UNIVERSITÀ DEL PIEMONTE ORIENTALE

PROGETTO AA 2024/25

Fase di progettazione

(dopo fase di specifica, precede l'implementazione)

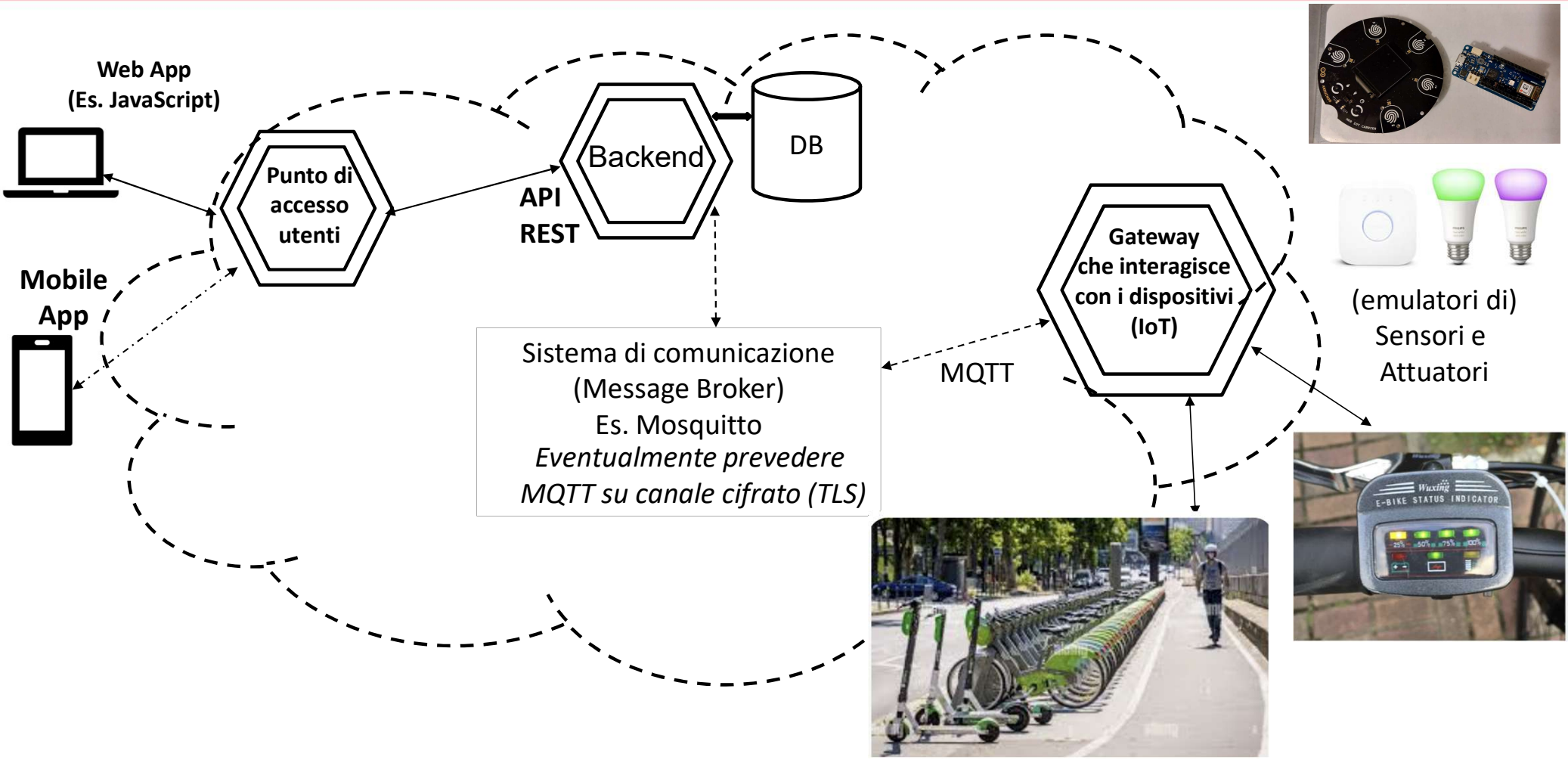
FASE 2: Progettazione

- Nella fase di specifica avete descritto COSA si doveva progettare
- Nella fase di progettazione dovrete definire COME realizzare il sistema

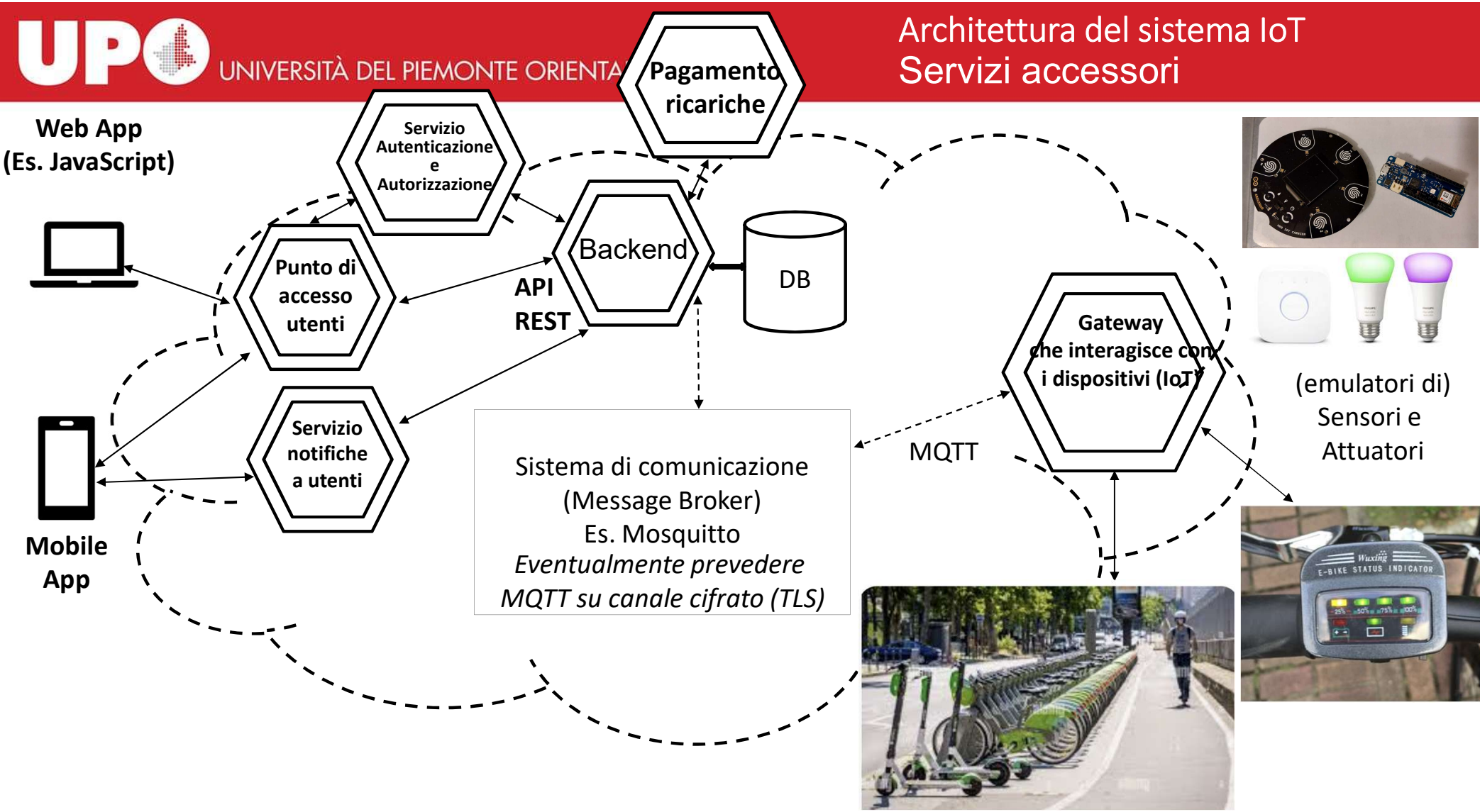
Alcuni suggerimenti su come impostare questa fase:

- Architettura basata su servizi il più possibile flessibile rispetto al deployment dei diversi servizi
- Alcuni servizi devono essere disponibili tramite interfaccia REST
- L'interazione può avvenire anche tramite comunicazione asincrona (pubblicazione via broker di eventi)

Architettura del sistema IoT



Architettura del sistema IoT
Servizi accessori



Componenti principali dell'architettura:

- Client App: può essere molto semplice (linea di comando) oppure eseguito nel browser (es. realizzato con javascript) oppure un'applicazione mobile: interagisce con un microservizio che funge da punto di accesso al sistema e che quale contatta il backend tramite le API REST.
- Backend: permette agli utenti di accedere per consultare le informazioni presenti nel DB (parcheggi e mezzi, ricariche e corse) o inserire / modificare i dati relativi alle varie entità gestite. Espone un'interfaccia REST. Comunica con il sottosistema IoT tramite il broker MQTT.
- DB: database contenente tutte le informazioni necessarie al funzionamento del sistema (es. dati corse, stato mezzi, informazioni sulle manutenzioni, …). Accessibile solo tramite il backend
- Sottosistema IoT - si occupa di rilevare misure tramite i sensori (livello carica batteria) che invia tramite broker MQTT e tramite lo stesso canale di comunicazione può accettare comandi per gli attuatori (es. sblocco mezzo per inizio corsa). Può essere quindi sia publisher che subscriber.

Cosa produrre nella fase di progettazione

- Per ciascun servizio: diagramma delle classi con i relativi attributi e metodi, diagrammi di sequenza per mostrare le interazioni tra diversi oggetti o tra i diversi microservizi ed eventualmente diagrammi di attività per mostrare la suddivisione dei compiti tra diversi thread di un certo servizio.
- Può essere utile strutturare in package le diverse classi (all'interno di ciascun microservizio) per evidenziarne i diversi componenti.

Cosa produrre nella fase di progettazione

Progettazione API REST (e documentazione …) e TOPIC MQTT

- Definire Risorse
- Definire la Rappresentazione delle risorse (che verrà scambiato tra client e server; noi useremo un formato JSON)
- Definire gli Endpoint / TOPIC e il formato dei messaggi inviati al broker
- Definire le possibili Azioni
- Definire i possibili Errori

Definizione delle Risorse (dal diagramma classi di dominio)

Siamo interessati ad accedere a singole risorse ma anche a collezioni di risorse (per queste ultime usiamo nomi plurali):

- Mezzi
- Corse
- Parcheggi
- Utenti (amministratore servizio / utilizzatori dei mezzi)

- Sensori (livello carica batterie, …)
- Attuatori (meccanismo di blocco/sblocco mezzo)

Rappresentazione delle risorse

Corsa: Nota: alcuni attributi potrebbero essere indefiniti in certe fasi

{"id_utente": 3,
"id_mezzo": EB14,
"id_parcheggio_prelievo": Park1,
"data_inizio": 25/03/2025-10.39,
"id_parcheggio_restituzione": Park2,
"data_fine": 25/03/2025-11.00
}

Rappresentazione delle risorse

Mezzo:

{"id_mezzo": EB15,

"stato": indisponibile,

"livello_ricarica": "15%"}

Endpoint

Definire le URI (Uniform Resource Identifier) corrispondenti ai possibili endpoint che permettono di accedere alle risorse; hanno in comune una URI base: http://api.mobishare.org/v1– può anche essere utile includere la versione

Esempio: **parcheggi** restituisce un array di oggetti "parcheggi"

Esempio: **parcheggi/Park1** restituisce l'elenco dei mezzi presenti nel parcheggio

Dato che gli utenti potrebbero voler visualizzare i loro dati ma anche i loro pagamenti o le loro soste potremmo utilizzare una URI gerarchica

Esempio: **utenti/{id_utente}/pagamenti** *oppure* **utenti/{id_utente}/profilo**

È anche possibile aggiungere dei parametri definendo così delle query:

Esempio: **utenti?tipo=sospeso** *oppure* **corse?data:25-05-2024**

Nota: per esperimenti in locale la URI base sarà semplicemente localhost:porta

Azioni CRUD sulle risorse

Possibili azioni sulle risorse:

- Visualizzare le corse di un dato utente o in una certa data
- Creare una nuova corsa / modificare dati corsa (es. data fine, costo corsa)
- Modificare il metodo di pagamento di un utente
- Registrare la rimozione / aggiunta di un mezzo in un parcheggio
- Modificare lo stato di un mezzo

Associamo ciascuna azione ad un «verbo» http – GET, POST, PUT, DELETE; inoltre consideriamo anche il tipo di errore da restituire in caso di fallimento:

200 (successo: GET nel body c'è la risorsa, PUT/POST nel body informazioni sull'esito), 201 (POST conferma creazione risorsa, restituisce id), 400: richiesta non valida (errore sintassi), 401: accesso non autorizzato, 404: risorsa non trovata, 500: Errore interno

VEDERE ESEMPIO SPECIFICA TRAMITE YAML / SWAGGER EDITOR

Tabella delle possibili azioni, con eventuale input e risposta (tipo utenti autorizzati)

Verbo http	Endpoint	Input	Output in caso di successo	Messaggio Errore	Descrizione
GET	/utenti/{idUtente}/ricariche/	Body: vuoto	Stato: 200 Body: lista ricariche	Stato: 500	Fornisce un array di ricariche saldo
GET	/parcheggi/{idParcheggio}/mezzi_elettrici	Body: vuoto	Stato: 200 Body: dati mezzi elettrici disponibili	Stato: 404 o 500	Fornisce lista mezzi elettrici prelevabili nel parcheggio
GET	/mezzi	Body: vuoto Parametro: stato, tipo, zona o parcheggio	Stato: 200 Body: lista mezzi in un certo stato	Stato : 500	Fornisce un array di mezzi di un certo tipo con un certo stato
POST	/corse	Body: nuova corsa: id utente, id mezzo, data inizio, parking	Stato: 201 Body: id della nuova corsa	Stato: 401, 500	Inserisce nuova corsa
PUT	/mezzi/{id}/stato	Body: modifica stato (e posizione) mezzo	Stato: 200 Body: vuoto	Stato: 401, 404, 500	Modifica stato di un mezzo

Tabella delle possibili azioni, con eventuale input e risposta

Verbo http	Endpoint	Input	Output in caso di successo	Messggio Errore	Descrizione
GET	/corse/	Body: vuoto Parametro: id_utente e/o id_mezzo e/o park inizio+park destinazione	Stato: 200 Body: dati dispositivi presenti nella serra	Stato: 500	Fornisce l'array delle corse che soddisfano i parametri indicati
DELETE	/mezzi/{id_mezzo}	Body: vuoto	Stato: 200	Stato : 404 o 500	Cancella un mezzo dal DB
DELETE	/parcheggi/	Non definito	Non Definito	Stato: 400	Azione vietata

Cosa produrre nella fase di progettazione

Progettazione TOPIC e messaggi MQTT

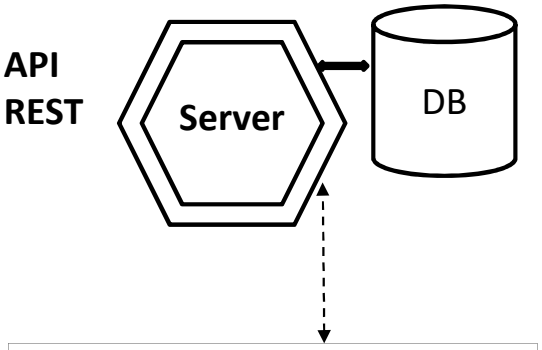
Parking/{id_parking}/Mezzi (da Gestore IoT a Broker) invio id mezzo e livello batteria nel messaggio

Parking/{id_parking}/StatoMezzi/{id_mezzo} (da Broker a Gestore IoT) messaggio per blocco/sblocco/cambio stato di un mezzo in un dato parcheggio

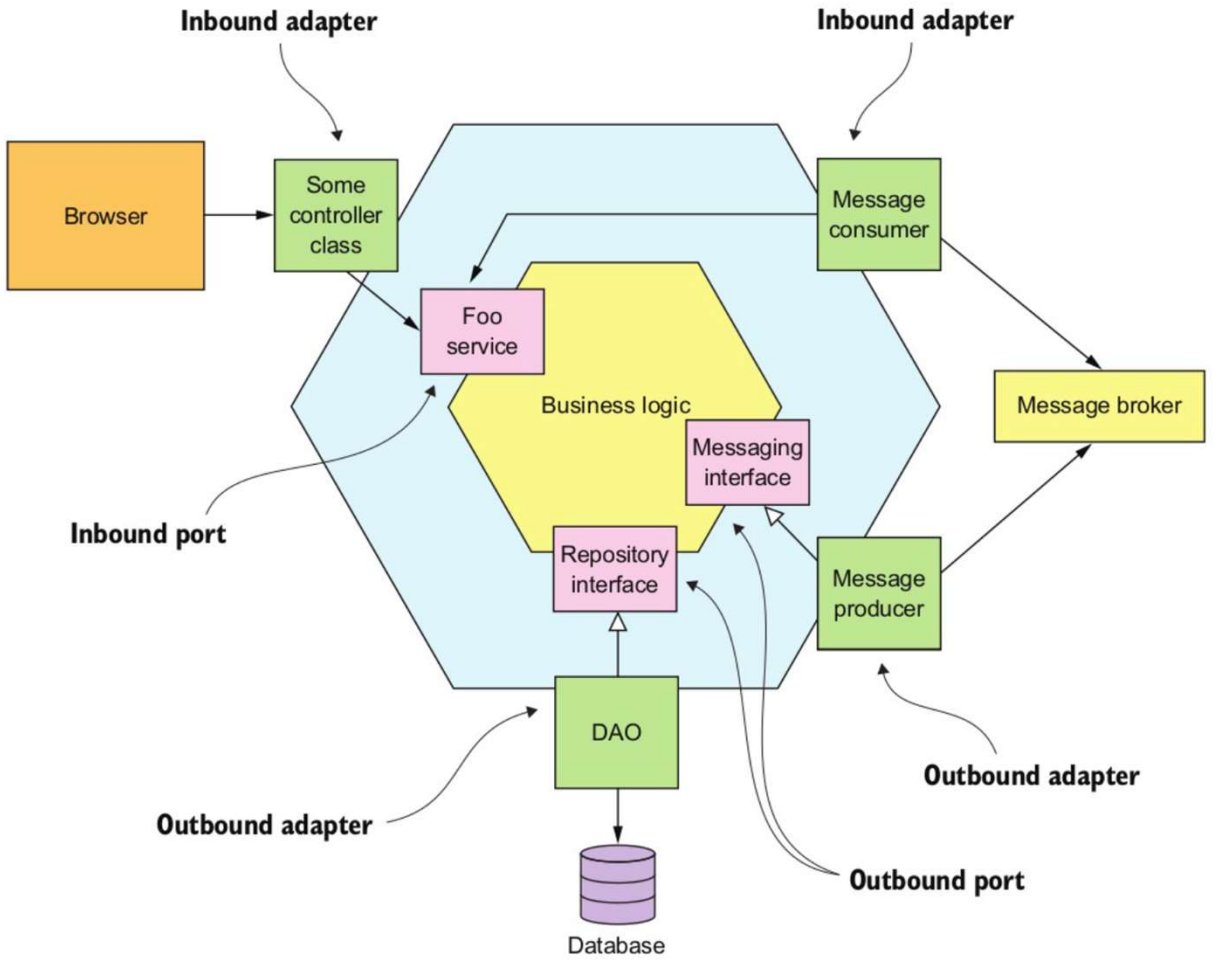
per sottoscrivere tutti i parcheggi (Broker) può usare una wildcard

Parking/+/Mezzi

oppure tutti gli eventi di cambio stato di un certo parcheggio
Parking/{id_parking}/StatoMezzi/#

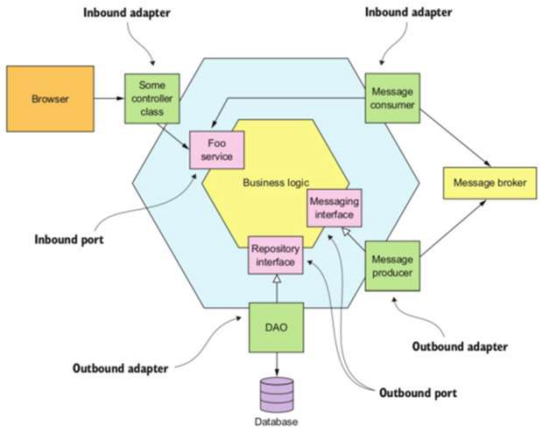


Sistema di comunicazione
(Message Broker)
Es. Mosquitto



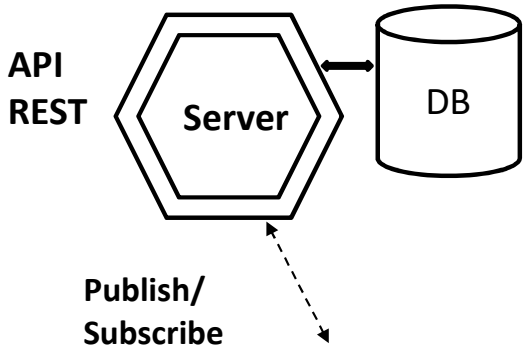
Classi: business logic e adapters

- Interfaccia della business logic: metodi che corrispondono a comandi che cambiano lo stato interno o a query che richiedono risposte
- Adapter
  - Inbound: costituiscono un'interfaccia per il mondo esterno che invia comandi e query al microservizio. Es. API REST
  - Outbound: costituiscono un'interfaccia verso sistemi esterni
    - Database
    - Broker



API REST ADAPTER (Inbound Adapter)

Classi che si occupano di gestire le richieste REST in arrivo dalle interfacce utente e richiamano i metodi offerti dal cuore del microservizio



Business Logic (gestore prenotazioni, soste o pagamenti e utenti)
Classi che si occupano di rispondere a comandi/query:

- crea corsa, cancella mezzo, modifica stato mezzo, dotazione parcheggio
- registra inizio/fine corsa e calcola importo pagamento
- registrazione nuovi utenti

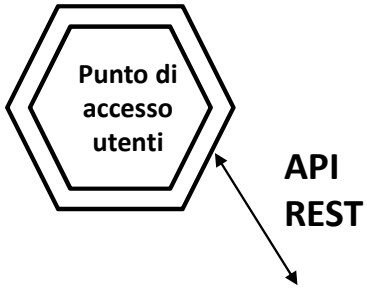
PERSISTENZA (outbound adapter)
Mappa operazioni CRUD sugli oggetti trattati dalla business logic su operazioni sul DB

Broker messaging ADAPTER (In/Outbound Adapter)

- In: riceve notifica livello batteria mezzo (via subscribe a broker)
- Out: invia conferma sblocco mezzo (via publish a broker)

View (UI)

Classi che si occupano di gestire le viste dell'interfaccia utente

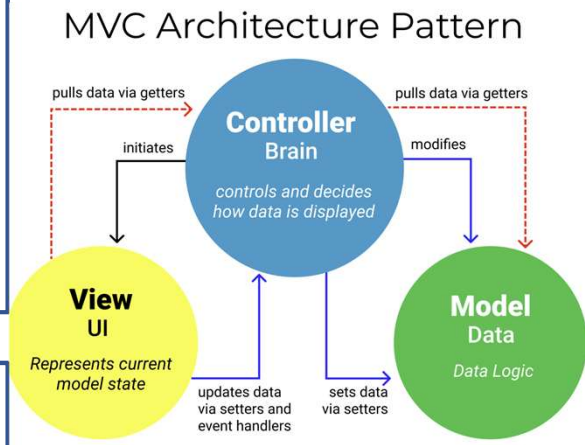


Controller

gestisce la sequenza di interazioni con l'utente (innescate dalla UI) e di conseguenza aggiorna il modello

Model (gestisce gli oggetti del dominio)

- Interagisce con l'outbound adapter
- Aggiorna le viste



Outbound Adapter verso il Servizio che le chiamate alle API REST

IoT ADAPTER (In/Outbound Adapter)
- In: rilevare misure (livello carica batteria)
- Out: comanda gli attuatori (es. sblocco bici, cambio colore spia luminosa: se emulo con lampadine: chiama API REST)

Applica criteri per variare stato degli attuatori (per esempio può immediatamente modificare la spia per indicare indisponibilità di un mezzo elettrico se ha la batteria scarica)
Gestisce interazione con spia luminosa e con meccanismo di blocco/sblocco
Gestisce un registro "locale" dei mezzi presenti (attenzione alla consistenza con il DB … potrebbero non essere sempre allineati).

Broker messaging ADAPTER (In/Outbound Adapter)
- Out: invia dati in/out da sbarre (via publish a broker) – e inoltra fine ricarica
- Out: Notifica all'amministratore eventuali malfunzionamenti
- In: riceve e attua comandi di blocco/sblocco