

A.A. 2021 - 2022

**M. CECCARELLI - E. D'ANGELIS - P. ZARRI**

# **LIBRARY SEAT RESERVATION**

**SYSTEM DESIGN DOCUMENT**



Introduzione	3
Obiettivo del sistema	3
Architettura software	4
Scomposizione in componenti	4
Breve Introduzione al REST	4
Architettura complessiva	5
Architettura nel dettaglio	6
Progettazione del Backend	7
Domain Model	7
Data Transfer Objects (DTO)	9
Data Access Objects (DAO)	10
Mapper	11
Controller	11
Endpoint	11
Annotazioni JPA E Persistenza	12
Altre annotazioni JPA	14
Elenco delle API REST	14
Testing	15
Deploy con Wildfly su Docker	16
Progettazione del Gateway	17
Domain Model	17
Annotazioni e Persistenza	18
Endpoint	18
Gateway: API REST	18
Progettazione del Frontend	19
Elementi fondamentali	19
Dipendenze	20
Diagramma di navigazione	21
Screenshot	22
Caso d'uso: Gestione della coda	26
Caso d'uso: Notifiche admin con RSocket	28

# INTRODUZIONE

## OBIETTIVO DEL SISTEMA

Il seguente documento presenta la modellazione relativa allo sviluppo di una applicazione web per la gestione delle prenotazioni di posti all'interno delle aule studio delle biblioteche di Firenze.

Gli obiettivi di maggior rilievo per il progetto sono:

- Un sistema di **gestione “a code”** delle richieste pervenute, strutturata nel seguente modo:
  - le richieste che vengono ricevute dal server devono essere trattate in modo “accodato” con il risultato di non sovraccaricare il server;
  - la presenza di un componente distribuito in stile gateway o proxy che intermedia le richieste;
  - un numero massimo di utenti che realizzano il caso d'uso;
  - una coda “esterna” con ticket ordinati e feedback circa la lunghezza della stessa e tempi di attesa per accedere al servizio.
- Due frontend distinti, strutturati nel seguente modo:
  - Uno per l'**utente semplice**
    - ▶ che realizza il caso d'uso della prenotazione e verifica delle proprie prenotazioni.
  - Uno per l'**utente amministratore (admin)**
    - ▶ che possiede un mini-pannello con calendario, avente riferimento del numero di prenotazioni per ogni fascia oraria;
    - ▶ gli amministratori non sono soggetti alla coda.

# ARCHITETTURA SOFTWARE

## SCOMPOSIZIONE IN COMPONENTI

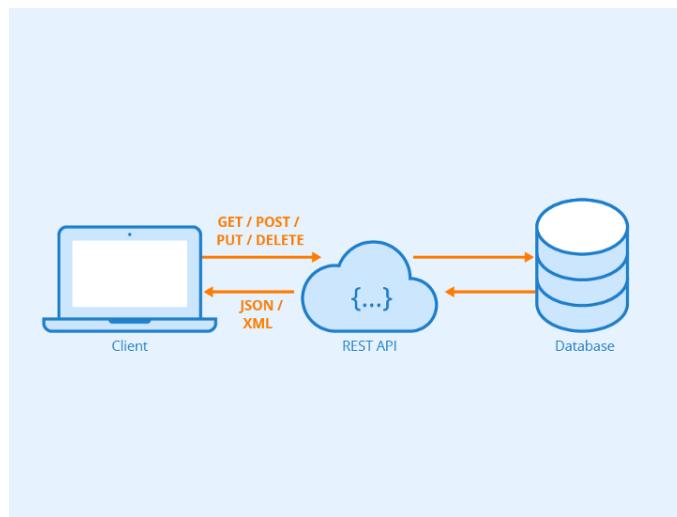
L'architettura software per questo progetto è di **natura distribuita**, in altri termini, il nostro sistema è suddiviso in **tre moduli**:

- **Modulo frontend**
- **Modulo backend**
- **Modulo gateway**

Il sistema verrà progettato su un'**architettura RESTful**.

## BREVE INTRODUZIONE AL REST

**REST** è l'acronimo di **REpresentational State Transfer** e, in generale, indica uno stile architetturale adatto a sviluppare **Servizi Web**.



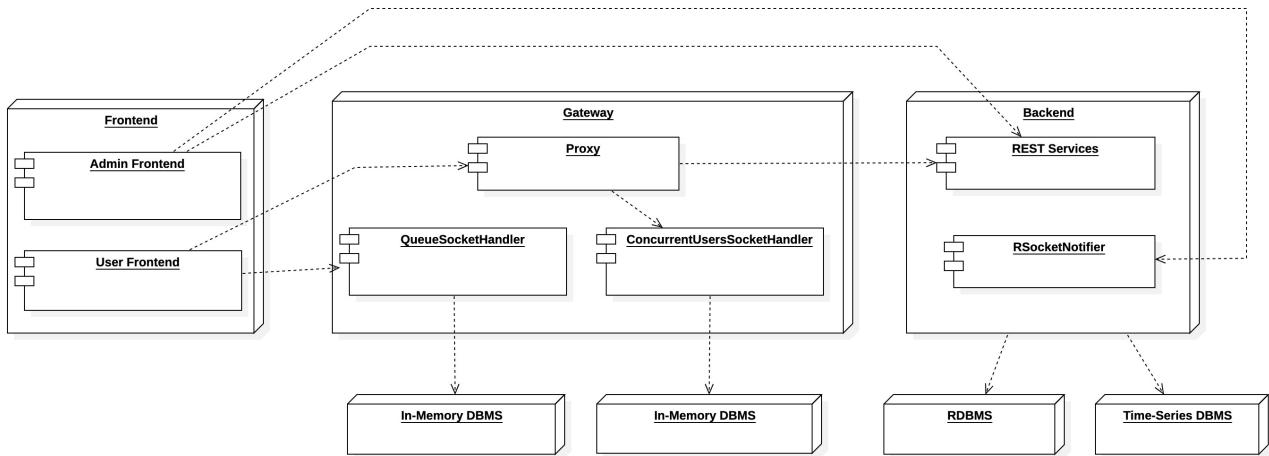
Un sistema software basato su un'**architettura RESTful** espone servizi tramite un insieme di **REST API** fruibili dai client tramite il protocollo **HTTP**. Una **REST API** è una collezione di **servizi REST**.

Un sistema basato su un'architettura RESTful deve seguire **alcuni principi**, tra cui:

- basarsi sullo stile Client/Server;
- l'utilizzo di HTTP(S) per lo scambio di messaggi;
- l'utilizzo del concetto di risorsa, identificabile tramite URI.

## ARCHITETTURA COMPLESSIVA

L'**architettura software complessiva** per questo progetto è la seguente:



Il modulo **Frontend** si presenta in due possibili realizzazioni:

- **Admin Frontend**: mette a disposizione dell'admin gli strumenti per la gestione delle biblioteche e delle relative prenotazioni;
- **User Frontend**: mette a disposizione dell'utente gli strumenti per la registrazione/login e per la prenotazione/cancellazione di prenotazioni in una o più biblioteche.

Il modulo **Gateway** fa da intermediario tra il *frontend* e il *backend* e si occupa della gestione della coda per l'accesso degli utenti al sistema di prenotazione.

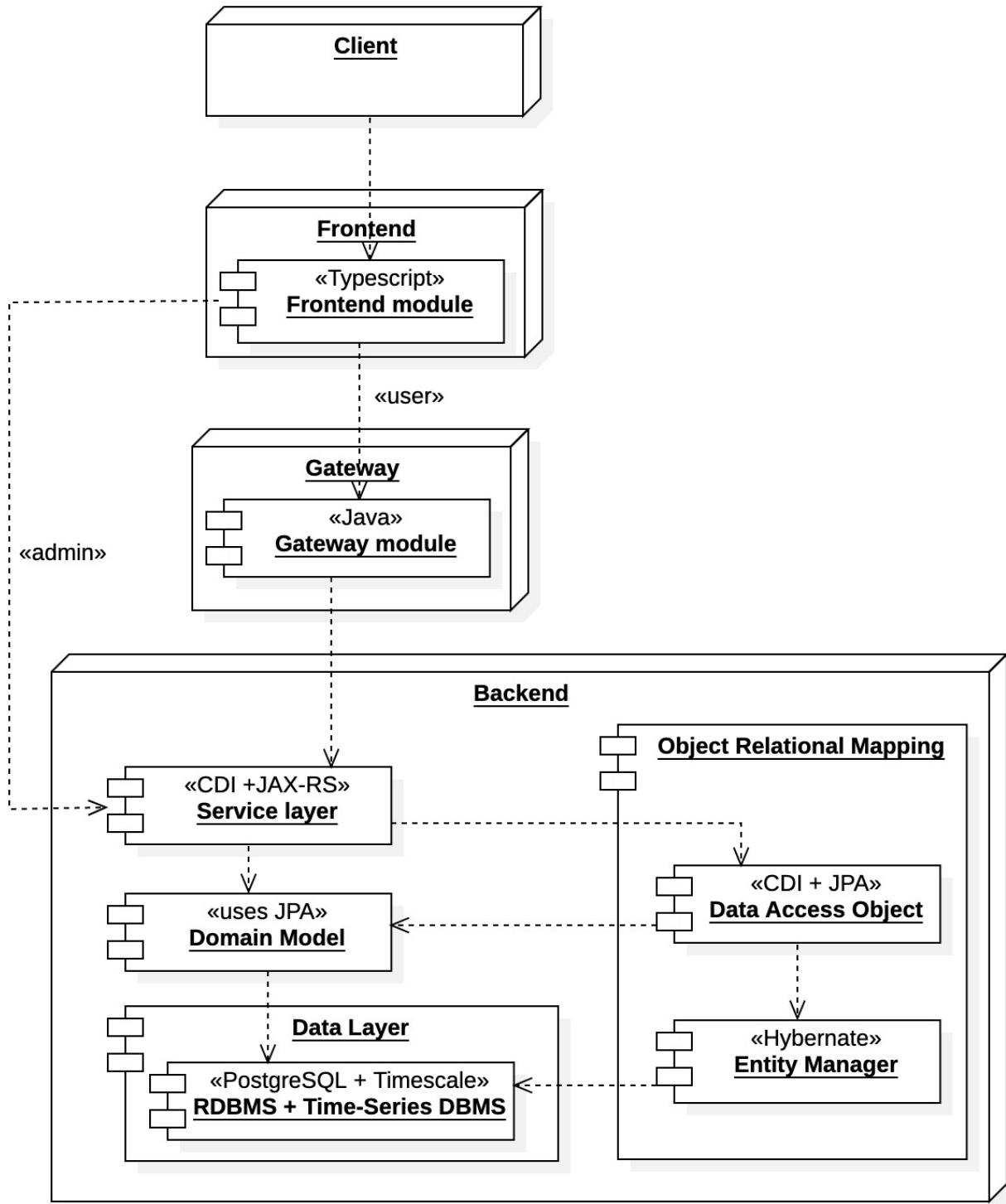
Il modulo **Backend** si interfaccia con il database ed espone le API REST al client.

In particolare l'Admin Frontend utilizza direttamente i servizi REST senza passare dalla coda e riceve gli aggiornamenti tempo reale sulle prenotazioni effettuate o cancellate da parte degli utenti (componente **RSocketNotifier**).

Tutte le richieste dell'User Frontend invece sono intercettate dal componente **Proxy** del Gateway, che stabilisce se inserire o meno un utente in coda (componente **ConcurrentUsersSocketHandler**). Nel caso quindi in cui un utente debba essere inserito in coda l'User Frontend fa uso del componente **QueueSocketHandler**.

## ARCHITETTURA NEL DETTAGLIO

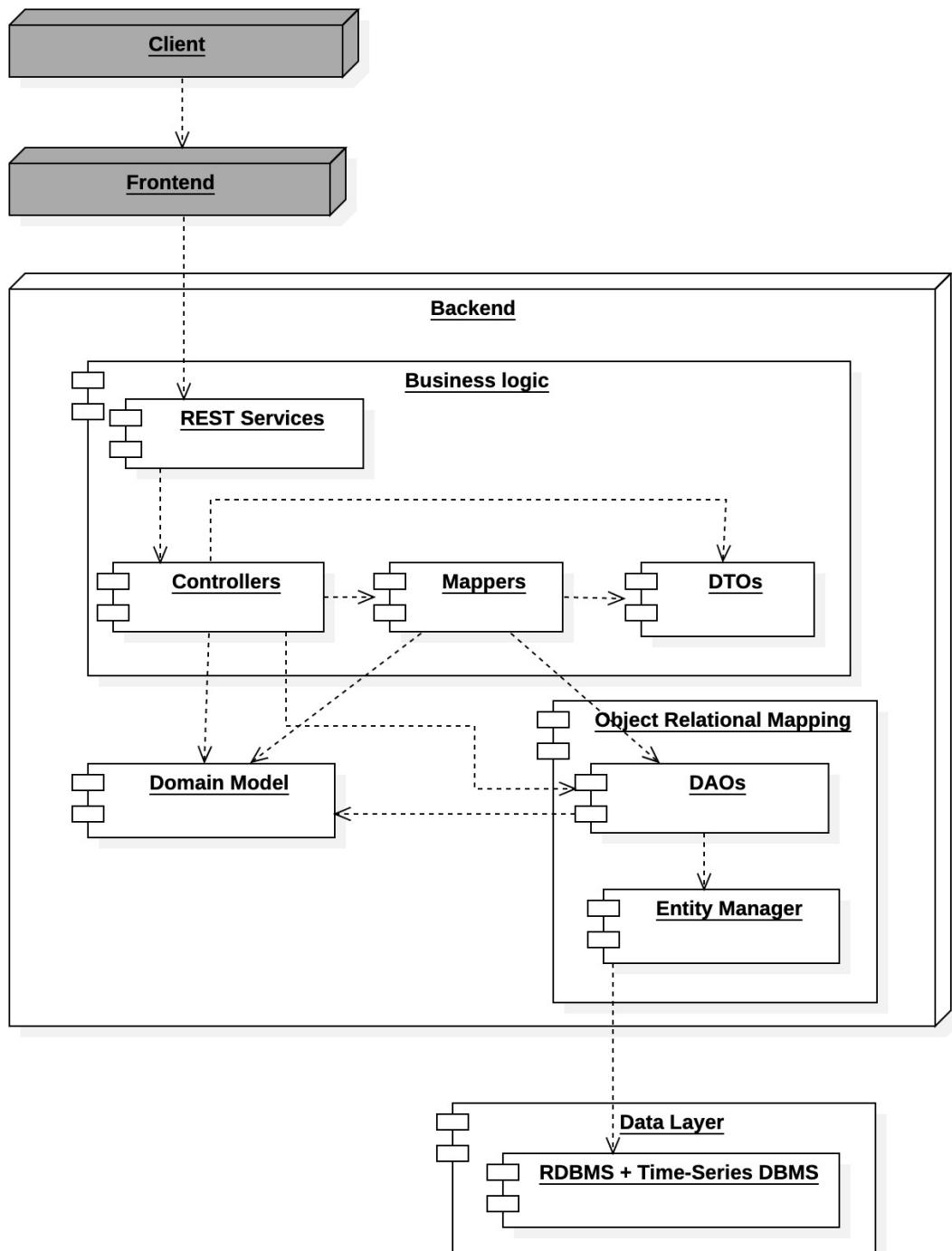
Prendendo in considerazione gli elementi già introdotti nella precedente sezione, si può pensare ad una configurazione più dettagliata della architettura software.



# PROGETTAZIONE DEL BACKEND

In questa sezione del documento viene descritta la progettazione del **Backend** basato su un'architettura **RESTful** e sul paradigma Client-Server. Il Backend è quella parte del sistema che risiede sul Server e che espone servizi tramite specifiche **REST API**.

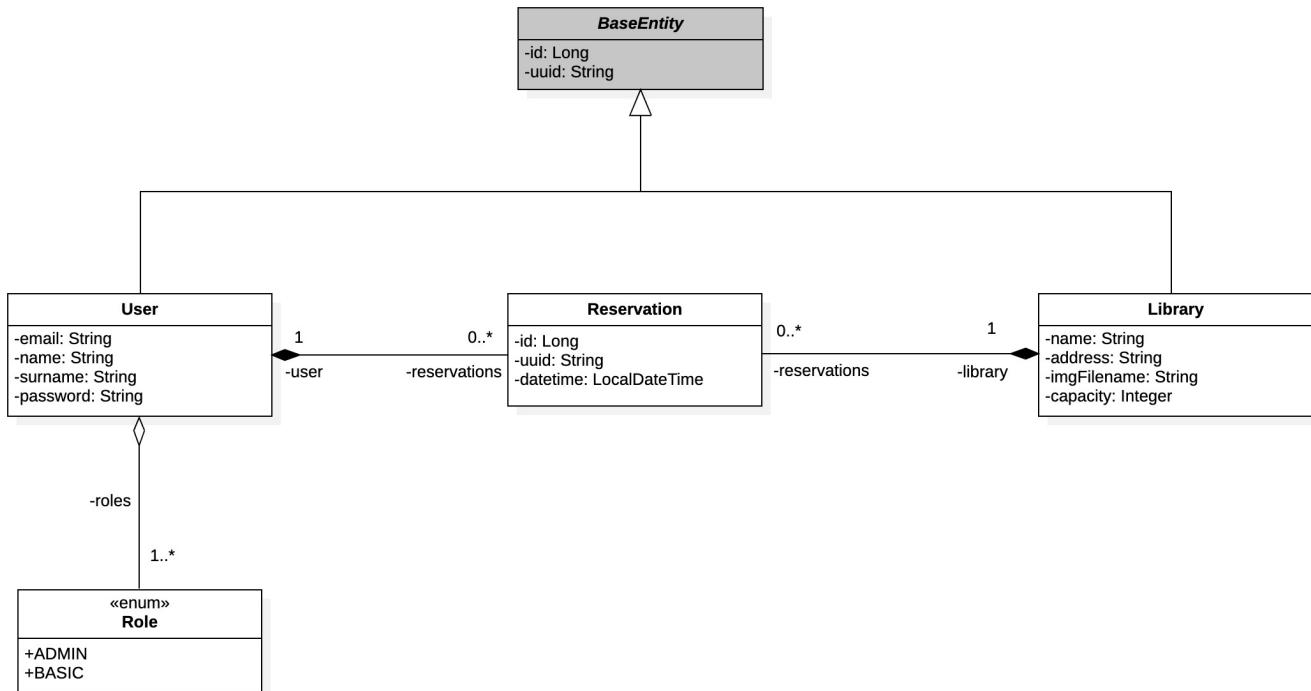
## DOMAIN MODEL



Il **Domain Model** rappresenta un componente fondamentale di un sistema software. Esso costituisce un modello concettuale che fornisce una rappresentazione formale del contesto operativo, descrivendo le entità che ne fanno parte e le loro relazioni.

Di seguito viene riportato il nostro modello di dominio di tipo *anemico* (che descrive solamente il contesto applicativo, modellandolo in termini di relazioni e di tipo):

- **BaseEntity**: classe astratta che viene estesa da **User** e **Library**;



- **User** rappresenta un utente e dispone dei seguenti attributi: un identificativo, un nome, un cognome, una password, una lista di ruoli e una lista di prenotazioni;
- **Library** rappresenta una biblioteca e dispone dei seguenti attributi: un identificativo, un nome, un indirizzo, una capienza, una lista di prenotazioni e il *filename* dell'immagine della biblioteca;
- **Reservation** rappresenta una prenotazione e dispone di un identificativo, una data con fascia oraria, un utente e una biblioteca;
- **Role** rappresenta un'enumerazione di possibili ruoli da assegnare a ciascun utente. Può essere *ADMIN* oppure *BASIC*.

## DATA TRANSFER OBJECTS (DTO)

L'acronimo **DTO** sta per **Data Transfer Object** e modella un oggetto software che rappresenta una versione semplificata di oggetti riferiti al modello di dominio.

Un DTO non ha una propria business logic e non espone metodi o funzionalità particolari. Svolge il ruolo di **trasportatore di dati** tra processi comunicanti con la finalità di ridurre la quantità di informazioni trasferite verso i chiamanti.

Per definire i DTO si parte dal modello di dominio sopra. Mostriamo di seguito i DTO implementati per il nostro progetto:

UserDTO
-id: long
-email: String
-name: String
-surname: String
-password: String
-roles: List<String>

ReservationDTO
-id: long
-userId: long
-userName: String
-userEmail: String
-libraryId: long
-libraryName: String
-datetime: String

LibraryDTO
-id: long
-name: String
-imgFilename: String
-address: String
-capacity: Integer

AdminNotificationDTO
-action: UserAction
-reservationId: Long
-libraryId: Long
-date: String
-notificationMessage: String

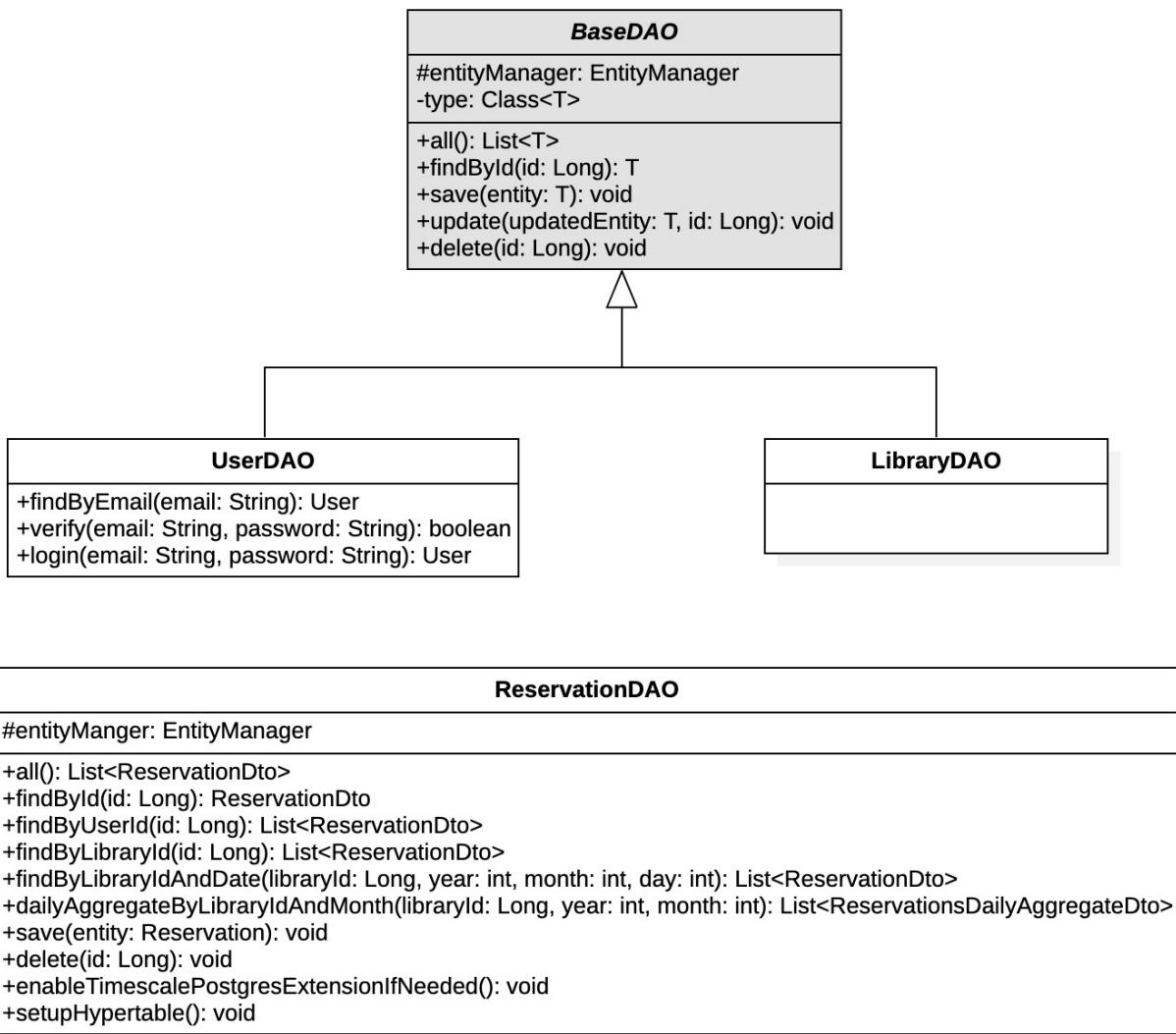
ReservationDailyAggregateDTO
-date: String
-countMorning: Integer
-countAfternoon: Integer

- **UserDTO**, **ReservationDTO** e **LibraryDTO** sono i DTO relativi alle entità descritte precedentemente;
- **AdminNotificationDTO** è il DTO che rappresenta una notifica di aggiunta o cancellazione di una prenotazione;
- **ReservationDailyAggregateDTO** è il DTO che rappresenta la divisione delle prenotazioni giornaliere per una biblioteca, raggruppate per mattina e pomeriggio.

## DATA ACCESS OBJECTS (DAO)

L'acronimo **DAO** sta per Data Access Object, e modella un oggetto software che fornisce un'interfaccia astratta verso il livello di persistenza ed espone un insieme di metodi utili per coprire le principali operazioni CRUD. Svolge il ruolo di intermediario tra le entità del modello di dominio e le tabelle "reali" del database, avvalendosi del supporto di un **EntityManager**.

Mostriamo di seguito i DAO implementati per il nostro progetto:



## MAPPER

Un **mapper** modella un oggetto software che espone alcuni metodi per “mappare” gli oggetti Java (entità del domain model) in DTO e viceversa. In genere si deve implementare i metodi:

- `convertToDto()` per la conversione entità -> DTO
- `transferToEntity()` per la conversione DTO -> entità

## CONTROLLER

I **controller** rappresentano dei componenti software in grado di assolvere il caso d’uso invocando o implementando direttamente i metodi caratterizzanti. Nello specifico:

- manipolano le entità dei Domain Model per intermediazione dei DAO
- elaborano ed interpretano istanze di DTO tramite intermediazione dei Mapper

## ENDPOINT

Tutti i servizi associati ad una certa risorsa sono gestiti da un **Endpoint**, che quindi può essere definito come un contenitore logico che raggruppa e organizza i servizi esposti e che li rende fruibili ai Client, in modo univoco tramite URI.

Per identificare gli Endpoint sono presenti tre **REST API**: una relativa agli **User**, una relativa alle **Library** e un’altra relativa alle **Reservation**. Si identificano quindi tre endpoint, chiamati rispettivamente:

- **UserRestServices** (`api/users/`)
- **LibraryRestServices** (`api/libraries/`)
- **ReservationRestServices** (`api/reservations/`)

In un capitolo successivo verranno poi descritte nel dettaglio le **API** utilizzate nel nostro progetto.

## ANNOTAZIONI JPA E PERSISTENZA

Di seguito sono riportate tutte le **annotazioni JPA** utilizzate nel backend durante l'implementazione delle classi del Domain Model.

### BaseEntity

La classe  **BaseEntity** è una classe che fa da superclasse per le varie entità del Domain Model. Il suo compito è quello di gestire gli identificativi (sia l'**uuid** usato a livello app che l'**id** a livello database) e quello di effettuare l'override del metodo `hashCode()` e `equals()` in modo che venga fatto un controllo di uguaglianza basato sull'**uuid**.

Le annotazioni utilizzate per questa classe sono:

- `@MappedSuperclass`
- `@Id` e `@GeneratedValue(strategy = GenerationType.IDENTITY)` per l'attributo `id`
- `@Column(unique = true)` per l'attributo `uuid`

### User

La classe  **User** estende la classe astratta  `BaseEntity`.

Le annotazioni utilizzate per questa classe sono:

- `@Entity`
- `@Table(name = "users")`
- `@ElementCollection(fetch = FetchType.EAGER)` per mappare la lista di ruoli
- `@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)` per indicare una relazione con cardinalità *uno-a-molti* con le prenotazioni

### Library

La classe  **Library** estende la classe astratta  `BaseEntity`.

Le annotazioni utilizzate per questa classe sono:

- `@Entity`
- `@Table(name = "libraries")`

- `@OneToMany(mappedBy = "library", cascade = CascadeType.ALL)` per indicare una relazione con cardinalità *uno-a-molti* con le prenotazioni

## Reservation

La classe **Reservation** non estende la classe astratta  `BaseEntity`, come le classi precedenti, a causa dell'integrazione con **TimescaleDB**.

Infatti TimescaleDB richiede che la colonna “temporale” della tabella (nel nostro caso l'attributo `datetime`) sia anche chiave primaria, ma JPA non consente di definire una chiave primaria composita estendendo una superclasse astratta che già definisce un `@Id`.

Dunque le annotazioni utilizzate per questa classe sono:

- `@Id, @SequenceGenerator(name="seq", sequenceName="db_reservations_seq", allocationSize=1)` e `@GeneratedValue(generator = "seq")` per l'attributo `id`. In questo caso non era possibile utilizzare la strategia `.GenerationType.IDENTITY` poiché JPA non la supporta sulle classi con chiave primaria composita
- `@IdClass(ReservationCompositeKey.class)` per definire la chiave primaria composita
- `@Id` e `@Column(columnDefinition = "TIMESTAMP", nullable = false)` per l'attributo `datetime`
- `@Column(nullable = false)` per l'attributo `uuid`
- `@ManyToOne(fetch = FetchType.LAZY)` per indicare una relazione con cardinalità *molti-a-uno* con l'utente
- `@ManyToOne(fetch = FetchType.LAZY)` per indicare una relazione con cardinalità *molti-a-uno* con la biblioteca

Con tali annotazioni la tabella relativa agli oggetti della classe **Reservation** avrà due colonne contenenti due `id`: uno dell'utente e uno della biblioteca.

La classe **ReservationCompositeKey** utilizzata nell'annotazione `@IdClass` rappresenta la chiave primaria composita per la classe `Reservation` e contiene i due attributi `id` e `datetime`.

## ALTRÉ ANNOTAZIONI JPA

Di seguito sono riportate tutte le **annotazioni JPA** utilizzate nel backend nelle altre classi:

- `@RequestScoped` per i controller e i mapper
- `@Singleton` e `@Startup` per i Bean inizializzati all'avvio dell'applicazione
- `@Provider` e `@Priority(Priorities.AUTHENTICATION)` per il filtro delle richieste HTTP

## ELENCO DELLE API REST

Presentiamo adesso le **API REST** per quanto riguarda **User**, **Library** e **Reservation**.

### User API

Le API per l'utente sono le seguenti:

- GET `api/users/` ritorna tutti gli utenti
- GET `api/users/{id}` ritorna l'utente con l'id specificato
- POST `api/users/login/` esegue il login
- POST `api/users/signup/` esegue la registrazione
- DELETE `api/users/delete/{id}` cancella l'utente con l'id specificato
- PUT `api/users/update/` aggiorna i dati di un utente

### Library API

Le API per le biblioteche sono le seguenti:

- GET `api/libraries/` ritorna tutte le biblioteche
- GET `api/libraries/{id}` ritorna la biblioteca con l'id specificato
- POST `api/libraries/add/` aggiunge una nuova biblioteca
- DELETE `api/libraries/delete/{id}` cancella la biblioteca con l'id specificato
- PUT `api/libraries/update/` aggiorna i dati di una biblioteca

## Reservation API

Le API per le prenotazioni sono le seguenti:

- GET `api/reservations/` ritorna tutte le prenotazioni
- GET `api/reservations/{id}` ritorna la prenotazione con l'id specificato
- GET `api/reservations/user/{id}` ritorna le prenotazioni dell'utente con l'id specificato
- GET `api/reservations/library/{id}` ritorna le prenotazioni della biblioteca con l'id specificato
- GET `api/reservations/library/{id}/{year}/{month}/{day}` ritorna le prenotazioni della biblioteca con l'id specificato e per la data specificata
- GET `api/reservations/stats/library/{id}/{year}/{month}` ritorna il conteggio delle prenotazioni divise per fasce orarie per la biblioteca con l'id specificato e per il mese specificato (per maggiore efficienza usa il time bucket di Timescale)
- POST `api/reservations/add/` aggiunge una nuova prenotazione
- DELETE `api/reservations/delete/{id}` cancella la prenotazione con l'id specificato

Alcune di queste **API REST** sono utilizzabili soltanto da utenti con privilegi di **ADMIN** tramite l'annotazione `@RolesAllowed` (ad esempio l'aggiunta o la cancellazione di una biblioteca).

## TESTING

Per il **testing** abbiamo utilizzato il framework **JUnit5** (per lo *unit testing*) e **Mockito** (per evitare problemi di dipendenze tra oggetti e aumentare l'isolamento).

Sono stati testati diversi **livelli dell'architettura (Domain Model, Business Logic e DAO)** e i loro componenti critici.

In particolare per il **Domain Model** è stata testata la superclasse  `BaseEntity` e `Reservation` per verificare la corretta identità, uguaglianza ed inizializzazione.

Per la **Business Logic** i test vengono eseguiti in isolamento e per eventuali dipendenze esterne sono stati utilizzati i Mocks.

Per i **DAO** sono stati testati i metodi CRUD (`find`, `save`, `update` e `delete`) ed eventuali altri metodi rilevanti. Per testare la persistenza è stato utilizzato un database *in-memory* (**HyperSQL**) tramite l'aggiunta del file `persistence.xml` nella cartella `test`.

## DEPLOY CON WILDFLY SU DOCKER

Per effettuare il deploy è stato utilizzata un'immagine Docker di Wildfly versione 24.0.0, modificata per il supporto a database PostgreSQL (su cui si basa anche TimescaleDB).

In particolare sono stati prodotti i seguenti file:

- **Dockerfile**: a partire dall'immagine Wildfly ufficiale (`jboss/wildfly:24.0.0.Final`), genera un'immagine Docker modificata per il supporto a PostgreSQL, tramite i seguenti step eseguiti con la CLI di `jboss`:

- Si scarica il driver PostgreSQL dalla repository Maven ufficiale;
- Si aggiunge il **modulo** PostgreSQL facendo riferimento al driver scaricato prima;
- Si aggiunge il **driver** PostgreSQL;
- Si specifica un **Datasource** principale da utilizzare poi nel progetto Java, con un URL di connessione del tipo:

```
jdbc:postgresql://${DB_HOST}:${DB_PORT}/${DB_NAME}
```

- `docker-compose.yml`: specifica due services, uno per il database PostgreSQL (con immagine di base `timescale/timescaledb:latest-pg13`) e l'altro per l'application server, con l'immagine Wildfly modificata tramite Dockerfile (descritto sopra).

Per avviare il progetto sarà dunque sufficiente avviare il file Docker Compose con il comando:

```
docker-compose up
```

che attiva e mette in comunicazione i singoli container.

All'interno del container basato su Wildfly è stato aggiunto un volume (`/workdir/deploy/wildfly/`) per il deploy automatico di file .WAR. Le seguenti porte sono state

esposte: **8080** per accedere all'applicazione, **9990** per la console admin, **5005** per il debugging e **7878** per eventuali comunicazioni tramite RSocket.

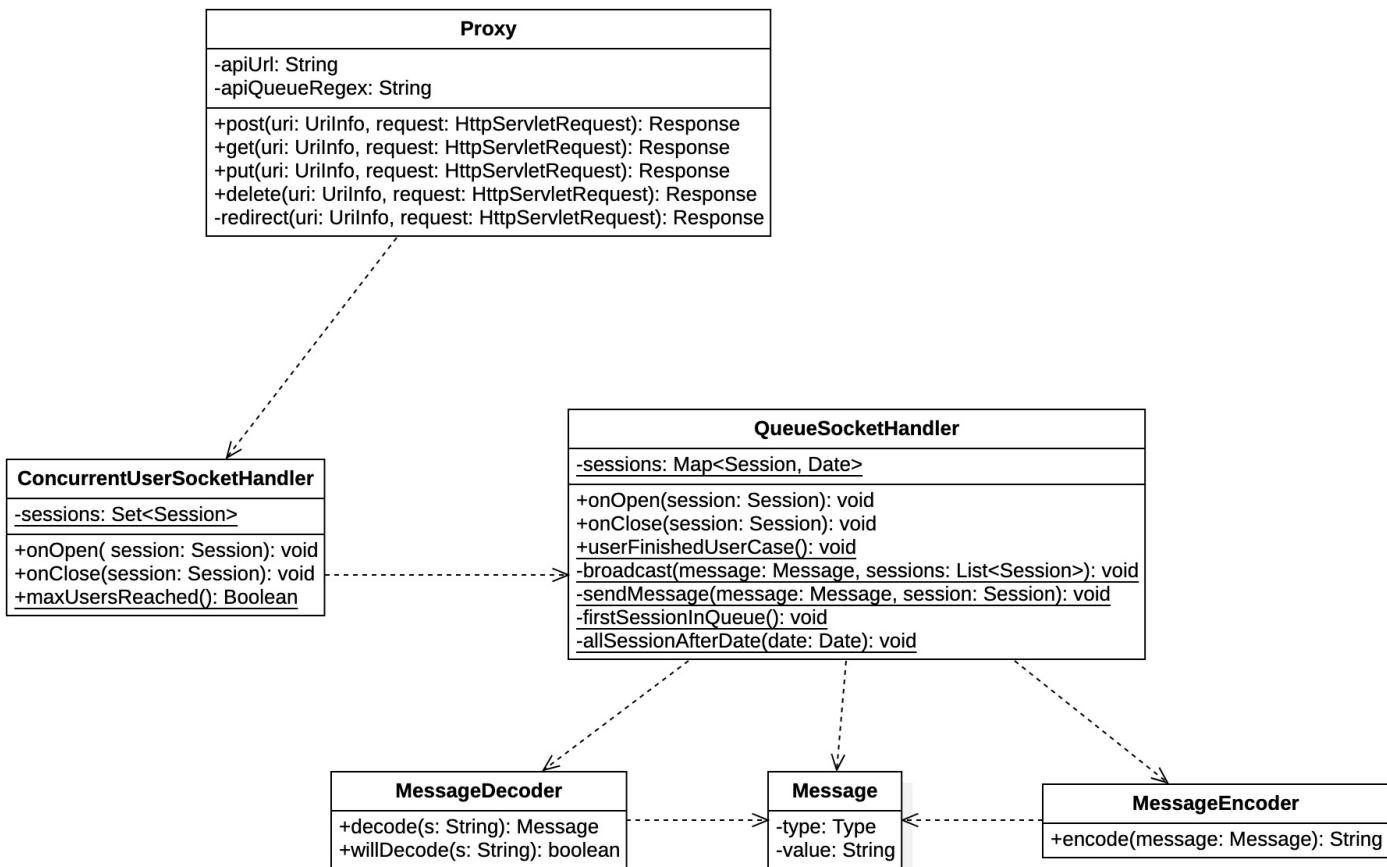
Come ultimo passo, per supportare la creazione di hypertable di Timescale tramite Hibernate direttamente all'interno del progetto, è stato registrato un dialect che estende quello esistente per Postgres e inoltre mappa il tipo OTHER (tipo di ritorno della chiamata SQL non nativa "SELECT create\_hypertable ...") in stringa. Questo dialect viene poi specificato nella proprietà `hibernate.dialect` del file `persistence.xml`.

## PROGETTAZIONE DEL GATEWAY

In questa sezione del documento viene descritta la progettazione del **Gateway** che fa da intermediario tra il *frontend* e il *backend* e si occupa della gestione della coda per l'accesso degli utenti al sistema di prenotazione.

### DOMAIN MODEL

Di seguito viene riportato il nostro modello di dominio per quanto riguarda il Gateway:



## ANNOTAZIONI E PERSISTENZA

Di seguito sono riportate tutte le **annotazioni JPA** utilizzate nel gateway durante l'implementazione delle classi del Domain Model:

- `@ServerEndpoint` per generare gli endpoint web socket nelle classi utilizzate per la gestione della coda (`ConcurrentUsersSocketHandler` e `QueueSocketHandler`)
- `@Provider` per il filtro delle richieste HTTP

## ENDPOINT

Tutti i servizi associati ad una certa risorsa sono gestiti da un **Endpoint**, che quindi può essere definito come un contenitore logico che raggruppa e organizza i servizi esposti e che li rende fruibili ai Client, in modo univoco tramite URI.

Per identificare gli Endpoint sono presenti due **URL WebSocket**: uno relativo alla coda e l'altro relativo agli utenti presenti all'interno dell'applicazione. Si identificano quindi due endpoint:

- **QueueSocketHandler** (`ws://<IP>:<PORT>/gateway/queue`)
- **ConcurrentUsersSocketHandler** (`ws://<IP>:<PORT>/gateway/concurrent-users`)

## GATEWAY: API REST

Presentiamo adesso le **API REST** per quanto riguarda il gateway:

- GET `gateway/api/{*}` reindirizza la richiesta al backend. In caso di richiesta relativa ad una biblioteca, esegue anche un controllo per stabilire se mettere in coda un utente;
- POST `gateway/api/{*}` reindirizza la richiesta al backend;
- PUT `gateway/api/{*}` reindirizza la richiesta al backend;
- DELETE `gateway/api/{*}` reindirizza la richiesta al backend.

# PROGETTAZIONE DEL FRONTEND

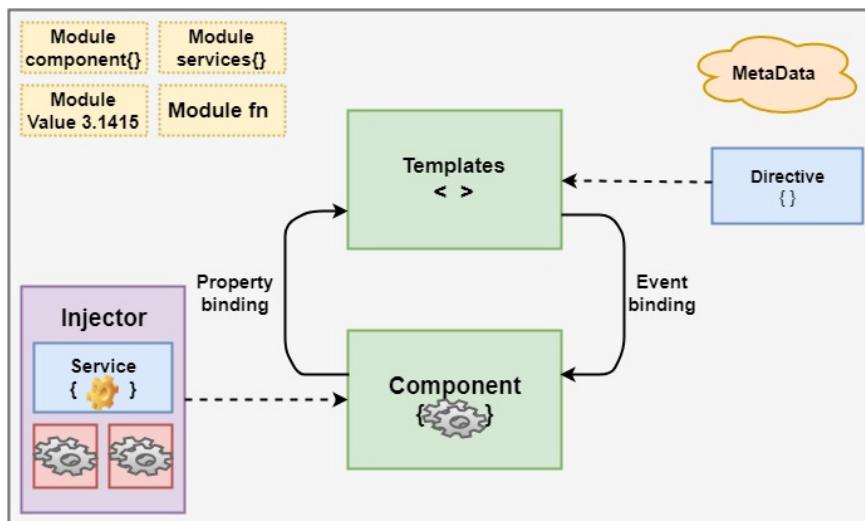
Il **Frontend** è stato sviluppato tramite il framework **Angular** con l'aiuto dei componenti forniti dalla libreria **Angular Material**.



Seguendo il workflow tipico dello sviluppo di un'applicazione Angular, il frontend è stato suddiviso in diversi componenti: ogni componente consiste in una cartella contenente:

- un **file HTML** che ne definisce la vista;
- un **file CSS**;
- un **file TS** che ne definisce il comportamento.

Il linguaggio di base è TypeScript. L'applicazione è stata realizzata come una Single Page Application (SPA) orientata ai servizi e con paradigma Model-View-Controller.

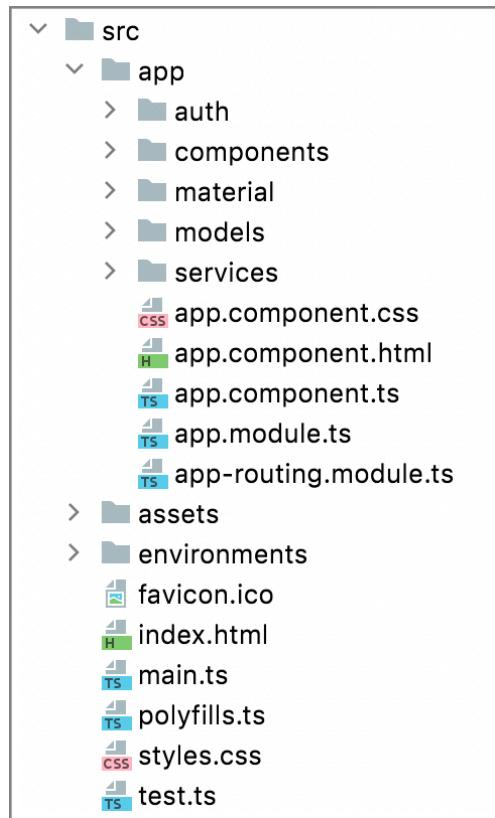


## ELEMENTI FONDAMENTALI

Gli elementi fondamentali sono:

- **Modello**: stato del sistema, legato in maniera bidirezionale ai servizi REST del backend;
- **Componente**: controllori del pattern con la responsabilità di attuare i comandi inviati dagli utenti e modificare lo stato del modello
- **Template**: frammenti di documenti HTML che rappresentano le viste (view di MVC); solitamente ad ogni componente è associato un template che definisce il layout di un componente

- **Property Binding:** permette di osservare un attributo di un componente e propagare i cambiamenti nella vista
- **Event Binding:** permette di associare un evento (es. click di un bottone) ad un metodo di un componente che controlla in quel momento la vista
- **Direttive:** istruzioni speciali associabili ad un template, in grado di modificarlo prima che venga renderizzato. Ne esistono due tipi:
  - **Strutturali:** incidono nel layout a livello di DOM
  - **Di Attributo:** alterano soltamente l'aspetto di un elemento già presente nel DOM
- **Servizi:** classi che mettono a disposizione dei metodi, riusabili in vari controller tramite dependency injection con `@Injectable`, allo scopo di implementare varie funzionalità (es: validazione campi di input, condivisione variabili di stato, logging, comunicazione con i servizi REST esposti da un backend)
- **Moduli:** raggruppano componenti, template e risorse per favorire la riusabilità.



Struttura del progetto

## DIPENDENZE

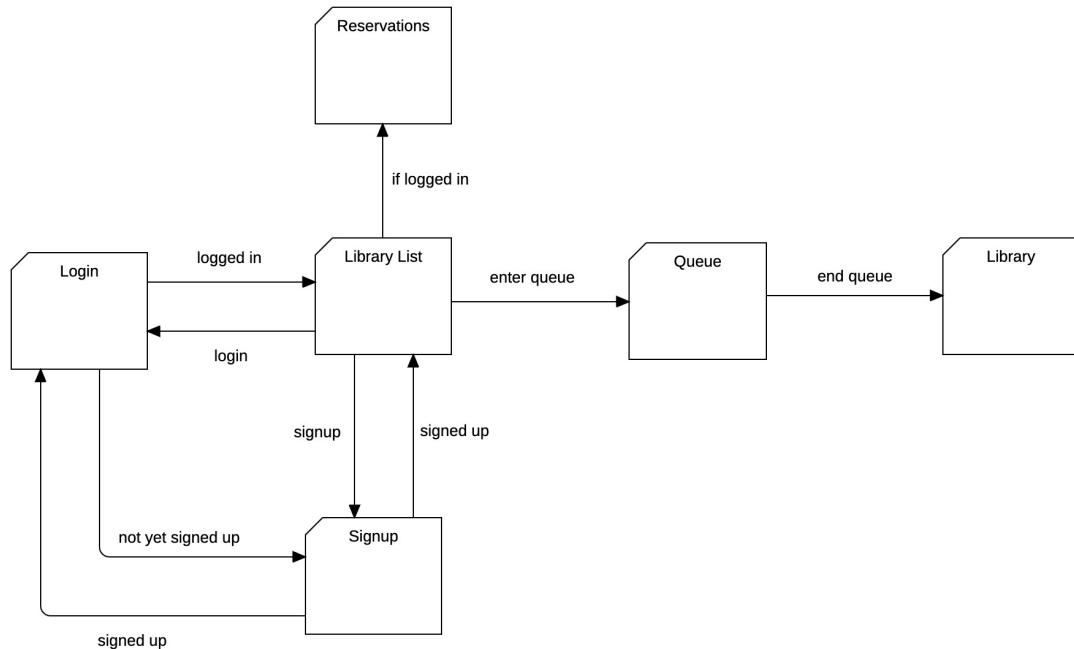
Il modulo frontend realizzato fa uso delle seguenti dipendenze esterne:

- **Angular Material:** design system per i componenti UI;
- **Leaflet:** libreria JavaScript per mappe interattive;
- **angularx-qrcode:** libreria per la generazione di QR code;
- **RSocket:** protocollo a livello applicativo per stream reattivi;
- **RxJS:** libreria JavaScript che mette a disposizione il tipo `Observable` e la funzione `pipe()`, che permette di compiere più di operazioni sul risultato in emissione.

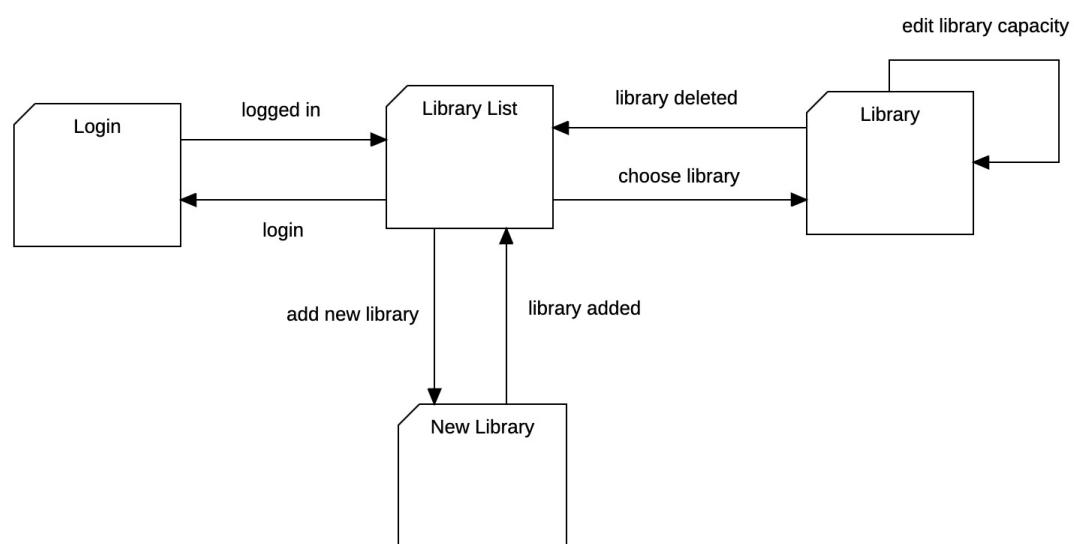
## DIAGRAMMA DI NAVIGAZIONE

Di seguito vengono mostrati i due **diagrammi di navigazione** (uno per l'**utente** e uno per l'**admin**), che elencano le pagine che costituiscono l'interfaccia del frontend e mostrano le transizioni che avvengono tra esse durante il normale utilizzo dell'applicazione.

### Diagramma di Navigazione per l'Utente



### Diagramma di Navigazione per l'Admin



## SCREENSHOT

Di seguito vengono riportati alcuni **screenshot** delle varie pagine della nostra applicazione.

Login

Email \*

Password \*

Login

Non sei registrato? [Clicca qui](#)

Pagina di Login

Signup

Nome \*

Cognome \*

Email \*

Password \*

Signup

Sei già registrato? [Clicca qui](#)

Pagina di Signup

Prenota il tuo posto in biblioteca

Cerca biblioteca...

Biblioteca Villa Bandini  
Via del Paradiso, 5, Firenze  
  
Capacità: 50 posti

Biblioteca Mario Luzi  
Via Ugo Schiff, 8, Firenze  
  
Capacità: 70 posti

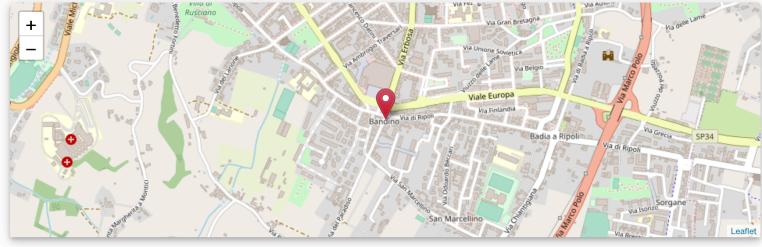
Biblioteca delle Oblate  
Via dell'Oriuolo, 24, Firenze  
  
Capacità: 60 posti

Biblioteca Palagio di Parte Guelfa  
Piazza della Parte Guelfa, Firenze  
  
Capacità: 30 posti

Biblioteca Biblioteca Biblioteca dei Biblioteca Dino

Pagina Home

Biblioteca Villa Bandini  
Via del Paradiso, 5, Firenze



Pagina di una biblioteca (1/2)

## LIBRARY SEAT RESERVATION

Tempo rimasto: 1:32

SET 2021						
D	L	M	M	G	V	S
SET		1	2	3	4	
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

**Venerdì 24 Settembre**

Occupazione mattina: 50 / 50

Occupazione pomeriggio: 26 / 50

Fascia 8.00 - 13.00  
 Fascia 13.00 - 19.00

**Prenota**

© Copyright LibrarySeatReservation 2021

Pagina di una biblioteca (2/2)

Utente 8934 ▾

Sei in coda...



Sei la prima persona in coda

Potrai accedere al servizio in circa 15 secondi

Una volta terminata l'attesa, verrai reindirizzato automaticamente.

© Copyright LibrarySeatReservation 2021

Pagina della coda

Utente 43 ▾

**Le mie prenotazioni**

Future	Passate	Biblioteca	QR Code	Elimina
8:00	Lunedì 27 Settembre	Biblioteca delle Oblate		
8:00	Lunedì 4 Ottobre	Biblioteca ISIS Leonardo da Vinci		
8:00	Venerdì 8 Ottobre	Biblioteca del Galluzzo		
13:00	Domenica 17 Ottobre	Biblioteca Pietro Thouar		
8:00	Giovedì 28 Ottobre	Biblioteca Dino Pieraccioni		

Items per page: 5 ▾ 1 – 5 of 5 | < < > > |

Pagina delle prenotazioni

## LIBRARY SEAT RESERVATION

Le mie prenotazioni

Future	Passate			
Ora	Data	Biblioteca	QR Code	Elimina
8:00	Lunedì 27 Settembre	Biblioteca delle Oblate		
8:00	Lunedì 4 Ottobre			
8:00	Venerdì 8 Ottobre			
13:00	Domenica 17 Ottobre			
8:00	Giovedì 28 Ottobre			

Lunedì 27 Settembre - ore 8:00  
Biblioteca delle Oblate

QR code relativo a una prenotazione

OTT 2021

D	L	M	M	G	V	S	
31	3	4	5	6	7	8	9
10	11	12	13	14	15	16	
17	18	19	20	21	22	23	
24	25	26	27	28	29	30	

### Prenotazioni per Mercoledì 20 Ottobre

Mattina (8.00-13.00)		Pomeriggio (13.00-19.00)	
Occupazione: 34 / 50			
Nome	Email	Nome	Email
Utente 4402	user4402@email.com	Utente 3297	user3297@email.com
Utente 196	user196@email.com	Utente 5167	user5167@email.com
Utente 2071	user2071@email.com		

1 - 5 of 34 | < < > >| Items per page: 5

### Modifica capacità biblioteca

Valore: 50

●

Modifica

### Elimina biblioteca

Verranno eliminate anche tutte le prenotazioni per questa biblioteca.

**Elimina biblioteca**

Dashboard Admin

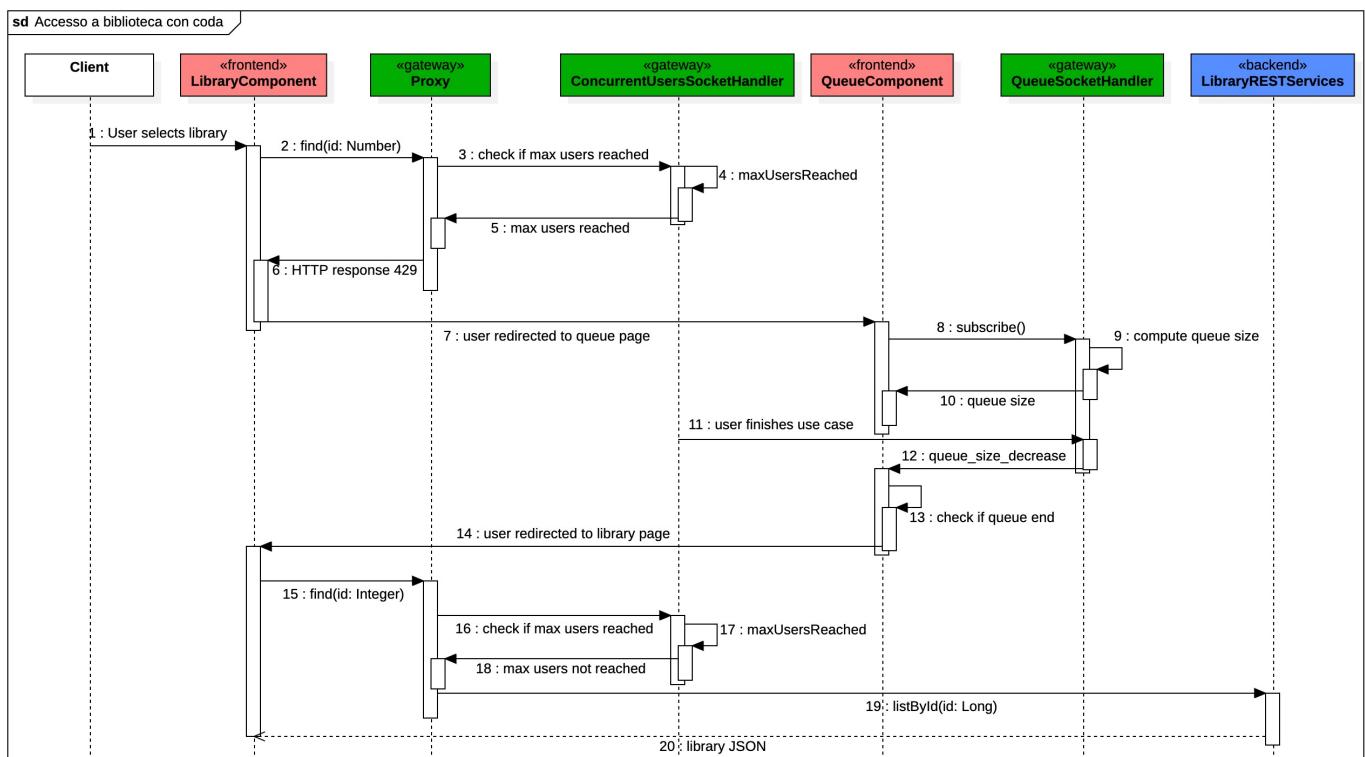
## LIBRARY SEAT RESERVATION

The screenshot shows a web-based administrative interface for library seat reservations. At the top, there is a dark blue header bar with the logo 'LIBRARY SEAT RESERVATION' on the left and a user dropdown menu on the right labeled 'Utente Admin (Admin)'. Below the header, the main content area has a white background. A central modal window titled 'Aggiungi Biblioteca' (Add Library) contains four input fields: 'Nome \*' (Name), 'Nome file immagine \*' (Image file name), 'Indirizzo \*' (Address), and 'Capacità \*' (Capacity). Each field is accompanied by a small descriptive label and a required asterisk (\*). To the right of the 'Capacità' field is a dropdown arrow icon. At the bottom right of the modal is a grey button labeled 'Aggiungi' (Add).

Pagina admin per l'aggiunta di una biblioteca

## CASO D'USO: GESTIONE DELLA CODA

Di seguito viene mostrato il **sequence diagram** del caso d'uso di un **accesso a una biblioteca nel caso in cui l'utente entri in coda** (per semplicità supponiamo che l'utente sia il primo in coda).



Le **sequenze di operazioni** eseguite per questo caso d'uso sono le seguenti:

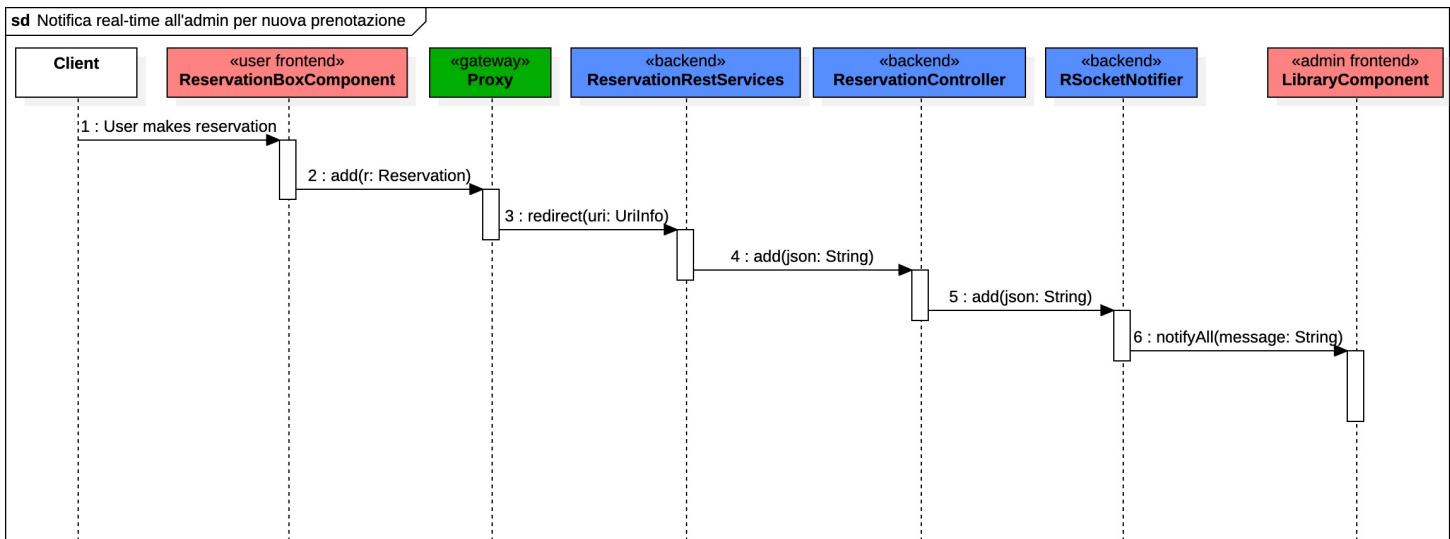
1. L'**utente** seleziona una biblioteca;
2. Il componente **LibraryComponent** tramite il servizio **LibraryService** del frontend richiede al gateway le informazioni sulla biblioteca scelta tramite REST API;
3. La classe **Proxy** del modulo gateway intercetta la richiesta e controlla se è stato raggiunto il numero di utenti in contemporanea;
4. La classe **ConcurrentUsersSocketHandler** del modulo gateway, che tiene traccia di tutti gli utenti che stanno prenotando un posto in biblioteca, controlla se è stato raggiunto il numero massimo di utenti in contemporanea;
5. Sempre la classe **ConcurrentUsersSocketHandler** restituisce il booleano per indicare se il numero è raggiunto o meno;

6. Il **Proxy** risponde alla richiesta HTTP con il codice di risposta **429 (Too Many Requests)**;
7. L'**utente** viene reindirizzato alla pagina della coda;
8. Viene attivato il componente **QueueComponent** che esegue il `subscribe()` e si connette tramite socket alla classe **QueueSocketHandler** del modulo gateway;
9. La classe **QueueSocketHandler**, che tiene traccia di tutti gli utenti che entrano ed escono dalla coda, calcola la dimensione della coda corrente;
10. Sempre la classe **QueueSocketHandler** invia un messaggio socket contenente la dimensione della coda, permettendo all'utente di visualizzare il numero di persone in coda davanti a lui insieme ad una stima del tempo di attesa;
11. Quando un altro utente ha terminato la fase di prenotazione e libera un posto, la classe **ConcurrentUsersSocketHandler** notifica anche **QueueSocketHandler**;
12. La classe **QueueSocketHandler** invia un messaggio socket al **QueueComponent** per notificare il decremento della coda;
13. Il componente **QueueComponent** controlla se la coda è terminata;
14. L'**utente** viene reindirizzato alla pagina della biblioteca;
15. Il componente **LibraryComponent** tramite il servizio **LibraryService** del frontend richiede al gateway le informazioni sulla biblioteca scelta tramite REST API;
16. La classe **Proxy** del modulo gateway intercetta la richiesta e controlla se è stato raggiunto il numero di utenti in contemporanea;
17. La classe **ConcurrentUsersSocketHandler** del modulo gateway controlla se è stato raggiunto il numero massimo di utenti in contemporanea;
18. Sempre la classe **ConcurrentUsersSocketHandler** restituisce il booleano per indicare se il numero è raggiunto o meno;
19. La classe **Proxy** del modulo gateway inoltra la richiesta al backend;
20. La classe **LibraryRESTServices** del modulo backend restituisce al **LibraryComponent** le informazioni richieste in formato JSON.

A questo punto l'utente ha a disposizione una finestra di **2 minuti** per effettuare una prenotazione, al termine dei quali verrà automaticamente terminato il caso d'uso.

## CASO D'USO: NOTIFICHE ADMIN CON RSOCKET

Di seguito viene mostrato il **sequence diagram** del caso d'uso dell'**arrivo di una notifica in real-time all'admin nel caso in cui un utente si sia prenotato per una determinata biblioteca** (supponiamo che l'admin sia collegato sulla dashboard delle stessa biblioteca su cui viene eseguita la prenotazione da parte dell'utente).



Le **sequenze di operazioni** eseguite per questo caso d'uso sono le seguenti:

1. L'**utente** effettua una prenotazione per una biblioteca tramite il frontend;
2. Il componente **ReservationBoxComponent** tramite il servizio **ReservationService** del frontend invia la richiesta POST HTTP al gateway;
3. La classe **Proxy** del modulo gateway intercetta la richiesta e la inoltra subito al backend;
4. La classe **ReservationRestServices** del modulo backend riceve la richiesta e chiama il metodo `add()` della classe **ReservationController**;
5. La classe **ReservationController** salva la prenotazione sul database utilizzando il relativo **ReservationDAO** e **ReservationMapper**. Nel caso di successo chiama il metodo statico `notifyAll()` della classe **RSocketNotifier**. Tale metodo riceve come parametro un oggetto JSON che contiene le informazioni sul tipo di notifica (aggiunta o cancellazione) e sulla prenotazione stessa.
6. La classe **RSocketNotifier** tramite il metodo `notifyAll()` invia un messaggio **Fire-and-Forget** a tutti gli admin in ascolto.

## LIBRARY SEAT RESERVATION

A questo punto il componente **LibraryComponent** all'interno della **dashboard dell'admin** viene aggiornato dinamicamente e viene mostrata a schermo una notifica.

The screenshot shows the Admin Dashboard with two main components. On the left is a calendar for September 2021, with days from 1 to 30 listed. The days 27, 28, 29, and 30 are highlighted in green circles, indicating they are reserved. On the right is a table titled "Prenotazioni per Lunedì 27 Settembre" (Reservations for Monday 27 September) showing four entries. The table has columns for Nome (Name), Email, and Elimina (Delete). The last entry is Utente 354 with email user354@email.com. Below the table are navigation arrows and a page size selector set to 5 items.

Dashboard admin prima della prenotazione da parte di un utente

The screenshot shows the Admin Dashboard after a new reservation has been made. The calendar now shows the day 27 highlighted with a red circle, while the other days (28, 29, 30) remain green. The reservation table now shows five entries. A notification at the bottom of the screen states: "Nuova prenotazione #90805 effettuata per il giorno 27-09-2021 alle 08:00".

Dashboard admin dopo la prenotazione da parte di un utente