



Gestione dei Big Data

Emanuele Mele, Matteo Aprile, Roberto Vadacca

2022/2023

Indice

1	Introduzione	2
1.1	Contesto	2
1.2	Obiettivi	2
2	Review tecnologica	3
2.1	NGSI-LD	3
2.2	Docker	3
2.3	MongoDB	4
2.4	IoT Agent	4
2.5	Context Broker: Orion-LD	4
3	Studio di fattibilità	5
3.1	Architettura	5
4	Progettazione	6
4.1	Data models	6
4.2	POST request	7
4.3	Docker	7
5	Sviluppo	9
5.1	Python	9
5.2	Flask	9
5.3	GUI	11
6	Conclusioni	12

Capitolo 1

Introduzione

1.1 Contesto

Questo progetto è stato ideato per **automatizzare** e rendere più fruibili tutti i dati utili ricavabili da una biblioteca.

Si creerà un prototipo di software che potrà eseguire il **tracciamento di libri e posti a sedere** presenti in una biblioteca. Tramite dei sensori si ricaveranno i dati per rendere più fruibile e veloce la ricerca di specifici libri ed il monitoraggio della disponibilità di posti liberi in biblioteca.

L'utente sarà in grado di visualizzare importanti informazioni della biblioteca attraverso un'**interfaccia grafica**. Si intende tenere traccia del numero di posti a sedere disponibili e occupati, inoltre, verrà messo a disposizione un catalogo di libri disponibili e in arrivo.

1.2 Obiettivi

Gli obiettivi di questo progetto sono:

- creazione di smart data model ad hoc
- emulazione dei sensori
- utilizzo di Orion Context Broker
- utilizzo di MondoDB per storicizzare i dati
- creazione di una GUI

Capitolo 2

Review tecnologica

2.1 NGSI-LD

NGSI-LD è un modello di informazioni e un'API per l'interrogazione e la sottoscrizione a informazioni di contesto utilizzato dalla piattaforma Fiware. Il modello di informazioni di NGSI-LD può essere considerato come la prima specifica formale, creata da un organismo di standardizzazione per i database a grafo.

L'API permette di supportare varie architetture:

- **Context Consumer:** effettua operazioni di recupero e/o interrogazione delle entità presenti nel Context Broker
- **Context Producer:** effettua operazioni di creazione, aggiornamento ed eliminazione di entità, proprietà e relazioni presenti nel Context Broker
- **Context Source:** rende disponibili delle entità per poterle rendere rilevabili dal Context Broker
- **Context Broker:** punto di accesso principale alle informazioni
- **Registry Server:** memorizza le informazioni sulla tipologia di dati del contesto

2.2 Docker

Docker semplifica il processo di deployment eseguendo programmi in ambienti isolabili, minimali e facilmente distribuibili. Il software permette di fare tutto ciò andando ad **impacchettare un'applicazione e le sue dipendenze** in un container. In caso di necessità è possibile eseguire e definire delle applicazioni Docker con **multi-container** tramite un file .yaml che configura i singoli servizi per ogni applicazione, ma che può avviarli tutti con un singolo comando.

2.3 MongoDB

MongoDB è un potente sistema di database open source e gratuito non relazionale, popolare per l'archiviazione di **alti volumi di dati** in documenti e collezioni in formato JSON. Grazie a questo modello dati è possibile soddisfare esigenze mutevoli senza dover ridefinire il modello concettuale.

2.4 IoT Agent

Per utilizzare e coordinare dispositivi IoT, Fiware mette a disposizione un IoT Agent grazie al quale i dispositivi sono in grado di **inviare i propri dati** e di interagire con il Context Broker utilizzando i **propri protocolli nativi**

Le richieste HTTP generate dal Context Brocker e dirette verso l'IoT Agent sono note come "**traffico verso sud**". Questo traffico è costituito da **comandi** impartiti a dei dispositivi attuatori, cioè che vanno ad **alterare lo stato del mondo reale** con le loro azioni.

Le richieste generate da un dispositivo IoT e trasmesse verso il Context Brocker attraverso un IoT Agent, sono note come "**traffico verso nord**". In questo modo vengono trasmesse le misure eseguite da un sensore per avere **informazioni sul mondo reale** che verranno riportate nei context data del sistema.

2.5 Context Broker: Orion-LD

Il Context Broker fornisce un **accesso facile alle informazioni di contesto** che vengono utilizzate dai Context Consumer. Questa architettura permette anche di **memorizzare** le informazioni presenti nelle entità fornite da un Context Producer, oppure derivanti da Context Sources esterne ricavate da Context Source Discovery che fa uso dell'API di NGSI-LD.

Ha il compito di **aggregare le informazioni** relative alle entità della richiesta, restituendo il risultato aggregato al Context Consumer. La cadenza della restituzione delle informazioni che partono dal Context Broker è configurabile, ad esempio, nel caso delle **subscriptions** che andranno a notificare il Broker ogni volta che si verifica un evento.

Nello specifico **Orion-LD** è un Context Borker che supporta NGSI-LD e NGSI-v2.

Capitolo 3

Studio di fattibilità

3.1 Architettura

Lo scenario che si presenta necessita di una rete di componenti che comunichino tra di loro per poter gestire al meglio i dati. Non avendo a disposizione sensori fisici, sono stati sviluppati dei sensori mock che, grazie all'emulazione di una determinata attività, possano trasmettere le loro POST request al **Context Broker** (Paragrafo 2.5). Queste richieste però non possono essere dirette subito al Broker ma prima verranno prese in carico dall'**IoT Agent** (Paragrafo 2.4) che dovrà andare ad interpretare i protocollo nativi dei sensori per poi redirigerli al Broker tramite porta 4041 : 4041.

Ogni sensore è contraddistinto da ID, proprietà e tipologia. Queste informazioni costituiscono un documento in formato **JSON-LD** che è un istanza del **data models** (4.1).

Per lo **storage dei dati** si è scelto l'uso di un database NoSQL, dato che, nella realtà, si sta parlando di Big Data (Paragrafo 2.3).

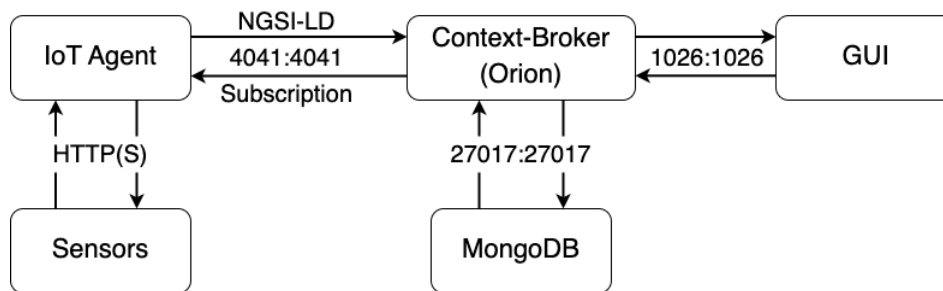


Figura 3.1: Architettura dell'applicativo

Capitolo 4

Progettazione

4.1 Data models

Per esprimere la struttura dei dati delle entità richieste dal caso d'uso è necessario creare dei **data models**. Per generare questi ultimi, ove possibile, sono stati utilizzati modelli preesistenti disponibili su <https://smartdatamodels.org> e nel caso di mancata disponibilità, sono stati creati sulla base dell'ontologia <https://schema.org/>. In questo progetto è stato necessario modellare da zero le entità **libro** e **biblioteca**, invece, per i modelli **persona** e **device** sono state aggiunte delle proprietà essenziali altresì mancanti.

```
Book:
  properties:
    isbn:
      description: The ISBN of the book.
      type: string
    x-ngsi:
      model: 'https://schema.org/Text'
      type: Property
      uri: 'https://schema.org/isbn'
      uri-prefix: 'https://schema.org/'
:
  #...more properties...
  type: object
  x-ngsi:
    uri: 'https://schema.org/Book'
    uri-prefix: 'https://schema.org/'
```

4.2 POST request

Ogni data model sarà quindi responsabile della definizione di un'entità e del suo aggiornamento in tempo reale tramite delle **POST request** che andranno a **simulare** i dispositivi fisici responsabili del monitoraggio dei cambiamenti nel mondo reale.

Il Context Broker, ad ogni lettura del sensore, utilizzerà un sistema di **subscription** al fine di **notificare** in modo asincrono tutti i subscriber dell'avvenuto cambiamento. Il sistema di subscription risulta fondamentale per poter tenere aggiornata la GUI, senza sovraccaricare l'applicativo. Per rispettare questa premessa si invierà una query ogni periodo di tempo Δt .

```
curl -L -X POST 'http://localhost:1026/ngsi-ld/v1/subscriptions/' \
-H 'Content-Type: application/ld+json' \
-H 'NGSILD-Tenant: openiot' \
--data-raw '{
  "description": "Notificami quando viene prelevato un libro",
  "type": "Subscription",
  "entities": [{"type": "Device"}],
  "watchedAttributes": ["isbn"],
  "notification": {
    "format": "keyValues",
    "endpoint": {
      "uri": "http://172.18.1.200:5050/sensorBooksNotification",
      "accept": "application/json"
    }
  },
  "@context": "http://context/ngsi-context.jsonld"
}'
```

4.3 Docker

Il deployment del sistema descritto nel paragrafo 3.1, è stato possibile grazie a **Docker** che permette di fornire un servizio isolato dalla macchina sulla quale lo si sta eseguendo, andando ad **impacchettare un'applicazione e le sue dipendenze** in un container (Figura 4.1).


















	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
✓ 	smartlib	-	Running (5/5)			■ ⋮
	db-mongo 5cbbff941cac 	mongo:4.4	Running	27017:27017 	17 minutes ago	■ ⋮
	fiware-ld-context 0060329f5598 	httpd:alpine	Running	3004:80 	17 minutes ago	■ ⋮
	smartlib acf67ed7e691 	smartlib-smart-lib	Running	5050:5050 	17 minutes ago	■ ⋮
	fiware-iot-agent d21fde4da736 	fiware/iotagent-js	Running	4041:4041  7896:7896 	17 minutes ago	■ ⋮
	fiware-orion 0b8c3b081e19 	fiware/orion-ld:1.1	Running	1026:1026 	17 minutes ago	■ ⋮

Figura 4.1: Container Docker dell'applicativo

Capitolo 5

Sviluppo

5.1 Python

Per lo sviluppo del Back-End del sistema si è deciso di utilizzare il linguaggio di programmazione Python, principalmente per la **facilità di sviluppo** e per le differenti **librerie interfacciabili con MongoDB**.

5.2 Flask

Python nativamente non permette agevolmente di lavorare con le chiamate derivanti dal protocollo HTTP, quindi si è fatto affidamento ad un suo framework: **Flask**. Attraverso questo framework è possibile sviluppare agilmente un'infrastruttura che gestisca le **chiamate REST**.

Il Back-End presenta principalmente due file python:

1. `app.py`: **gestisce le chiamate REST**, occupandosi di cosa far visualizzare quando si inserisce un determinato url e quali informazioni passare alla pagina.
2. `mongo.py`: interfaccia costruita ad hoc ed in grado di:
 - **instaurare una connessione** con il database MongoDB
 - **eseguire query** mirate al recupero di informazioni filtrate.

La struttura di Flask è costituita dai due file principali e due cartelle, `templates` e `static`, che contengono rispettivamente i file HTML delle pagine e le immagini utilizzate all'interno di esse. Nella figura 5.1 è possibile visualizzare la struttura del framework.

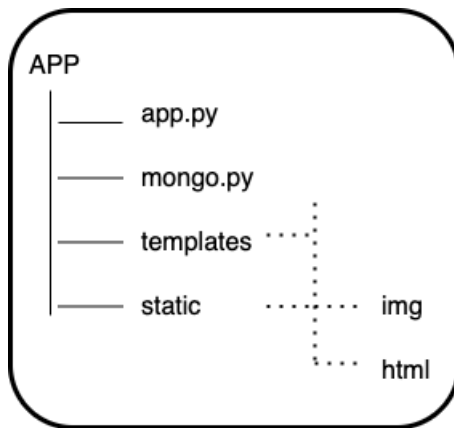


Figura 5.1: Struttura dell'applicazione Flask.

Il file `mongo.py` dispone di una serie di funzioni sviluppate con l'intenzione di **gestire query e connessione** al database MongoDB. Di seguito sono riportate le dichiarazioni delle funzioni che compongono l'interfaccia, da cui è possibile dedurre le rispettive query ed utilizzi all'interno dell'applicazione.

```

#connect to db
def mongo_connection()
#connection to collection entities
def mongo_connection_entities()

# DB OPERATION
def get_seats(entitiesCollection):
def random_seats(entitiesCollection)
def change_seats(entitiesCollection ,number_seats)
def change_book_availability(entitiesCollection ,isbn)
def list_all_category(entitiesCollection)
def list_all_books(entitiesCollection)
def list_all_books_filtered_by_available(entitiesCollection ,available)
def list_all_books_filtered_by_isbn(entitiesCollection ,isbn)
def list_all_books_filtered_by_autor(entitiesCollection ,autor)
def list_all_books_filtered_by_title(entitiesCollection ,title)
def list_all_books_filtered_by_category(entitiesCollection ,category)
def login(entitiesCollection ,email ,passw)
def insert_new_person(entitiesCollection ,email ,passw ,name ,surname ,faculty)
def insert_new_book(entitiesCollection , isbn , title , description , autor , category)
def insert_new_category(entitiesCollection , categoryName)
def remove_category_by_name(entitiesCollection , categoryName)

```

5.3 GUI

Il Front-End dell'applicazione è stato sviluppato utilizzando i classici linguaggi di markup **HTML** e di formattazione **CSS** accompagnati dal linguaggio di programmazione **JavaScript** e dal framework **Bootstrap** nella sua versione 4.0.

Bootstrap ha permesso di sviluppare un **infrastruttura modulare** e quindi di avere più componentistiche statiche utilizzabili come dei blocchi da inserire all'interno di ogni singola pagina. Le pagine HTML statiche utilizzano al loro interno degli elementi appartenenti a Flask in modo da poter avere una visualizzazione in **tempo reale** del contenuto del database. Come è stato possibile visualizzare nell'immagine 5.1 i file HTML sono presenti all'interno della cartella templates.

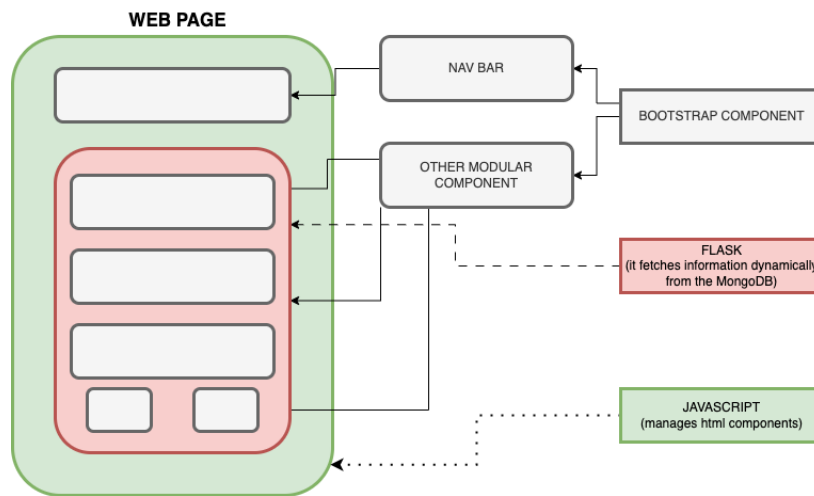


Figura 5.2: Gestione delle pagine HTML.

Nella figura 5.2 è possibile vedere come la pagina statica HTML viene composta e gestita. I componenti HTML e CSS sono modulari e costruiti attraverso Bootstrap, questi componenti vengono creati un'unica volta e poi importati in ogni pagina differente.

Il framework Flask permette di dare un **tasso di replicazione** ai componenti in base a ciò che si vuole mostrare, ovvero permette di estrarre i dati dal back-end e di fornirli al front-end, ciò per fornire una visualizzazione completa delle informazioni presenti nel database.

In fine i componenti JavaScript permettono di dare **dinamicità alla pagina** e di controllare alcune interazioni utente con i componenti in gioco. Anche il sistema di notifica derivante dai sensori è gestito da JavaScript.

Capitolo 6

Conclusioni

Dati gli obiettivi proposti nel capitolo 1 sono stati rispettati i seguenti punti:

- creazione di smart data model ad hoc per ogni entità dello scenario, data l'impossibilità di usarne di preesistenti
- emulazione dei sensori fisici tramite POST request
- connessione del Context Broker Orion come punto cardine del sistema
- utilizzo di MondoDB per storicizzare i dati
- creazione di una GUI