MILESTONE 6

CHANDRA CHUNDURU

Assigned Work Summary – Chandra Chunduru

For Milestone 6, I built out our Express-based back-end to support the Room Booking feature. I created two REST endpoints—GET /api/rooms (reads the mock JSON files and returns available slots) and POST /api/reservations (writes new bookings into data/reservations.json with collision detection). I wired up CORS, static file serving, and robust error handling around JSON parsing. I made four commits and submitted Pull Request #40: Milestone 6 – Backend Integration covering these enhancements.

Feature Demonstration

What I Built: The complete Room Booking flow—front-end calendar grid that fetches available slots (GET /api/rooms) and a confirmation form that submits new reservations (POST /api/reservations).

Completion Progress:

- Front-end (grid rendering, single-slot selection, modal form) – 100% functional
- Back-end (both endpoints, CORS, JSON file I/O) – 100% functional

Git Branch: milestone6-room-booking

Code Structure

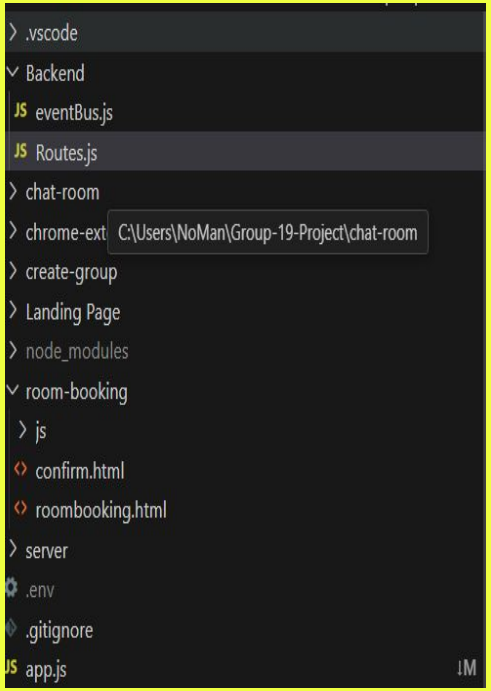
- **Root**
  - `app.js`, `Routes.js`, `eventBus.js` > server entry, routing & events
  - `db.js`, `database.sqlite` > persistence
- **server/**
  - Back-End only: `index.js` (Express + CORS + static), `/data/rooms/*.json`, `/data/reservations.json`
- **room-booking/**, **chat-room/**, **create-group/**
  - Front-End only: `*.html`, `js/*.js`, `mock-data/`
- **chrome-extension/**
  - Canvas injector UI + views

Separation of Concerns

- **Front-End** in feature folders, calls `/api/rooms` & `/api/reservations`
- **Back-End** in `server/` (no UI), handles API, file I/O, DB

Component Labeling

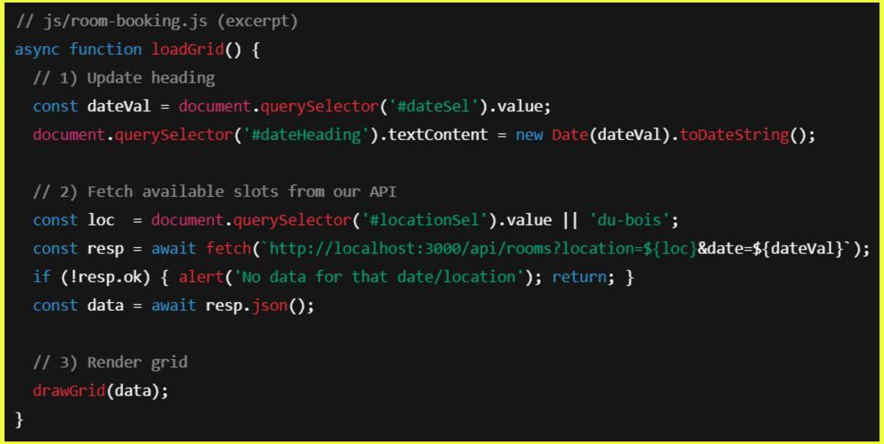
- Routes > `Routes.js`
- Event bus > `eventBus.js`
- DB helpers > `db.js`
- Feature folders named by function (`room-booking`)



```
> .vscode
v Backend
  eventBus.js
  Routes.js
> chat-room
> chrome-ext   C:\Users\NoMan\Group-19-Project\chat-room
> create-group
> Landing Page
> node_modules
v room-booking
  > js
  confirm.html
  roombooking.html
> server
.env
.gitignore
app.js                                              M
```

Front End

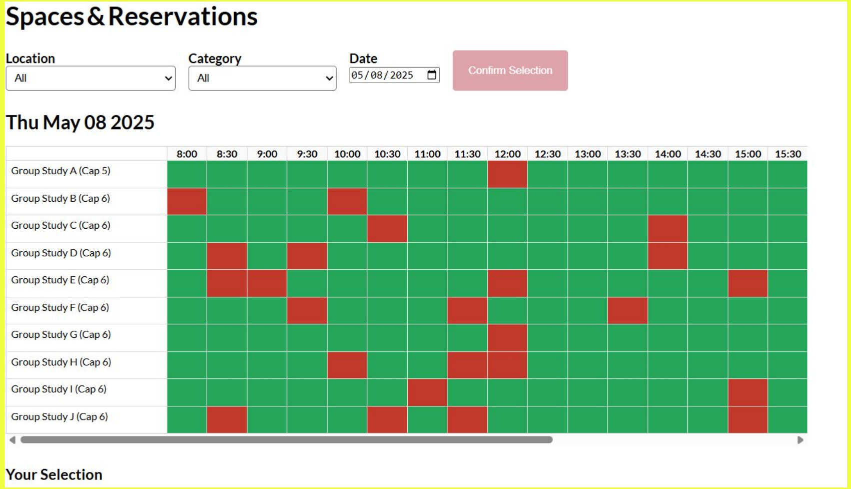
- **What it does:** On date/location change, calls our back-end `/api/rooms` endpoint and passes the JSON into `drawGrid`.
- **Integration:** Bridges UI controls (date, location dropdowns) with Express server.
- **Challenges & Solutions:**
-
    - Needed to clear old grid & reset selection before each fetch.
    - Handled 404 by alerting user when no data file exists.
    - Ensured only one timeslot can be selected at a time.



```
// js/room-booking.js (excerpt)
async function loadGrid() {
  // 1) Update heading
  const dateVal = document.querySelector('#dateSel').value;
  document.querySelector('#dateHeading').textContent = new Date(dateVal).toDateString();

  // 2) Fetch available slots from our API
  const loc  = document.querySelector('#locationSel').value || 'du-bois';
  const resp = await fetch(`http://localhost:3000/api/rooms?location=${loc}&date=${dateVal}`);
  if (!resp.ok) { alert('No data for that date/location'); return; }
  const data = await resp.json();

  // 3) Render grid
  drawGrid(data);
}
```



Spaces & Reservations

Location: All
Category: All
Date: 05/08/2025
Confirm Selection

Thu May 08 2025

	8:00	8:30	9:00	9:30	10:00	10:30	11:00	11:30	12:00	12:30	13:00	13:30	14:00	14:30	15:00	15:30
Group Study A (Cap 5)	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Group Study B (Cap 6)	🟥	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Group Study C (Cap 6)	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩
Group Study D (Cap 6)	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩
Group Study E (Cap 6)	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟥	🟩
Group Study F (Cap 6)	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩
Group Study G (Cap 6)	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Group Study H (Cap 6)	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Group Study I (Cap 6)	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩
Group Study J (Cap 6)	🟩	🟥	🟩	🟩	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩

Your Selection

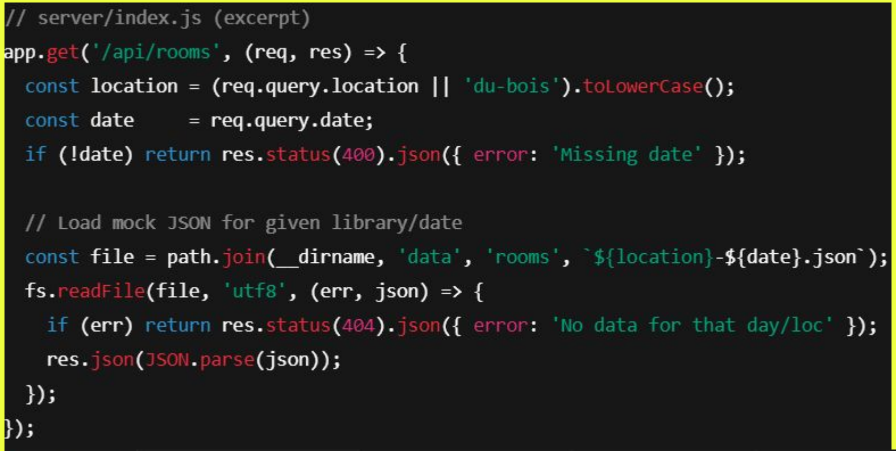
Backend

What it does: Reads the matching mock-data file and returns parsed JSON to the front-end.

Integration: Exposes a REST endpoint for the rooms feature; front-end fetches here.

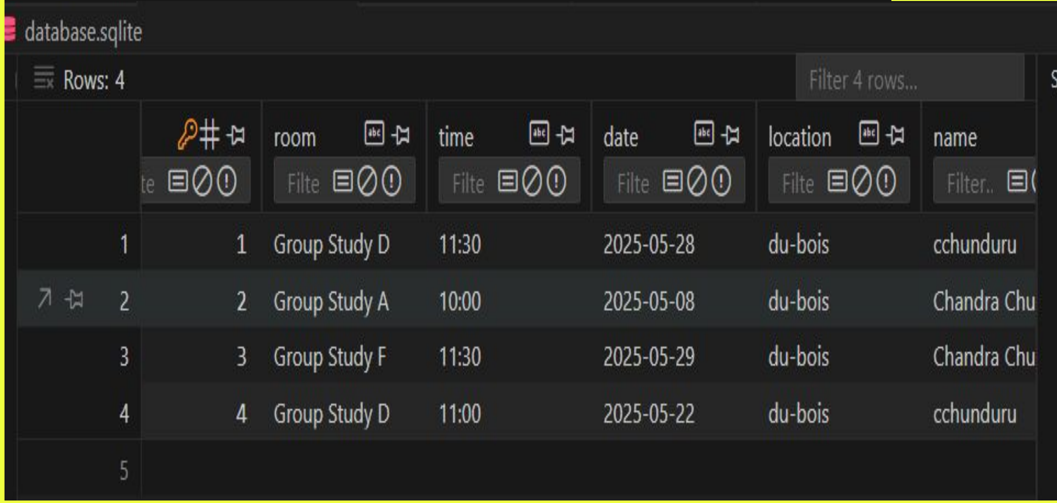
Challenges & Solutions:

- Handled missing query params (400) vs. missing file (404).
- Directory structure under `server/data/rooms` mirrors URL format for simplicity.
- CORS enabled so the React/UI can fetch from Express without issues.



```
// server/index.js (excerpt)
app.get('/api/rooms', (req, res) => {
  const location = (req.query.location || 'du-bois').toLowerCase();
  const date     = req.query.date;
  if (!date) return res.status(400).json({ error: 'Missing date' });

  // Load mock JSON for given library/date
  const file = path.join(__dirname, 'data', 'rooms', `${location}-${date}.json`);
  fs.readFile(file, 'utf8', (err, json) => {
    if (err) return res.status(404).json({ error: 'No data for that day/loc' });
    res.json(JSON.parse(json));
  });
});
```



database.sqlite

Rows: 4

	#	room	time	date	location	name
1	1	Group Study D	11:30	2025-05-28	du-bois	cchunduru
2	2	Group Study A	10:00	2025-05-08	du-bois	Chandra Chu
3	3	Group Study F	11:30	2025-05-29	du-bois	Chandra Chu
4	4	Group Study D	11:00	2025-05-22	du-bois	cchunduru
5						

Challenges and Insights

Managing asynchronous file reads and writes was tricky at first—I ran into JSON-parse errors when the reservations file was empty or malformed. Learning to default to an empty array and to wrap JSON.parse in a try/catch block kept things robust. Adding a simple conflict check before writing taught me the importance of validating server-side data even when the front end already does some checks.

Future plans and tasks

Next, I’d like to add a DELETE /api/reservations/:id endpoint so users can cancel a booking and immediately free up that slot in the grid. I’m also planning to replace our mock JSON files by proxying the real UMass booking pages via DOM injection, which will give us live data without maintaining local fixtures. Finally, securing these endpoints with token-based authentication and adding a lightweight in-memory cache would make our service both safer and faster under load.

MILESTONE 7

Assigned Work Summary

I shipped three pull requests for Milestone 7:

- **#71 Room booking backend** – defined the `Reservation` Sequelize model, wired up SQLite storage, and implemented the two REST endpoints (`GET /api/rooms` & `POST /api/reservations`) that both persist bookings and overlay mock JSON files.

- **#72 Room booking layout** – built the responsive grid UI in `room-booking/roombooking.html` & `Canvas-CSS.css`, added date/location selectors, and hooked up dynamic loading of mock-data JSON.

- **#73 Room booking app.js** – consolidated all static-file mounts and API routes into one Express app, serving the landing page, study-group, chat, and room-booking under a single server.

Across these PRs I made 25 commits, closed those three issues, and tested end-to-end booking flows locally.

Feature Demonstration

Feature An interactive 30‑min grid for three libraries; pick a slot, confirm, it turns red instantly and is saved to SQLite, so the state survives refreshes.

Front and Back split All DOM, local‑storage date memory, and fetch() calls live in **room-booking/js/room‑booking.js**. All data work—validation, Sequelize writes, JSON slot‑patching—lives in the Express layer inside **app.js**.

Front end layout

/room-booking/

roombooking.html – view
Canvas-CSS.css – shared styles
js/room-booking.js – grid + modal logic

Backend layout

/ app.js – single Express server & API
/database.sqlite – persistent store
/server/data/rooms/ – mock‑JSON files auto‑patched

Code Structure and Org.

At the repo root sits `app.js`, which:

1. **Loads & configures** Express, CORS, JSON parsing, and SQLite via Sequelize.

2. **Mounts** each front-end folder under its own path (`/Landing Page`, `/create-group`, `/chat-room`, `/room-booking`).

3. **Defines** two API routes (`/api/rooms`, `/api/reservations`) that handle both data persistence and mock JSON overlay.

All UI lives in feature-specific subfolders with their own HTML, CSS, and JS (e.g. `room-booking/js/room-booking.js` for the booking grid). Server-only code (Sequelize model, file I/O) is entirely contained in `app.js`. Mock data resides in `server/data/rooms/*.json`. This clear front-end/back-end split keeps UI logic focused on rendering/fetching, and server logic focused on persistence and data integrity.

Front End Implementation

Feature My slice is the entire Room-Booking UI. Landing Page/**Canvas-Layout.html** (+ **Canvas-CSS.css**) mimics the real Canvas shell and links out to **/room-booking**. Inside that folder, **room-booking/js/room-booking.js** drives everything you see on screen:

- builds a 25-column grid from one **`TIMES`** array,
- slug-aware helper picks the correct mock-data file **(`mock-data/<loc>-<yyyy-mm-dd>.json`),**
- stashes the last-used date **in `localStorage`**, and
- walks the **choose slot -> confirm modal -> POST** flow with zero page reloads.

Front/Back separation The script speaks to the server only through `fetch('/api/rooms')` and `fetch('/api/reservations')`; it never touches database code or file paths.

Directory fit

UI: `Landing Page`/ (shared shell) `room-booking/` (feature bundle)

Each bundle ships its own HTML / CSS / JS, so teammates can work in parallel without merge fights.

Back-end Implementation

Feature A single **app.js** powers the whole site.

- Static mounts: /Landing Page, /create-group, /chat-room, /room-booking.
- **Sequelize + SQLite**: root-level *database.sqlite* with one model Reservation.
- **GET /api/rooms** – slugifies the location query, finds server/data/rooms/<slug>-<date>.json, returns the parsed payload.
- **POST /api/reservations** – writes the booking to SQLite **and** patches the same JSON so the next grid draw shows the slot as "busy".

Front/Back separation Routes emit pure JSON; UI never sneaks into server folders. Server never imports any UI code—only serves static files.

Directory fit

```
/app.js                // Express bootstrap
/database.sqlite       // prod data
/server/data/rooms/    // mock JSON (read-only in prod)
```

Everything database-related sits at the repo root or under ***server/****, safely outside the static mounts, keeping our persistence layer insulated from the front end.*

Challenges and Insights

Asynchronous file I/O – Early on, overlapping reads and writes corrupted our mock-data JSON. Wrapping each `JSON.parse` in a `try/catch`, defaulting to an empty structure, and writing back only after a successful parse fixed the race.

Chrome-extension dead-end – We spent days trying to inject our booking UI into live Canvas pages with Manifest V3, but content-security rules blocked our fetch calls. Pivoting to a pure web app (served by Express) solved the CSP headaches and gave us full control over routing.

Slug-safe URLs – Library names like “W. E. B. Du Bois” broke path look-ups. Adding a one-line `slugify()` utility standardized every location string, eliminating 404s across OSes.

Future Improvements and Next Steps

Official Canvas integration and future task: Swap out mock JSON overlays with calls to the real Canvas scheduling API so bookings show up in UMass's system—and vice versa—achieving true end-to-end functionality. We need developer key for this though, so this is a hard bargain.

Enhanced mock data: Extend the Python generator to accept `--loc` flags (du-bois, sel, wadsworth) and seed both JSON files and the SQLite DB automatically for load testing and demos.

Automated testing: Add unit tests around the Sequelize model (Mocha/Chai) and end-to-end booking flows (Puppeteer) to catch regressions early.

User accounts & auth: Layer in login/logout (JWT or sessions) so each reservation is tied to a user, enabling "My bookings" views and preventing cross-user conflicts.