

A.A. 2021 - 2022

LIBRARY SEAT RESERVATION

Docente

Prof. Enrico Vicario

Supervisori

- Dott. Boris Brizzi
- Ing. Jacopo Parri
- Ing. Samuele Sampietro



INDICE

Analisi dei requisiti	5
Requisiti funzionali	5
Gestione degli utenti	5
Gestione delle biblioteche	5
Gestione delle prenotazioni	5
Requisiti non funzionali	6
Requisiti di dominio	6
Casi d'uso	7
Casi d'uso dell'utente	7
Casi d'uso dell'Admin	9
Modello di dominio concettuale	10
Mockup	21
System Design	22
Obiettivo del sistema	22
Architettura software	23
Scomposizione in componenti	23
Breve Introduzione al REST	24
Architettura complessiva	25
Architettura nel dettaglio	25
Progettazione del Backend	27
Domain Model	28
Data Transfer Objects (DTO)	29
Data Access Objects (DAO)	30
Mapper	31
Controller	31
Autenticazione	31
Endpoint	32
Annotazioni JPA E Persistenza	32

Annotazioni CDI e JAX-RS	34
Altre annotazioni	35
Elenco delle API REST	35
Testing	36
Deploy con Wildfly su Docker	37
Progettazione del Gateway	38
Domain Model	38
Annotazioni JSR 356 e JAX-RS	39
Autorizzazione	39
Endpoint	39
Gateway: API REST	40
Progettazione del Frontend	40
Elementi fondamentali	41
Dipendenze	42
Diagramma di navigazione	42
Screenshot	43
Caso d'uso: Gestione della coda	48
Caso d'uso: Notifiche admin con RSocket	51
Conclusioni e sviluppi futuri	53
Approfondimento: TimescaleDB	55
Hypertables	55
Vantaggi	57
Approfondimento: Websocket vs RSocket	59
WebSocket	59
RSocket	60
Approfondimento: HyperSQL	63
Modalità di utilizzo	63
Vantaggi	65
Performance	65
Bibliografia	67

ANALISI DEI REQUISITI



ANALISI DEI REQUISITI

REQUISITI FUNZIONALI

Possiamo suddividere le funzionalità principali che il sistema deve fornire nelle seguenti categorie:

- Gestione degli utenti;
- Gestione delle biblioteche;
- Gestione delle prenotazioni.

Di seguito viene riportata un'analisi più dettagliata delle funzionalità.

GESTIONE DEGLI UTENTI

Il sistema deve permettere la gestione di risorse di tipo **Utente**:

- Il sistema deve permettere la **registrazione** di un Utente.
- Il sistema deve permettere il **login/logout** di un Utente.

GESTIONE DELLE BIBLIOTECHE

Il sistema deve permettere la gestione di risorse di tipo **Biblioteca**:

- Il sistema deve permettere l'**aggiunta** di una nuova Biblioteca;
- Il sistema deve permettere la **modifica** della capienza di una Biblioteca;
- Il sistema deve permettere la **cancellazione** di una Biblioteca;
- Il sistema deve permettere la **consultazione** delle informazioni di una Biblioteca.

GESTIONE DELLE PRENOTAZIONI

Il sistema deve permettere la gestione di risorse di tipo **Prenotazione**:

- Il sistema deve permettere la **creazione** di una nuova Prenotazione;

- Il sistema deve permettere la **cancellazione** di una Prenotazione.
- Il sistema deve permettere la **consultazione** delle prenotazioni effettuate.

REQUISITI NON FUNZIONALI

- Il sistema dovrà avere una **natura distribuita**, ovvero deve essere diviso in tre moduli: un **modulo frontend**, un **gateway** che intermedia le richieste per la gestione della coda e un **modulo backend**;
- Il sistema deve essere realizzato su una **architettura RESTful**;
- Il modulo frontend dovrà essere implementato utilizzando **Angular (TypeScript)** [1];
- Il modulo gateway e il modulo backend dovranno essere implementati utilizzando **Java Enterprise Edition (JEE)** [2];
- La logica del modulo **gateway** deve essere indipendente dal contesto applicativo.

REQUISITI DI DOMINIO

Di seguito sono riportati i requisiti di dominio:

- Una risorsa di tipo **Utente** deve prevedere i seguenti attributi:
 - un identificativo
 - una email
 - un nome
 - un cognome
 - una password
 - una lista di ruoli
- Una risorsa di tipo **Biblioteca** deve prevedere i seguenti attributi:
 - un identificativo
 - un nome
 - un indirizzo

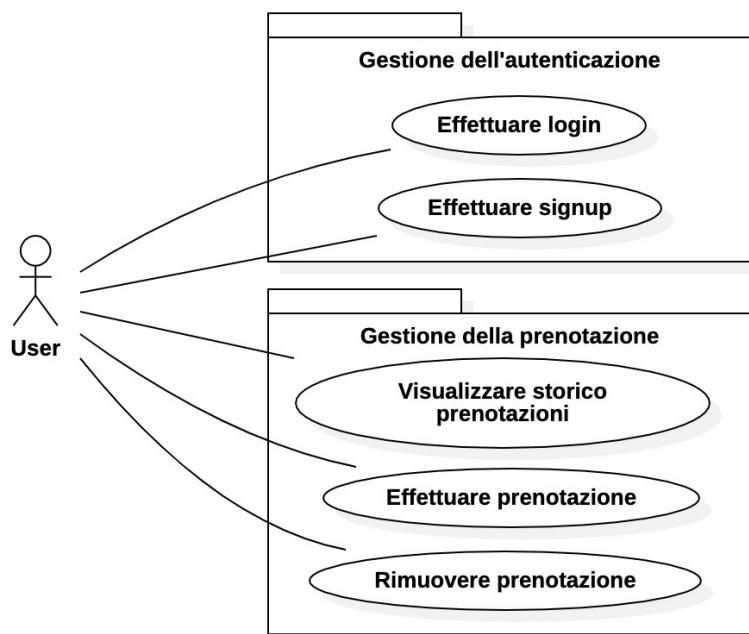
- una capienza
- Una risorsa di tipo **Prenotazione** deve prevedere i seguenti attributi:
 - un identificativo
 - un identificativo dell'utente che ha effettuato la prenotazione
 - un identificativo della biblioteca relativa alla prenotazione
 - una data e una fascia oraria

CASI D'USO

Il modello dei casi d'uso rappresenta un'astrazione di alto livello del modello di interazione e definisce il comportamento funzionale che il sistema offre a chi lo utilizza. Ci sono due elementi fondamentali:

- gli **attori**, cioè coloro che interagiscono in maniera diretta con il sistema (persone, dispositivi o altri sistemi);
- i **casi d'uso**, cioè un'astrazione delle interazioni che avvengono tra gli attori e il sistema (funzionalità che il sistema offre a chi lo utilizza).

CASI D'USO DELL'UTENTE



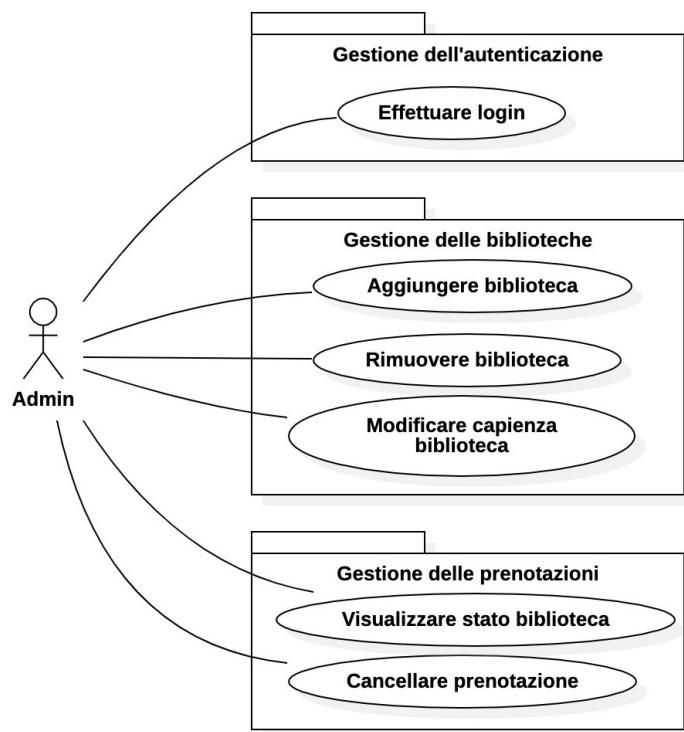
Esempio di **casi d'uso dell'utente** per quanto riguarda la **gestione dell'autenticazione**:

Caso d'uso: Effettuare il login
ID: UC01
Attori: Utente
Sequenza degli Eventi:
<ol style="list-style-type: none"> 1. L'utente accede alla pagina di login; 2. L'utente inserisce le proprie credenziali; 3. Il frontend prepara i dati da inviare al backend; 4. Il frontend invia una richiesta al backend; 5. Il backend verifica la correttezza delle credenziali inserite; 6. Il backend invia una risposta al frontend comunicando l'esito dell'operazione; 7. Il frontend utilizza le informazioni ricevute dal backend per autenticare l'utente.
Postcondizioni:
<ol style="list-style-type: none"> 1. L'utente è correttamente autenticato nell'applicazione.

Esempio di **casi d'uso dell'utente** per quanto riguarda la **gestione della prenotazione**:

Caso d'uso: Effettuare una prenotazione
ID: UC02
Attori: Utente
Precondizioni:
<ol style="list-style-type: none"> 1. L'utente è autenticato correttamente nell'applicazione.
Sequenza degli Eventi:
<ol style="list-style-type: none"> 1. L'utente seleziona una biblioteca; 2. Se il massimo numero di utenti è stato raggiunto, l'utente viene messo in coda; 3. Raggiunto il suo turno, l'utente può accedere alla pagina di prenotazione relativa alla biblioteca scelta; 4. L'utente può vedere la disponibilità della biblioteca per i vari giorni e fasce orarie; 5. L'utente può scegliere un giorno e una fascia oraria; 6. L'utente esegue la prenotazione per la data selezionata; 7. Il frontend prepara i dati da inviare al backend; 8. Il frontend invia una richiesta al backend; 9. Il backend salva la prenotazione dell'utente all'interno del database e invia una risposta al frontend comunicando l'esito; 10. Il frontend utilizza l'informazioni ricevute per comunicare visivamente l'esito all'utente.
Postcondizioni:
<ol style="list-style-type: none"> 1. L'utente ha effettuato una prenotazione per la biblioteca scelta.

CASI D'USO DELL'ADMIN



Esempio di **casi d'uso dell'admin** per quanto riguarda la **gestione dell'autenticazione**:

Caso d'uso: Effettuare il login
ID: UC03
Attori: Admin
Sequenza degli Eventi:
<ol style="list-style-type: none"> 1. L'admin accede alla pagina di login; 2. L'admin inserisce le proprie credenziali; 3. Il frontend prepara i dati da inviare al backend; 4. Il frontend invia una richiesta al backend; 5. Il backend verifica la correttezza delle credenziali inserite; 6. Il backend invia una risposta al frontend comunicando l'esito dell'operazione; 7. Il frontend utilizza le informazioni ricevute dal backend per autenticare l'admin.
Postcondizioni:
<ol style="list-style-type: none"> 1. L'admin è correttamente autenticato nell'applicazione.

Esempio di **casi d'uso dell'admin** per quanto riguarda la **gestione delle biblioteche**:

Caso d'uso: Aggiungere una biblioteca
ID: UC04
Attori: Admin
Sequenza degli Eventi:
<ol style="list-style-type: none"> 1. L'admin accede alla pagina Home; 2. L'admin può cliccare sul bottone "Aggiungi nuova biblioteca"; 3. L'admin viene reindirizzato alla pagina per compilare i dati relativi alla biblioteca da aggiungere; 4. L'admin inserisce le informazioni necessarie; 5. Il frontend prepara i dati da inviare al backend; 6. Il frontend invia una richiesta al backend; 7. Il backend salva la biblioteca all'interno del database e invia una risposta al frontend comunicando l'esito; 8. Il frontend utilizza l'informazioni ricevute per comunicare visivamente l'esito all'admin.
Postcondizioni:
<ol style="list-style-type: none"> 1. La biblioteca è correttamente aggiunta all'elenco delle biblioteche.

MODELLO DI DOMINIO CONCETTUALE

Basandosi sulle informazioni fornite dai casi d'uso, delineati precedentemente, possiamo definire un modello di dominio:

Nome	Proprietà	Descrizione
Library	<ul style="list-style-type: none"> • un identificativo • un nome • un indirizzo • una capacità 	Oggetto che comprende tutte le informazioni relative ad una biblioteca.
Reservation	<ul style="list-style-type: none"> • un identificativo • un identificativo dell'utente • un identificativo della biblioteca • una data e una fascia oraria 	Oggetto che comprende tutte le informazioni relative ad una prenotazione.
User	<ul style="list-style-type: none"> • un identificativo • una email • un nome • un cognome • una password • una lista di ruoli 	Oggetto che comprende tutte le informazioni relative ad un utente.

MOCKUP





Logo

Area riservata Login Signup

Prenota il tuo posto in biblioteca

Cerca biblioteca...

Biblioteca 1

Via prova 1 - Firenze

Biblioteca 2

Via prova 2 - Bagno a Ripoli

Biblioteca 3

Via prova 3 - Scandicci

Logo

Footer

*Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.*



A Web Page

A set of five white navigation icons on a grey background: a double arrow pointing left, a single arrow pointing up, a double arrow pointing right, a single arrow pointing down, and a house icon.

<http://www.seatreservation.com/admin-home>

A circular logo containing the word "Logo".

Pinco Pallino Admin ▾

Prenota il tuo posto in biblioteca

Cerca biblioteca...

Aggiungi Nuova Biblioteca

Biblioteca 1

Via prova 1 - Firenze

Biblioteca 2

Via prova 2 - Bagno a Ripoli

Biblioteca 3

Via prova 2 - Scandicci

Logo

Footer

Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.

A small circular icon with a speech mark and a checkmark inside.

A Web Page

http://www.seatreservation.com/user-login

Logo

Area riservata Login Signup

Email

Password

Login

Non sei registrato, registrati qui

Logo

Footer

adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.

A Web Page

http://www.seatreservation.com/signup

Logo

Area riservata Login Signup

Nome

Cognome

Email

Password

Signup

Logo

Footer

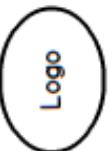
*Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.*

The diagram illustrates a wireframe of a web application for library seat reservations. The interface is divided into several sections:

- Header:** A grey header bar contains a back arrow, a search icon (magnifying glass), and a logo.
- Left Sidebar:** A vertical grey sidebar on the left features a 'Logo' placeholder, a 'Le mie prenotazioni' section with a large rectangular frame, and a 'Footer' section at the bottom.
- Main Content Area:** The central area contains three identical card components, each representing a reservation:
 - Card 1:** Shows 'Biblioteca 1' and 'Data Orario' with a trash bin icon.
 - Card 2:** Shows 'Biblioteca 1' and 'Data Orario' with a trash bin icon.
 - Card 3:** Shows 'Biblioteca 1' and 'Data Orario' with a trash bin icon.
- Bottom Footer:** A dark footer bar at the bottom right contains a 'Logo' placeholder and a small gear icon.

A URL 'http://www.seatreservation.com/my-reservations' is also present in the sidebar.

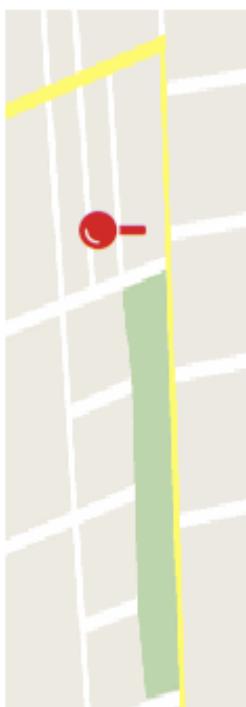
A Web Page <http://www.seatreservation.com/prenotazione>



Logo



Area riservata Login Signup



Biblioteca 1
Via prova 1 - Firenze

S	M	T	W	T	F	S
27	28	29	30	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	31
1	2	3	4	5	6	7

8:00 - 13:00 12 / 50

13:00 - 19:00 50 / 50

Prenota

Logo

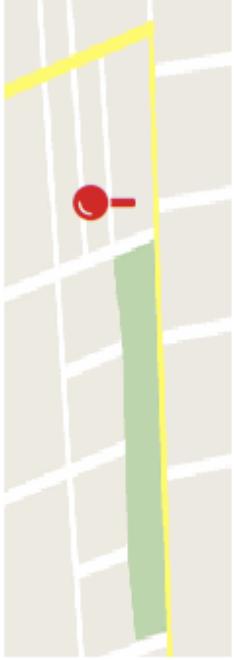
Footer

Lorem ipsum dolor sit amet, consectetur adipisciing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

A Web Page <http://www.seatreservation.com/admin/prenotazione>



Pinco Pallino Admin ▾



Biblioteca 1
Via prova 1 - Firenze

S	M	T	W	T	F	S
27	28	29	30	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	31
1	2	3	4	5	6	7

Capienza 60 posti - + ↕

8:00 - 13:00 13:00 - 19:00

Occupazione 16/60

Pippo Pluto	Trash icon
Paperino Quo	Trash icon
Samuele Ceccherini	Trash icon
Topolino Qua	Trash icon

[Elimina Biblioteca](#)

Logo

Footer

Logo

Lore ipsum dolor sit amet, consectetur adipisciing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

A Web Page

http://www.seatreservation.com/add-library

Logo

Pinco Pallino Admin ▾

Nome biblioteca

Indirizzo

Capienza

Aggiungi

Logo

Footer

■

Lore ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.



Area riservata Login Signup

Sei in coda...

Ci sono

60

persone prima di te

potrai accedere al servizio in: 2 min



Footer

Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.



SYSTEM DESIGN DOCUMENT



SYSTEM DESIGN

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.

L'utilizzo di una coda si rende necessario quando le risorse sono limitate o quando il traffico web aumenta improvvisamente. Nel nostro caso erano possibili due scenari **funzionali**:

- Una coda per ogni biblioteca;
- Una coda generale per l'accesso al sistema di prenotazione.

È stato scelto di intraprendere la seconda possibilità, in quanto nel nostro caso l'intento era quello di prevenire sovraccarichi al sistema dovuti a un grande numero di richieste contemporanee relative a biblioteche diverse.

Gli scenari **architetturali** possibili erano i seguenti:

- Modulo frontend come parte attiva del funzionamento della coda (es. informato dal server reagisce con un redirect);
- Modulo frontend completamente indipendente dal modulo gateway (es. il gateway ha un suo frontend specifico per la pagina della coda).

È stata scelta la prima opzione, che fornisce maggiore libertà di scelta e consente con più facilità l'aggiunta di funzionalità specifiche alla coda (es. timer per completare il caso d'uso).

Ispirandosi al portale di vaccinazione regionale, il nostro sistema deve essere strutturato nel seguente modo:

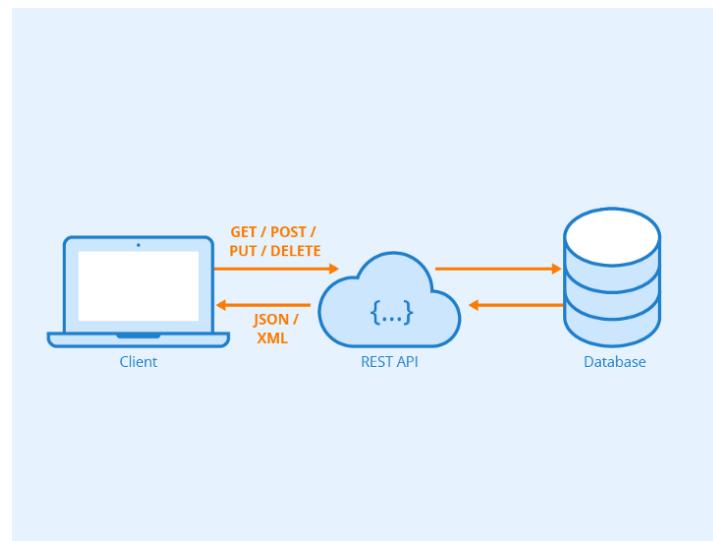
- le richieste che vengono ricevute dal server devono essere trattate in modo “accodato” con il risultato di non sovraccaricare il server;
- la presenza di un componente distribuito in stile gateway o proxy che intermedia le richieste;
- un numero massimo di utenti che realizzano il caso d'uso;

- una dimensione massima della lunghezza della coda, che se superata restituisce un messaggio di errore;
 - una coda “esterna” con ticket ordinati e feedback circa la lunghezza della stessa e tempi di attesa per accedere al servizio.
- **Due frontend**, 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.
- Utilizzo delle seguenti tecnologie:
 - **Backend e Gateway**: Java Enterprise Edition (JEE) [2];
 - **Frontend**: Angular (TypeScript) [1] con Angular Material [3] per il design;
 - **WebSocket e RSocket** [4] (vedi approfondimento) per la comunicazione real-time tra i moduli;
 - **JWT** [5] per la gestione dell’autenticazione e dell’autorizzazione;
 - **Docker** [6] per la divisione in container (**Wildfly** [7] e database PostgreSQL [8]);
 - **Timescale** [9] per gestione di query temporali in maniera efficiente (vedi approfondimento).

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**: frontend, backend e gateway.



Esempio di architettura RESTful

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

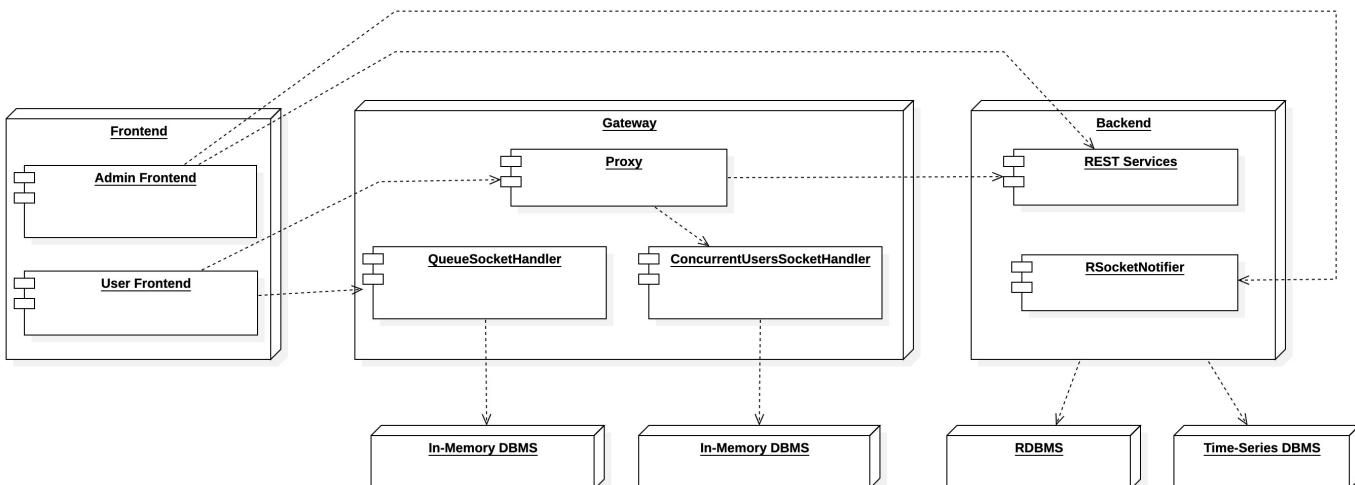
BREVE INTRODUZIONE AL REST

REST è l'acronimo di **REpresentational State Transfer** e, in generale, indica uno stile architettonurale 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**.

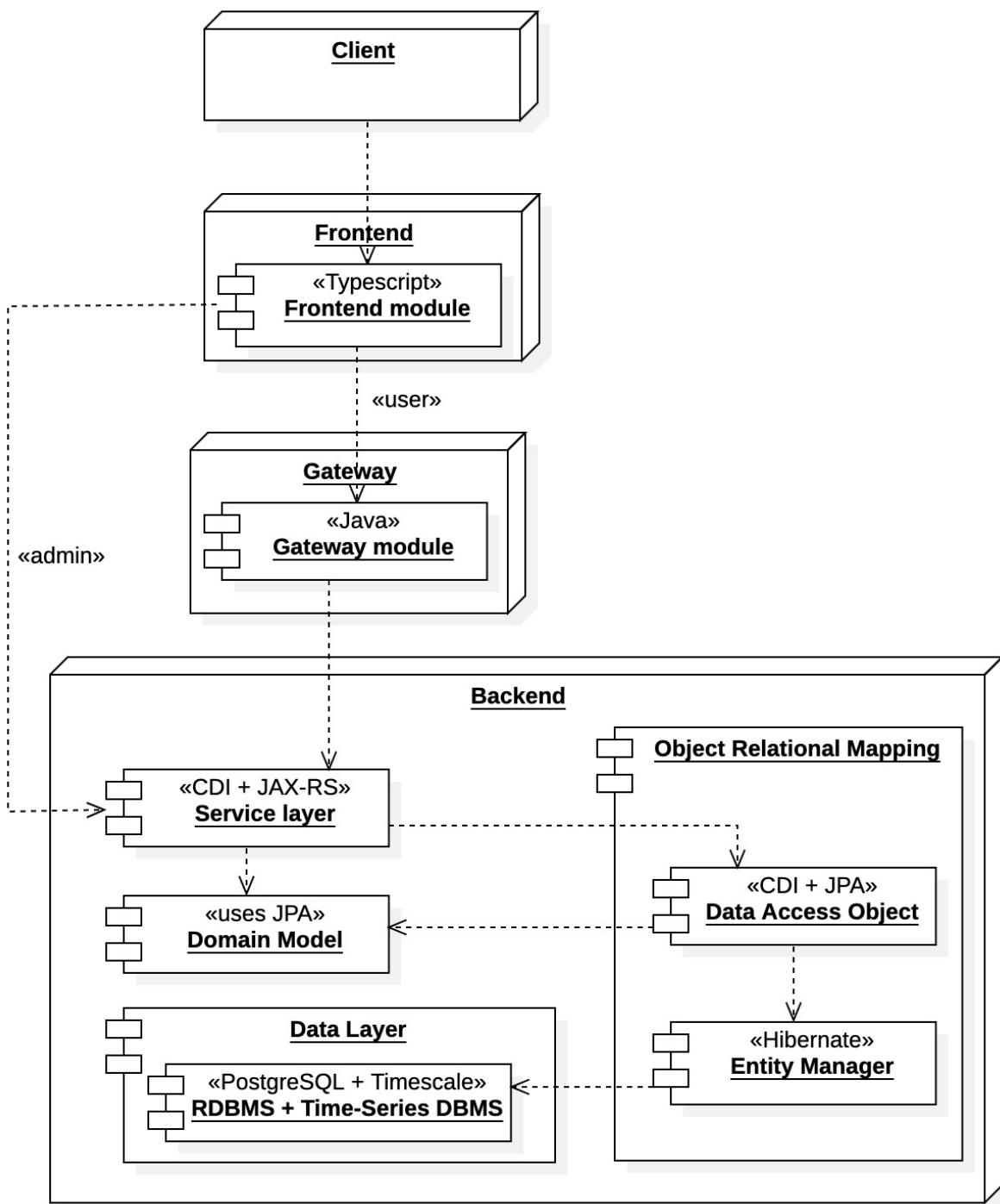
ARCHITETTURA NEL DETTAGLIO

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

Come si vede dall'immagine i moduli principali sono tre: Frontend, Gateway e Backend. Di particolare interesse è l'architettura del Backend che risulta suddiviso in vari layer:

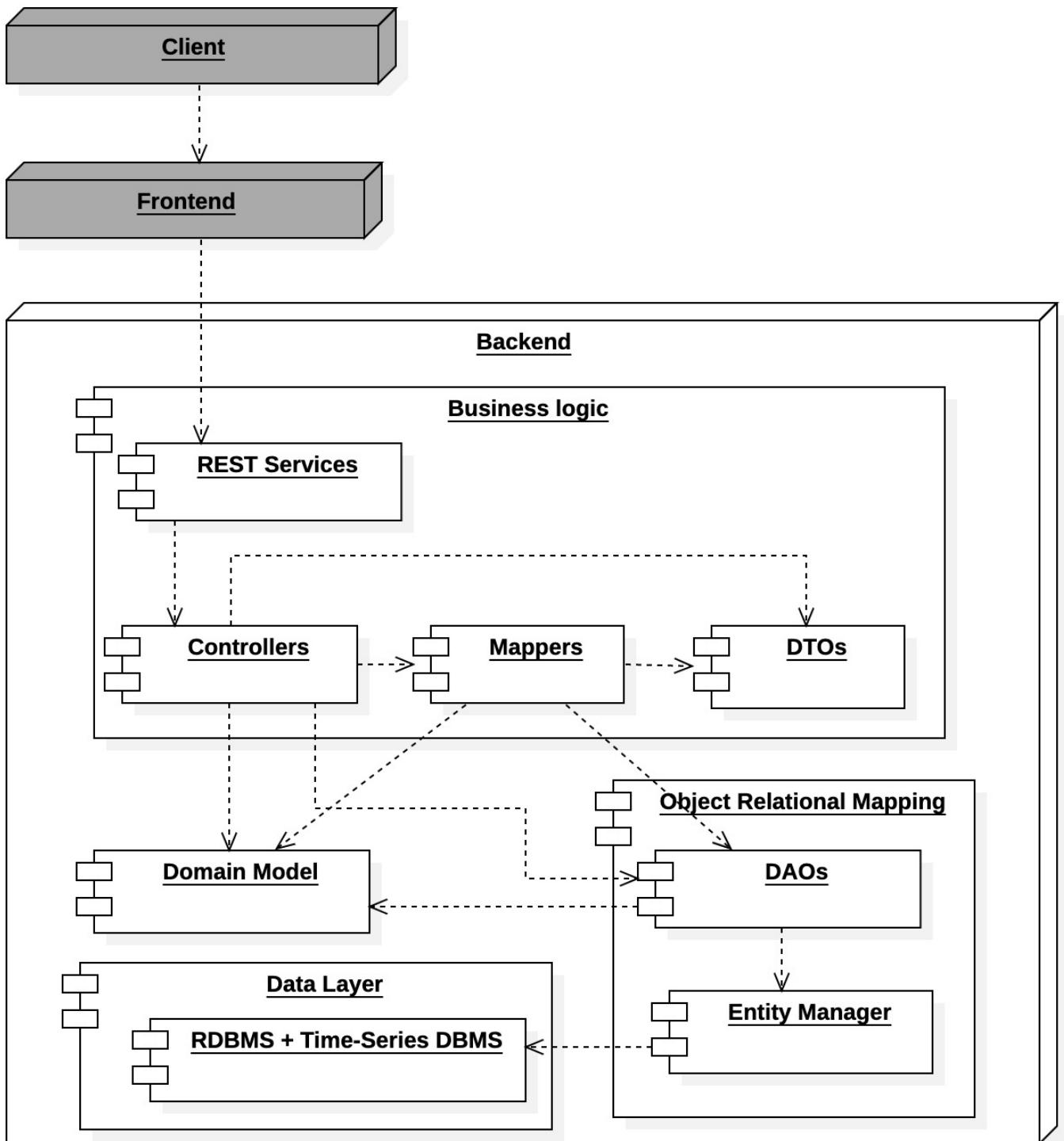
- **Service Layer**: è la *business logic* del progetto, composta da controller, mapper, DTO e servizi REST;

- **Domain Model:** descrive le entità che fanno parte del progetto e le loro relazioni;
- **Data Layer:** definisce l'accesso al database;
- **Object Relational Mapping (ORM):** strato che gestisce l'adattamento tra la struttura relazionale del database e la struttura della logica di dominio orientata agli oggetti. Formato da:
 - **DAO:** disaccoppia la business logic dal problema della persistenza, ed espone un insieme di metodi utili per coprire le principali operazioni CRUD;
 - **Entity Manager:** interagisce con il DBMS.



PROGETTAZIONE DEL BACKEND

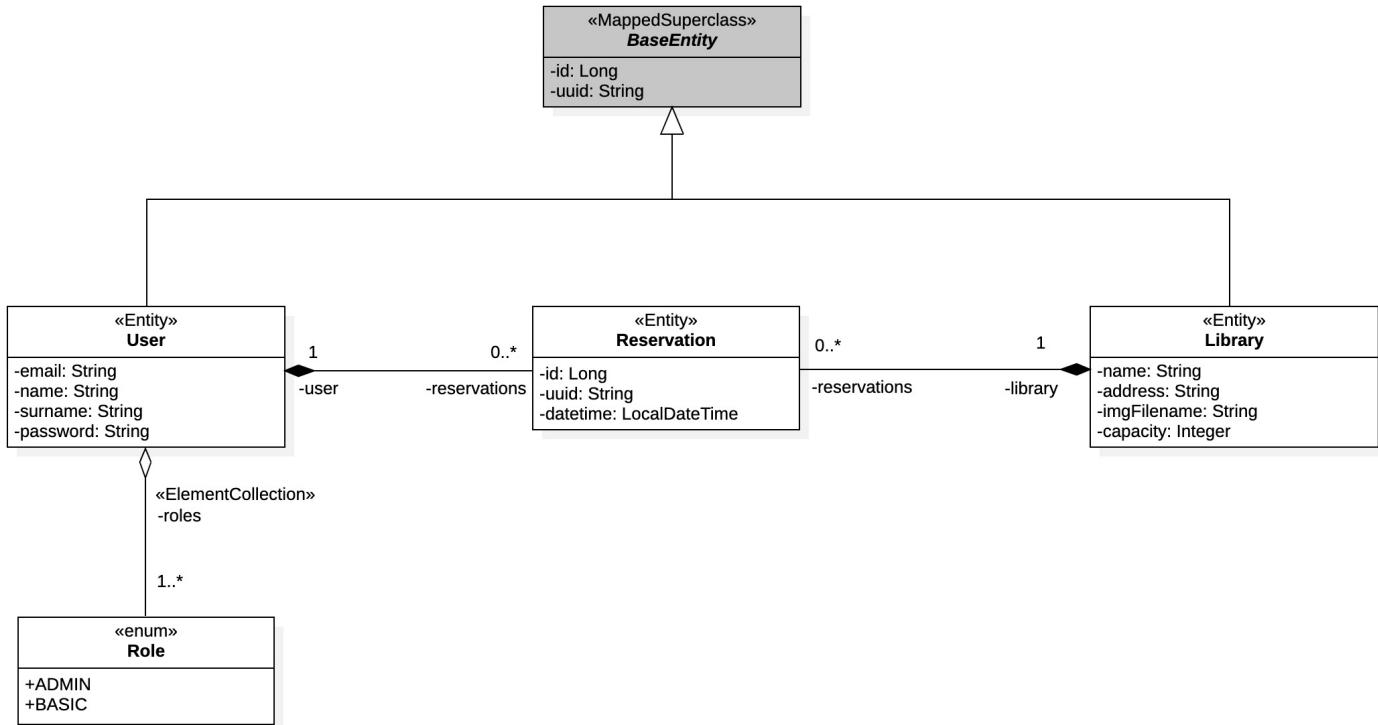
In questa sezione del documento viene descritta la progettazione del **Backend** basato su un'architettura **RESTful** e sul paradigma Client-Server. Il Backend è quella parte del sistema che risiede sul Server e che espone servizi tramite specifiche **REST API**.



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*. Si è reso necessario l'utilizzo di due classi di

utenza in quanto il sistema richiede una figura (**ADMIN**) per la gestione delle biblioteche e di un utente (**BASIC**) che possa usufruire delle fuzionalità della piattaforma.

La classe **Reservation** non estende la classe astratta *BaseEntity*, come le classi precedenti, a causa dell'integrazione con **TimescaleDB** [9]. Questa è una scelta di design che verrà spiegata in seguito.

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

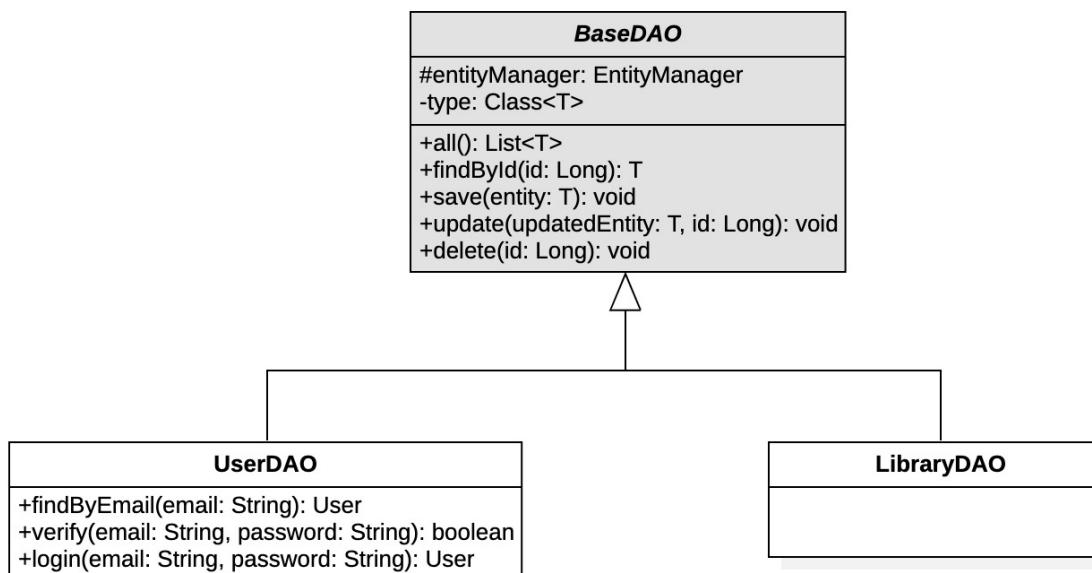
ReservationDailyAggregateDTO
-date: String
-countMorning: Integer
-countAfternoon: Integer

- **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 il conteggio 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:



ReservationDAO
#entityManger: EntityManager
+all(): List<ReservationDto>
+findById(id: Long): ReservationDto
+findByUserId(id: Long): List<ReservationDto>
+findByLibraryId(id: Long): List<ReservationDto>
+findByLibraryIdAndDate(libraryId: Long, year: int, month: int, day: int): List<ReservationDto>
+dailyAggregateByLibraryIdAndMonth(libraryId: Long, year: int, month: int): List<ReservationsDailyAggregateDto>
+save(entity: Reservation): void
+delete(id: Long): void
+enableTimescalePostgresExtensionIfNeeded(): void
+setupHypertable(): void

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

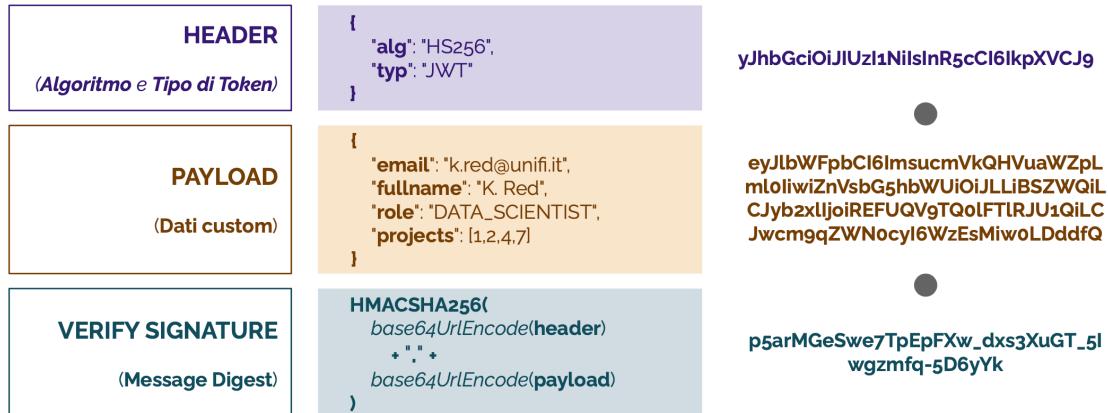
AUTENTICAZIONE

Per l’autenticazione è stato utilizzato lo standard **JSON Web Token** (JWT) [5] del protocollo **OAuth 2.0** [10] tramite la libreria **Java JWT** (JJWT) [11].

In particolare, il backend si occupa di generare un **token** (stringa alfanumerica che viene “firmata” e “verificata” con un algoritmo di cifratura, ad esempio **HMAC256**).

L’autenticazione viene eseguita nei seguenti passi:

- login: viene inviata dal frontend una richiesta HTTP(S) contenente *email* e *password*
- il backend controlla le credenziali e risponde inviando un *token* al client valido per sei ore (adottando un algoritmo di generazione del digest basato su una chiave segreta), che lo salva
- il client può quindi inoltrare tale token in ogni successiva richiesta HTTP(S)
- il backend verifica che questo sia corretto (verifica il digest)
- logout: viene rimosso il token lato client



Esempio di token JWT per l'autenticazione

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`.

ANNOTAZIONI CDI E JAX-RS

Di seguito sono riportate tutte le **annotazioni CDI** [12] e **JAX-RS** [13] 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

ALTRÉ ANNOTAZIONI

- @Secured per annotare le API che richiedono autenticazione
- @GatewayAuthorizationRequired per annotare le API che richiedono un token speciale dal Gateway (ad es. per verificare che l'utente abbia effettivamente atteso in coda)

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 [9])
- 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).

Vengono anche fatti ulteriori controlli di sicurezza tramite il `SecurityContext JAX-RS` [13], per assicurare ad esempio che un utente non possa effettuare o cancellare prenotazioni per altri utenti.

TESTING

Per il **testing** abbiamo utilizzato il framework **JUnit5** [14] (per lo *unit testing*) e **Mockito** [15] (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** [16] - vedi approfondimento) 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 [6] di **Wildfly** [7] versione 24.0.0, modificata per il supporto a database **PostgreSQL** [8] (su cui si basa anche TimescaleDB [9]).

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 [22] 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 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.

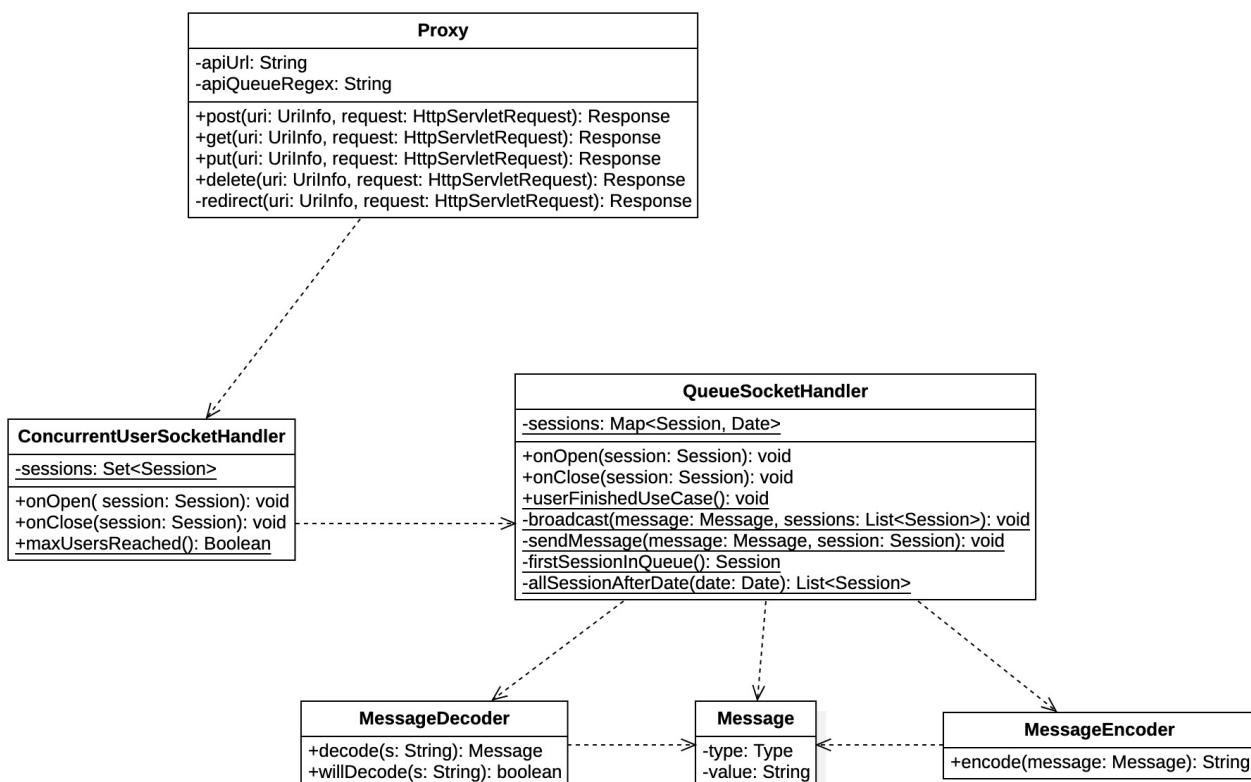
Le operazioni di attivazione di Timescale e creazione della hypertable sono delegate al DAO delle prenotazioni.

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 JSR 356 E JAX-RS

Di seguito sono riportate tutte le **annotazioni JSR 356 (Java API for WebSocket)** [17] e **JAX-RS** [13] 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

AUTORIZZAZIONE

Per assicurare che l'utente BASIC non possa bypassare il Gateway inviando richieste non autorizzate direttamente al Backend, è stato utilizzato un altro token JWT. In particolare:

- Il Frontend fa una richiesta al Gateway per poter accedere al servizio di prenotazione;
- Il Gateway riceve la richiesta e genera un token valido per due minuti (adottando un algoritmo di generazione del digest basato su una chiave segreta condivisa con il Backend) ed inoltra la richiesta al Backend insieme al token creato;
- Il Backend riceve la richiesta e il token, e verifica che questo sia corretto (verifica il digest con la stessa chiave segreta con cui era stato cifrato nel Gateway);
- Se il token risulta corretto, l'utente può accedere al servizio richiesto. In caso contrario, viene generato un errore HTTP 401 Unauthorized.

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

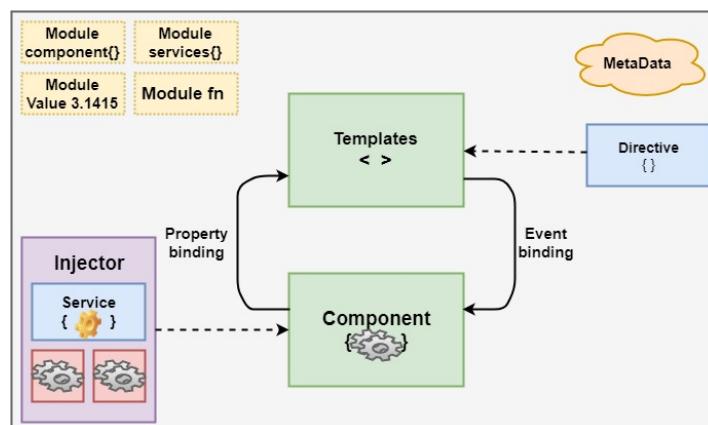
Il **Frontend** è stato sviluppato tramite il framework [Angular \[1\]](#) con l'aiuto dei componenti forniti dalla libreria [Angular Material \[3\]](#).



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** per lo stile;
- un **file TS** che ne definisce il comportamento.

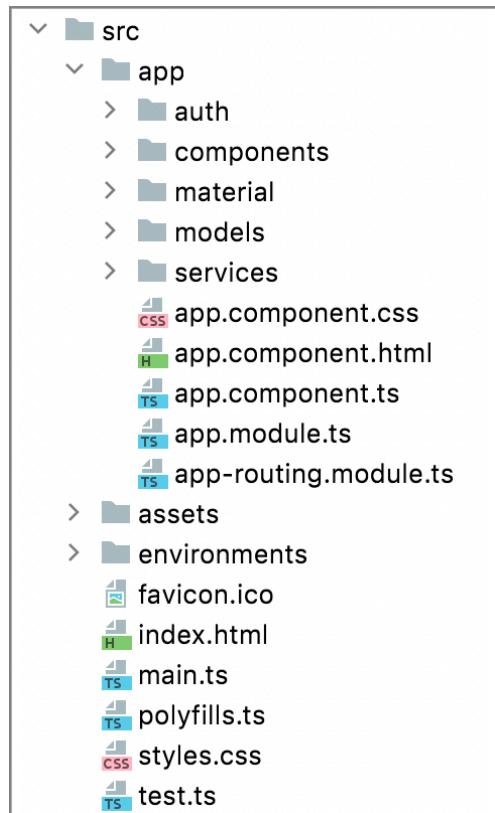
Il linguaggio di base è [TypeScript \[18\]](#). L'applicazione è stata realizzata come una Single Page Application (SPA) orientata ai servizi e con paradigma Model-View-Controller.



ELEMENTI FONDAMENTALI

Gli elementi fondamentali sono:

- **Modello**: stato del sistema, legato in maniera bidirezionale ai servizi REST del backend;
- **Componente**: “controllori” del pattern MVC, 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 [3]**: design system per i componenti UI;
- **Leaflet [19]**: libreria JavaScript per mappe interattive;
- **angularx-qrcode [20]**: libreria per la generazione di QR code;
- **RSocket [4]**: protocollo a livello applicativo per stream reattivi;
- **RxJS [21]**: libreria JavaScript che mette a disposizione il tipo `Observable` e la funzione `pipe()`, che permette di compiere più 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

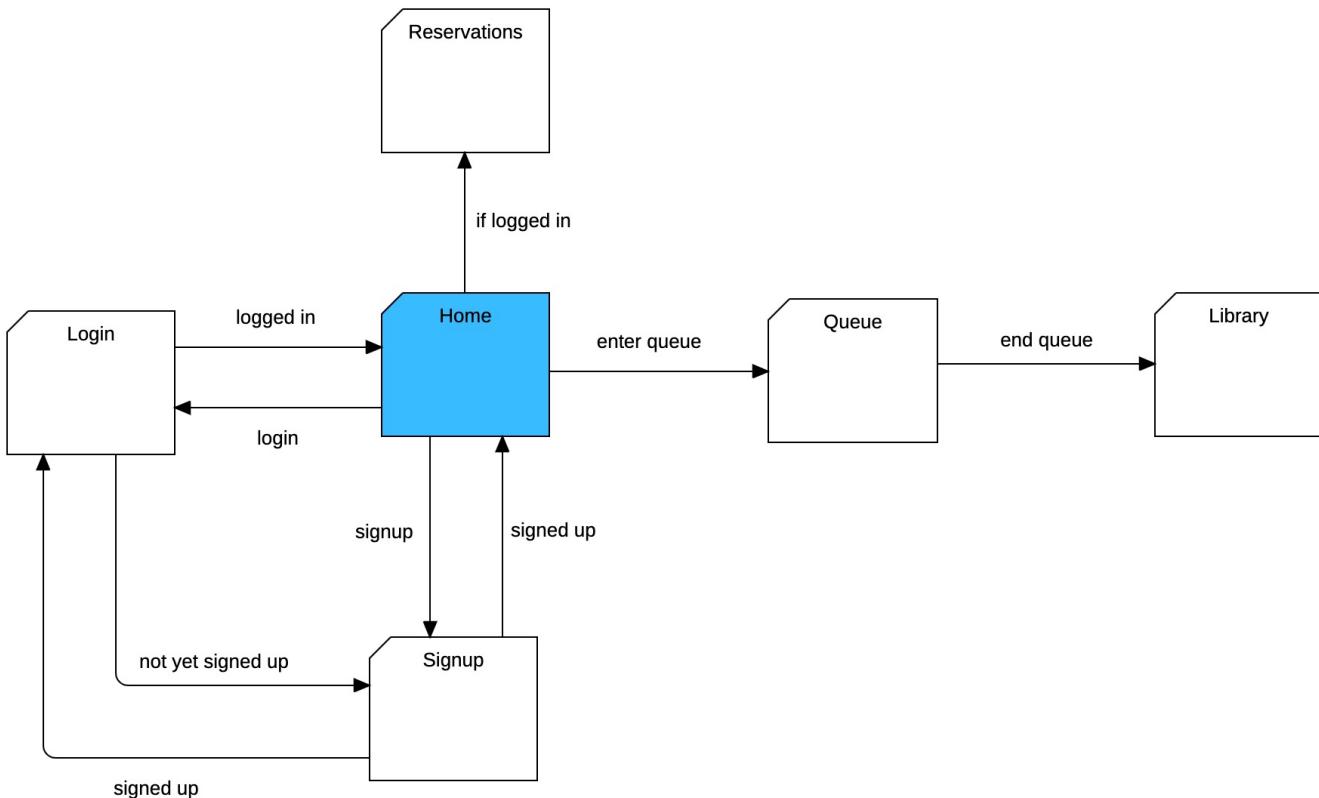
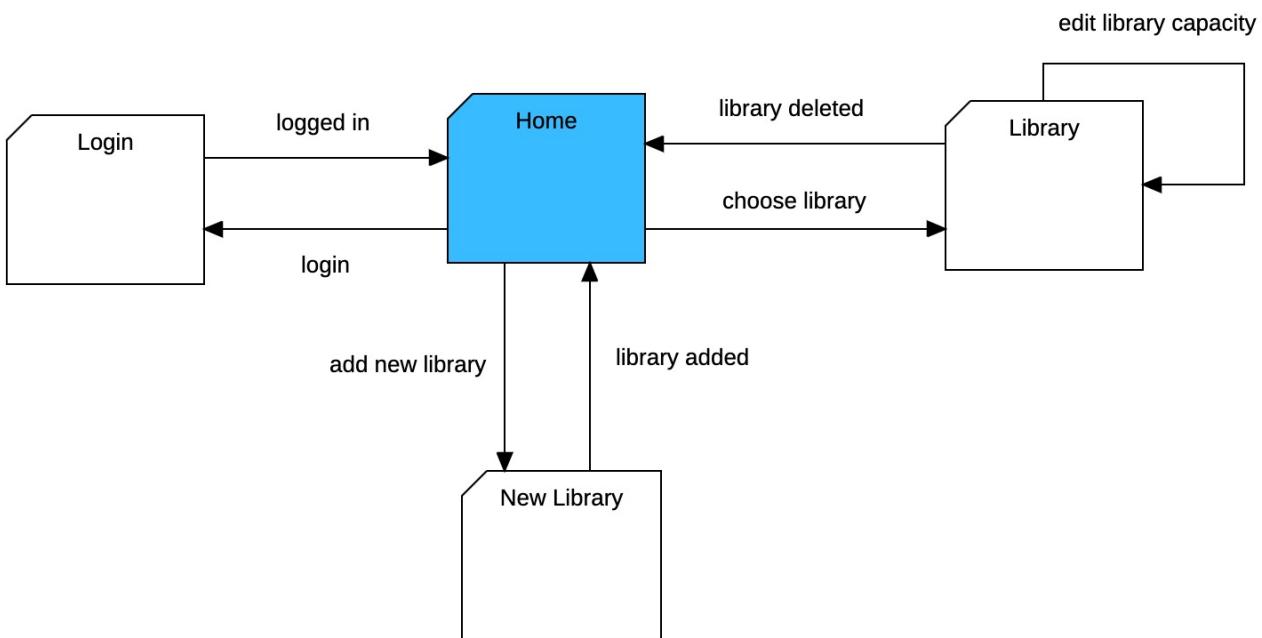
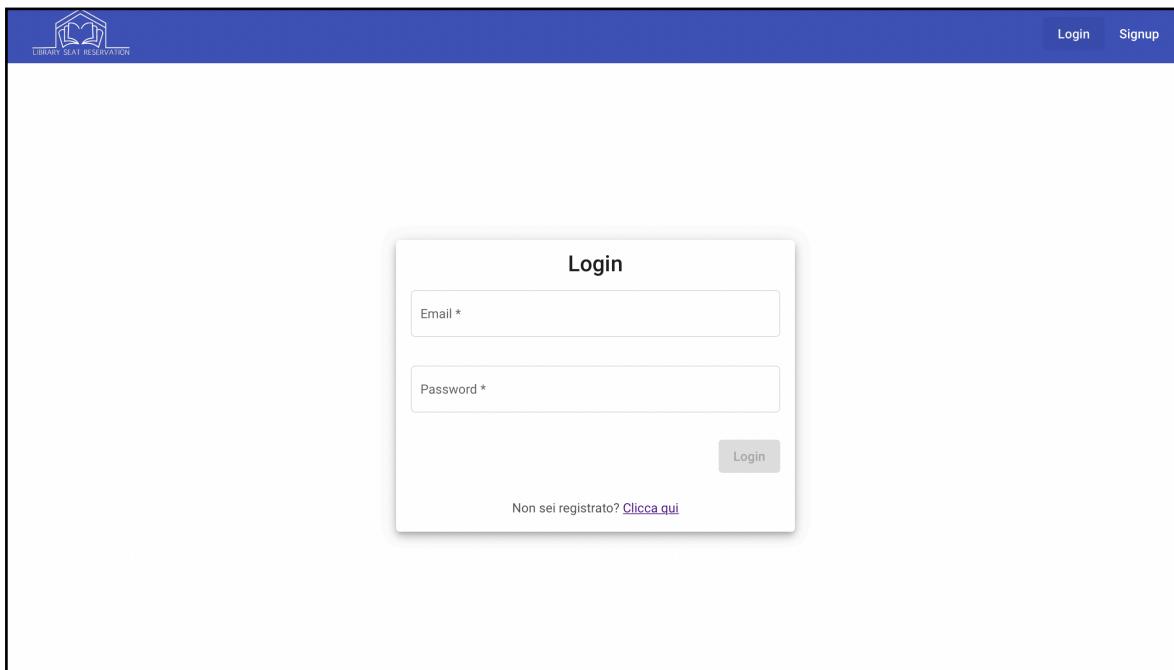


Diagramma di Navigazione per l'Admin



SCREENSHOT

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



Pagina di Login

The screenshot shows the 'Signup' form within a white modal box. At the top center is the word 'Signup'. Below it are four input fields with labels: 'Nome *', 'Cognome *', 'Email *', and 'Password *'. To the right of the 'Password' field is a 'Signup' button. At the bottom left of the modal is a link 'Sei già registrato? [Clicca qui](#)'.

Nome *

Cognome *

Email *

Password *

Signup

Sei già registrato? [Clicca qui](#)

Pagina di Signup

The screenshot shows the main home page with a blue header bar. In the center, there is a search bar with the placeholder 'Cerca biblioteca...' and a magnifying glass icon. Below the search bar, there are four library cards, each featuring a thumbnail image, the library name, its address, and its seating capacity.

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

LIBRARY SEAT RESERVATION

The screenshot shows the library's name, address, and a map of the surrounding area.

Biblioteca Villa Bandini
Via del Paradiso, 5, Firenze

A map of the neighborhood around Villa Bandini is displayed, showing streets like Viale Europa, Via di Ripoli, and various landmarks.

Pagina di una biblioteca (1/2)

The screenshot shows the seat reservation calendar for Friday, September 24, 2021.

Tempo rimasto: 1:32

SET 2021						
D	L	M	M	G	V	S
				1	2	3
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)

The screenshot shows a message indicating the user is in a queue.

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.

Utente 8934

© Copyright LibrarySeatReservation 2021

Pagina della coda

LIBRARY SEAT RESERVATION

Utente 43 ▾

Le mie prenotazioni

Future	Passate			
Ora	Data	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

Utente 43 ▾

Le mie prenotazioni

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

Lunedì 27 Settembre - ore 8:00
Biblioteca delle Oblate



5 | < < > > |

QR code relativo a una prenotazione

LIBRARY SEAT RESERVATION

OTT 2021

D	L	M	M	G	V	S
31	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

Prenotazioni per Mercoledì 20 Ottobre

Mattina (8.00-13.00) **Pomeriggio (13.00-19.00)**

Nome	Email	Elimina
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	

Occupazione: **34 / 50**

1 - 5 di 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
Utente Admin (Admin) ▾

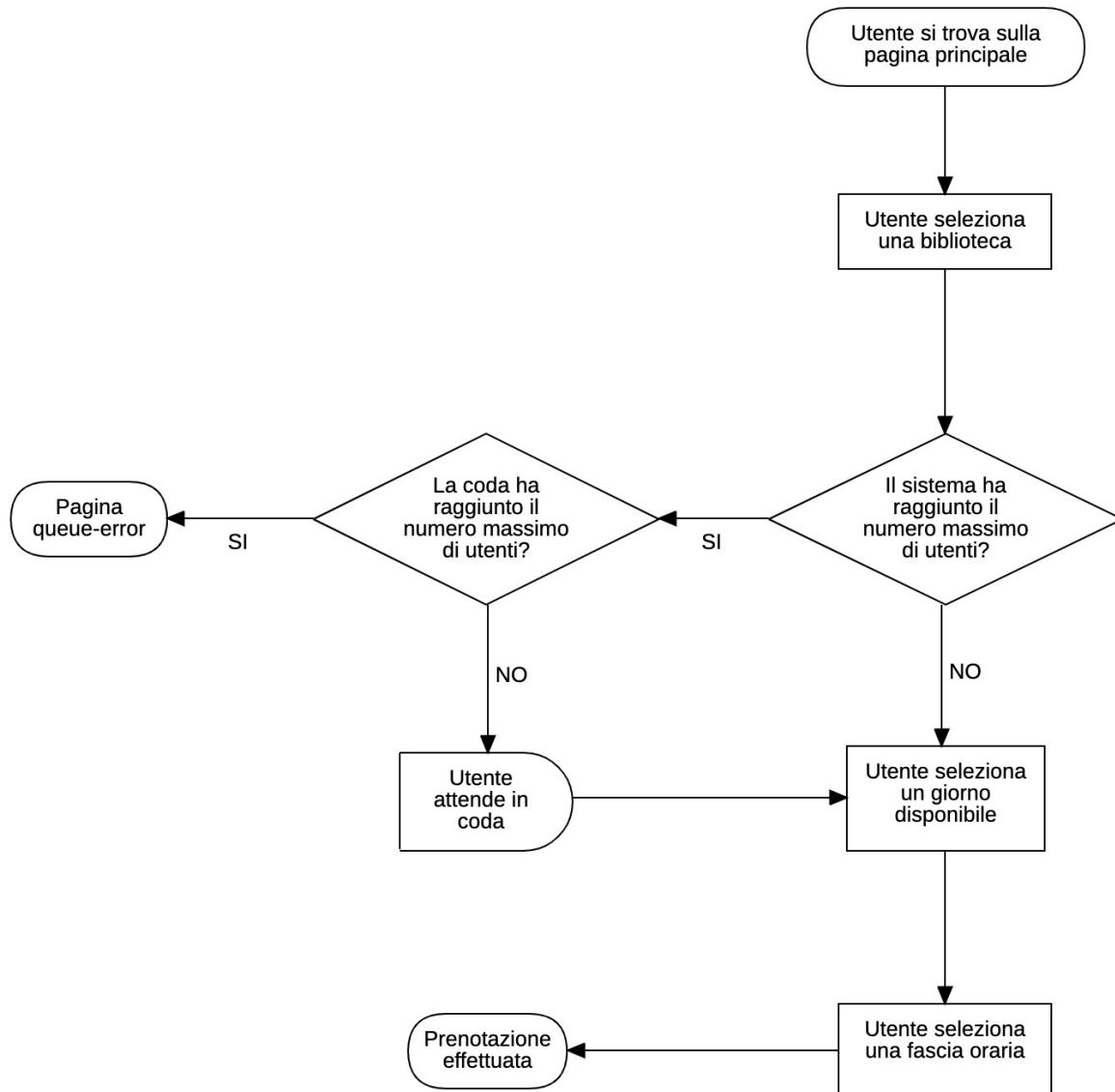
Aggiungi Biblioteca

Aggiungi

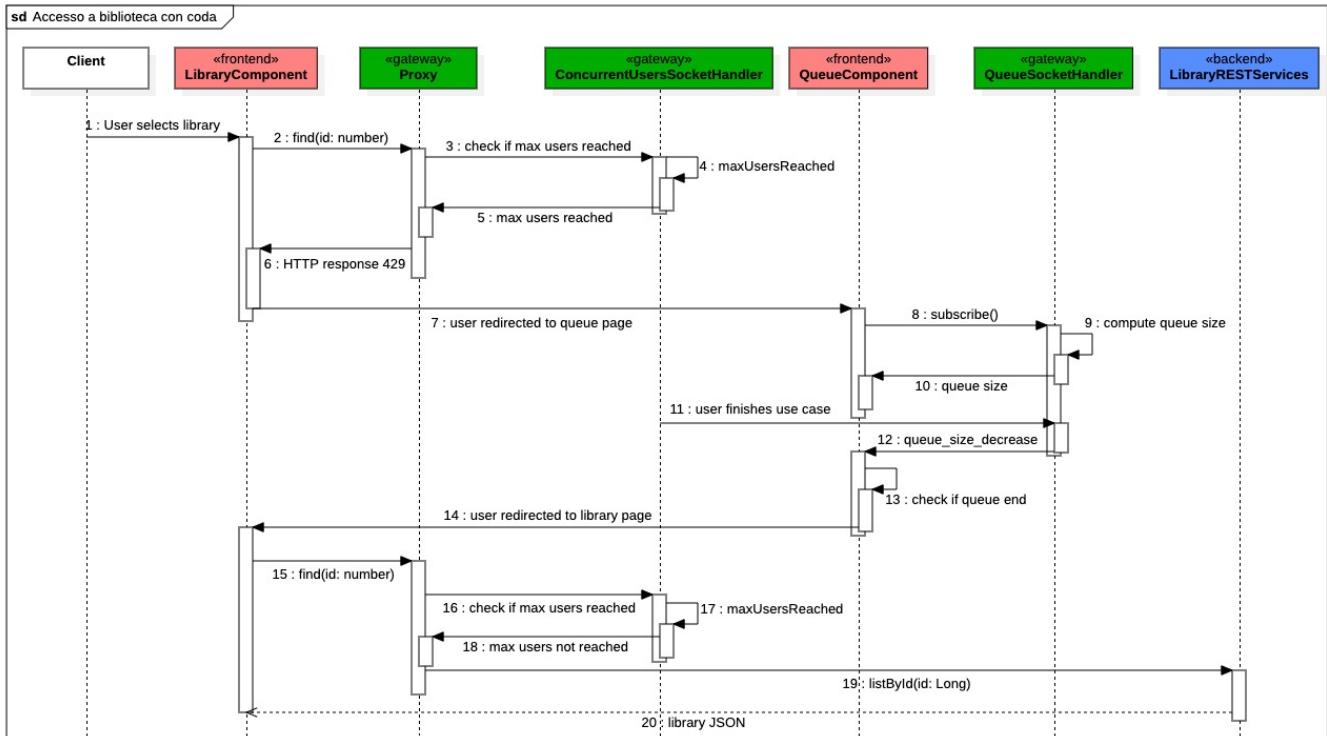
Pagina admin per l'aggiunta di una biblioteca

CASO D'USO: GESTIONE DELLA CODA

Di seguito viene mostrato il **diagramma di flusso** 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).



Di seguito viene mostrato anche il **sequence diagram** dello stesso caso d'uso (per semplicità si omette il controllo sulla dimensione massima della coda):



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

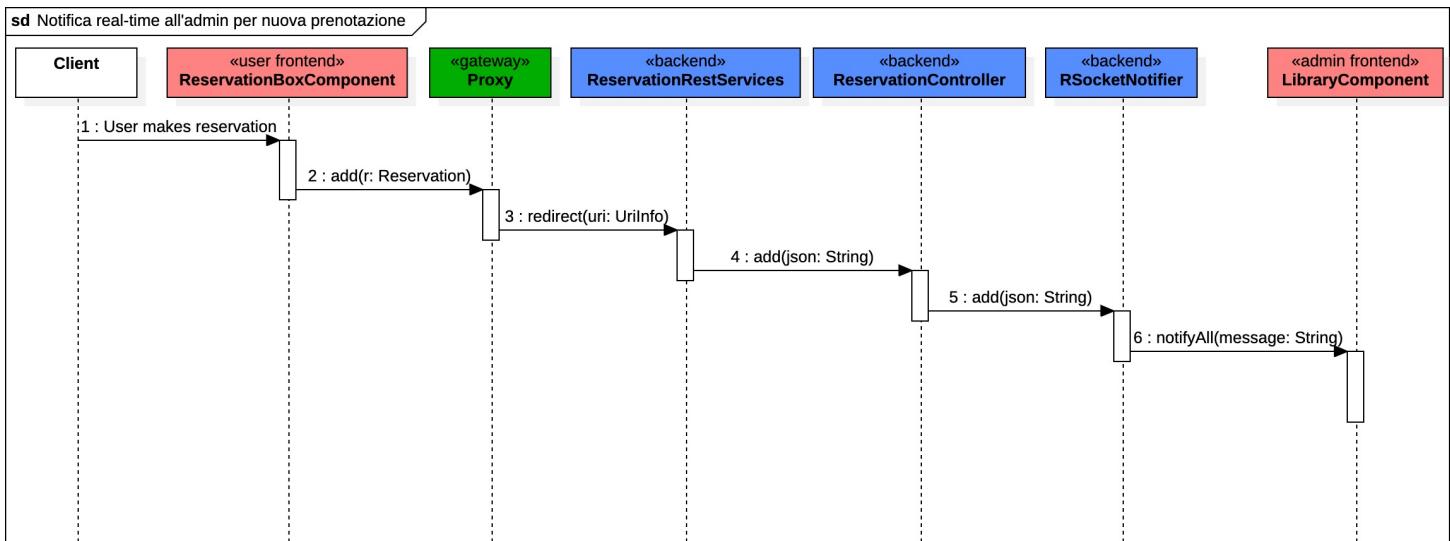
1. L'**utente** seleziona una biblioteca;
2. Il componente **LibraryComponent** tramite il servizio **LibraryService** del frontend richiede al gateway le informazioni sulla biblioteca scelta tramite REST API;
3. La classe **Proxy** del modulo gateway intercetta la richiesta e controlla se è stato raggiunto 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 un 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 (per il caso d'uso scelto si suppone di sì, essendo il primo in coda);
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 un 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 escluso dal 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**, passando i dati della prenotazione;
5. La classe **ReservationController** salva la prenotazione sul database utilizzando il relativo **ReservationDAO** e **ReservationMapper**. In caso di successo, invoca anche 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**, che tiene traccia di tutti i client in ascolto sulla porta 7878 con protocollo RSocket, con il metodo `notifyAll()` invia un messaggio di tipo **Fire-and-Forget** a tutti gli admin connessi alla dashboard.

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

The screenshot shows two main sections. On the left is a monthly calendar for September 2021. The days from 27 to 30 are highlighted in green, indicating they are reserved. On the right is a table titled "Prenotazioni per Lunedì 27 Settembre" (Bookings for Monday 27 September). It shows two time slots: "Mattina (8.00-13.00)" and "Pomeriggio (13.00-19.00)". The "Occupazione" status is shown as "49 / 50". The table lists five users with their names, emails, and a red trash icon for deletion. Navigation buttons and a page size selector are at the bottom.

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			

Prenotazioni per Lunedì 27 Settembre		
Mattina (8.00-13.00)		Pomeriggio (13.00-19.00)
Occupazione: 49 / 50		
Nome	Email	Elimina
Utente 3905	user3905@email.com	
Utente 3801	user3801@email.com	
Utente 80	user80@email.com	
Utente 354	user354@email.com	

46 – 49 of 49 | < < > >| Items per page: 5

Dashboard admin prima della prenotazione da parte di un utente

This screenshot is similar to the previous one but shows a change in the calendar. The day 27 is now highlighted in red, indicating it is fully booked. The rest of the days (28-30) remain green. The booking table shows the same data as before, with the "Occupazione" status now at "50 / 50". A new message at the bottom of the screen reads: "Nuova prenotazione #90805 effettuata per il giorno 27-09-2021 alle 08:00".

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			

Prenotazioni per Lunedì 27 Settembre		
Mattina (8.00-13.00)		Pomeriggio (13.00-19.00)
Occupazione: 50 / 50		
Nome	Email	Elimina
Utente 3905	user3905@email.com	
Utente 3801	user3801@email.com	
Utente 80	user80@email.com	
Utente 354	user354@email.com	
Utente 18	user18@email.com	

46 – 50 of 50 | < < > >| Items per page: 5

Nuova prenotazione #90805 effettuata per il giorno 27-09-2021 alle 08:00

Dashboard admin subito dopo la prenotazione da parte di un utente, con notifica a schermo e aggiornamento real-time sulla disponibilità della biblioteca

CONCLUSIONI E SVILUPPI FUTURI

Il sistema realizzato implementa tutte le specifiche indicate nell'analisi dei requisiti, fornendo all'utente una piattaforma per la gestione delle prenotazioni di posti all'interno delle aule studio delle biblioteche di Firenze.

Il sistema implementa inoltre anche un meccanismo di gestione della coda in modo da evitare il sovraccarico sul server, con **parametri configurabili** da parte dello sviluppatore in base alle necessità:

- L'URL relativo alle API REST del backend (API_URL);
- L'espressione regolare che indica le chiamate API "critiche" da proteggere con la coda (API_QUEUE_REGEX);
- Il numero massimo di utenti che il sistema può ospitare contemporaneamente (MAX_CONCURRENT_USERS);
- La lunghezza massima della coda (MAX_QUEUE_SIZE).

Il modulo Gateway potrebbe quindi diventare un modulo **indipendente e universale**, adattabile ai vari casi d'uso, tenendo però conto che il modulo Frontend è parte integrante del sistema e dovrà essere adattato anch'esso alle varie situazioni.

Data la configurabilità del sistema, un possibile sviluppo futuro potrebbe essere quello di estenderne l'utilizzo ad una regione più ampia (es. Toscana).

Altri sviluppi futuri potrebbero prevedere l'aggiunta di **più fasce orarie** (e non solo mattina/pomeriggio). Sarebbe inoltre interessante anche poter offrire una prenotazione specifica per una determinata **aula** di una biblioteca (es. Aula A della biblioteca *Villa Bandini*).

TIMESTEALDB



APPROFONDIMENTO: TIMESCALEDB

TimescaleDB [9] è un database relazionale open-source **Full SQL** per i dati time-series, che promette di “scalare” con performance che prima erano raggiungibili solo dai database NoSQL.



TimescaleDB è sostanzialmente una estensione di **PostgreSQL** [11] in grado di offrire un insieme di operazioni relative ai *time-series* data, cioè ad un set di dati annotato *temporalmente*. In quanto derivato da PostgreSQL, TimescaleDB risulta supportato da molti linguaggi di programmazione, tra cui **Java** e **Python**.

HYPERTABLES

Le **hypertables** sono ciò che consente a TimescaleDB di lavorare in modo così efficace con i dati delle serie temporali, rendendo l'**archiviazione** e l'**interrogazione di dati** operazioni estremamente veloci su scala peta-byte.

TimescaleDB infatti partiziona automaticamente i dati delle *time-series* in **blocchi** o **sotto-tabelle**, sia in base al tempo che allo spazio.

Una **hypertable** è quindi un livello di astrazione che consente di interrogare e accedere ai dati da **tutti i blocchi**, come se fossero in un'unica tabella: i comandi inviati all'hypertable vengono infatti applicati a tutti i blocchi che appartengono a quella hypertable.

La creazione di una hypertable è un processo suddiviso in due fasi:

1. Inizialmente si utilizza un'istruzione SQL **CREATE TABLE** per creare una tabella relazionale;

VANTAGGI

Uno dei maggiori vantaggi di TimescaleDB è il fatto che **supporta il linguaggio SQL** in modo nativo, riducendo molto la curva di apprendimento, pur offrendo una **serie di funzioni** che non si trovano nei database relazionali tradizionali.

Queste funzioni hanno lo scopo di fornire due vantaggi chiave:

- maggiore facilità d'uso per l'analisi delle *time-series*;
- prestazioni migliorate.

Una di queste funzioni critiche per le serie temporali è `time_bucket()`.

`time_bucket()` viene utilizzato per **aggregare periodi di tempo** di dimensioni arbitrarie (es. 5 minuti, 1 giorno). Essenzialmente, `time_bucket()` è una versione più potente della funzione standard `date_trunc()` di PostgreSQL, in quanto consente anche la scelta di intervalli di tempo arbitrari.

Oltre a consentire query di serie temporali più flessibili, `time_bucket()` consente anche di scriverle in modo più semplice a livello sintattico:

```
SELECT time_bucket('5 minutes', time) AS five_min, avg(cpu)
  FROM metrics
 GROUP BY five_min
 ORDER BY five_min DESC;
```

Esempio di query SQL con `time_bucket` per estrarre la media temporale (colonna `time`) della colonna `cpu` ad intervalli di 5 minuti.

`time_bucket()` risulta molto utile per creare dashboard o visualizzazioni di dati *time-series*, e in generale per trasformare le osservazioni “raw” in aggregati di livello superiore con intervalli di tempo prefissati.

WEBSOCKET VS RSOCKET



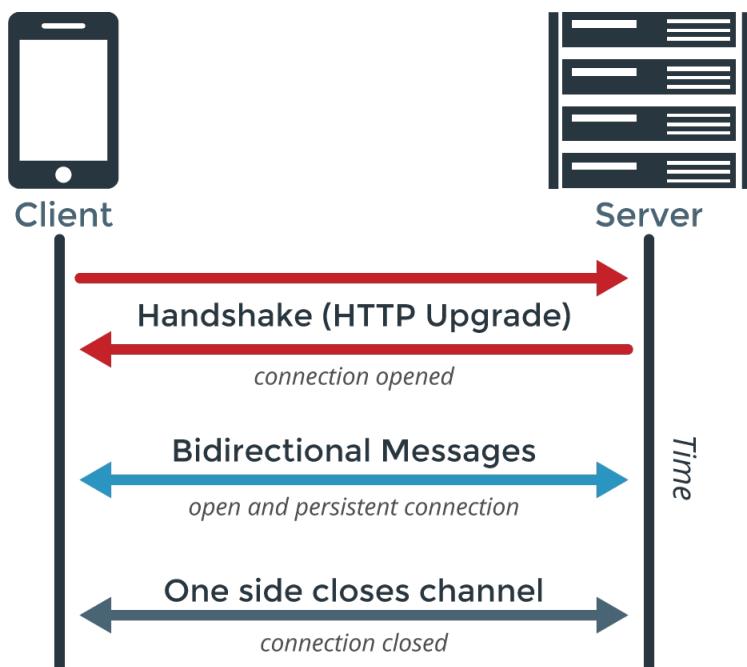
APPROFONDIMENTO: WEBSOCKET VS RSOCKET

Sia **WebSocket** che **RSocket** [4] sono protocolli di comunicazione bidirezionale, multiplex e duplex, in grado di inviare dati da un server a un client riutilizzando lo stesso canale di connessione. Il vantaggio di questi protocolli, rispetto ad HTTP classico, consiste nella possibilità da parte del server di inviare contenuti ad un client (es. browser) senza prima dover essere sollecitato dal client stesso (o viceversa).

I due protocolli funzionano però a livelli diversi, e le differenze vengono elencate di seguito.

WEBSOCKET

WebSocket è un protocollo di comunicazione di basso livello basato su Transmission Control Protocol (**TCP**). In pratica definisce come un flusso di byte viene trasformato in frame, ma senza offrire nessuna semantica a livello di applicazione: spetta allo sviluppatore creare un protocollo applicativo per interagire con il WebSocket.



Per stabilire una connessione WebSocket, il client invia una richiesta di handshake ed il server invia una risposta di avvenuta connessione.

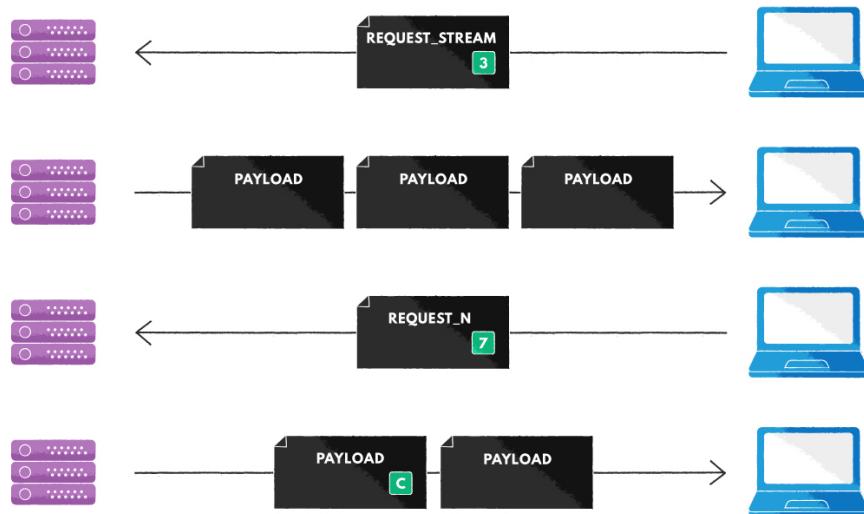
Una volta stabilita la connessione, il client ed il server possono inviare dati tramite il WebSocket in entrambe le direzioni.

RSocket

RSocket è un protocollo a livello applicativo, totalmente asincrono, che implementa la [specifica Reactive Streams](#), con supporto a caratteristiche avanzate come *framing*, ripresa automatica della sessione (*session resumption*) e *backpressure* a livello di applicazione. Lo scopo di questo protocollo è quello di ridurre la latenza nelle comunicazioni e i costi dell'infrastruttura.



RSocket è inoltre **agnostico rispetto al livello di trasporto utilizzato**: può infatti essere eseguito su diversi trasporti di flusso di byte come TCP, WebSocket, HTTP/2 o Aeron, e permette quindi di cambiare il livello di trasporto sottostante in base, ad esempio, alla capacità del dispositivo ricevente o alle esigenze di prestazione — a differenza di WebSocket che invece è vincolato all'utilizzo del livello di trasporto TCP.

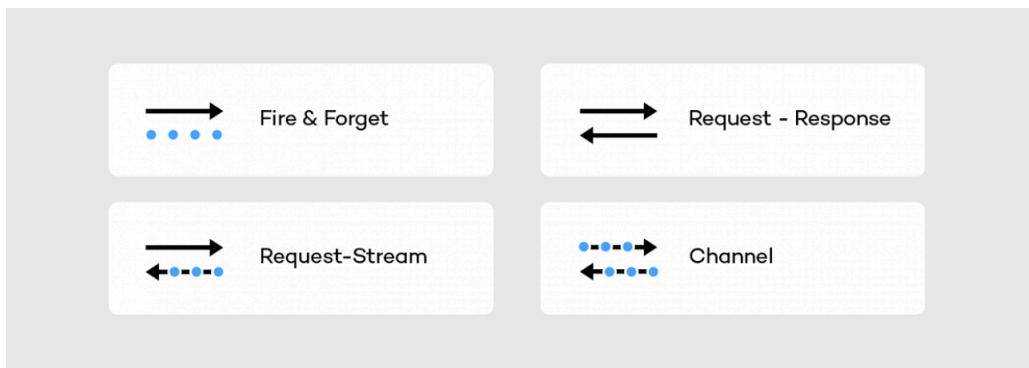


La gestione dei flussi di dati, in particolare i dati "live" il cui volume non è predeterminato, richiede un'attenzione particolare in un sistema asincrono. Il problema più rilevante riguarda il consumo di risorse, che deve essere controllato in modo tale da evitare situazioni in cui se un server invia degli eventi più velocemente di quanto il client sia in grado di elaborarli, alla fine il client esaurirà le risorse e sarà costretto a chiudere la connessione.

RSocket ha un supporto integrato per il controllo del flusso, che aiuta a evitare queste situazioni riducendo il carico, chiamato **backpressure** non bloccante: il destinatario può specificare la quantità di dati che desidera consumare, entro un intervallo di tempo predefinito, e non ne riceverà di più fino a che non notifica al mittente che è pronto ad elaborarne altri.

RSocket supporta le seguenti interazioni:

- **Fire-and-forget:** un'ottimizzazione del classico metodo *request/response* HTTP, da utilizzare quando non è necessario attendere e associare una risposta a ogni relativa richiesta;
- **Request-Response:** simili ad HTTP, sono flussi di una sola risposta ottimizzati in modo che il consumatore attenda un messaggio di risposta dal server, ma senza mai bloccarsi in modo sincrono;
- **Request-Stream:** il client invia un singolo frame al server e riceve l'intero stream di dati, in maniera asincrona (*push*);
- **Channel:** canale bidirezionale, con un flusso di messaggi in entrambe le direzioni.



HYPERSQL



APPROFONDIMENTO: HYPERSQL

HSQL Database Engine (HSQLDB o HyperSQL) è un *database management system* relazionale (**RDBMS**) scritto completamente in Java.

È leggero, veloce, supporta il multithreading ed ha la capacità di lavorare sia in modalità *in-memory* che *in-process*, mettendo a disposizione delle applicazioni diverse modalità di utilizzo (server, web server e servlet). Include anche un programma da riga di comando (**SqlTool**) per interagire con il database in linguaggio SQL.

Può essere installato sia scaricando il **HSQLDB JDBC Driver** (file `.jar` da aggiungere nel *classpath* del progetto), oppure utilizzando **Maven** [22] (specificando l'id `org.hsqldb.hsqldb`).

Si tratta di uno strumento adatto per supportare quelle applicazioni che non richiedono tutte le caratteristiche offerte da altri DBMS (es. MySQL), o per fare pratica con framework come **JPA**.

MODALITÀ DI UTILIZZO

- **Modalità Server:** il database viene eseguito in una *Java virtual machine* (JVM) e ascolta le connessioni da programmi che stanno sullo stesso computer o su altri computer all'interno della rete. Esistono tre modalità server, basate sul protocollo utilizzato per le comunicazioni tra client e server:

- **Server:** viene utilizzato un protocollo di comunicazione proprietario, generalmente in ascolto sulla porta TCP 9001 — questa è la modalità più veloce e consigliata per avviare il database. Il formato dell'URL di connessione è del tipo:

```
jdbc:hsqldb:hsqldb://db
```

- **Web Server:** questa modalità viene utilizzata quando l'accesso al computer che ospita il server del database è limitato al protocollo HTTP (ad esempio in caso di restrizioni imposte da un firewall) — in questo caso viene avviato un server Web speciale che consente ai client di connettersi tramite HTTP. Il formato dell'URL di connessione è del tipo:

```
jdbc:hsqldb:http://db
```

- **Servlet:** come Web Server, ma questa modalità viene utilizzata quando è un *servlet engine* separato, come **Tomcat** o **Resin**, a fornire l'accesso al database — questa modalità però può servire solo un singolo database alla volta.
- **Modalità in-process (standalone):** questa modalità utilizza il *filesystem* per eseguire il database come parte dell'applicazione Java, in particolare nella stessa *Java Virtual Machine* (JVM). Questo ha il risultato di favorire accessi più rapidi, poiché i dati non vengono convertiti e inviati in rete, ma lo svantaggio principale è che non è possibile connettersi al database dall'esterno dell'applicazione: non è quindi possibile gestire il contenuto del database mentre l'applicazione è in esecuzione. Il formato dell'URL di connessione è del tipo:

`jdbc:hsqldb:file:db`

- **Modalità in-memory:** non prevede persistenza, ma il database viene allocato per intero nella RAM. Questa modalità risulta utile nel caso di elaborazione interna dei dati dell'applicazione oppure in fase di testing. In questo caso, il formato dell'URL di connessione è del tipo:

`jdbc:hsqldb:mem:db`

In tutti i casi, HSQL mette a disposizione un utente con privilegi di amministratore con le seguenti credenziali:

Username: sa

Password: stringa vuota

Ad esempio, una connessione ad un database HSQL in-memory può essere effettuata in Java aggiungendo queste proprietà al file `persistence.xml`:

```
<property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
<property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:standalone"/>
<property name="javax.persistence.jdbc.user" value="sa"/>
<property name="javax.persistence.jdbc.password" value=""/>
```

VANTAGGI

- Supporto molto esteso per la sintassi standard di SQL (2016), inclusa la maggior parte delle funzionalità opzionali;
- Tabelle in memoria per operazioni più veloci;
- Tabelle basate su disco per set di dati di grandi dimensioni;
- Tabelle con supporto a dati esterni (es. file CSV) possono essere utilizzate come tabelle SQL.

PERFORMANCE

HyperSQL ha più opzioni di distribuzione e persistenza che ne influenzano le prestazioni:

- Il **tipo di tabella** (**MEMORY**, **CACHED** o **TEXT**) indica come vengono archiviati i dati di ciascuna riga della tabella e come vi accede il database;
- La modalità **in-process** o **server** indica in che modo l'applicazione accede ai dati del database;
- Il **modello di transazione** indica come e quando diverse sessioni (connessioni) rimangono in attesa l'una con l'altra.

In particolare le prestazioni dei diversi **tipi di tabella** sono:

- Le tabelle **MEMORY** offrono le massime prestazioni. Tutti i dati sono in memoria e ogni campo di ogni riga è un oggetto di memoria che può essere letto dal database senza alcuna conversione. Quando i dati vengono aggiornati, viene scritto su disco solo un record di log, con un overhead molto basso;
- Le tabelle **CACHED** hanno prestazioni inferiori rispetto alle tabelle MEMORY. I dati per questo tipo di tabella provengono da una cache che contiene un sottoinsieme di tutte le righe in tutte le tabelle CACHED. La riduzione delle prestazioni è dovuta al fatto che la dimensione della cache delle righe è in genere inferiore al totale delle righe di tutte le tabelle CACHED: per questo le righe vengono spesso eliminate dalla cache e altre righe vengono lette dal disco e convertite in oggetti di memoria.
- Le tabelle **TEXT** hanno somiglianze sia con le tabelle CACHED che con le tabelle MEMORY. Gli indici vengono mantenuti in memoria, mentre i dati vengono

mantenuti su disco e memorizzati nella cache come le tabelle CACHED. Poiché i dati vengono archiviati come valori separati da virgola (file CSV), la lettura e la scrittura dei dati richiede più tempo rispetto alla stessa operazione in formato binario.

Le prestazioni della modalità **in-process**, rispetto alla modalità **server**, sono:

- L'accesso *in-process* avviene nello stesso spazio di memoria dell'applicazione: non c'è alcuna conversione dei dati o *overhead* dovuto al trasferimento;
- La modalità *server*, al contrario, ha uno spazio di memoria diverso rispetto all'applicazione. I dati vengono convertiti in un flusso di byte, trasferiti sulla rete e quindi riconvertiti in oggetti: questo introduce una latenza, da sommare all'*overhead* dovuto all'elaborazione aggiuntiva necessaria per la conversione;
- L'accesso in modalità *server* può essere velocizzato utilizzando le **stored procedures** SQL, che consentono di incapsulare un'intera transazione in un'unica istruzione SQL.

HyperSQL supporta due **modelli di transazione** per il controllo della concorrenza: **multiversion concurrency control (MVCC)** e **two-phase locking (2PL)**. Le differenze in termini di prestazioni sono:

- HyperSQL ha il supporto completo al *multithreading*, per cui se la maggioranza delle operazioni sono di *lettura*, le prestazioni sono molto elevate in tutti i modelli di transazione;
- Se è presente una quantità significativa di operazioni di aggiornamento, le prestazioni del modello di blocco 2PL possono essere ricondotte al caso in cui ci sia un singolo thread: ci sono infatti dei *lock* sia in lettura che in scrittura;
- Sotto le stesse condizioni, MVCC offre prestazioni notevolmente maggiori rispetto al 2PL, poiché non vengono utilizzati *lock* di lettura, mentre i *lock* di scrittura vengono mantenuti solo sulle singole righe aggiornate. Più thread possono leggere e aggiornare il database.

In sintesi, le prestazioni più veloci si ottengono in genere con la combinazione di tabelle **MEMORY**, accesso **in-process** e modello di transazione **MVCC**. Se è richiesto un utilizzo ridotto della memoria, alcune tabelle possono essere definite come tabelle **CACHED**, mantenendo invece le tabelle a cui si accede più frequentemente come tabelle **MEMORY**. Se l'accesso è di tipo **server**, è possibile utilizzare le *stored procedures* SQL per ridurre la latenza dovuta ai round trip della rete.

BIBLIOGRAFIA

- [1] <https://angular.io>
- [2] <https://www.oracle.com/java/technologies/java-ee-glance.html>
- [3] <https://material.angular.io>
- [4] <https://rsocket.io>
- [5] <https://jwt.io>
- [6] <https://www.docker.com>
- [7] <https://www.wildfly.org>
- [8] <https://www.postgresql.org>
- [9] <https://www.timescale.com>
- [10] <https://oauth.net/2/>
- [11] <https://github.com/auth0/java-jwt>
- [12] <https://docs.oracle.com/javaee/6/tutorial/doc/gjbnr.html>
- [13] <https://www.oracle.com/technical-resources/articles/java/jax-rs.html>
- [14] <https://junit.org/junit5/>
- [15] <https://site.mockito.org>
- [16] <http://hsqldb.org>
- [17] <https://www.oracle.com/technical-resources/articles/java/jsr356.html>
- [18] <https://www.typescriptlang.org>
- [19] <https://leafletjs.com>
- [20] <https://github.com/Cordobo/angularx-qrcode>
- [21] <https://rxjs.dev>
- [22] <https://maven.apache.org>