stato di automazione del sistema è stato necessario definire in primis la struttura del progetto e una prima fase di sviluppo prototipale evolutivo che facesse emergere le reali necessità.

A fronte di ciò, e nel rispetto di quanto definito nella sezione relativa ai requisiti non funzionali, si è resa necessaria la definizione di:

- una modalità di gestione automatizzata delle dipendenze
- una modalità di gestione automatizzata della presenza di più progetti in un singolo progetto
- una pipline per gestire il testing automatizzato con due varianti differenti
- una pipline per fare il buildind di tutto il progetto

Alla necessità di **gestire in maniera automatica le dipendenze e la multiprogettualità** ci si è affidati all'utilizzo di npm, lo standard de facto nel mondo javascript. npm mette a disposizione la tecnologia degli workspaces per gestire più progetti in uno: con `npm init -w workspacename` viene inizializzato un nuovo progetto npm con un proprio file di configurazione (*package.json*), e in cui si può lavorare come se fosse un progetto a se stante con le proprie pipeline e dipendenze. È possibile accedere ai comandi del workspace dal progetto principale con `npm run command -w workspacename`. Gli workspace configurati in questo progetto sono il modulo *core*, *warehouse-service*, e *delivery-checker*. La tecnologia degli npm workspace permette anche una gestione ottimizzata e automatica delle dipendenze: le librerie saranno scaricate nella cartella `node_modules` del progetto pricipale e, solo nel caso sia necessaria la stessa libreria ma con una versione differente, questa viene scaricata nella cartella `node_modules` dei progetti che ne hanno bisogno nella versione corretta. I vari moduli di uno stesso progetto possono dipendere gli uni da gli altri, e la tecnologia degli workspaces permette di utilizzare i meccanismi di import e export come se fossero dipendenze come le altre.

```
//importazione di DeliveryCheckerWarehousemanActionHandler dal core in
delivery-checker
import { DeliveryCheckerWarehousemanActionHandler } from
'core/dist/client/DeliveryCheckerActionHandler'
```

npm permette di definire nel file `package.json` alcuni comandi personalizzati che possono essere lanciati semplicemente con `npm run comando-da-lanciare`. Questa opportunità è stata sfruttata per rispettare la volontà di mantenere semplice l'interfacciamento con le funzionalità di automazione.

La pipeline per la **gestione del testing automatizzato** è stata in astratto così definita:

1. lancio di un db di test con lo stesso schema del db di deployment.
2. popolazione del database di test.
3. esecuzione dei test.
4. A seconda della modalità di test scelta può essere eseguito lo shutdown del database o meno. Questo risulta utile perchè in presenza di test non passati possono essere controllati i valori presenti nel db.

Sono stati definiti due comandi per il testing secondo le due possibili modalità previste: `test-not-shutdown-db` e `test-shutdown-db`. Entrambi sfruttano docker e docker-compose per il loro funzionamento. Nel file `docker-compose.yaml` troviamo definito un servizio db_dev che lancia un'istanza di Mongodb e settando opportunamente i volumi del container come definito, permette di popolare il database come dichiarato nel file json contenuto nella cartella `/db_init_dev/data`. Gli step eseguiti quindi saranno:

```
## elimina vecchi file creati eseguendo la pipeline
rm -rf ./core/src/test/data &&
## lancio del db
sudo docker-compose up -d db_dev &&
## messa a disposizione delle delivery che popolano il db anche agli script
di test
cp -R ./db_init_dev/data ./core/src/test/data &&
## lancio dei test
export ENV=dev && cd core && npx jest
```

La pipeline per l'**automazione del deployment** dell'intero sistema invece prevedono principalmente le seguenti fasi:

1. traspilazione del modulo core
2. build di tutte le applicazioni che devono essere raggiungibili attraverso WarehouseService
3. lancio del database con opportuna popolazione
4. lancio del servizio

Anche in questo caso sono stati utilizzati docker e docker-compose.

Nel file `docker-compose.yaml` sono definite le istruzioni utili al deployment del sistema:

```
  warehouse-service:
    build:
      context: ./
      dockerfile: ./warehouse_service.dockerfile
    ports:
      - '5000:3000'
    depends_on:
      - "db"
```

Viene indicato in maniera dichiarativa che verrà lanciato un servizio *warehouse-service* che utilizza un immagine descritta nel dockerfile *./warehouse_service.dockerfile*, disponibile sulla porta 5000 facendo il binding tra la porta 3000 dell'applicazione del servizio e quella del container. Prima di lanciare il servizio docker-compose avvierà una fase di discovering per il detect di un servizio *db* che se non presente sarà lanciato nelle modalità previste dal `docker-compose.yaml` (in questo caso analoghe a quelle descritte per il lancio del db di test).

Il dockerfile *./warehouse_service.dockerfile* ci indica poi come viene costruito il container del servizio:

```
FROM node:19-slim

EXPOSE 3000

RUN mkdir /warehouse
COPY . ./warehouse
WORKDIR warehouse
RUN npm install
RUN npm run compile-core
RUN npm run deploy --workspaces
CMD npm start -w warehouse-service
```

Come base viene utilizzata un'immagine che permette di eseguire applicazioni nodejs presa da dockerhub. A questo punto viene copiata tutta la cartella del progetto escluse le cartelle descritte nel `.dockerignore`. In particolare vengono ignorati i `node_modules` cioè i pacchetti installati tramite npm, le dipendenze del progetto (si predilige una installazione pulita). Dopodichè appunto vengono installate le dipendenze mancanti tramite npm. A seguire vengono eseguiti i comandi che corrispondono ai punti 1 2 dell'elenco puntano che descrive la procedura:

1. `compile-core` permette di transpilare il codice typescript presente nel modulo core dando origine a una cartella build a cui hanno accesso i progetti dipendenti dal modulo
2. `npm run deploy --workspaces` comando npm che permette di eseguire `npm run deploy` in ogni workspace del progetto (quindi in ogni sottoprogetto). Di ogni applicazione resa disponibile da WarehouseService, per il momento solo DeliveryChecker, viene generato un sito web statico con tutte le risorse necessarie situate in un'unica cartella che viene copiata nella cartella `./public` di *warehouse-service*.

Nell'ultima riga del dockerfile viene stabilito il comando da lanciare all'avvio del container `npm start -w warehouse-service` che esegue npm start in *warehouse-service* lanciando il server express che implementa il servizio.

Lanciando il comando `docker-compose up --build warehouse-service` è possibile lanciare il servizio, facendo il build dell'immagine docker che comprende tutti i passaggi descritti precedentemente, e se non ancora attivo lanciando il servizio *db*.

Deployment

Per lanciare il sistema sarà necessario avere installato sulla propria macchina *git*, *docker* e *docker-compose*.

0. Installare git, docker e docker-compose
1. Il primo passo è clonare o scaricare il repository da qui
2. Lanciare il comando `docker-compose up --build warehouse-service` (nelle successive esecuzioni sarà possibile omettere l'opzione --build). Provvederà al lancio del servizio. Se speficicato il flag `--build` sarà necessario attendere qualche minuto per aspettare che la build sia completata
3. Accedere a delivery checker tramite l'indirizzo `indirizzo.del.server.o.localhost:5000/delivery-checker`

Guida ed esempi di utilizzo

Per mostrare come utilizzare il sistema tramite app DeliveryChecker è stato realizzato un video che è possibile vedere qui. Sono stati registrati in particolare i seguenti scenari di utilizzo.

- Accesso tramite alias
- Selezione delle spedizioni
- Edit del prodotto con acquisizione del lock da parte dello scrittore
- Edit della nota testuale con acquisizione del lock da parte dello scrittore
- Cambio spedizione corrente
- Disconnessione di un'applicazione

Conclusioni

È stato realizzato un sistema che supporta il lavoro dei magazzinieri di un'azienda di logistica in cui all'interno del magazzino e in particolare per questo progetto il task principale da considerare era il controllo dei prodotti in ingresso da parte dei magazzinieri.

Questo ha portato all'implementazione di un servizio WarehouseService che gestisce le informazioni del magazzino, e di un'applicazione DeliveryChecker che permette ai magazzinieri di interfacciarsi con il sistema e eseguire i task previsti.

In linea generale ci si ritiene soddisfatti del prodotto realizzato: il sistema risulta solido grazie alla suite di test automatizzata che lo accompagna, e anche la messa in produzione risulta agevole senza installazioni complicate. L'applicazione risulta semplice e non ha presentato in fase di validazione comportamenti inspettati, rispondendo adeguatamente in tutti gli scenari previsti. Manca certamente una fase di test con utenti e dati reali per verificarne il comportamento se utilizzata con utenti non esperti, e per monitorare le performance generali del sistema a fronte di un aumento degli utenti e con una mole di dati reale. Solo a seguito di questi ipotetici test ci si potrebbe esprimere in merito alla valutazione delle performance.

In tutto l'arco del progetto si è cercato di sviluppare una soluzione che fosse pronta a possibili aperture future che necessitano ancora un'analisi approfondita del dominio, e anche in questo caso ci si ritiene soddisfatti di quanto prodotto. Tenendo come riferimento lo schema di *delivery-checker* si dovrebbe avere la possibilità di aggiungere senza troppi problemi altre applicazioni e accedere a tutte le comodità definite in fase di automazione. Il modulo *core*, condensando tutte le principali funzionalità, può essere agevolmente esteso ed è pronto a sopportare anche un eventuale refactoring potendosi appoggiare anche alla suite di test di cui è dotato.

Ci si ritiene soddisfatti anche delle tecnologie apprese e delle competenze maturate durante il progetto:

- Si ha acquisito una conoscenza approfondita di WebSocket e soprattutto della libreria socket.io.
- Si è consolidata la conoscenza di Javascript ed npm ampliandola con il meccanismo degli workspace.
- È stato imparato Typescript in maniera entry level. Comunque si ritiene di averne fatto un utilizzo sapiente durante il progetto, non tanto in termine di codice scritto, sicuramente migliorabile, quanto per la selezione delle porzioni di codice da scrivere con questo linguaggio.
- Si ha avuto la possibilità di studiare le problematiche di gestione della concorrenza in ambito distribuito, della gestione del partizionamento arrivando a definizione di un comportamento omogeneo del sistema anche a fronte di eventuali errori.
- Si ritiene di aver acquisito un certa maturità anche nella progettazione e nello sviluppo delle pipline di automazione e di test. In particolare i test non sono stati pensati solo in vista della verifica delle funzionalità, ma per accompagnare lo sviluppo del sistema nei suoi stadi iniziali.

Con occhio critico e volenteroso di imparare dagli errori si ritiene di aver gestito il progetto in maniera non sempre ottimale. Soprattutto nella fase iniziale il progetto ha subito diverse fasi di stallo a causa delle tante novità da apprendere in contrasto con la fretta di arrivare subito a una soluzione definitiva, sia in termini di design del progetto che di sviluppo delle pipeline di automazione.

Sarebbe stato meglio avere un approccio più graduale e prototipale, e magari, almeno all'inizio, non iniziare subito a lavorare in test-driven, ma prototipare un'interfaccia grafica per interagire con il sistema, che è

un'attività in cui si ha molta più esperienza, e che avrebbe permesso di sviluppare qualche funzionalità e far emergere gli aspetti critici dello sviluppo con più chiarezza.

Nel futuro si procederà con uno studio del dominio applicativo per vedere quali saranno le prossime parti da sviluppare: sicuramente manca tutta una parte di interazione con i clienti per la creazione delle spedizioni, lo stoccaggio dei prodotti verificati in magazzino e l'uscita dal magazzino dei prodotti stoccati.