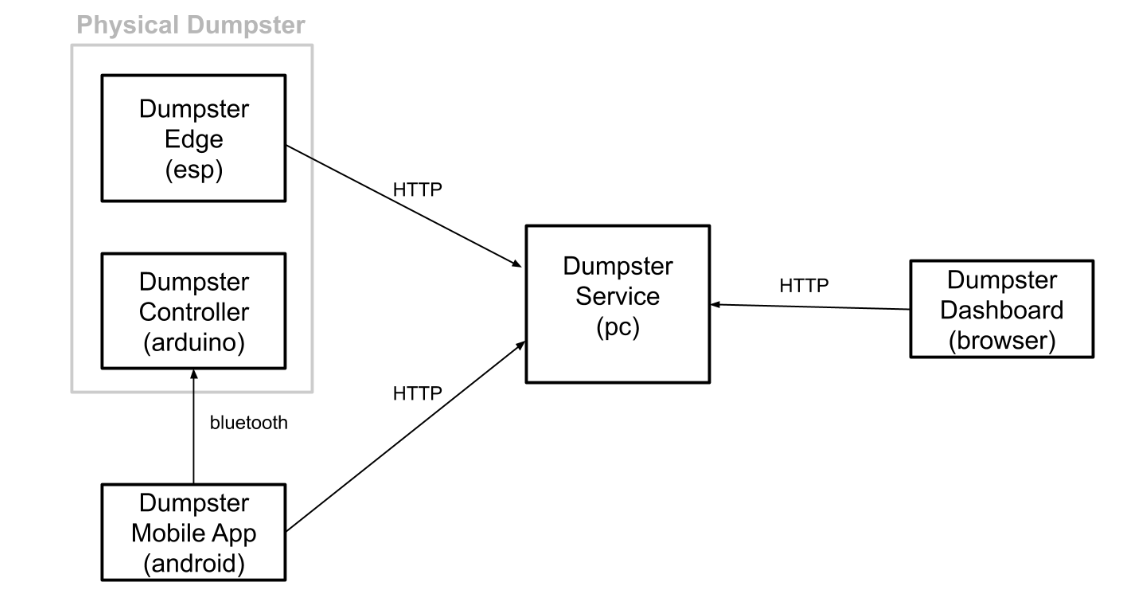
**RELAZIONE Progetto #3 – *Smart Dumpster***

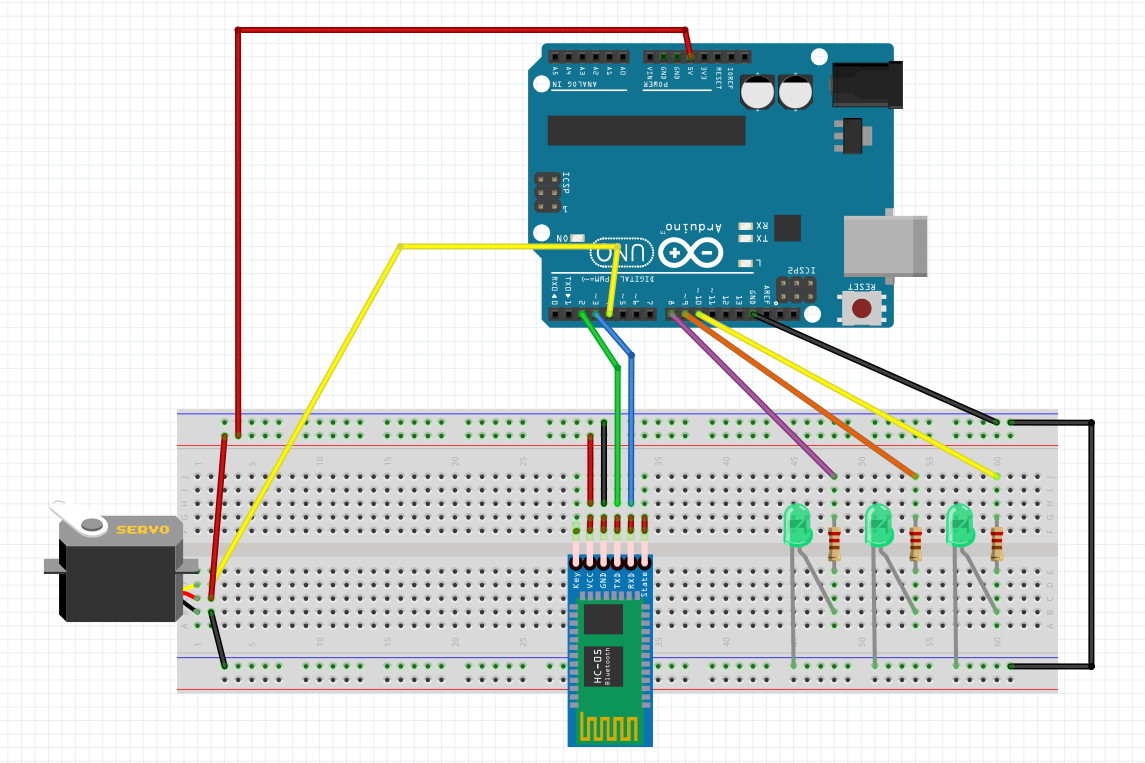
Autori: Matteo Ragazzini, Marta Spadoni

Le specifiche richiedono di realizzare uno smart dumpster, un sistema iot caratterizzato da 5 sottosistemi: Dumpster controller, Dumpster mobile app, Dumpster edge , Dumpster service e Dumpster dashboard, questi comunicheranno tra loro attraverso diverse tecnologie di comunicazione. Riportiamo di seguito lo schema esplicativo fornitoci dalle specifiche e successivamente analizziamo in dettaglio ogni componente dell’ecosistema.



**Dumpster Controller**

Questo componente basato su piattaforma arduino rappresenta idealmente (insieme al dumpster edge, trattato in seguito) il dumpster fisico, composto da un servo che simula il portellone del cassonetto, il modulo HC-05 che permette la comunicazione via bluetooth con l’app mobile e tre led che corrispondono ai 3 tipi di rifiuti che è possibile depositare. Alleghiamo di seguito lo schema fritzing rappresentante i collegamenti della bord.



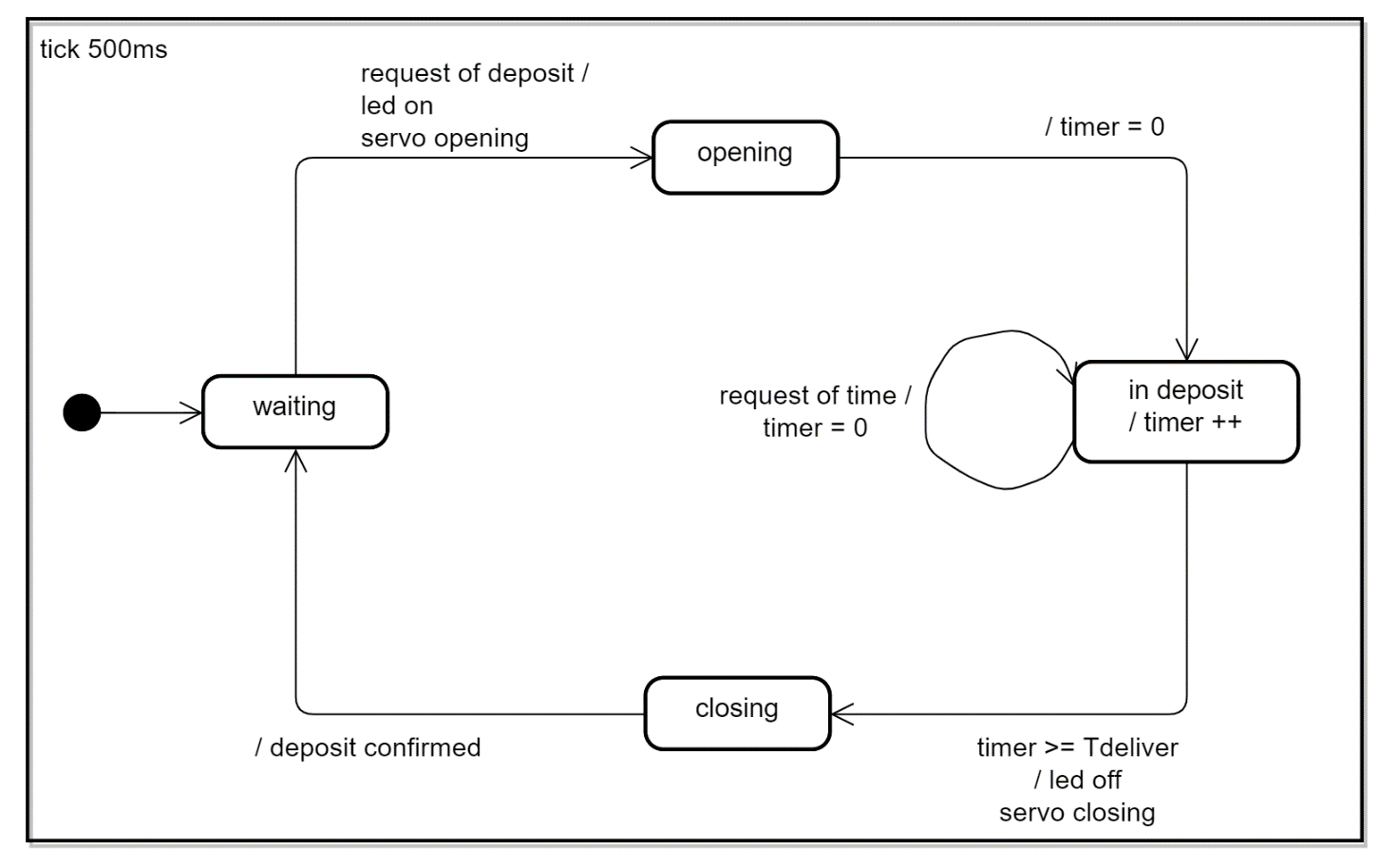
Per la realizzazione del controller veniva richiesta una logica in termini di macchina a stati finiti sincrona. Abbiamo quindi identificato i 4 stati che caratterizzano il suo funzionamento :

* *waiting*, stato durante il quale il dumpster aspetta una richiesta di deposito, questa viene comunicata dall’app sulla connessione bluetooth e il messaggio ricevuto è la posizione nell’array del led corrispondente al tipo di rifiuto selezionato;
* *opening*, stato di transizione nel quale viene acceso il led corrispondente al tipo di rifiuto da depositare e aperto il portellone, questo stato, come quello di *closing*, è necessario per dare il tempo al servo di assumere la posizione indicata;
* *in deposit*, fase in cui viene contrallata la durata del deposito, da noi settata a 5s, e gestite eventuali richieste, provenienti via bluetooth, di ulteriore tempo. Per realizzare il timer abbiamo sfruttato l’archittetura sincrona della nostra soluzione, considerando come tempo di tick 500ms e contando il numero di iterazioni che la macchina fa in questo stato, arrivati a 10 terminiamo il deposito. Nel caso in cui si registri una richiesta di tempo aggiuntivo, azzeriamo la variabile che tiene il conto delle iterazioni, concedendo così all’utente ulteriori 5s;
* *closing,* secondo stato di transizione nel quale viene spento il led acceso nella fase *opening*, chiuso il portellone, e notificata l’applicazione dell’avvenuto deposito con un messaggio sul canale bluetooth.

Avendo un solo task da eseguire, abbiamo sfruttato l’architettura super-loop di wiring per andare a richiamare ogni 500ms il metodo tick del controller, evitando quindi di inserire uno scheduler.

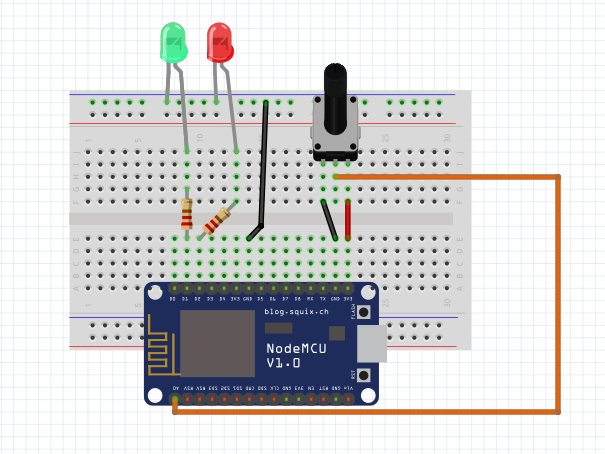
Per la comunicazione via bluetooth con l’app abbiamo sfruttato le classi messe a disposizione nel repository del corso, queste wrappano la SoftwareSerial creata sui pin ai quali è collegato il modulo e forniscono i metodi necessari allo scambio di messaggi.

Riportiamo di seguito il diagramma degli stati.



**Dumpster Edge**

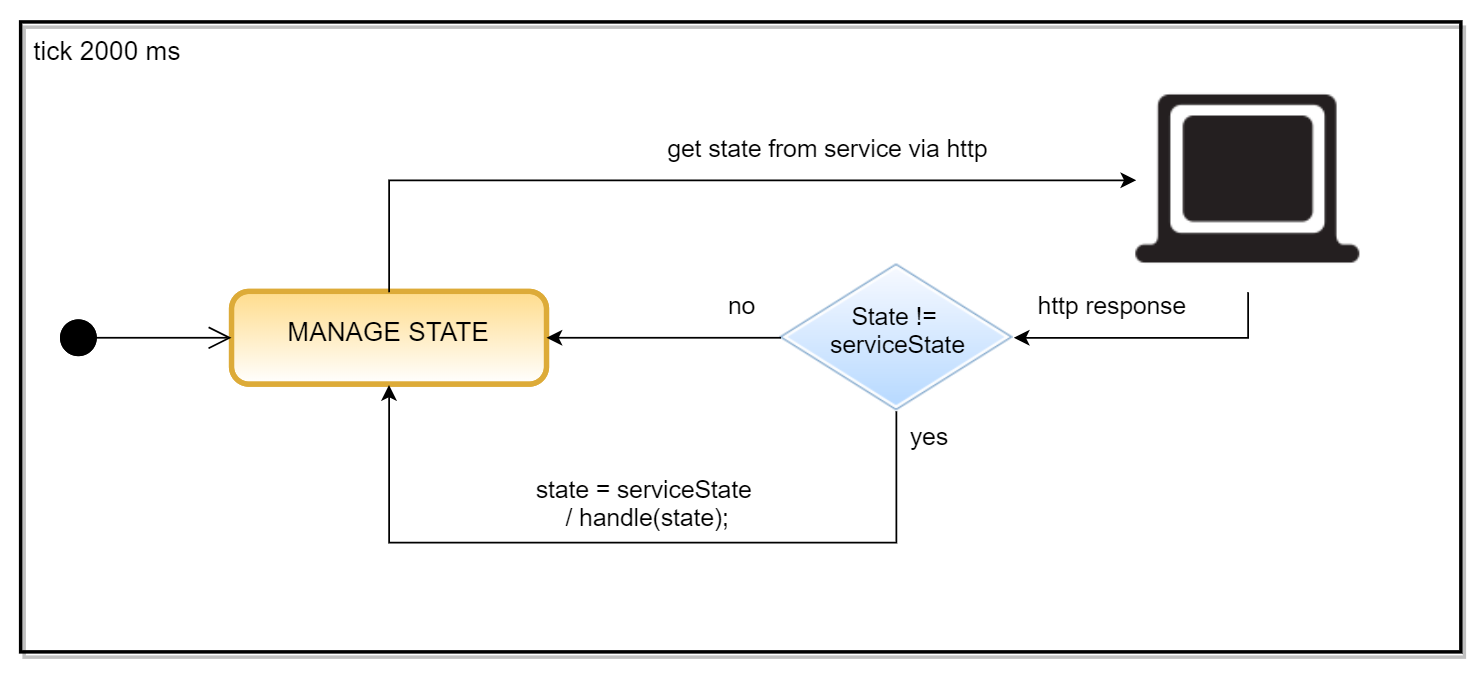
Il dumpster edge realizzato tramite scheda ESP8826 wifi, va a completare la parte fisica del dumpster aggiungendo una comunicazione http con il service, cuore dell’ecosistema. L’edge presenta 2 led, uno verde ed uno rosso rappresentanti lo stato del dumpster, rispettivamente *available*  e *not available* e un potenziometro che simula un sensore di peso posizionato all’interno del cassonetto, questo misura l’aumento del peso complessivo a seguito di ogni deposito. Riportiamo di seguito lo schema fritzing di questo componente.



Per la realizzazione del dumpster edge, non vi era nessun vincolo sul tipo di architettura da utilizzare, così abbiamo proceduto andando ad analizzare il comportamento dello stesso per capire quale, secondo noi, fosse la soluzione migliore da adottare. Considerando la specifica che permette la variariazione dello stato del dumpster direttamente dalla dashboard, abbiamo identificato la necessità di una continua richiesta di aggiornamento da parte dell’ESP verso il service, per conoscere il vero stato attuale del sistema. Abbiamo quindi ritenuto che l’architettura più adatta fosse una specie di semplice architettura ad eventi e reazioni sincrona, gli eventi sono i diversi cambi di stato che possono avvenire, ad esempio da available e not available, mentre le reazioni sono gli handle associati ai nuovi stati registrati. Il sincronismo dell’architettura è dato dal fatto che la funzione manageState(), responsabile del comportamento sopra descritto, viene richiamata dal loop ogni 2sec. Questo tempo è stato trovato effettuando numerosi test e riteniamo che sia un buon compromesso fra reattività del sistema e funzionamento dello stesso. Infatti, abbiamo notato che con un tempo ridotto alcune richieste GET e POST non andavano a buon fine. La prima tipologia di messaggio http viene appunto utilizzata per ricevere lo stato corrente del dumpster, mentre la seconda serve a comunicare al server, con un file JSON, i dati aggiornati all’ultimo deposito effettuato, in particolare: peso netto del deposito, disponibilità residua, e peso complessivo. Avendo a disposizione il potenziometro che misura unicamente il peso complessvo, calcoliamo il peso netto come la variazione tra il precedente peso complessivo memorizzato e quello registrato al momento del calcolo. Inoltre, abbiamo mappato i 1023 livelli del pot in una scala di peso che varia da 0 a circa 500kg, usando la funzione map.

L’invio dei dati avviene come reazione alla ricezione dello stato “DEPOSIT”, in questo modo evitiamo di fare POST “inutili” e di reagire a variazioni del peso del pot non realmente corrispondenti ad un deposito.

Abbiamo inoltre fatto la scelta di mantenere sempre attiva un’unica connessione http con il service al fine di garantire una maggiore stabilità del sistema e delle comunicazioni, infatti ngrok non permette di creare più di 20 connessioni al minuto.

Riportiamo di seguito il diagramma del dumpster edge.

**Dumpster Mobile App**

La Dumpster mobile app è il sistema su piattaforma Android che permette all’utente di interagire con il cassonetto e in particolare di realizzare un deposito. Questo componente comunica sia con il Controller via bluetooth sia con il Service (descritto nei particolari in seguito) via HTTP. Per utilizzare il bluetooth e facilitare la gestione delle connessioni http ci siamo serviti rispettivamente della libreria e delle classi messe a disposizione nel repository del corso.

L’applicazione dopo essersi assicurata che l’utente abbia sia il bluetooth acceso che una connessione ad internet, mostra la schermata iniziale. In questa è presente un unico bottone che serve a dare inizio ad un nuovo deposito.

La prima fase del processo di deposito consiste in una richiesta via GET http di un token al Dumpster Service, ricevuto il token l’applicazione mostra una nuova activity, dove viene gestito il deposito vero e proprio. In questa sezione come prima cosa si instaura la connessione con il modulo HC-05 di Arduino, andando a creare una piconet dove l’app è il dispositivo master mentre il controller è lo slave. Creato il canale di comunicazione si permette all’utente di scegliere la tipologia di rifiuto che desidera depositare, questa viene comunicata nella modalità descritta nel Dumpster Controller. A scelta avvenuta, nell’app, si attiva il bottone che permette di richiedere al Controller tempo aggiuntivo per completare il deposito.

La fine del deposito viene comunicata dal controller con un messaggio sul canale, quest’ultimo è anche l’unico messaggio che l’app riceve via bluetooth quindi nella callback “onMessageReceived()” del *Listener* che gestice il canale, eseguiamo direttamente la fase finale del deposito. Quest’ultima è caratterizzata da: la chiusura del canale con il controller, la notifica al server attraverso una POST http della terminazione del deposito e la comunicazione all’utente che il tutto è avvenuto con successo.

La richiesta iniziale del token può avere tre risultati: un HttpResponse con codice di stato 200, indice del fatto che si è ottenuto il token, un HttpResponse con codice 202, segnale che la richiesta è stata negata perché il Dumpster è in stato *not\_available* e infine, HttpResponse nullo indice del fatto che la richiesta non può essere eseguita perché non si riesce a contattare il Service. Avremmo voluto utilizzare per il caso del Dumpster non disponibile un codice di errore http più consono ma abbiamo notato che quando il service inviava codici di errore (4\*\* o 5\*\*) l’HttpResponse risultava nullo non permettendoci quindi di differenziare questo caso da quello del Service non contattabile

Immagine che contiene screenshot

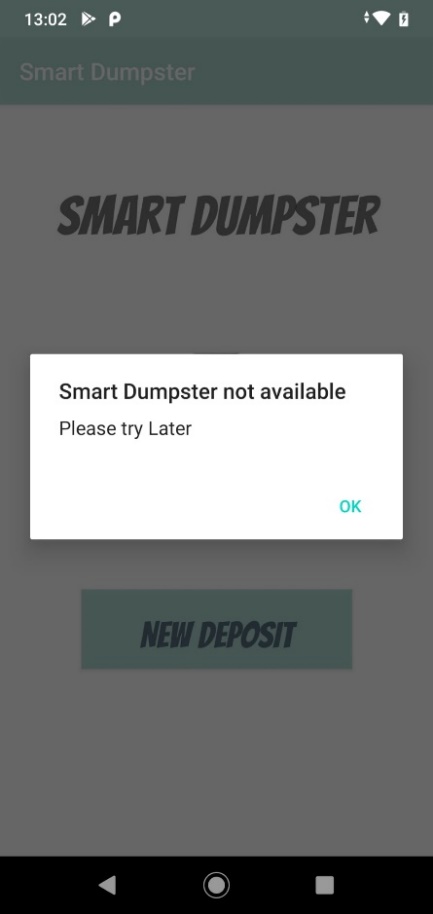
Descrizione generata automaticamente 

Immagine che contiene screenshot

Descrizione generata automaticamente Immagine che contiene screenshot

Descrizione generata automaticamente

**Dumpster Service**

Il Dumpster Service è il server http del sistema, comunica con l’app, l’edge e la dashboard e pertanto è la parte che permette la “sincronizzazione” tra i 5 componenti. Per realizzare il service ci siamo serviti del toolkit Vert.x presentatoci nel corso. Attraverso quest’ultimo abbiamo realizzato una REST Api per gestire le diverse richieste provenienti dai componenenti. In particolare, per differenziare le richieste abbiamo creato 3 route: “/api/esp”, “/api/app”, “/api/dash” una per ogni client.

La comunicazione con il Dumpster Edge avviene attraverso due tipi di messaggi http: una GET e una POST. L’edge utilizza la GET per allinearsi ai cambiamenti di stato avvenuti per via degli altri componenti, infatti, il service risponde a tale richiesta con una stringa contenente lo stato in cui si trova il Dumpster. Alla ricezione di una POST, invece, il Service “scompatta” il file JSON contenuto nel body, aggiorna i dati sulla disponibilità del Dumpster e sul numero dei depositi avvenuti dall’ultimo reset e infine, memorizza il nuovo deposito. Dato che l’esp invia anche la disponibilità corrente, quando questa risulta a zero il Service cambia lo stato in *not\_available*. Nel costruire l’architettura del sistema, per quanto riguarda la memorizzazione dei depositi, abbiamo pensato a due possibili soluzioni: scrittura su file o creazione di un database. Visto che i dati non presentavano una struttura troppo complessa abbiamo preferito scegliere la prima soluzione. Scriviamo su file ogni qualvolta l’esp effettua una POST, in questo modo ci assicuriamo di non perdere neanche un deposito e per avere la certezza che ad ogni POST corrisponda un deposito reale, e non una variazione immotivata del peso, abbiamo scelto di introdurre un nuovo stato fittizio, “DEPOSIT”, questo indica appunto, la necessità di avere i dati riguardo il nuovo deposito(vedere DumpsterEdge per maggiori dettagli). Il Dumpster Service entra in tale stato quando l’app gli comunica che è stato realizzato un nuovo deposito.

La cuminicazione tra Service e App avviene per due motivi: richiesta del Token per realizzare un deposito e comunicazione della terminazione dello stesso. L’app come già detto nella sua sezione, utilizza nel primo caso una GET mentre nel secondo una POST. Il comportamento del service alla richiesta di un token è molto semplice: controlla lo stato corrente del Dumpster, se questo è *available* e non è già stato richiesto iniziato un deposit, concede il token, rispondendo con un messaggio http con codice di stato 200, segno appunto che la richiesta è andata a buon fine, altrimenti risponde con codice 202, segno che la richiesta è stata accettata ma non può essere soddisfatta. Alla ricezione della POST invece, il Dumpster Service, cambia il suo stato interno, settandolo in “DEPOSIT” per le motivazioni dette sopra.

La Dumpster Dashboard contatta il Service per: comunicargli eventuali cambiamenti di stato (vedi la sezione Dumpster Dashboard), conoscere lo stato del Dumpster e ottenere delle statistiche sui depositi. Nel primo caso la Dashboard utilizza una POST e il service aggiorna il suo stato a seconda del booleano ricevuto, se “true” lo setta ad *available* altrimenti a *not\_available.* Per la Dashboard lo stato del Dumpster è caratterizzato anche da tutti i dati sulla disponibilità residua e il numero dei depositi, per questo quando il service riceve una GET sulla route “/api/dash”, risponde con un file JSON, contenete tutte queste informazioni. Dovendo però gestire un’ulteriore richiesta GET dalla dashboard, per evitare il conflitto con la precedente, abbiamo dovuto introdurre la route “/api/dash/statistics”. Per quanto riguarda la produzione delle statistiche potevamo scegliere di farle computare o alla Dashboard stessa, inviandogli i dati “grezzi”, o, come poi abbiamo deciso, al Service, per via di una maggiore facilità di manipolazione dei dati. Come già detto abbiamo deciso di memorizzare su file i singoli depositi, ognuno caratterizzato dal proprio peso e dalla data in cui è avvenuto; abbiamo fatto questa scelta, nonostante le statistiche richieste dalle specifiche riguardino le giornate e non i singoli depositi, al fine di lasciare l’opportunità futura di realizzare statistiche più dettagliate. Le specifiche inoltre, lasciavano un certo grado di libertà per quanto riguarda l’arco temporale su cui produrre le statistiche, noi abbiamo scelto di settarlo ad una settimana, ma la nostra soluzione permette sia di ampliarlo che ridurlo senza problemi. Per mantenere i dati relativi solo agli ultimi 7 giorni, quando registriamo il primo deposito di una nuova giornata controlliamo se sul file sono presenti dati troppo vecchi e in tal caso li eliminiamo, riscrivendo su di esso solo i depositi recenti. Dovendo però fare dei resoconti giornalieri abbiamo creato la classe “DailyLog” caratterizzata dalla data a cui si riferisce il resoconto, il totale dei depositi effettuati in tale data e il loro numero. Per realizzare le statistiche iniziamo con la lettura del file e la creazione di una lista di Deposit, di seguito li raggruppiamo per data e per ogni insieme creaiamo e aggiorniamo il DailyLog corrispondente producendo con essi una lista. Ottenuta la lista dei DailyLog, il service risponde alla GET ricevuta con un JSONArray contenente tanti JSON quanti i DailyLog della lista.

**Dumpster Dashboard**

La Dumpster Dashboard nel nostro progetto è una WebApp. Oltre ad HTML e il framework Bootstrap per la parte di CSS, utilizziamo Javascript, e in particolare il framework JQuery, per sfruttare i metodi AJAX, così da aggiornare dinamicamente la pagina web. Secondo le specifiche la dashboard deve permettere di monitorare la disponibilità, la quantità corrente e il numero di depositi effettuati dall’ultimo reset oltre che permettere di forzare lo stato ad un dei due valori possibili (available o not\_available). In ultimo, la dashboard deve mostrare l’andamento dell’utilizzo del Dumpster negli ultimi N giorni, N da noi fissato a 7 come detto nella sezione Dumpster Service. Per mostrare in maniera chiara e immediata i dati richiesti dalle specifiche abbiamo deciso di sfruttare dei grafici, in particolare, abbiamo utilizzato la libreria CanvasJS. Per mostrare lo stato del Dumpster Edge e in particolare la percentuale di disponibilità abbiamo utilizzato un grafico a Donut, in rosso indichiamo il grado di occupazione del Dumpster mentre in verde la disponibilità residua, inoltre, con una label laterale indichiamo il numero di depositi effettuati. Per quanto riguarda l’andamento dell’utilizzo del Dumpster abbiamo deciso di riassumere in un unico grafico sia la quantità totale in kg che il numero di depositi registrati giornalmente, sfruttando un grafico a colonne con due assi verticali. In questo modo evidenziamo anche il rapporto fra peso e numero di depositi. Per aggiornare la pagina coerentemente ai tempi generali del sistema abbiamo deciso di richiedere i dati ogni 2 secondi. Come UI per forzare lo stato mettiamo a disposizione due bottoni alla cui pressione associamo l’invio di una POST al Service contenente il rispettivo stato.

Immagine che contiene screenshot, monitor, computer

Descrizione generata automaticamente