Aplições Distribuidas pela Internet

MEEC

School resources management system

Authors:

Miguel Rocha (93138) miguel.a.rocha@tecnico.ulisboa.pt
José Graça (96256) josemendesgraca@tecnico.ulisboa.pt

Group 8

2025/2026 – 1st Semester, P1

Contents

1 Introduction

This document presents the design and implementation of a School Resources Management System developed for the ADInt course. The system follows a modular, service-oriented architecture comprising four Flask microservices—Message, Room, Food, and Check-in—coordinated by a client-facing gateway. Each service exposes REST endpoints (and, where appropriate, XML-RPC) and persists state in SQLite via SQLAlchemy; the Room service additionally integrates with the Fenix API to populate room schedules. The gateway manages user authentication (Fenix OAuth), proxies requests to the backing services, and supports QR-code–driven interactions. This report consolidates the unified REST API, enumerates internal and public endpoints, and specifies the underlying data models with their entities, relationships, constraints, and defaults. It also summarises initial data, background behaviours, and cross-service flows (messaging and friendships, room events, restaurant reservations and ratings, and user presence via check-ins), providing an implementation-faithful reference for maintainers and evaluators. Other than that, the final project also has all the functionalities of the first project stage implemented. The databases had to be changed to accommodate the new functionalities so the server were also adapted for this stage. Other than that, not much changes from the first project implementation.

2 Unified REST API Documentation

Internal Endpoints (Client-facing)

Endpoint	Methods and Description
`/`	GET – Login screen.
`/mainScreen`	GET – Main dashboard (requires login).
`/messages`	GET – Message interface (requires login).
`/login`	GET – Initiate OAuth login with Fenix.
`/login/callback`	GET – OAuth callback handler.
`/logout`	GET – Logout user.
`/qrreaderout`	GET – QR code scanner (public).
`/qrreaderin`	GET – QR code scanner (authenticated).
`/public_info?data=<qr_data>`	GET – Process QR code (public).
`/private_info?data=<qr_data>`	GET – Process QR code (authenticated).
`/user_courses`	GET – Get user's enrolled courses from Fenix.
`/bots/create/<number_of_bots>`	GET – Create test bot users.

REST API Endpoints (Proxies to Other Services)

Endpoint	Methods, Description, and Proxy Target

`/api/user/profile`	GET – Get current user profile (local).
`/api/friends`	GET – Get user's friends list (Message App).
`/api/chat/<friend_username>`	GET – Get chat with friend (Message App).
`/api/whereis/<username>`	GET – Get user's check-in location (Check-in App + Room App).
`/api/user/add_friend`	POST – Add friend (Message App).
`/api/send_message_resquest`	POST – Send message (Message App).
`/api/user/reserve`	POST – Reserve meal at restaurant (Food App).
`/api/user/rate`	POST – Rate restaurant (Food App).

Message App (Port 5010)

Endpoint	Methods and Description
`/api/User/<username>`	GET – Check if user exists. Returns: 200 if exists, 404 if not.
`/api/<username>/friends`	GET – Get user's friends. Returns: JSON array of friend usernames.
`/api/<username>/inbox`	GET – Get received messages. Returns: JSON array of messages.
`/api/chat/<user_nick>/<friend_nick>`	GET – Get conversation between two users. Returns: JSON array of messages (ordered by date).
`/api/<username>/sent`	GET – Get sent messages. Returns: JSON array of messages.
`/api/add_user`	POST – Add new user. Body: {username, pwd}. Auth: Secret required.
`/api/add_friend`	POST – Create friendship. Body: {username, friend_username, pwd}. Auth: Secret required.
`/api/send_message`	POST – Send message. Body: {sender, receiver, content, pwd}. Auth: Secret required.
`/api/user/<username>`	DELETE – Delete user.

Room App (Port 5001)

Endpoint	Methods and Description

/api/room/<room_name>/schedule	GET – Get room schedule by name. Returns: JSON {menu: schedule}.
/api/scrape/<room_tecnico_id>	GET – Scrape schedule from Fenix API. Returns: JSON success message.
/api/room/<room_id>/internal_id	GET – Get events by internal DB ID. Returns: JSON {events: [...]}.
/api/room/<room_tecnico_id>/events	GET – Get events by Tecnico ID. POST – Add event to room. DELETE – Delete specific event. Body (POST): {course, start_time, end_time, event_type, date}. Body (DELETE): {date, start_time, course}.
/api/room/<room_tecnico_id>	GET – Get room info by Tecnico ID. POST – Create new room. Body: {tecnico_id, name, capacity, schedule}.
/api/room/<room_name>	GET – Get room info by name. Returns: JSON room details.

Food App (Port 5000)

Endpoint	**Methods and Description**
/api/<restaurant_name>/menu	GET – Get restaurant menu. Returns: JSON {menu: "..." }.
/api/restaurant/<restaurant_name>/<rating>	GET – Update restaurant rating (1–5). Returns: JSON success + new rating.
/api/restaurants/<restaurant_name>/reserve	POST – Create reservation. Body: {date}.

Check-in App (Port 8000)

Endpoint	**Methods and Description**
/api/whereis/<username>	GET – Get user's current location. Returns: JSON {status, location?, date?}.
/api/checkin	POST – Check-in user. Body: {username, location}.
/api/checkout	POST – Check-out user. Body: {username}.

3 Functionalities

3.1 Main App

The main app is the connection point in this project, it works as server, client for the other apps REST server, and stores a database of users, all apps that deal with users identify users by username.

Main app's code can be found in client.py

3.1.1 Login Screen

In the login screen the user may click the login button and redirect to IST login page. When login is completed the user is redirected to the main screen, if the user is known the database loads the user info, if it is a new user, IST api is contacted to retrieve the available user data.

Still in login screen the user may access the public information about rooms and restaurants by scanning a QRcode that encodes text for "room:roomID" or "restaurant:name".

3.1.2 Main Screen

After a successful login the user reaches the main app where the following functionalities are present:

- **Message App Button** - A button in available to redirect to message screen.
- **Logout Button** - This button de-authenticates user and redirects to login screen.
- **QR Code Reader** - Redirects to the QR Code Reader page, which loads a camera and as soons as it detects a QR Code, reads the code's structure and sends it to the server to get processes
- **Actions** - Changes according to read QR Code type. If the read QR Code is of type room, it allows the user to either check-in or check-out. If the QR Code is of type restaurant, it allows the user to make a reservation on a specific date, and only after making the reservation does it allow the user to rate the restaurant.
- **Information** - Also changes according to the read QR Code type. If the read QR Code is of type room, it displays a table showing the classes that take place in that room on that day, as well as whether the user is enrolled in those classes or not and if that class is currently happening, has already happened or will take place later in the day. If the read QR Code is of type restaurant, it displays the menu by default and a messages regarding the success of your reservation and rating.

3.1.3 Message Screen

In this webbapp, users may show a QRcode that other users may read through the main screen that once read, befriends both friends, and stores friendships in MessageApp server.

The may choose a friend from the friend list, and this will fill the massage receiver, search for the location of friend on checkin database, show it, and show message history like a normal text app.

Sending a message stores message in Message app database.
The app refreshes incoming messages every 10 seconds.

3.2 MessageApp

This app acts as server for the main app, database for messages, users and friendships, message and friendships tables are related to user for ease of access.

this app has got an admin server on its end point.

Data access uses SQLAlchemy with a SQLite database at `db/messagesdb.sqlite`. The metadata base is created with `Base = declarative_base()` and tables are materialized by `Base.metadata.create_all(engine)`. The session factory is provided by `Session = sessionmaker(bind`

Table `user` (Class `User`)

Description: Application user account (logical identity for sending/receiving messages).

Column	Type	PK/FK	Default	Null?
`id`	Integer	PK	–	No
`username`	String	UNIQUE	–	No

Notes:

- The `username` column is unique and non-nullable.

Table `message` (Class `Message`)

Description: A direct message between two users, storing sender/receiver user IDs and content.

Column	Type	PK/FK	Default	Null?
`id`	Integer	PK	–	No
`sender_id`	Integer	FK → `user.id`	–	No
`receiver_id`	Integer	FK → `user.id`	–	No
`content`	String	–	–	No
`created_at`	DateTime	–	`datetime.utcnow()`	Yes

Relationships:

- `sender_user`: many-to-one to `User` (via `sender_id`); backref `sent_messages`.
- `receiver_user`: many-to-one to `User` (via `receiver_id`); backref `received_messages`.

Notes:

- There are no `sender` or `receiver` *text* columns; use the relationships (`m.sender_user.username`, `m.receiver_user.username`) to access usernames.
- No explicit cascade or `ondelete` behavior is configured (database defaults apply).

Table `friendship` (Class `Friendship`)

Description: Represents a friendship link between two users (self-referential association).

Column	Type	PK/FK	Default	Null?
`id`	Integer	PK	–	No
`user1_id`	Integer	FK → `user.id`	–	No
`user2_id`	Integer	FK → `user.id`	–	No

Relationships:

- `user1`: many-to-one to `User` (via `user1_id`).
- `user2`: many-to-one to `User` (via `user2_id`).

Notes:

- The model treats friendship as an (intended) undirected pair, but no uniqueness/symmetry constraint is enforced. As-is, duplicates or self-friendships can be inserted unless additional constraints are added (e.g., `UniqueConstraint(min(user1_id,user2_id), max(user1_id,us`

Cardinality and Integrity

- **Messages:** Each `Message` has exactly one sender and one receiver (two *many-to-one* links to `User`). A `User` can send/receive zero or many messages ($1 : N$ in each role).
- **Friendships:** Many-to-many between `User` and `User` modeled by `Friendship`.
- **Referential integrity:** `Message.sender_id`, `Message.receiver_id`, `Friendship.user1_id`, `Friendship.user2_id` are `NOT NULL` and reference `user.id`.
- **Deletions:** No cascade rules are defined; deleting a `User` with dependent rows will violate FKs unless related `Message`/`Friendship` rows are removed first or proper cascade/`ondelete` is configured.

Model Summary

- **Entities:** `User`, `Message`, `Friendship`.
- **Key attributes:** `User.id`, `Message.id`, `Friendship.id`.
- **Foreign keys:** `Message.sender_id` → `User.id`, `Message.receiver_id` → `User.id`, `Friendship.user1_id` → `User.id`, `Friendship.user2_id` → `User.id`.
- **Constraints:** `User.username` is `UNIQUE` and `NOT NULL`.
- **Relevant defaults:** `Message.created_at` = `UTC now (datetime.utcnow())`.

3.3 Room app

This app acts as server for the client server, scrapes IST api, filling rooms and respective events automatically, storing rooms and events in related tables, storing rooms without classes as study rooms. The app has the REST capability of modifying events.

Data access uses SQLAlchemy with a SQLite database at `db/roomdb.sqlite`. The metadata base is created with `Base = declarative_base()` and tables are materialized by `Base.metadata.cre`
The session factory is provided by `Session = sessionmaker(bind=engine)`.

Table `room` (Class `Room`)

Description: Physical or logical room, with capacity, schedule payload, and type.

Column	Type	PK/FK	Default	Null?
`id`	Integer	PK	–	No
`tecnico_id`	Integer	–	–	No
`name`	String	–	–	No
`capacity`	Integer	–	–	Yes
`schedule`	String	–	–	Yes
`room_type`	String	–	`'study'`	Yes

Relationships:

- `events`: one-to-many to `Event`, declared as `Room.events = relationship("Event", order_by=Event.date, back_populates="room")`.

Notes:

- `tecnico_id` is not marked as unique; duplicates are allowed unless constrained elsewhere.
- `room_type` defaults to `'study'`.

Table `event` (Class `Event`)

Description: Scheduled event associated with a room on a specific date.

Column	Type	PK/FK	Default	Null?
`id`	Integer	PK	–	No
`course`	String	–	–	No
`start_time`	String	–	–	No
`end_time`	String	–	–	No
`event_type`	String	–	`'lecture'`	Yes
`room_id`	Integer	FK → `room.id`	–	No
`date`	Date	–	–	No

Relationships:

- `room`: many-to-one to `Room`, with `back_populates="events"`.

Notes:

- `start_time` and `end_time` are stored as `String`; consider `Time`/`DateTime` for temporal semantics.
- The `__repr__` references `self.name`, which is not a column of `Event` (likely a minor typo).

Cardinality and Integrity

- **Cardinality:** One `Room` has zero or many `Event` ($1 : N$); each `Event` belongs to exactly one `Room`.
- **Referential integrity:** `Event.room_id` is `NOT NULL` and references `room.id`.
- **Ordering:** `Room.events` is ordered by `Event.date`.

Model Summary

- **Entities:** `Room`, `Event`.
- **Key attributes:** `Room.id`, `Event.id`.
- **Foreign key:** `Event.room_id` $\rightarrow$ `Room.id`.
- **Relevant defaults:** `Room.room_type` = `'study'`, `Event.event_type` = `'lecture'`.

3.4 Food App

This app acts as a server for both reservations and evaluations which belong in the restaurant's table. Reservations are related to the restaurants.

Data access uses SQLAlchemy with a SQLite database at `db/fooddb.sqlite`. The metadata base is created with `Base = declarative_base()` and tables are materialized by `Base.metadata.cre`
The session factory is provided by `SessionFactory = sessionmaker(bind=engine)`.

Table `restaurant` (Class `Restaurant`)

Description: Main entity representing a restaurant, including menu, reservation count, and aggregate rating.

Column	**Type**	**PK/FK**	**Default**	**Null?**
`id`	Integer	PK	–	No
`name`	String	–	–	Yes
`nr_reservations`	Integer	–	–	Yes
`menu`	String	–	–	Yes
`rating`	Float	–	`0`	Yes
`number_of_ratings`	Integer	–	`0`	Yes

Relationships:

- `reservations`: one-to-many to `Reservation`, with `back_populates="restaurant"` and ordering by `Reservation.date`.

Notes:

- `rating` stores the running average; `number_of_ratings` maintains the count to update the average incrementally.
- No explicit cascade behavior is configured (defaults apply).

Table `reservation` (Class `Reservation`)

Description: Stores a reservation associated with a restaurant at a given date/time.

Column	Type	PK/FK	Default	Null?
`id`	Integer	PK	–	No
`restaurant_id`	Integer	FK → `restaurant.id`	–	No
`date`	DateTime	–	`datetime.utcnow()`	Yes

Relationships:

- `restaurant`: many-to-one to `Restaurant`, with `back_populates="reservations"`.

Cardinality and Integrity

- **Cardinality:** One `Restaurant` has zero or many `Reservation` ($1 : N$); each `Reservation` belongs to exactly one `Restaurant`.
- **Referential integrity:** `Reservation.restaurant_id` is `NOT NULL` and references `restaurant.id`.
- **Logical ordering:** Reservations are exposed ordered by `date` via the relationship's `order_by`.

Model Summary

- **Entities:** `Restaurant`, `Reservation`.
- **Key attributes:** `Restaurant.id`, `Reservation.id`.
- **Foreign key:** `Reservation.restaurant_id` → `Restaurant.id`.
- **Relevant defaults:** `Restaurant.rating` = 0, `Restaurant.number_of_ratings` = 0, `Reservation.date` = `UTC now`.

3.5 Check-InApp

This app acts as a server for the check in or check out, stores them as rows incrementally without deleting and validations are made by order of the check in. Data access uses SQLAlchemy with a SQLite database at `db/check.sqlite`. The metadata base is created with `Base = declarative_base()` and the table is materialized by `Base.metadata.create_all(engine)`. The session factory is provided by `Session = sessionmaker(bind=engine)`.

Table `checkin` (Class `CheckIn`)

Description: Stores user presence events. A single table records both "check-in" and "check-out" events, distinguished by the `spec` column (`"Checked In"` or `"Checked Out"`).

Column	Type	PK/FK	Default	Null?
`id`	Integer	PK	–	No
`username`	String	–	–	Yes
`location`	String	–	–	Yes
`spec`	String	–	–	Yes
`date`	Date	–	–	Yes

Notes:

- Business logic enforces allowed values for `spec` (`"Checked In"` / `"Checked Out"`) in the application layer; there is no DB-level constraint.
- `date` is set to the current date in application code for both check-ins and check-outs.
- The `__repr__` dynamically prints `CheckIn` or `CheckOut` based on `spec`; this does not imply a separate table.

Cardinality and Integrity

- **Entities:** Single entity `CheckIn`; no foreign keys or relationships.
- **Event stream:** User presence is modeled as an ordered stream of events per `username`; the latest row per user determines current status.

Model Summary

- **Entity:** `CheckIn`.
- **Key attribute:** `CheckIn.id`.
- **Relevant semantics:** `spec` $\in$ {`"Checked In"`, `"Checked Out"`} (enforced in code).