STRUTTURA DELIVEROO.JS

index.js

il file è il punto di partenza del progetto:

Definisce la porta del server prendendo il valore `PORT` definito nel file ***process.env***, oppure usando il valore di default `8080`.

Prova ad attivare la connessione con il DB di Redis implementato nel file ***src/redisClien***.

Mette in ascolto il server, definito nel file ***src/httpServer***, alla porta definita

Mette in ascolto il sistema ioServer, definito nel file ***src/ioServer*** nel server

SRC

redisClient.js

Importa dal file ***.env*** le credenziali di accesso per il DB creato con Redis: `REDIS_URL`
crea un nuovo cliente usando le seguenti credenziali.

implementa i listener a

- `myGrid.'agent score'` che aggiorna il punteggio del player
- `myGrid.'agent created'` che verifica se esiste già il player nel database, se si ricarica il suo punteggio, altrimenti crea un nuovo record.

Dove con `myGrid` intendiamo l'istanza della classe `Grid` definita nel file ***grid.js***

httpServer.js

Crea il server partendo dall'istanza app definita nel file ***app.js***

app.js

Va a definire le risorse da inviare al client; nel file si definisce che i dati da inviare sono contenuti all'interno della directory: ***packages/@unitn-asa/deliveroo-js-webapp***

ioServer.js

Questo file implementa il gestore degli eventi che si attiva quando un nuovo socket si connette al server tramite **Socket.IO**; e anche l'invio di tutti i dati del gioco alla socket.

La nozione base è che quando una socket si collega al server essa solleva l'evento `socket.'connection'` passando come attributo a se stessa: `socket`.

Prima di tutto il file va a definire l'istanza del *`Server`* di Soket.IO con la variabile `io`; poi continua con la definizione del listener all'evento evento `socket.'connection'` con attributo `socket` che:

1) Autentifica la Socket:
Quando un nuovo socket si connette, il server autentica il socket utilizzando l'oggetto `myAuthenticator` un'istanza della classe `Authentication` definita nel file ***deliveroo/Authentication.js***:

```
const myAuthenticator = new Authentication(myGrid)
```

Dove `myGrid` è l'oggetto `Grid` importato da ***grid.js***

Ad ogni connessione di una nuova socket avviene l'autenticazione mediante il metodo `myAuthenticator.authenticate(socket)`.
Essa verifica se la socket era già associata ad un agente oppure se era una nuova socket ne crea uno nuovo, e ritorna l'agente salvato nell'istanza `me`.

Dopo l'autenticazione la socket emette un evento `socket.'hi'` aggregato agli attributi: `socket.id`, `me.id` ed `me.name`.

2) Check 'god'
Se l'agente è identificato con `name` uguale a `'god'`:

- vengono modificati i parametri `me.config.PARCELS_OBSERVATION_DISTANCE` e `me.config.AGENTS_OBSERVATION_DISTANCE` e sono impostati a `'infinite'` permettendo così una visione completa di tutti gli agenti e pacchi.
  Poi viene emesso il segnale `socket.'config'` passando come argomento `me.config`.

- vengono abilitati ulteriori 3 listener:
  - a `socket.'create parcel'` che viene emesso con due attributi `x`, `y`. Il listener quindi genera un pacco in quelle coordinate mediante `myGrid.createParcel(x,y)`.

  - a `socket.'dispose parcel'` che viene emesso con due attributi `x`, `y`.
    Il listener controlla se ci sono pacchi nella casella di coordinate ed eventualmente le elimina con `myGrid.deleteParcel()`. Successivamente emette un segnale `myGrid.'parcel'`.

  - a `socket.'tile'` che viene emesso con due attributi `x`, `y`. Il listener serve per cambiare natura della casella indicata dalle coordinate:
    da ***bloccato*** → a ***spawner***
    da ***spawner*** → a ***delivery***
    da ***delivery*** → a ***standard***
    da ***standard*** → a ***bloccato***

3) Presa ed Invio dati della Mappa alla Socket

I dati della mappa sono salvati all'interno dell'oggetto `myGrid` importato dal file ***grid.js***:

Quindi prima passa in rassegna tutte le celle della griglia: `myGrid.getTiles()` mediante un ciclo for:

- Se la casella non è bloccata emette un segnale `socket.'tile'` con i dati della casella ed inserisce la casella in un array `tiles`.
- Se la casella è bloccata emette un segnale `socket.'not_tile'` con le coordinate della casella

Una volta passate tutte le tabelle viene emesso il segnale `socket.'map'` con attributi l'array `tiles`, la larghezza e lunghezza della mappa `myGrid`.

Inoltre definisce il listener all'evento `myGrid.'tile'` emesso da una casella quando essa viene modificata ed l'evento ha come attributo la casella modificata. Il listener con le stesse modalità definite precedentemente controlla se la casella è bloccata o meno ed emette il segnale `socket.'tile'` o `socket.'not_tile'`.

4) Presa ed Invio dati dell'Agente alla Socket

I dati dell'agente associato alla socket: `me` vengono trasmessi alla socket mediante l'emissione dell'evento `socket.'you'` con attributo appunto `me`.

Inoltre viene implementato il listener all'evento `me.'agent'` sollevato dall'agente ogniqualvolta le sue coordinate o il suo punteggio cambiano; il listener semplicemente emette l'evento `socket.'you'` con attributo appunto `me`.

5) Presa ed Invio dati dei Dintorni dell'Agent alla Socket

I dati dei dintorni dell'agente corrispondono agli altri agenti e pacchi nelle vicinanze dell'agente; per passare questi dati viene implementato il listener a `me.'parcels sensing'` ed a `me.'agents sensing'` che sono emessi con i rispettivi argomenti: `parcels`: pacchi nelle vicinanze ed `agents`: agenti nelle vicinanze.

I listener semplicemente chiamano rispettivamente gli eventi `socket.'parcels sensing'` ed `socket.'agents sensing'` con i corrispondenti attributi: `parcels` e `agents`.

Infine invoca i metodi `me.emitParcelSensing()` e `me.emitAgentSensing()` che definiscono `parcels` e `agents` e sollevano gli eventi `me.'parcels sensing'` ed a `me.'agents sensing'`.

6) Si mette in ascolto dei comandi Client

Vengono implementati i listeners dei segnali `socket.'move'`, `socket.'pickup'` e `socket.'putdown'` inviati dal client per comandare il suo agente.
I listener vanno a richiamare metodi di movimento, di raccolta e scarico della classe `Agent` implementato nel file ***deliveroo/Agent.js***

Inoltre i 3 eventi possono essere emessi con un terzo argomento: `aknowledgementCallback` una funzione che, se passata, viene invocata dal listener con argomento il risultato del metodo eseguito per avvisare il mittente `socket` del successo del suo comando.

7) Reindirizzare i messaggi provenienti dalle Socket

Vengono implementati 3 listeners:

- a `socket.'say'` che gestisce l'evento in cui `socket` invia un messaggio ad un preciso agente. Infatti l'evento viene allegato con gli argomenti `msg`: contenuto del messaggio ed `toId` che indica l'id dell'agente a cui inviare il messaggio.

  Per passare il messaggio il listener cicla tutte le socket dell'agente `toId` ed per ognuno socket destinataria: `socketDest` emette un evento: `socketDest.'msg'` passando i dati dell'agente mittente: `me.id` e `me.name` ed il messagio `msg`.

  Inoltre `socket.'say'` può essere emesso con un terzo argomento: `aknowledgementCallback` una funzione che se passata viene invocata dal listener con argomento `successful` per avvisare il mittente `socket` del successo dell'invio del messaggio.

- a `socket.'ask'` che gestisce l'evento in cui `socket` invia una domanda ad un preciso agente. Infatti l'evento viene allegato con gli argomenti `msg`: contenuto del messaggio, `toId` che indica l'id dell'agente a cui inviare il messaggio e `replayCallback` una funzione che permette al destinatario di inviare una risposta.

  Per passare la domanda il listener cicla tutte le socket dell'agente `toId` ed per ognuno socket destinataria: `socketDest` emette un evento: `socketDest.'msg'` passando i dati dell'agente mittente: `me.id` e `me.name`, il messaggio `msg` e la funzione di risposta `replayCallback`.

- a `socket.'shout'` che gestisce l'evento in cui `socket` invia un messaggio a TUTTI gli agenti. Infatti l'evento viene allegato con gli argomenti `msg`.

  Per passare il messaggio il listener emette un evento broadcast socket.'msg' passando i dati dell'agente mittente: `me.id` e `me.name` ed il messagio `msg`.

  Inoltre `socket.'shout'` può essere emesso con un terzo argomento: `aknowledgementCallback` una funzione che se passata viene invocata dal listener con argomento `successful` per avvisare il mittente `socket` del successo dell'invio del messaggio.

8) Condivisione di path

Permette ad una socket di condividere un path con tutte le altre socket collegate al suo stesso agente.

Per far ciò implementa il listener a `socket.'path'` con allegato l'attributo `path`, il listener allora cicla tutte le socket associate all'agente `me`: `socketDest` ed invia il path sollevando il segnale `socketDest.'path'` allegando `path`.

9) Broadcast log di un client

Permette di avvisare tutte le socket il log di un client mediante l'emissione dell'evento broadcast `socket.'log'` con argomenti: il tempo di log, la socket id del log, l'id ed il name dell'agente con cui è loggato ed eventualmente altri argomenti.

grid.js

Va a leggere quale mappa è stata scelta tramite la variabile `MAP_FILE` ed definisce la mappa di gioco

`MAP_FILE` viene inizializzata con il valore settato in ***config.js***; altrimenti se non è presente viene inizializzato con il valore definito in ***process.env***; se anche qui non è definito viene inizializzato al valore di default: `default`.

Una volta capito che mappa caricare prende i dati relativi alla mappa scelta dalla cartella ***/levels/maps/*** e va a creare mediante la mappa l'istanza `grid` della classe `Grid` definita nel file ***/deliveroo/Grid.js***. Questa istanza viene esportata e sarà la mappa di gioco.

Inoltre va a

- chiamare la funzione generatrice dei pacchetti `parcelsGenerator(grid)` definita nel file ***workers/parcelsGenerator.js.***
- chiamare n volte la funzione che gestisce terzi agenti `randomlyMovingAgent(grid)` definita nel file ***workers/randomlyMovingAgent.js***. Dove n è uguale a `RANDOMLY_MOVING_AGENTS` definito in ***config.js***; altrimenti se non è presente viene inizializzato con il valore definito in ***process.env***; se anche qui non è definito viene inizializzato al valore di default: 0.

workers

parcelsGenerator.js

Implementa la funzione `parcelsGenerator` generatrice dei pacchetti.
Prende in input l'istanza `grid` della classe `Grid` definito nel file ***/deliveroo/Grid.js***

Essa definisce un listener all'evento `myClock.PARCELS_GENERATION_INTERVAL` e:

- va a controllare se si è già raggiunto il numero di pacchi massimi: `PARCELS_MAX`
- se non è stato raggiunto controlla se ci sono caselle spawner libere
- se ci sono caselle spawner libere va a generare un pacchetto su una di esse scelta a caso

```
grid.createParcel( tile.x, tile.y )
```

Dove `myClock` è il clock di gioco importato da ***/deliveroo/Clock.js*** e `PARCELS_GENERATION_INTERVAL` può assumere i valori `'1'`, `'2'`, `'5'`, `'10'` che sono i vari eventi emessi da `myClock`.

Le variabili `PARCELS_GENERATION_INTERVAL` e `PARCELS_MAX` sono definite come nel file ***process.env***, altrimenti come nel file ***config.js*** altrimenti sono settati a un valore di default.

randomlyMovingAgent.js

Implementa la funzione `randomlyMovingAgent` che gestisce i terzi agenti
Prende in input l'istanza `myGrid` della classe `Grid` definita nel file ***/deliveroo/Grid.js*** ed una stringa `name`.

La funzione crea un nuovo agente `myGrid.createAgent(name)` e lo fa muovere random ogni lasso di tempo `RANDOM_AGENT_SPEED`.
`RANDOM_AGENT_SPEED` può assumere i valori `'1'`, `'2'`, `'5'`, `'10'` che sono i vari eventi emessi da `myClock`. Il suo valore è definito come nel file ***process.env***, altrimenti come nel file ***config.js*** altrimenti sono settati a un valore di default.

deliveroo

É una cartella contenente le implementazioni degli oggetti di DELIVEROO.JS

Observable.js

Implementa la classe `Observable` i cui oggetti riescono a notificare agli altri componenti del sistema quando un certo campo all'interno dell'oggetto cambia.
I metodi di questa classe sono:

- `interceptValueSet("campo", "evento_emesso")`
  Questo metodo consente di specificare un campo all'interno dell'oggetto che viene monitorato. Se il campo specificato non esiste viene creato. Ogni volta che il campo viene modificato l'oggetto emette un evento mediante la chiamata di un secondo metodo: `emitOnePerTick()`

- `emitOnePerTick("event", "argomento1")`
  Questo metodo consente di emettere un evento dall'oggetto `Observable`. La particolarità è che l'evento specificato viene emesso solo alla fine del ciclo di event loop ed una sola volta anche se il metodo viene chiamato più volte. Inoltre è possibile specificare un numero arbitrario di eventi che vengono trasmessi con l'evento.

- `emitAccumulatedAtNextTick("event","argomento1")`
  Questo metodo consente di emettere un evento dall'oggetto `Observable`. La particolarità è che l'evento viene emesso solo alla fine del ciclo di event loop. Inoltre se il metodo viene chiamato più volte esso permette di emettere il segnale solo una volta e passando tutti gli argomenti specificati nelle varie chiamate del metodo.

Xy.js

Questo file implementa la classe `Xy`, che estende `Observable` aggiungendo 2 proprietà: `x` e `y`.
Quindi gli oggetti `Xy` sono degli oggetti `Observable`, che possono emettere eventi, e che sono caratterizzati da una posizione spaziale.
I metodi che implementa sono:

- `static distance(a={x,y}, b={x,y}) && distance(other={x,y})`
  Metodo di classe che restituisce la distanza tra i due punti: `a` ed `b`.
  Metodo di istanza che restituisce la distanza tra l'istanza e il punto `other`

- `static equals(a={x,y}, b={x,y}) && equals(other={x,y})`
  Metodo di classe che verifica se due punti: `a` ed `b` coincidono
  Metodo di istanza che verifica se un punto `other` coincide con la posizione dell'istanza

Clock.js

Questo file implementa la classe `Clock`, che estende `Observable`, la quale rappresenta il clock nel gioco.

Per tenere traccia del tempo essa usa il metodo javascript `setInterval()` che esegue una funzione ogni lasso di tempo. Essa ritorna un valore ***id*** che può essere usato come parametro nella funzione `clearInterval()` per fermare le esecuzioni della funzione.

La classe `Clock` ha quattro proprietà:

- `#base`: definisce l'intervallo di tempo tra clock, essa è posta uguale alla variabile `CLOCK`
- `#id`: salva l'id dell'intervallo eseguito
- `#ms`: tiene traccia del tempo trascorso in millisecondi
- `#isSynch`: tiene traccia se il clock è sincronizzato con il gioco

I metodi della classe `Clock` sono:

- `start()`: il metodo chiama `this.#id=setInterval(()=>{//function},this.#base)`, quindi ogni volta che passano `#base` millisecondi viene eseguita la funzione `//function`. `//function` semplicemente prima mette in false `#isSynch` per evitare che il tempo in cui viene eseguita incida anche sul gioco. Poi incrementa `#ms` di `#base` per tenere traccia del tempo trascorso. Successivamente emette l'evento `clock.'frame'`, che quindi segnala al resto del sistema che è passato un clock; inoltre fa dei controlli per vedere quando sono passati 1, 2, 5, 10 secondi per eventualmente emettere i segnali `clock.'1'`, `'2'`, `'5'`, `'10'`. Infine viene riportato a true `#isSynch`.

- `stop()`: il metodo chiama `clearIntervall(this.#id)` per fermare il ciclo di clock

- `synch(delay)`: il metodo permette di sincronizzare il codice con il clock di gioco. Infatti esso viene risolto se e solo se `#isSynch` è true, altrimenti il metodo aspetta un nuovo segnale `clock.'frame'` Inoltre permette di attendere un certo periodo di tempo: `delay` misurata in millisecondi basato sul clock del programma.

`CLOCK` definisce l'intervallo di tempo tra ogni clock in millisecondi. `CLOCK` può assumere il valore definito in ***process.env***, altrimenti quello definito nel file ***config.js***. Se nessuno dei due valori è definito esso viene impostato a 50 millisecondi.

Infine nel file viene definita l'istanza **myClock** della classe `Clock`, essa viene esportata e poi riutilizzata in tutto il progetto

Parcel.js

Questo file implementa la classe `Parcel`, la quale estende la classe `Xy` . La classe `Parcel` definisce l'entità di gioco del pacco. I pacchi possono spawnare in giro per la mappa, il giocatore deve passarci sopra per raccoglierli e portarli fino a una determinata casella per ricevere come ricompensa dei punti.

Gli attributi sono:

- `#lastId`: variabile di classe, memorizza l'ultimo id associato ad una istanza di parcel
- `id`: variabile di istanza che definisce il suo id che è composto dalla lettera '`p`' + il valore di `#lastId`
- `reward`: indica il valore in punti associato all'istanza Parcel. Inizializzato con un valore casuale nell'intervallo `PARCEL_REWARD_AVG - PARCEL_REWARD_VARIANCE` a `PARCEL_REWARD_AVG + PARCEL_REWARD_VARIANCE`.
- `carriedBy`: riferimento all'istanza `Agent` che sta attualmente trasportando il Parcel.

Invece i metodi sono:

- `followCarrier`(): metodo privato che aggiorna le coordinate del pacco affinché corrispondano con quelle dell'`Agent` che lo trasporta.

- `decay()`: metodo privato che decrementa di uno il valore di `reward`, se `reward` è uguale o minore di 0 emette il segnale `parcel.'expired'` e disabilita il listener che lo chiama.

- Il costruttore che in questo caso ha un ruolo fondamentale. Infatti esso oltre che ha inizializzare le variabili di istanza:
  - aggiunge `interceptValueSet('carriedBy')` e `interceptValueSet('reward')`, metodo ereditato da `Observable` che solleva un segnale `parcel.'carriedBy'` o `parcel.'reward'` ogniqualvolta la variabile `carriedBy` o `reward` viene modificata.
  - aggiunge la variabile `lastCarrier` che serve per tenere memoria dell'ultimo portatore.
  - aggiunge il listener al segnale `parcel.'carriedBy'` il quale a sua volta
    - o abilita il listener del portatore `carriedBy` al segnale `parcel.'xy'`, emesso dall'`Agent` trasportatore quando si muove, che chiama il metodo `followCarrier()`.
    - o disabilita il listener dell'ultimo portatore `lasrCarrier` al segnale `parcel.'xy'`
  - aggiunge il listener al segnale ad uno dei seguenti segnali `clock.'1'`, `'2'`, `'5'`, `'10'` emessi dall'oggetto `myClock` definito in ***Clock.js***.
    Per scegliere il segnale da catturare viene usata la variabile `PARCE_DECADING_INTERVALL` che può essere inizializzata con uno di questi quattro valori oppure con il valore `'infinite'` che non corrisponde ad nessun segnale emesso e quindi il listener non viene mai chiamato e quindi nemmeno il metodo `decay()` rendendo la durata del pacco appunto infinita.

Queste tre variabili: `PARCEL_REWARD_AVG`,`PARCEL_REWARD_VARIANCE`,`PARCE_DECADING_INTERVALL` possono assumere il valore definito in ***process.env***, altrimenti quello definito nel file ***config.js***. Se nessuno dei due valori è definito essi vengono impostati ad un valore di default.

Tile.js

Il file implementa la classe `Tile`, la quale estende la classe `Xy` e rappresenta una casella nella griglia di gioco. Esistono 4 tipologie di caselle:

- Casella **Standard**: i vari agenti possono passare sulla casella
- Casella **Bloccata**: i vari agenti NON possono passare sulla casella
- Casella di **Spawn Pacchi**: sulla casella possono spawnare dei pacchi.
- Casella di **Delivery**: casella sulla cui gli agenti possono consegnare i pacchi raccolti ed ottenere in cambio un putaggio.

Gli attributi sono:

- `#grid`: l'istanza `Grid` a cui questa cella appartiene
- `#blocked`: un flag che indica se la cella è bloccata
- `#delivery`: un flag che indica se la cella è delivery
- `#parcelSpawner`: un flag che indica se la cella è parcel spawn
- `#locked`: un flag che indica se la cella è occupata da un agente.

I metodi sono:

- Il costruttore che accetta come parametri il riferimento alla griglia, le coordinate x e y della cella e i flag opzionali per `#blocked`, `#delivery` e `#parcelSpawner`.

- `block()`: blocca la cella e restituisce true. Restituisce false se la cella era già bloccata.
- `unblock()`: sblocca la cella e restituisce true. Restituisce false se la cella era già sbloccata

- `lock()`: rende la cella occupata e restituisce true. Restituisce false se la cella era già occupata
- `unlock()`: rende la cella libera e restituisce true. Restituisce false se la cella era già libera.

- `delivery(value)`: imposta il flag `#delivery` con il valore booleano di `value`

- `parcelSpaener(value)`: imposta il flas `#parcelSpawner` con il valore booleano di `value` .

Tutti i metodi quando cambiano uno dei valori della casella emettono il segnale `#grid.'tile'` tramite il metodo `emitOnePerTick()` ereditato da `Observable` passando come attributo loro stessi.

Grid.js

Il file implementa la classe `Grid` che definisce la griglia di gioco. La griglia di gioco è intesa come l'insieme delle caselle della mappa, l'insieme degli agenti sulla mappa e l'insieme dei pacchi presenti sulla mappa.

Gli attributi sono:

- `#tiles`: una matrice bidimensionale di oggetti `Tile` che rappresentano la vera e propria griglia di gioco.
- `#agents`: una mappa di oggetti `Agent` presenti sulla griglia con i loro id come chiave
- `#parcels`: una mappa di oggetti `Parcel` presenti sulla griglia con i loro id come chiave

I metodi sono:

- costruttore che prende come parametro una matrice bidimensionale formata da 0, 1 e 2 che verrà ricalcata per definire la griglia di gioco mediante la seguente mappatura:
  - 0 → Cella Bloccata
  - 1 → Cella di Spawn Parcel
  - 2 → Cella di Delivery

  Fatto ciò quindi inizializza la matrice `#tiles` e le mappe `#agents`, `#parcels`.

- `getTiles([x1, x2, x3, x4])`: restituisce un iterable di oggetti Title all'interno delle coordinate specificate
- `getTiles(x, y)`: restituisce l'oggetto Tile alle coordinate specificate

- `getMapSize()`: restituisce le dimensioni della griglia di gioco

- `getAgentIds()`: restituisce un array di id degli agenti nella griglia
- `getAgents()`: restituisce un iterabile degli oggetti Agent nella griglia
- `getAgent(id)`: restituisce l'oggetto Agent con l'id specificato

- `createAgent(options)`: crea un nuovo agente: `me` nella griglia con le opzioni specificate e lo restituisce. Aggiunge l'agente creato nella mappa `#agents`, emette il segnale `grid.'agent created'` passando come parametro `me` ed imposta gli ascoltatori degli eventi:

  - listener a `grid.'parcel'` e `me.'xy'` che chiamano il metodo `me.emitParcelSensing()`. Questo metodo serve per verificare e notificare se qualche pacco è nella vicinanze dell'agente

  - listener a `me.'xy'` che emette un evento `grid.'agent xy'`

  - listener a `me.'score'` che emette un evento `grid.'agent score'`

  - listener a `grid.'agent xy'`, `grid.'agent score'` e `grid.'agent deleted'` che chiamano la funzione `me.emitAgentSensing()`. Questo metodo serve per verificare e notificare se qualche altro agente è nelle vicinanze dell'agente `me`.

- `deleteAgent(agent)`: elimina l'agente specificato dalla griglia rimuovendolo da `#agents`, rimuove anche tutti i listener ad esso associati, genera l'evento `grid.'agent deleted'` ed invoca il metodo `unlock()` della cella su cui si trova per liberarla.

- `getParcels()`: restituisce un iterabile dei pacchi sulla griglia

- `getParcelsQuantitaty()`: restituisce il numero di pacchi sulla griglia.

- `createParcels(x,y)`: crea un nuovo pacco nelle coordinate specificate (x, y) nella griglia. Aggiunge il pacco alla mappa `#parcels`, emette il segnale `parcel.'parcle'` passando come parametro in nuovo pacco ed imposta gli ascoltatori agli eventi:

    - listener a `parcel.'expired'` emesso dal pacco quando esaurisce il tempo che chiama il metodo `deleteParcel(id)` per eliminarlo.

    - i listener a `parcel.'carriedBy'`, `parcel.'reward'` e `parcel.'xy'` che a loro volta emettono l'evento `grid.'parcel'`. Quindi ogni qualvolta il pacco viene modificato la griglia emette l'evento `grid.'parcel'`

- `deleteParcel(id)`: elimina il pacco con l'id specificato dalla griglia rimuovendolo da `#parcels`

Posponer.js

Questo file implementa la classe Posponer, che ha il compito di wrappare una funzione per poterla trattare come una funzione posticipata.
Quindi permette multiple chiamate alla funzione senza che essa venga eseguita, ma invece esse vengono raggruppate ed eseguite in un'unica chiamata.

Gli attributi sono:

- `finallyDo`: che è la variabile a cui sarà associata la referente della funzione de wrappare durante l'inizializzazione.
- `toBeFired`: inizializzata a false, indica se la funzione può essere eseguita
- `accumulateArgs`: è un array che ha il compito di salvare i vari argomenti passati dalle varie chiamate della funzione.

I metodi sono:

- `.atNextTick()`: la funzione viene chiamata tramite `process.nextTick()`
- `.atSetImmedite()`: la funzione viene chaamta tramite `setImmediate()`
- `.atSetTimeout()`: la funzione viene chiamata tramite setTimeout()
- `at(promise)`: la funzione viene chiamata tramite `promise.then()`, quindi viene invocata dopo l'esecuzione della promise.

Agents.js

Il file implementa la classe `Agent`, che estende `Xy` e rappresenta un player del gioco. Ogni singolo agente può navigare all'interno della mappa interagendo con altri agenti e pacchi.

Gli attributi sono:

- `#lastId`: variabile di classe che tiene traccia dell'ultimo id usato
- `#grid`: riferimento alla griglia in cui l'agente è situato
- `id`: identificativo univoco dell'istanza agente
- `name`: il nome dell'agente
- `sensing`: un set di agenti che sono nei pressi dell'agente soggetto
- `score`: il punteggio accumulato dall'agente
- `#carryingParcels`: insieme di pacchi che l'agente sta attualmente trasportando
- `config`: importa tutte le configurazioni di gioco come definite nel file ***config.js***
- `moving`: un flag che indica se l'agente si sta muovendo o no

i metodi sono:

- il costruttore oltre ad inizializzare tutti i parametri, posizionare il player in una casella random e generare un segnale `agent.'xy'`:
    - aggiunge `interceptValueSet('score')`, metodo definito in `Observable` e che emette un segnale `agent.'score'` ogniqualvolta l'attributo score viene modificato.
    - definisce i listener di `agent.'score'` e `agent.'xy'` i quali a loro volta emettono un segnale `agent.'agent'`.
- `emitAgentSensing()`: cerca e salva in un array tutti gli agenti presenti entro una certa distanza dall'agente soggetto. La distanza è definita dalla variabile `AGENT_OBSERVATION_DISTANCE` definita in `config`. Successivamente emette un segnale `agent.'agent sensing'` ed allega come attributo l'array di agenti nelle vicinanze.
- `emitParcelSensing()`: cerca e salva in un array tutti i pacchi presenti entro una certa distanza dall'agente soggetto. La distanza è definita dalla variabile `AGENT_OBSERVATION_DISTANCE` definita in `config`. Successivamente emette un segnale `agent.'parcel sensing'` ed allega come attributo l'array di pacchi nelle vicinanze.
- `tile()`: ritorna l'oggetto `Tile` su cui è posizionato l'agente
- `stepByStep(incr_x, incr_y)`: la funzione divide uno spostamento definito da `incr_x` e `incr_y` in più passi e successivamente esegue tutti i passi mediante un ciclo for.
  Il numero di passi in cui è diviso viene definito dalla variabile `MOVEMENT_STEPS` che è definita o nel file ***process.env*** oppure in ***config.js***.
  Per rendere il movimento più fluido viene definita in `config` la variabile `MOVEMENT_DURATION`, che indica la durata totale dello spostamento, la quale viene divisa per `MOVEMENT_STEPS` trovando così il lasso di tempo tra uno step e l'altro.
  Ogni iterazione del ciclo for viene ritardata mediante il metodo `synch()` di `myClock` affinché venga eseguito uno step ogni lasso di tempo.
- `move(incr_x, incr_y)`: questo metodo permette all'agente di muoversi sulla griglia di uno spostamento definito da `incr_x` e `incr_y`. Esso controlla il flag `move` per verificare che l'agente non sia già in movimento: se è in movimento termina, se non lo è esegue lo spostamento mettendo il flag a true, chiamando il metodo `stepByStep()` e riportando il flag a false.
  Infine libera tramite il metodo `unlock()` la cella in cui era precedentemente situato l'agente.

- `up()`: muove in modo sincrono a `myClock` l'agente verso l'alto mediante la chiamata `this.move(0,1)`
- `down()`: muove in modo sincrono a `myClock` l'agente verso il basso mediante la chiamata `this.move(0,-1)`
- `left()`: muove in modo sincrono a `myClock` l'agente verso sinistra mediante la chiamata `this.move(-1,0)`
- `right()`: muove in modo sincrono a `myClock` l'agente verso destra mediante la chiamata `this.move(1,0)`

- `pickUp()`: implementa in modo sincrono a `myClock` la raccolta di pacchi presenti su una cella. Esso prende da `#grid` tutti i pacchi presenti sulla griglia mediante `getParcels()`; scorre tutti i pacchi e confronta le loro coordinate con quelle dell'agente. Se corrispondono ed il pacco non è già raccolto da qualche altro agente (controllo su `carriedBy` del pacco) aggiunge il pacco ad un array di pacchi raccolti.
  Successivamente se l'array di pacchi raccolti presenta almeno un elemento viene emesso l'evento `agent.'pickup'` trasmettendo come argomento l'array di pacchi raccolti.

- `putDown()`: implementa in modo sincrono a `myClock` la deposizione dei pacchi trasportati da un agente e la loro conversioni in punti.
  Se viene passato un array di id di pacchi il metodo deposita solo i pacchi selezionati.
  La deposizione di ogni pacco prevede: la sua rimozione da `#carryingParcels`, porre a null il suo attributo `carriedBy`, una volta controllato che l'agente sia su una casella delivery aggiungere il valore di `reward` del pacco all'attributo `score`, l'aggiunta del pacco in un array di pacchi depositati e la rimozione del pacco dalla griglia mediante `#grid.deleteParcel()`.
  Successivamente se sono stati depositati dei pacchi viene emesso il segnale `agent.'putdown'` trasmettendo come argomenti l'agente stesso e l'array dei pacchi depositati.

Sensor.js

Il file implementa la classe `Sensor` che estende dalla classe `Xy`.
Le istanze di Sensor riescono a rilevare gli agenti e pacchi nelle sue vicinanze.

Classe inutilizzata.

Movable.js

Il file implementa la classe Movable che estende dalla classe `Xy`.
Essa definisce dei metodi che permettono all'oggetto di muoversi all'interno di una griglia

Classe inutilizzata.

Authentication.js

Il file implementa la classe Authentication usata per l'autenticazione dei players.

Gli attributi sono:

- `idToAgentAndSockets`: una mappa che associa gli `id` degli agenti con il rispettivo `score` dell'agente e le `sockets` associate ai singoli agenti.
- `grid`: un riferimento alla griglia del gioco.

I metodi sono:

- `registerSocketAndGetAgent(id,name,socket)`: questo metodo:
  - estrae nella variabile `entry` il record corrispondente a `id` nel `idToAgentAndSockets`; se non esiste aggiunge un record vuoto associato a quell'`id`.
  - aggiunge `socket` a `entry.sockets`.
  - ricerca nella griglia di gioco: `grid` un agente associato a `id`: `grid.getAgent(id)`. Salva il ritorno del metodo nella variabile `me`.
  - se `me` è uguale a `null`; quindi non è stato trovato nessun agente associato a quel `id`, allora:
    - viene creato un nuovo agente sulla mappa: `grid.createAgent(id, name)`.
    - aggiorna `me.score` con lo `entry.score`.
    - definisce il listener a `me.'score'` emesso quando lo score dell'agente viene modificato; il listener aggiorna `entry.score` a `me.score`.
  - Infine ritorna `me`.

- `getAgent(id)`: ritorna l'oggetto agente: `agent` associato al fornito `id`.

- `getSockets(id)`: ritorna un iterabile con tutte le sockets associate all'agente: `agent` associato ad `id`.

- `authenticate(socket)`: questo metodo gestisce l'autenticazione degli agenti dividendoli in 2 casi:
  - **Nuovo Utente**: la socket non invia nessun token, quindi è un nuovo utente. Viene generato un nuovo `id` per la socket e viene salvato il `name` per l'agente dato dalla socket.
    Viene quindi generato un nuovo token usando come payload `id`, `name` e la variabile `SUPER_SECRET` definita nel file ***process.env***.
    Viene infine emesso il segnale `socket.'token'` con attributo il token creato.
  - **Vecchio Utente**: la socket invia un token, quindi l'utente ha già fatto un accesso al gioco. Il token viene decodificato, se il token è valido vengono estratte i valori di `id` e `name`, altrimenti viene tornato errore e la socket viene disconnessa.
    Al momento della disconnessione viene tolta `socket` dall'array `sockets` associato all'`id` estratto dal token invalido nella mappa `idToAgentAndSockets`. Poi viene controllato se sockets è restato vuoto, nel caso il server aspetta tot millisecondi poi l'oggetto agente viene rimosso dalla griglia: `grid.deleteAgent()`.
    Il tot millisecondi che vengono aspettati sono specificati da `AGENT_TIMEOUT` definita nel file ***process.env***, altrimenti settata a un valore di default: `10000`.

  Alla fine di entrambe le 2 procedure, se la socket non viene disconnessa, viene aggiunto all'array `sockets` associato ad `id` in `idToAgentAndSockets` una nuova socket: `socket`; mediante la chiamata di `registerSocketAndGetAgent(id,name,socket)`

.env

File che contiene informazioni chiave per il funzionamento del gioco. Per questo non viene condiviso su github. Nella repository è presente anche un file di esempio .env chiamato .env.example.

Essa può contenere:

- **`MAP_FILE`** : indica il nome della mappa da caricare sulla griglia di gioco
  - ➢ **valore** di **default**: `'default_map'`

- **`PARCELS_GENERATION_INTERVAL`** : indica l'intervallo di tempo tra la generazione di un pacco e quello successivo.
  - ➢ **valore** di **default**: `'2s'`
- **`PARCELS_MAX`** : indica il numero massimo di pacchi che possono essere presenti sulla mappa
  - ➢ **valore** di **default**: `'infinite'`
- **`PARCEL_REWARD_AVG`** : indica il valore medio della ricompensa dei pacchi
  - ➢ **valore** di **default**: `30`
- **`PARCEL_REWARD_VARIANCE`** : indica la variazione massima della ricompensa di un singolo pacco rispetto al valore medio.
  - ➢ **valore** di **default**: `10`
- **`PARCE_DECADING_INTERVALL`** : indica l'intervallo di tempo per cui la ricompensa del pacco diminuisce
  - ➢ **valore** di **default**: `'infinite'`

- **`RANDOMLY_MOVING_AGENTS`** : indica il numero di agenti 'boot' presenti sulla mappa
  - ➢ **valore** di **default**: `0`
- **`RANDOM_AGENT_SPEED`** : indica l'intervallo di tempo tra due mosse di un agente boot
  - ➢ **valore** di **default**: `'2s'`
- **`MOVEMENT_STEPS`** : indica in quanti step viene division uno spostamento di un agente
  - ➢ **valore** di **default**: `1`

- **`CLOCK`** : indica la durate in millisecondi di un clock di gioco
  - ➢ **valore** di **default**: `50`

- **`SUPER_SECRET`** : indica la chiave su cui vengono creati i token
  - ➢ **nessun valore** di **default**
- **`AGENT_TIMEOUT`** : indica il tempo di attesa per un agente senza socket prima di essere eliminato.
  - ➢ **valore** di **default**: `10000`

- **`REDIS_URL`**: indica l'URL per accedere al database
  - ➢ **nessun valore** di **default**

- **`LEVEL`**: indica il livello di gioco con cui si definirà il file ***config.js***
  - ➢ **valore** di **default**: `'2'`

- **`PORT`** : indica la porta su cui il server si mette in ascolto
  - ➢ **valore** di **default**: `8080`

config.js

File in cui è definito l'oggetto config che contiene le configurazioni del livello del gioco.

viene inizializzata una versione default di config con:

```
MAP_FILE : 'default_map';
PARCELS_GENERATION_INTERVAL: '2s'
PARCELS_MAX : '5'
MOVEMENT_STEPS : 2
MOVEMENT_DURATION : 50
AGENT_OBSERVATION_DISTANCE : 5
PARCELS_OBSERVATION_DISTANCE : 5
AGENT_TIMEOUT : 10000
PARCEL_REWARD_AVG : 30
PARCEL_REWARD_VARINCE : 10
PARCEL_DECADING_ITERVAL : '1s'
RANDOMLY_MOVING_AGENTS : 2
RANDOME_AGENT_SPEED : '2s'
CLOCK : 50
```

Questa versione di default però può venire sovrascritta se nel file ***.env*** è presente l'attributo **`LEVEL`**; esso va a specificare un livello nella cartella ***/levels***.

packages/unitn-asa

deliveroo-js-webapp

include tutti i file statici inviati al client sul web.

index.html

definisce il file html mandato al client. All'interno è incluso lo script ***deliveroo.js*** che gestisce tutto il gioco lato client.

deliveroo.js

Implementa il gioco lato client, e per far ciò usa le seguenti librerie

- `socket.io-client` come usato per la comunicazione WebSocket
- `EventEmitter` per la gestione degli eventi
- `THREE` per il rendering 3D
- `OrbitControls` per il controllo della telecamera
- `CSS2DRenderer` per il rendering di elementi HTML/CSS nella scena 3D
- `GUI` per elementi dell'interfaccia utente grafica

Possiamo dividere il codice di deliveroo.js in:

1) Creazione e Set-Up scena 3D

Per usare una visione 3D usa la libreria **three.js**; quindi prima inizializza la variabile `scena` come una scena 3D.

Viene configurata una telecamera prospettica posizionata a `(-1,2,2)`, e viene creato un `CSS2DRenderer` per il rendering di elementi HTML/CSS in 2D.

Aggiunge un listener per il ridimensionamento della finestra, che aggiorna il rapporto della telecamera e le dimensioni del render di conseguenza.

Poi passa alla configurazione della **telecamera**: definendo il suo controllore `controls` e settare delle sue proprietà come distanza minima e massima ed angoli.

Definisce all'interno della funzione `createPanel()` il pannello di gioco contenente le informazioni sulla partita; esso:

- inizializza la variabile `panel` con una nuova istanza di `GUI`
- definisce 3 cartelle nel pannello:
  - `tokenFolder`: contenente il token del giocatore
  - `chatFolder`: contenente la chat di gioco
  - `leaderboardFolder`: contenente la classifica dei punti dei vari agent.
- definisce l'oggetto `players` come una mappa contenente tutti i giocatori
- definisce 2 funzioni:
  - `processMsg(id,name,msg)`: definisce un oggetto `line` costituito da una intestazione formata da `id+' '+name` ed un messaggio: `msg`. Aggiunge alla cartella `chatFolder` la coppia `line` ed `id+' '+name` .

- `updateLeaderboard(agent)`: essa controlla se l'agente è già presenta nella lista dei giocatori: `players` e:
  - Se **non** esiste viene creato un oggetto `player` con chiave pari a `nome` di `agent` ed il valore il `score` dell'agente.
    Viene quindi creato un nuovo controllore: `controller` che rappresenta l'agnete nella leaderboard; essa ha come parametro l'`agent.score`.
    il controllore viene aggiunto alla lista `players` con chiave `agent.name`.
  - Se esiste viene aggiornato il punteggio del giocatore chiamando il metodo `.setValue(agent.score)` sul controllore dell'agente associato in `players`.

Successivamente viene chiamata la stessa funzione `createPanel()` per eseguirla.

Definisce ed aggiunge alla scena 2 frecce che rappresentano i versori dell'asse `x` e `z` della scena 3D

2) Definizione delle classi per gli oggetti 3D:

`onGrid`: classe che rappresenta un oggetto su una griglia in una scena 3D.
attributi:

- `#mesh`: indica la forma geometrica dell'oggetto nella scena 3D
- `#x` e `#y`: salvano le coordinate dell'oggetto sulla griglia
- `#carriedBy`: tiene traccia del portatore dell'oggetto, se non è portato da nessuno è settato ad `undefined`.
- `#text`: salva un testo associato all'oggetto: Questo verrà mostrato al disopra dell'oggetto nella scena 3D all'interno di una etichetta
- `#div`: rappresenta un elemento HTML `<div>`, che viene usato per definire l'etichetta sopra l'oggetto nella scena 3D
- `#label`: rappresenta un oggetto `CSS2DObject` associato all'elemento `#div`. Esso permette di aggiungere l'etichetta all'oggetto nella scena 3D.

metodi:

- il costruttore che prende in ingresso una forma geometrica: `mesh`, delle coordinate `x` ed `y` ed un testo `text`. Poi:
  - Aggiunge `mesh` alla lista degli oggetti cliccabili: `clickables`
  - Inizializza `#mesh` con `mesh`, le coordinate `#x` ed `#y` con `x` e `y`
  - Inizializza la `#mesh.position`: `x = x*1.5`; `y=0.5` e `z=-y*1.5`
  - Inizializza `#text`, `#div` e `#label` con il testo `text`
  - Definisce il listener all'evento `animator.'animate'` che si occupa di spostare l'oggetto in modo fluido mediante il metodo `#mesh.position.lerp`. Controlla la posizione dell'oggetto e verifica se è posizionato su una casella oppure in una posizione intermedia e aggiusta di conseguenza la velocità dello spostamento.

- `opacity(opacity)`: definisce l'opacità dell'oggetto settando i parametri `#mesh.material.opacity` e `#label.element.style.visibility` uguali a `opacity`

- `color(color)`: definisce il colore dell'oggetto settando il parametro `#mesh.material.opacity` a `color`

- `pickup(agent)`: implementa la funzione di raccolta dell'oggetto da parte di un agente: `agent`:
  - Essa aggiorna `#carriedBy` ponendolo uguale ad `agent`.
  - Aggiunge a `#mesh` del portatore il suo mesh
  - Aggiunge all'attributo `carrying` del portatore il record `this.id`, `this`.
  - Viene rimosso il suo mesh dalla scena: `scene.remove(this.#mesh)`
  - Azzera le sue coordinate `#x` e `#y`
  - Aggiorna le coordinate della posizione del suo mesh ponendo `#mesh.position.x` e `#mesh.position.z` a `0`; ed `#mesh.position.y` ad `0.5` indicando che esso è sollevato dalla griglia

- `putdown(x,y)`: implementa la funzionalità di scarico dell'oggetto da parte del suo portatore:
  - Rimuove dal `#mesh` del portatore il su mesh: this.`#mesh`
  - Rimuove l'oggetto dal `carrying` del portatore
  - Aggiorna le sue coordinate `#x` e `#y` ad `x` e `y`
  - Aggiorna la posizione del suo `#mesh`: `position.x` e `position.z` vengono messi uguali alle coordinate del mesh del portatore; mentre `position.y` viene posta a `0.5`.
  - Ripristina `#carriedBy` ad `undefined`
  - Aggiunge a `scene` il suo `#mesh`

- `removeMesh()`: implementa la funzionalità di resettare il mesh dell'oggetto e rimuoverlo dalla scena.

`PathPoint`: la classe definisce un oggetto punto nello spazio 3D.
attributi:
- `sphere`: attributo che contiene la forma geometrica sfera

metodi:
- il costruttore prende come parametri delle coordinate `x` e `y`. Definisce la forma geometrica sfera ed il materiale di essa; per poi definire il mesh sfera che viene associato all'attributo `sphere`. La sfera viene inizializzata come trasparente.
  Poi posiziona la sfera alle coordinate 3D: `(x*1.5, 0.1, -y*1.5 )`

- `show()`: rende visibile la sfera settando `sphere.material.opacity` a `1`

- `hide()`: rende invisibile la sfera settando `sphere.material.opacity` a `0`

`Tile`: la classe estende `onGrid` e rappresenta una cella nello spazio 3D

attributi:

- `#delivery`: flag che indica se la casella è una casella di consegna: colore **rosso**
- `#parcelSpawner`: flag che indica se la casella è una casella di spawn: colore **verde chiaro**
- `#blocked`: flag che inidca se la casella è bloccata: colore **nero**
- `pathPoint`: un punto associato alla casella; esso viene usato per definire un percorso. Infatti il server può inviare un percorso sulla mappa e le caselle che sono comprese nel percorso sono evidenziate mediante questo punto.

metodi:

- il costruttore prende come parametri **`x`**, **`y`** e un flag **`delivery`**. Definisce la forma geometrica di un cubo schiacciato: come una vera casella e definisce il colore in base al valore di **`delivery`**.
  Poi inizializza un oggetto mesh: `cube` con la geometria ed il colore scelto e lo aggiunge a `scene`.
  Richiama il costruttore di `onGrid` che salva le coordinate **`x`** e **`y`** in `#x` e `#y` , la forma `cube` in `#mesh` ed aggiorna la coordinata `#mesh.position.y` a `0`.
  Infine inizializza `pathPoint` con un nuovo punto nelle coordinate **`x`** e **`y`**.

`Parcel`: la classe estende `onGrid` e rappresenta un pacco nello spazio 3D

attributi:

- `id`: identificatore del pacco
- `#reward`: valore di ricompensa del pacco

metodi:

- il costruttore prende come parametri **`id`**, **`x`**, **`y`**, **`carriedBy`** e **`reward`**. Definisce la forma geometrica di un perfetto cubo e definisce un colore casuale.
  Poi inizializza un oggetto mesh: `parcel` con la geometria ed il colore scelto e lo aggiunge a `scene`.
  Richiama il costruttore di `onGrid` che salva le coordinate **`x`** e **`y`** in `#x` e `#y` , la forma `parcel` in `#mesh`.
  Inizializza i parametri id e # reward con **`id`** e **`reward`**. Infine controlla il parametro **`carriedBy`** e in caso fosse definito chiama la funzione `getOrCreateAgent(carriedBy)`.
- viene ridefinito il metodo set del parametro `#reward` che va ad aggiornare sia il parametro stesso ma anche il valore del parametro `#text` ereditato da `onGrid`.

`Agent`: la classe estende `onGrid` e rappresenta un agente nello spazio 3D

attributi:

- `id`: identificatore del pacco
- `carrying`: definisce una mappa contenente i pacchi raccolti dall'agente salvando record formati dall'id dell'oggetto e l'oggetto stesso.
- `#name`: nome dell'agente
- `#score`: punteggio dell'agente

metodi:

- il costruttore prende come parametri **`id`**, **`x`**, **`y`**, **`name`** e **`score`**. Definisce la forma geometrica di un cono e definisce un colore casuale.
  Poi inizializza un oggetto mesh: `mesh` con la geometria ed il colore scelto e lo aggiunge a `scene`.

Richiama il costruttore di `onGrid` che salva le coordinate `x` e `y` in `#x` e `#y` , la forma `parcel` in `#mesh` e le informazioni `id` e `score` in `#text`.
Inizializza i parametri `id`, `#score` e `#name` con `id`, `name` e `score`.

3) Setup connessione

Controlla i parametri dell'URL alla ricerca del parametro `name` che viene usato per inizializzare la variabile `name`. Se il parametro è presente vuol dire che la socket è già collegata ad un agente nel gioco, altrimenti vuol dire che è una nuova socket.

Se il parametro name non è presente allora viene mostrato a schermo un pop-up in cui è richiesto all'utente di inserire un nome: il valore inserito viene salvato in `name`.

Dichiara la variabile `token` e la inizializza con il valore ritornato dal `checkCookieForToken(name)` `checkCookieForToken(name)` è una funzione che controlla se esiste il cookie `'token_'+name`:
se **si** ritorna il valore associato; se **no** ritorna il valore di default: `""`

Inizializza la variabile socket con una connessione Socket.IO definendo vari attributi nelle sezioni:

- sezione `extraHeaders` definisce l'attributo `'x_token'` e lo inizializza con `token`
- sezione `query` definisce l'attributo `'name'` e lo inizializza con il parametro `name` dell'URL

Infine inizializza la variabile `me` con un agente default: `getOrCreateAgent('loading',name,0,0,0)`

4) Setup mappa di gioco

Dichiara le variabili per definire i vari componenti della mappa di gioco, e le inizializza implementando i listener agli eventi emessi da `socket`:

- Definisce un array vuoto di oggetti cliccabili: `clickables`
- Dichiare le variabili `WIDTH` e `HEIGHT`: larghezza e altezza della mappa di gioco.
- Implementa il listener a all'evento `socket.'map'` ed inizializza `WIDTH` e `HEIGHT` con i valori degli attributi larghezza ed altezza dell'evento.
- Dichiara le variabili:
  - `agents`: una mappa degli agenti presenti sul gioco. Esso mappa gli oggetti `Agent` con chiave il loro stesso `id`.
  - `parcels`: una mappa dei pacchi presenti sul gioco. Esso mappa gli oggetti `Parcel` con chiave il loro stesso `id`.
  - `tiles`: una mappa delle caselle presenti sul gioco. Esso mappa gli oggetti `Tile` con chiave la somma delle loro coordinate: `x+y*1000`.
- Definisce le funzioni per andare a prendere ed inserire elementi nelle 3 mappe: `agents`, `parcels`, `tiles`:
  - `setTile(x,y,delivery)`: controlla se in `tiles` c'è già una casella in quelle coordinate:
    - se si sovrascrive l'oggetto chiamando il costruttore `new Tile(x,y,delivery)`
    - se no crea l'oggetto `new Tile(x,y,delivery)` e lo inserisce in `tiles`.
  - `getTile(x,y)`: controlla se in `tiles` c'è una casella in quelle coordinate, se no ne crea una: `new Tile(x,y)`; ed infine ritorna la casella in quelle coordinate.
  - `getOrCreateParcel(id,x,y,carriedBy,reward)`: controlla se in `parcels` c'è già un pacco con quel `id`; se non c'è lo crea chiamando `new Parcel(id, x, y, carriedBy, reward)` e lo inserisce in `parcels`. Infine ritorna il pacco associato a quel `id`.

- `deleteParcel(id)`: tramite `getOrCreateParcel()` trova il pacco associato a `id` e su esso chiama il metodo `romoveMash()` per rimuovere la sua forma dalla scena. Infine rimuove il pacco anche da `parcels`.
- `getOrCreateAgent(id,name,x,y,score)`: controlla se in `agents` c'è già un agente con quel `id`; se non c'è lo crea chiamando `new Agent(id,name,x,y,score)` e lo inserisce in `agents`. Infine ritorna l'agente associato a quel `id`.

- Definisce ed implementa il metodo `window.getMap()` che stampa in console ed ritorna la struttura della mappa sotto forma di una stringa. La stringa contiene un quadrato di numeri dove ogni numero rappresenta una casella e in base al suo valore rappresenta anche il tipo di casella:
  - **0** => Bloccata
  - **1** => Delivery
  - **2** => Spawner
  - **3** => Standard

  Per definire questa stringa essa cila `tiles` controllando ogni volta la tipologia della cella.

5) Implementa i listener agli eventi sollevati dalla socket:

Ciò permette al client di sapere cosa avviene all'interno del gioco e aggiornare quindi le sue informazioni. I listener implementati sono:

- a `socket.'connect'`: definisce il testo dello span contenuto in ***index.html*** con id pari a `'socket.id'` ponendolo pari a `'socket.id'+socket.id`.

- a `socket.'diconect'` e ad `socket.'connect_error'` che stampano in console gli errori di connessione.

- a `socket.'token'` che viene emessa dopo l'autenticazione, esso prende il token passato come argomento dell'evento e lo salva nel cookie `'token_'+name`.

- a `socket.'log'` che stampa alcuni parametri degli argomenti passati con l'evento.

- a `socket.'not_tile'` emesso per indicare una casella bloccata; il listener prende le coordinate passate con l'evento e le usa per definire un casella bloccata: `getTile(x,y).blocked=true`

- a `socket.'tile'` emesso per indicare una casella non bloccata; il listener prende le coordinate passate con l'evento e i flag `delivery` e `parcelSpawner` per definire una determinata casella: `getTile(x,y).blocked = true` `getTile(x,y).delivery = delivery` `getTile(x,y).parcelSpawner = parcelSpawner`.

- a `socket.'msg'` emesso dal server per consegnare un messaggio; il listener stampa in console il messaggio e chiama la funzione `processMsg()` passando come parametri gli attributi dell'evento

- a `socket.'path'` emesso dal server per evidenziare un percorso passato come argomento dell'evento: `path`. Il listener prima cicla `tiles` accedendo tutte le caselle per chiamare il metodo `hide()` sul loro attributo `pathPoint`. Successivamente cicla tutte le coordinate contenute in `path`, accede alle caselle in quelle coordinate e rende visibile il `pathPoint` chiamando il metodo `show()`.

- a **socket.'config'** emesso dal server per definire le configurazioni dell'agente trasmesse come argomento **config**. Il listener prima di tutto imposta il testo dello span con id "config", definito nel file ***index.html***, pari al contenuto di **config**. Poi inizializza 3 variabili:
  - **AGNETS_OBSERVATION_DISTANCE = config.AGNETS_OBSERVATION_DISTANCE**
  - **PARCELS_OBSERVATION_DISTANCE = config.PARCELS_OBSERVATION_DISTANCE**
  - **CONFIG = config**

- a **socket.'you'** emesso dal server ogniqualvolta i dati dell'agente vengono modificati con allegati i seguenti argomenti: **id**, **name**, **x**, **y** e **score** dell'agente. Il listener:
  - Prima di tutto imposta il testo di 3 span definiti nel file ***index.html***:
    - span con id "agnet.id" con **id**.
    - span con id "agnet.name" con **name**.
    - span con id "agnet.xy" con **x** e **y**.
  - Aggiorna l'agente **me** con i dati inviati al listener.
  - Passa in rassegna tutte le caselle ciclando **tiles**; calcola la distanza tra **me** e la singola casella; confronta se la distanza è maggiore di **AGNETS_OBSERVATION_DISTANCE** e di **PARCELS_OBSERVATION_DISTANCE** e in base a ciò posta l'opacità della casella a 1 o ad 0.1.
  - Infine chiama la funzione **updateLeaderboard(me)** per aggiungerlo sulla board degli agenti.

- a **socket.'agents sensing'** emesso con argomenti **sensed**: un array di agenti presenti nei dintorni dell'agente della socket: **me**. Il listener
  - Definisce **sensed_ids** la lista di id degli agenti contenuti in **sensed**
  - Cicla tutti gli agenti in **agents** e guarda se l'id dell'agente **NON** corrisponde all'id di **me** o se **NON** è presente in **sensed_ids**; nel caso una delle due condizioni è vera viene settato l'attributo **opacity** dell'agente a 0, rendendolo invisibile.
  - Infine cicla tutto **sensed** e per ogni *agente* chiama **var agent = getOrCreateAgent()**. Dove **agent** e l'istanza **Agent** associata all'*agente* ciclato; così facendo se l'*agente* non era presente nella lista **agents** viene aggiunto. Poi aggiorna tutti i parametri di **agent** con quelli dell'*agente* e se lo **score** viene aggiornato viene chiamata la funzione **updateLeaderboard(agent)**.

- a **socket.'parcels sensing'** emesso con argomenti **sensed**: un array di pacchi presenti nei dintorni dell'agente della socket: **me**. Il listener
  - Definisce **sensed_ids** la lista di id dei pacchi contenuti in **sensed**
  - Cicla tutti i pacchi in **parcels** e guarda se il loro id **NON** è presente in **sensed_ids**; nel caso il pacco viene eliminato con **deleteParcel()**.
  - Cicla tutto **sensed** e per ogni *pacco* chiama **var was = getOrCreateParcel()**. Dove **was** e l'istanza **Parcel** associata al *pacco* ciclato; così facendo se il *pacco* non era presente nella lista **parcels** viene aggiunto.
    Poi controlla se **carriedBy** del *pacco* e **true** e **was.carriedBy** è **false**; in questo caso vuol dire che il pacco è stato raccolto quindi il listener salva in **agent** l'agente portatore del pacco e chiama **was.picked(agent)**.
    Poi controlla se **carriedBy** del *pacco* e **false** e **was.carriedBy** è **true**; in questo caso vuol dire che il pacco è stato posato quindi il listener chiama **was.picked(x, y)**, dove **x, y** sono le coordinate del *pacco*. Infine aggiorna tutti gli altri attributi di **was** con quelli del *pacco*.

6) Definisce i comandi dell'utente

I primi comandi implementati sono quelli speciali solo per agent con nome `'god'`:

- Definisce il flag `enable_tile_mode` inizializzato a false e che viene posto a true quando il pulsante shift viene premuto.
- Implementa un gestore di eventi che controlla i click dell'utente sulla scena 3D, controlla se il click è stato effettuato su un oggetto cliccabile in `clickables` ed eventualmente:
  1. converte la posizione 3D in coordinate della griglia: `x` e `y`.
  2. controlla il valore del flag `enable_tile_mode` ed
     - Se **true**: emette il segnale `socket.'tile'` con argomenti `x` e `y`.
     - Se **false**: verifica se esiste un pacco nelle coordinate `x` e `y` tramite un controllo su `parcels`. In base a ciò viene emesso il segnale:
       - `socket.'dispose parcel'` con argomenti `x` e `y` per eliminare il pacco resente
       - `socket.'create parcel'` con argomenti `x` e `y` per creare un nuovo pacco.

Successivamente vengono implementati i comandi standard:

- Per essi viene definita, ed inizializzata a `null`, la variabile `action`: che verrà associata alla funzione che implementa l'azione da svolgere.
- Viene definita la funzione asincrona `start_doing()` che esegue un ciclo while finché `action` è diverso da `null`. Nel ciclo while:
  - Viene chiamata `action()` il cui risultato è salvato nella variabile `res`
  - Viene controllato il valore di `res` e se è `false` il while si interrompe e la funzione si arresta
- Viene implementato il listener al rilascio dei tasti che setta `action` a `null`
- Viene implementato il listener al click dei tasti il quale inizializza `action` in base al tasto premuto:
  - **'Q'** => `action` emette l'evento `socket.'picket'` e restituisce come valore l'esito dell'operazione.
  - **'E'** => `action` emette l'evento `socket.'putdown'` e restituisce come valore l'esito dell'operazione.
  - **'W'** => `action` emette l'evento `socket.'move'` con argomento `'up'` e restituisce come valore l'esito dell'operazione.
  - **'A'** => `action` emette l'evento `socket.'move'` con argomento `'left'` e restituisce come valore l'esito dell'operazione.
  - **'S'** => `action` emette l'evento `socket.'move'` con argomento `'down'` e restituisce come valore l'esito dell'operazione.
  - **'D'** => `action` emette l'evento `socket.'move'` con argomento `'right'` e restituisce come valore l'esito dell'operazione.

7) ANIMAZIONE

Definisce una variabile `animator` come istanza di **EventEmitter**.
Definisce e chiama la funzione `animate()` che è il cuore dell'animazione user interface. È il corrispettivo del `main()`.

Nella funzione `animate()`:

- Emette il segnale `animator.'animate'`
- Sposta la telecamera per centrarla sull'agente corrispondente alla socket del client: `me`
- Aggiorna i controlli della telecamera: `controls.update()` ed aggiorna i render
- E attraverso `requestAnimationFrame(animate)` chiama se stessa all'infinito