Progetto d'esame: laboratorio di PISSIR

Progettazione logica

Sommario

- Architettura del sistema
  - Diagramma UML
  - Dettagli dei Moduli
    - Moduli di Comunicazione
  - Tecnologie e Strumenti utilizati
  - Sicurezza e Affidabilità
- Sviluppo
  - DB
- Topic
  - Topic messages

Architettura del sistema

Il sistema è composto da più architetture che interagiscono tra loro:

1. **Architettura Client-Server:**
   - **Frontend:** è scritto in JavaScript e utilizza Node.js per l'esecuzione e con il framework Express.js mostriamo un'interfaccia grafica realizzata come pagina web. Inoltre Express funge da **client REST** per il backend. Gli utenti interagiscono con il sistema tramite il frontend, il quale effettua chiamate REST al server backend.
   - **Backend:** Il server backend è scritto in Java utilizzando il framework **Javalin**. Questo server espone delle API REST per ricevere richieste dal frontend e interagire con il database SQLite.
   - **Database:** Il database utilizzato è **SQLite**, che viene manipolato esclusivamente dal backend tramite le API REST.
2. **Architettura IoT (Internet of Things):**
   - È presente un sistema IoT, costituito da numerosi dispositivi che si collegano a un broker **MQTT Mosquitto**.
   - I dispositivi IoT, **sensor, monitor, mobile e mwbot**, sono tutti client MQTT conformi alla versione **MQTTv3** e utilizzano la libreria **Eclipse Paho** in Java per la gestione della comunicazione.
   - **Comunicazione IoT:** Tutti i dispositivi (client MQTT) inviano e ricevono dati dal broker Mosquitto. La comunicazione tra il backend e i dispositivi IoT avviene tramite MQTT per operazioni in tempo reale.

Schema dell'architettura:

- **Client-Server**: Frontend (Client REST) <-> API REST <-> Backend (Java Javalin, SQLite)
- **IoT**: Dispositivi IoT (Sensori, Monitor, MWBot, Mobile) <-> MQTT Broker (Mosquitto)

Diagramma UML

Dettagli dei Moduli

- **Frontend (Node.js, Express.js):** Gestisce le richieste degli utenti e le interazioni via interfaccia Web, comunicando con il backend tramite RESTful API.
- **Backend (Java, Javalin):** Espone API REST per il frontend e interagisce con il database SQLite. Si occupa delle operazioni di backend.
- **MQTT Broker (Mosquitto):** Centralizza la comunicazione tra i dispositivi IoT (sensori, monitor, mobile, mwbot).
- **Sensori, Monitor, MWBot, Mobile (Java, Eclipse Paho Client MQTT):** Dispositivi che inviano e ricevono messaggi al broker MQTT.

Moduli di Comunicazione

- **Frontend e Backend (REST API):** Il frontend comunica con il backend tramite chiamate RESTful API. Queste API sono implementate nel server backend (Java con Javalin) e consentono l'interscambio di dati tra l'interfaccia utente e la logica di business del server. Il backend interagisce con il database SQLite per ottenere o salvare i dati.

- **IoT (MQTT):** Il sistema IoT è gestito tramite un broker MQTT (Mosquitto). I dispositivi IoT (sensori, mobile, monitor, mwbot) sono client MQTT, utilizzando la libreria **Eclipse Paho** per inviare e ricevere messaggi al broker. Il backend può interagire con questi dispositivi IoT in tempo reale, ricevendo dati da sensori o inviando comandi tramite il broker MQTT.

Tecnologie e Strumenti utilizati

- **Javalin:** Javalin è un framework per Java che ti consente di creare API REST veloci ed efficienti, successore di SparkJava Core, utilizzato da Backend e Debug Gradle applications.
- **Node.js e Express.js:** Il server frontend è scritto in Node.js con il middleware Express.js, che gestisce le richieste HTTP da parte degli utenti tramite l'interfaccia Web, le elabora per inoltrarle al server backend tramite le chiamate REST.
- **SQLite:** è il DBMS utilizzato per la sua semplicità e la capacità di essere facilmente integrato in applicazioni leggere. Tutte le operazioni sul database vengono eseguite dal backend tramite API REST.
- **MQTT:** I dispositivi IoT e il backend utilizzano questo protocollo per la comunicazione in tempo reale con il broker **Mosquitto**. **Eclipse Paho** è la libreria Java scelta per gestire la connessione MQTT dei vari client.
- **Autenticazione:** Gli utenti si autenticano con **email e password** per un semplice accesso al sistema.
- **Deployment:** Il sistema è composto da diverse applicazioni Java, tutte gestite tramite **Gradle**. Per semplificare il lancio del sistema, ho creato uno **script bash** che permette di avviare tutte le applicazioni (i vari **JAR files**) separatamente, simulando l'ambiente di produzione.

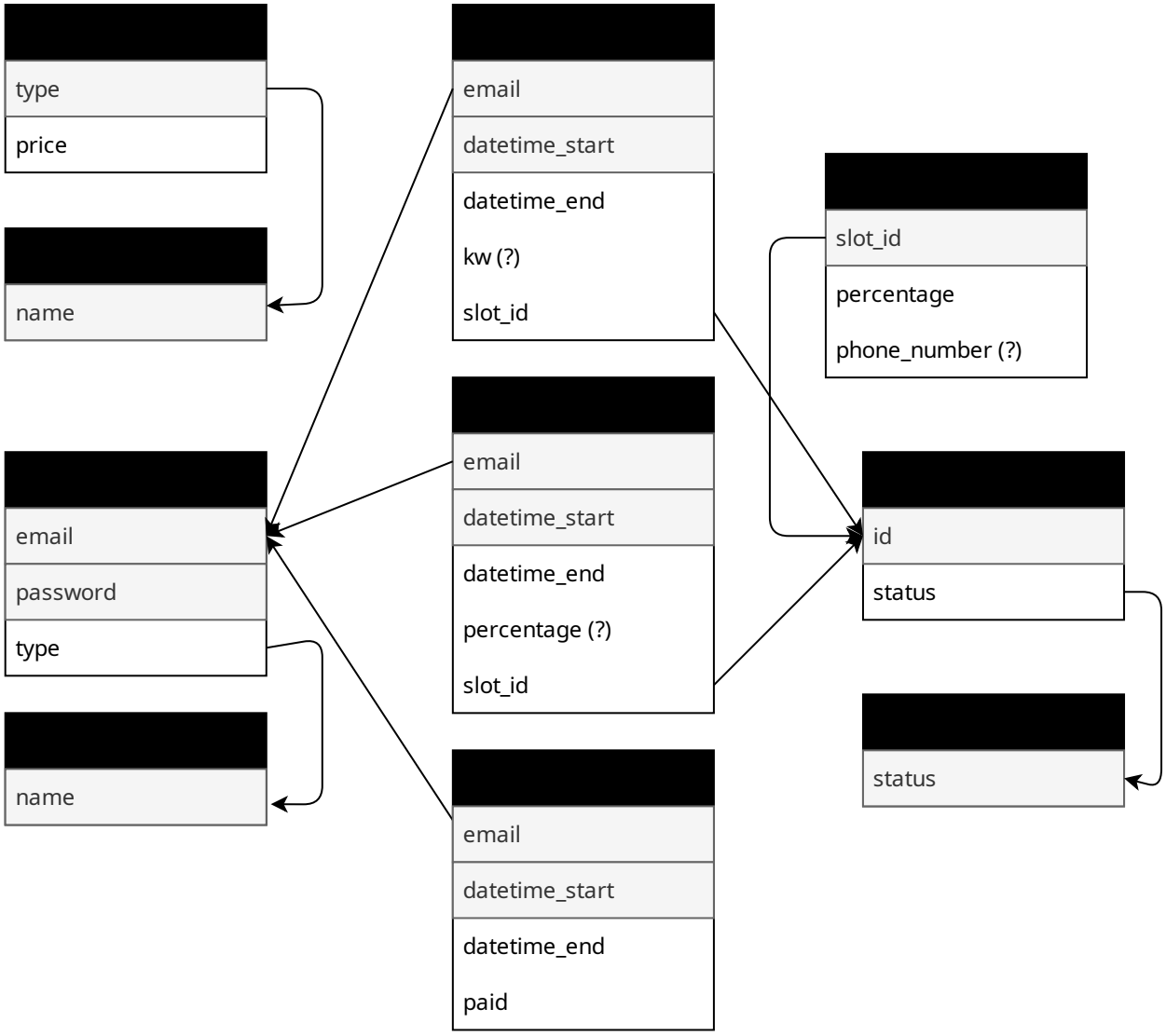
Sicurezza e Affidabilità

Ho implementato un'app Gradle atto a monitorare per tenere traccia dello stato dei vari client MQTT in tempo reale.

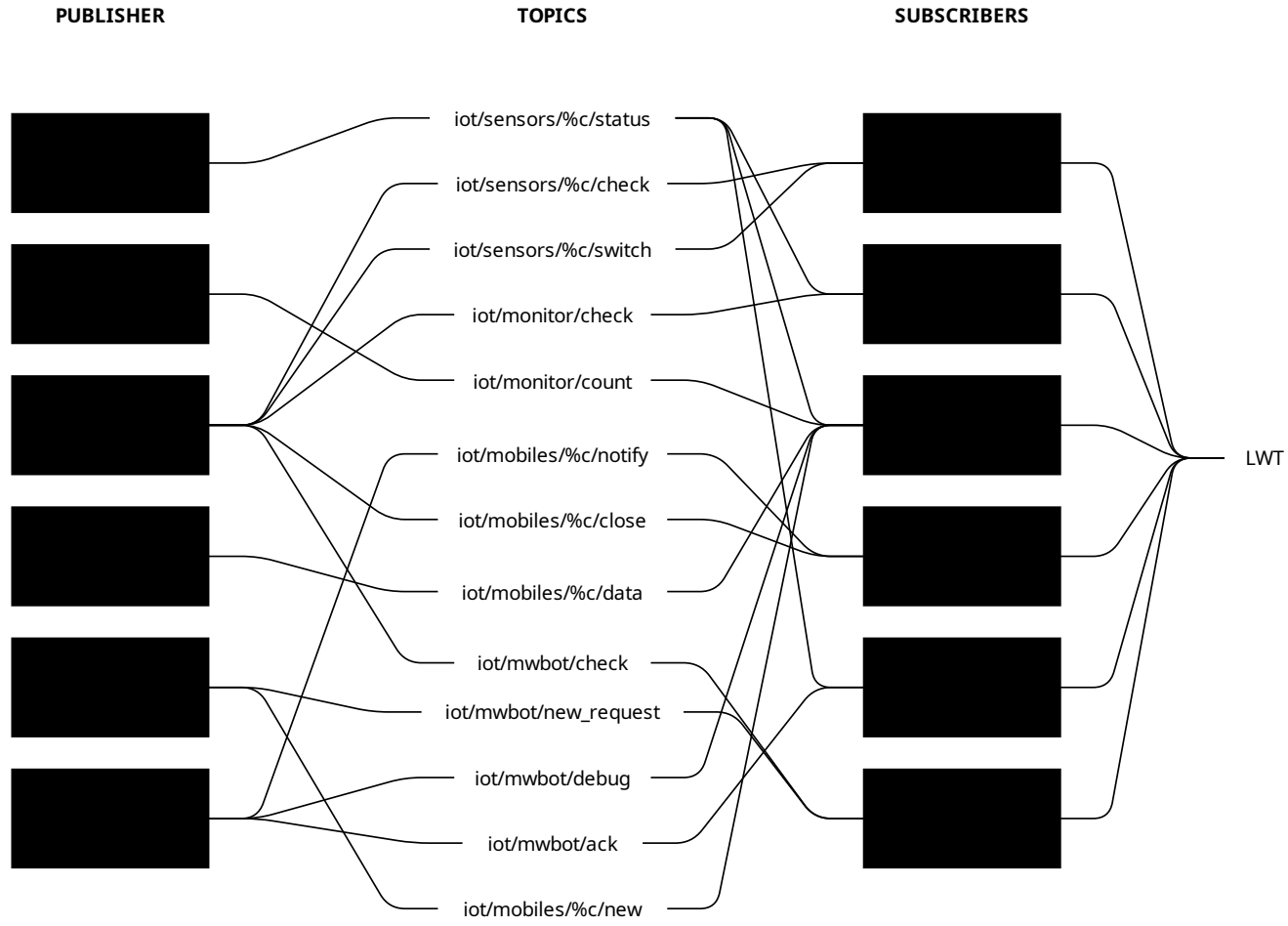
Sviluppo

DB

- **Schema DB:** smartparkingDB.schema


type
price
name
email
password
type
name
email
datetime_start
datetime_end
kw (?)
slot_id
email
datetime_start
datetime_end
percentage (?)
slot_id
email
datetime_start
datetime_end
paid
slot_id
percentage
phone_number (?)
id
status
status

Topic



Topic messages

- **iot/sensors/sensor_[1-9][0-9]*/status**: sensore pubblica il proprio stato occupazionale.

```
{
  "slot_id": "[1-9][0-9]+",
  "status": "FREE" | "OCCUPIED"
}
```

- **iot/sensors/sensor_[1-9][0-9]*/check**: topic utilizzato per "pingare" il sensore per avere il suo stato occupazionale.

```

```

- **iot/sensors/sensor_[1-9][0-9]*/switch**: topic utilizzato per "pingare" il sensore di cambiare stato occupazionale da `FREE` a `OCCUPIED` o viceversa.

```

```

- **iot/monitor/check**: topic utilizzato per "pingare" il monitor per avere il numero di posti liberi.

```

```

```
```

- **iot/monitor/count**: monitor pubblica il numero di posti liberi.

```
{
  "id": "monitor",
  "count": "[0-9]+"
}
```

- **iot/mobiles/[0-9]{10}/notify**: topic utilizzato per "notificare" il mobile/cellulare che l'MWbot ha finito di ricaricare la macchina.

```
{
  "message": "Car charged to ([0-9]|([1-9][0-9])|(100))% in slot [1-9][0-
9]* with 0|[1-9][0-9]* KW"
}
```

- **iot/mwbot/[0-9]{10}/close**: topic utilizzato per "chiudere" il processo mobile/cellulare.

```
```

- **iot/mwbot/[0-9]{10}/data**: topic utilizzato dai dispositivi mobile/cellulare per inviare dati al broker MQTT. (utile per il debug)

```
{
  "id": "[0-9]{10}",
  "message": ""
}
```

- **iot/mwbot/check**: topic utilizzato per "pingare" il MWBot per avere il suo stato lavorativo e altre informazioni.

```
```

- **iot/mwbot/new_request**: topic utilizzato per notificare l'MWbot di una nuova richiesta dagli utenti da aggiungere alla coda di attesa.

```
{
  "slot_id": "[1-9][0-9]*",
  "percentage": ([0-9]|([1-9][0-9])|(100)),
  "phone_number": "[0-9]{10}" | ""
}
```

- **iot/mwbot/debug**: topic utilizzato per scopi di debug per l'MWbot per avere il suo stato lavorativo e altre informazioni.

```
{
  "status": "IDLE" | "MOVING" | "CHARGING",
  "position": [1-9][0-9]*,
  "model": "<model name car>>",
  "percentage": ([0-9]|([1-9][0-9])|(100))
}
```

- **iot/mwbot/ack**: topic utilizzato per l'MWbot di notificare il backend che una richiesta di ricarica è stata completata.

```
{
  "slotId": [1-9][0-9]*,
  "kw": 0|[1-9][0-9]*
}
```

- **iot/mobiles/[0-9]{10}/new**: topic utilizzato dal debug per la creazione automatica dei dispositivi mobile/cellulare.

```
{
  "phoneNumber": "[0-9]{10}"
}
```