Tutoring App – Full Project Documentation

This document describes the Tutoring App end to end: its goals, architecture, data model, API design, frontend structure, security, deployment, and developer workflows. It's written as a practical guide, with diagrams, code examples, and tips learned during implementation.

1. What this project does

The Tutoring App is a full-stack platform where students discover tutors and book sessions, tutors manage availability and sessions, and admins oversee users and content. It focuses on a clean UX with strong guardrails against double booking, predictable data models, and a documented REST API.

Key capabilities:

- Register/login using email (admins) or student index (students/tutors) + password
- Role-based access control: student, tutor, admin
- Tutors manage recurring or date-specific availability
- Students browse/ filter available sessions and book/cancel
- Tutors/admins create, update, cancel sessions; conflict checks prevent overlaps
- Admins manage users and roles in bulk
- Responsive UI using Vue 3 + Vuetify, state via Pinia, API calls via Axios

Use cases in a nutshell:

- A new student visits the landing page, registers, and logs in. They browse all available sessions across subjects and tutors, filter by date, and book a slot. The booking instantly updates the UI and prevents others from double-booking the same slot.
- A tutor logs in, opens the Availability page, and defines recurring weekly availability (e.g., Mondays 09:00–12:00) plus a few date-specific blocks for exam season. They then create specific session offerings within those windows. The system blocks conflicts at creation time.
- An admin logs in, opens the Users Admin screen, filters all tutors, bulk-updates a few roles, and audits the day's sessions. Admins can also create tutors directly with assigned subjects.

2. System architecture

Three logical tiers:

- Frontend (Vue 3 + Vite + Vuetify + Pinia)
- Backend (Node.js + Express)
- Database (MongoDB via Mongoose)

High-level diagram (SVG in repo): `docs/images/architecture.svg`.

Highlights:

- API is namespaced under `/api/*` and protected with JWT Bearer tokens
- CORS configured for local dev (5173/5174) and easy production hardening
- Mongoose schemas encode validation and indexes for efficient queries
- Error handling middleware centralizes API error responses

Non-goals and boundaries:

- This app does not yet include email delivery, payments, or real-time websockets; those are listed in the roadmap. All user-facing updates are currently driven by standard HTTP requests and optimistic UI patterns.

- The backend is a monolith. Services are modularized by route and model folders but a microservices split is intentionally out of scope.

3. Technology stack

- Frontend: Vue 3, Vite, Vuetify 3, Vue Router, Pinia, Axios, js-cookie
- Backend: Node.js, Express, Mongoose, jsonwebtoken, bcryptjs, cors, helmet, morgan, dotenv
- DB: MongoDB (local or Atlas), with Mongoose models for Users, Sessions, Availability

4. Getting started

Prerequisites:

- Node.js 18+
- MongoDB instance

Environment variables (Backend `.env`):

- `MONGODB_URI` – Mongo connection string
- `JWT_SECRET` – strong random secret
- `JWT_EXPIRE` – e.g. `7d`
- `NODE_ENV` – `development` or `production`
- `PORT` – default 5000

Backend quickstart:

- Files: `src/backend/` (entry `server.js`)
- Scripts: see `src/backend/package.json`

Frontend quickstart:

- Files: `src/frontend/` (entry `src/main.js`)
- Vite dev server typically on 5173
- Set `VITE_API_URL` in `src/frontend/.env` for non-default API base

Local development tips:

- When running frontend and backend separately, ensure CORS allows dev origins `http://localhost:5173` and `http://localhost:5174`. This is already set in `server.js` for non-production NODE_ENV.
- Seed data is available via scripts in `src/backend/` (e.g., `seed.js`, `create_test_sessions.js`). Use them to quickly populate tutors and sessions.

5. Data model

All schemas live in `src/backend/models`.

5.1 User

- Fields: `name`, `email`, optional `studentIndex`, `password` (hashed), `role` (`student|tutor|admin`), profile fields (`avatar`, `bio`, `subjects`, `experienceLevel`, ratings, `qualifications`), `availability` preferences, `preferences` (notification settings)
- Security: passwords are bcrypt-hashed in a `pre('save')` hook
- Methods: `matchPassword(plain) => boolean`

Example (excerpt from `User.js`):

```
userSchema.pre('save', async function(next) {
  if (!this.isModified('password')) return next()
  const salt = await bcrypt.genSalt(10)
  this.password = await bcrypt.hash(this.password, salt)
  next()
})
```

5.2 Session

- Represents a concrete tutoring slot (date + start/end time)
- Fields: `title`, `subject`, `tutor`, `student` (nullable), `date`, `startTime`, `endTime`, `duration`, `status` (`available|booked|completed|cancelled`), `meetingLink`, `notes`
- Indexes on `tutor/date`, `student/date`, `status/date`
- Conflict checks in routes prevent double bookings

Important invariant: For a given tutor and date, no two sessions may overlap on time. Overlap detection is handled server-side using `$or` clauses comparing the new slot's start/end bounds to existing sessions.

5.3 Availability

- Tutors declare recurring or specific-date availability window(s)
- Fields: `tutor`, `dayOfWeek`, `startTime`, `endTime`, `isRecurring`, `specificDate`, `isAvailable`, `timezone`
- Virtual: `isFuture` convenience check
- Indexes on `(tutor, dayOfWeek)` and `(tutor, specificDate)`

Best practice: Keep recurring availability as the source of truth for routine weeks, and only create `specificDate` entries for exceptions (e.g., holidays, special exam preps).

6. Backend API

Base URL: `/api` (default host `http://localhost:5000`)

6.1 Auth

- POST `/auth/register` – public; creates a student by default
  - Body: `{ name, email, password }`
  - Returns: `{ success, data: { _id, name, email, role, token } }`
- POST `/auth/login` – public; email for admins or studentIndex for students/tutors
  - Body: `{ identifier, password }`
  - Returns: `{ success, data: { _id, name, email, studentIndex, role, token } }`
  - Example request:

    ```
    POST /api/auth/login HTTP/1.1
    Content-Type: application/json

    { "identifier": "admin@example.com", "password": "admin123" }
    ```

  - Example response:

```
{
  "success": true,
  "data": {
    "_id": "66c7e...",
    "name": "Admin User",
    "email": "admin@example.com",
    "role": "admin",
    "token": "<JWT>"
  }
}
```

- GET `/auth/me` – private; Bearer token
    - Returns user profile

Error modes:

- 400 if invalid registration payload
- 401 if login credentials are invalid or token verification fails

6.2 Users (admin where noted)

- GET `/users` – admin; optional `?role=tutor|student|admin`
- POST `/users` – admin; create with role and optional subjects
- GET `/users/:id` – private; owner or admin
- PUT `/users/:id` – private; owner can update profile fields, admin can change role
- DELETE `/users/:id` – admin
- PUT `/users/:id/role` – admin; change a single user role
- PUT `/users/roles/bulk` – admin; bulk updates
- GET `/users/tutors` – public; list active tutors

Example: bulk role update

```
PUT /api/users/roles/bulk
Authorization: Bearer <token>
Content-Type: application/json

{
  "updates": [
    { "userId": "66c7e1...a1", "role": "tutor" },
    { "userId": "66c7e1...b2", "role": "student" }
  ]
}
```

Response

```
{
  "success": true,
  "data": {
    "updated": [
      { "userId": "66c7e1...a1", "name": "Alice", "email": "alice@x.com", "newRole":
"tutor" }
```

```
    ],
    "errors": [
      { "userId": "66c7e1...b2", "error": "Cannot demote self from admin" }
    ]
  },
  "message": "1 users updated successfully, 1 failed"
}
```

Notes:

- Only admins may create users with roles other than student. Students self-register via `/auth/register` and are assigned the `student` role by default.
- Admins cannot demote their own role away from `admin` for safety.

6.3 Sessions

- GET `/sessions` – private; filter by `status`, `tutor`, `student`, `date`
- POST `/sessions` – tutor/admin; creates a session if within availability and non-conflicting
- GET `/sessions/:id` – private
- PUT `/sessions/:id` – private; only tutor owner or admin
- DELETE `/sessions/:id` – tutor/admin
- PUT `/sessions/:id/book` – student; transition available → booked
- PUT `/sessions/:id/cancel` – private; student/tutor/admin can cancel (becomes available)
- GET `/sessions/available` – public; browse available, filter by `subject`, `tutor`, `date`
- Availability helpers under `/sessions/availability/*` for bulk creation and updates

Booking flow diagram: see `docs/images/booking_flow.svg`.

Common pitfalls and responses:

- Attempt to create a session outside declared availability → 400 with guidance to set availability first
- Attempt to book an already booked session → 400
- Attempt to update a session by a non-owner (non-tutor, non-admin) → 401/403

Request/response examples

Create a session (tutor)

```
POST /api/sessions
Authorization: Bearer <tutor-token>
Content-Type: application/json

{
  "title": "Algorithms 101",
  "subject": "Algorithms",
  "date": "2025-09-10T00:00:00.000Z",
  "startTime": "10:00",
  "endTime": "11:00",
  "duration": 60
}
```

Response (201)

```
{
  "success": true,
  "data": {
    "_id": "66c8...",
    "title": "Algorithms 101",
    "subject": "Algorithms",
    "tutor": { "_id": "66c7...", "name": "Tutor T" },
    "student": null,
    "date": "2025-09-10T00:00:00.000Z",
    "startTime": "10:00",
    "endTime": "11:00",
    "duration": 60,
    "status": "available"
  }
}
```

Book a session (student)

```
PUT /api/sessions/66c8.../book
Authorization: Bearer <student-token>
```

Response

```
{
  "success": true,
  "data": {
    "_id": "66c8...",
    "status": "booked",
    "tutor": { "_id": "66c7...", "name": "Tutor T" },
    "student": { "_id": "77a1...", "name": "Student S" }
  }
}
```

6.4 Availability

- GET `/availability/:tutorId?` – public; list availability for all or a tutor
- GET `/availability/my/slots` – tutor; own slots
- POST `/availability` – tutor; create slot
- PUT `/availability/:id` – tutor; update slot
- DELETE `/availability/:id` – tutor; delete slot
- GET `/availability/:tutorId/date/:date` – public; slots for a specific day

Notes:

- The date-specific endpoint returns both recurring slots matching the weekday and any specificDate ranges that fall within the given 24h window. Further filtering to exclude already booked sessions can be added by joining with Sessions.

Example payloads

Create recurring Monday slot (tutor)

```
POST /api/availability
Authorization: Bearer <tutor-token>
Content-Type: application/json

{ "dayOfWeek": "monday", "startTime": "09:00", "endTime": "12:00", "isRecurring":
true }
```

Create date-specific slot

```
POST /api/availability
Authorization: Bearer <tutor-token>
Content-Type: application/json

{ "specificDate": "2025-09-15T00:00:00.000Z", "dayOfWeek": "monday", "startTime":
"14:00", "endTime": "16:00", "isRecurring": false }
```

6.5 Auth middleware

- `protect` reads Bearer token, verifies JWT, attaches `req.user`
- `authorize(...roles)` restricts routes by role

Example: `src/backend/middleware/auth.js` .

7. Security

- JWT with `Authorization: Bearer <token>`
- Passwords hashed with bcrypt; never returned in responses
- Role-based access on sensitive routes
- Helmet for secure headers, CORS for allowed origins, Morgan for request logs in dev
- Error handler returns consistent payloads; no stack traces in production
- Environment secrets stored in `.env` (never commit)

Token handling:

- On the frontend, tokens are persisted in cookies. The Axios request interceptor attaches the `Authorization` header for API calls. A 401 from the API triggers a logout + redirect to `/login` .
- Consider HttpOnly cookies and CSRF protections if you move token storage server-side.

8. Frontend overview

Entry: `src/frontend/src/main.js` wiring Vue, Vuetify, Router, Pinia.

Key directories:

- `views/` – route pages like Login, Dashboard, Sessions, Availability
- `components/` – reusable UI: cards, lists, calendars, dialogs
- `stores/` – Pinia stores: `auth` , `sessions` , `availability` , etc.
- `services/` – Axios client and resource services (`authService` , `userService` , `sessionService`)
- `router/` – route definitions and guards
- `design/` – tokens.css and theme

8.0 Router & views

Representative routes (names may vary):

- `/` → `HomeView.vue`
- `/login` → `LoginView.vue`
- `/register` → `RegisterView.vue`
- `/dashboard` → `DashboardView.vue` (role-aware)
- `/sessions` → `SessionsView.vue` (browse/filters)
- `/my-sessions` → `MySessionsView.vue` (booked by me / created by me)
- `/availability` → `AvailabilityView.vue` (tutors)
- `/admin/users` → `admin/UsersView.vue`
- `/admin/sessions` → `admin/SessionsView.vue`

8.1 Store structure

`stores/auth.js` :

- State: `{ user, token, isAuthenticated, loading, error }`
- Actions: `initializeAuth` , `register` , `login` , `logout` , `setAuthData` , `clearAuthData` , `updateUser` , `clearError`
- Getters: role helpers (`isAdmin` , `isTutor` , `isStudent`)

`stores/sessions.js` (not shown in snippets but implied):

- State: list, loading, filters
- Actions: load sessions, book/cancel, create/update/delete
- Derivations: upcoming/past partitions

8.2 Components overview

- `TutorCard.vue` – tutor summary: name, subjects, rating
- `SessionsList.vue` – list with status chips and actions
- `CalendarComponent.vue` – calendar or list of availability (optional FC integration)
- `NotificationComponent.vue` – snackbar/toast wiring
- `CreateUserDialog.vue` – admin create wizard

8.3 UX details

- Form validation: Vuetify inputs with simple rule arrays
- Feedback: `v-alert` for errors, `v-snackbar` for success
- Motion: subtle route transitions and animated decorative background

8.1 Auth flow (frontend)

- `stores/auth.js` keeps `user` , `token` , `isAuthenticated` , persisted in cookies
- Axios interceptor attaches token; 401 clears session and redirects to `/login`
- LoginView demonstrates email/studentIndex + password UX

Router guards (recommended):

- Protect authenticated routes by checking `authStore.isAuthenticated` before entering. Redirect to `/login` if missing.
- Optionally add role-based route meta and guard (e.g., meta: { requiresAdmin: true }).

8.2 Sessions UX

- List upcoming sessions, filter, paginate
- Tutors create slots respecting availability; conflicts rejected by API

- Students book available slots; instant UI feedback via snackbars

Example (service call):

```
// sessionService.bookSession
async function onBook(id) {
  const { success, data } = await sessionService.bookSession(id)
  if (success) showSnackbar('Booked successfully')
}
```

8.3 Availability UX

- Tutors define weekly recurring blocks or specific-date overrides
- Calendar-style components present slots grouped by day/time

9. Code examples

9.1 Express server and route mounting (excerpt from `server.js`):

```
app.use('/api/auth', require('./routes/auth'))
app.use('/api/users', require('./routes/users'))
app.use('/api/sessions', require('./routes/sessions'))
app.use('/api/availability', require('./routes/availability'))
```

9.2 Login by email or student index (excerpt from `routes/auth.js`):

```
const { identifier, password } = req.body
let user
if (identifier.includes('@')) {
  user = await User.findOne({ email: identifier }).select('+password')
} else {
  user = await User.findOne({ studentIndex: identifier }).select('+password')
}
```

9.3 Session conflict detection (excerpt from `routes/sessions.js`):

```
const conflicting = await Session.findOne({
  tutor: req.user.id,
  date: { $gte: new Date(date), $lt: new Date(new Date(date).getTime() + 86400000)
},
  status: { $in: ['available', 'booked'] },
  $or: [
    { startTime: { $lte: startTime }, endTime: { $gt: startTime } },
    { startTime: { $lt: endTime }, endTime: { $gte: endTime } },
    { startTime: { $gte: startTime }, endTime: { $lte: endTime } }
  ]
})
```

9.5 Availability creation (excerpt from `routes/availability.js`):

```
router.post('/', protect, authorize('tutor'), async (req, res) => {
  const { dayOfWeek, startTime, endTime, isRecurring, specificDate, timezone } =
req.body
  // Validate mandatory fields, prevent duplicates
  const existing = await Availability.findOne({ tutor: req.user._id, dayOfWeek,
startTime, endTime, specificDate })
  if (existing) return res.status(400).json({ success: false, message: 'This
availability slot already exists' })
  const availability = await Availability.create({ tutor: req.user._id, dayOfWeek,
startTime, endTime, isRecurring, specificDate, timezone })
  return res.status(201).json({ success: true, data: availability })
})
```

9.4 Axios API client with interceptors (`src/frontend/src/services/api.js`):

```
api.interceptors.request.use(config => {
  const token = Cookies.get('token')
  if (token) config.headers.Authorization = `Bearer ${token}`
  return config
})
```

10. Error handling conventions

- Success responses: `{ success: true, data, message?, count? }`
- Error responses: `{ success: false, message }` with appropriate status
- 401: missing/invalid token; 403: insufficient role; 404: not found; 400: validation; 500: server

11. Performance notes

- Query indexes on frequently filtered fields (status/date, tutor/date)
- Avoid N+1 by using Mongoose `populate` selectively
- Keep payloads lean for list endpoints; add pagination if datasets grow

Index cheatsheet:

- Sessions: `{ tutor: 1, date: 1 }` , `{ student: 1, date: 1 }` , `{ status: 1, date: 1 }`
- Availability: `{ tutor: 1, dayOfWeek: 1 }` , `{ tutor: 1, specificDate: 1 }`

Pagination and limits:

- If the number of sessions grows large, implement query params `?limit=50&skip=0` and sort by `{ date: 1, startTime: 1 }` .
- For tutors listing, consider server-side search by subject with case-insensitive regex and an index on `subjects` if growth warrants.

12. Local development

Recommended ports:

- Backend: 5000 (configurable)
- Frontend: 5173 (Vite)

Useful endpoints:

- `GET /api/health` quick status JSON

Seed scripts:

- `src/backend/seed.js` and `create_*` helpers for test data

Quality of life:

- Enable `morgan('dev')` in development for request tracing
- Use the `/api/health` endpoint to verify server readiness from frontend dev server

Seeding patterns:

- `create_admin.js` — create an admin with a known email/password
- `create_test_sessions.js` — generate sessions for a tutor on a date range
- `create_test_availability.js` — recurring slots for demonstration
- `clear_availability.js` — cleanup helpers while iterating

13. Testing

- Unit-test models and route guards first (not included by default). Suggested libs: Jest + supertest.
- Smoke test flows: login, list tutors, create session, book/cancel session.

Suggested test matrix:

- Auth: register → login → me; failure on wrong password; 401 when missing token
- Sessions: create within availability; conflict rejects; update by owner vs non-owner; delete by admin
- Booking: double-book protection; cancel by student vs tutor vs admin; status transitions
- Availability: create duplicate slot rejects; update/delete ownership

Minimal supertest example (pseudo):

```
it('prevents creating overlapping sessions', async () => {
  await asTutor.post('/api/sessions').send({ date: '2025-09-10', startTime: '10:00',
endTime: '11:00', duration: 60, title: 'A', subject: 'X' })
  const res = await asTutor.post('/api/sessions').send({ date: '2025-09-10',
startTime: '10:30', endTime: '11:30', duration: 60, title: 'B', subject: 'X' })
  expect(res.status).toBe(400)
})
```

14. Security checklist for production

- Rotate strong `JWT_SECRET`; set `JWT_EXPIRE`
- Restrict CORS origins to your production domain
- Serve over HTTPS; terminate TLS at ingress/reverse proxy
- Ensure MongoDB is not publicly exposed; use network rules / VPC peering
- Enable structured logging and log rotation

Privacy considerations:

- Avoid including sensitive PII in logs
- If adding analytics later, provide granular opt-in and respect user roles

15. Deployment overview

- Containerize backend (Node 18+) and front-end static build (Vite → /dist)

- Recommended: separate hosting for API and static frontend; configure CORS
- Provide environment via a secrets manager or platform variables

Minimal Docker sketch (backend):

```
FROM node:18-alpine
WORKDIR /app
COPY src/backend/package*.json ./
RUN npm ci --only=production
COPY src/backend ./
ENV NODE_ENV=production PORT=5000
EXPOSE 5000
CMD ["node", "server.js"]
```

Static frontend hosting:

- Build with `npm run build` (Vite) → serve `/dist` via static host or CDN (e.g., Netlify, Azure Static Web Apps, Vercel, NGINX)
- Set `VITE_API_URL` to your API base in the hosting environment

16. Roadmap ideas

- Email notifications (booking confirmations, reminders)
- Payments & invoices
- In-app video sessions
- Analytics dashboard and reports
- i18n and accessibility audits

Stretch goals:

- Calendar drag-and-drop for availability (FullCalendar integration)
- ICS export of sessions; reminders via email/SMS
- Tutor discovery: recommendation and search relevance tuning

17. Appendix – Screens and flows

- Login: supports student index or email; shows test credentials in dev
- Dashboard: role-aware quick links
- Sessions: browse/book, manage, cancel; permissions enforced
- Availability: day-by-day overview with create/edit/delete for tutors

Edge cases to consider:

- Timezone mismatches: prefer storing dates in UTC and formatting at the client. `startTime` / `endTime` use HH:MM 24-hour local times; ensure consistency when mixing with `date`.
- Partial failures in bulk updates: always return both successes and errors, as in Users bulk role updates.
- Network flakiness: front-end should display clear retry prompts and not duplicate bookings.

Glossary:

- Availability: declared windows when a tutor is open for sessions; not yet a session.
- Session: a concrete bookable offering tied to a date and time window.

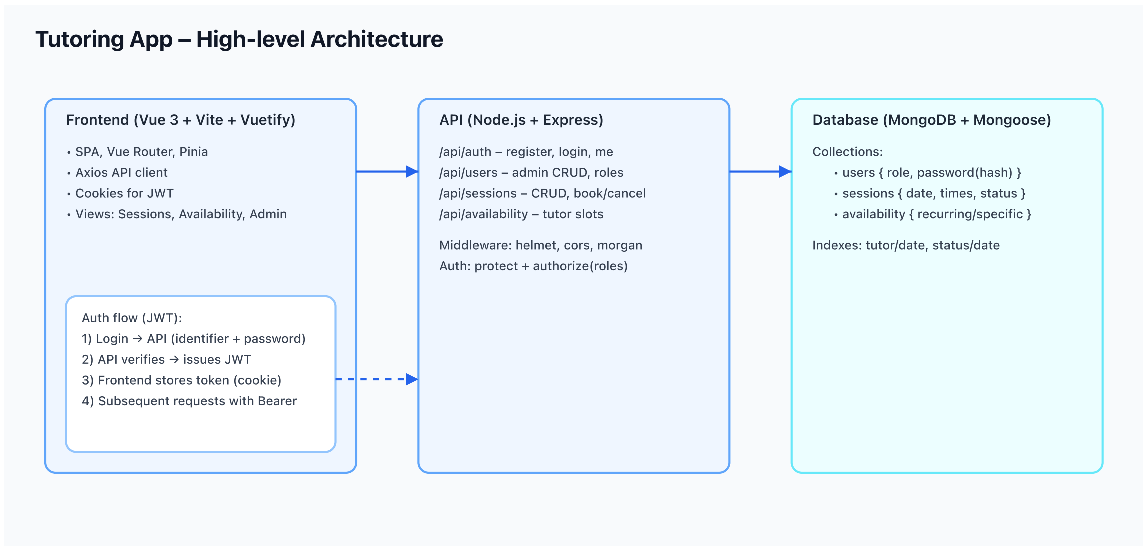
- Booking: the action of assigning a student to an available session (status becomes `booked`).

Troubleshooting:

- 401 on every request: check that `JWT_SECRET` matches between token issuance and verification; ensure Authorization header is attached by Axios.
- CORS blocked: verify `origin` in `server.js` matches the dev server or production domain.
- Mongo connection fails: verify `MONGODB_URI`, network access, and that the cluster allows your IP.

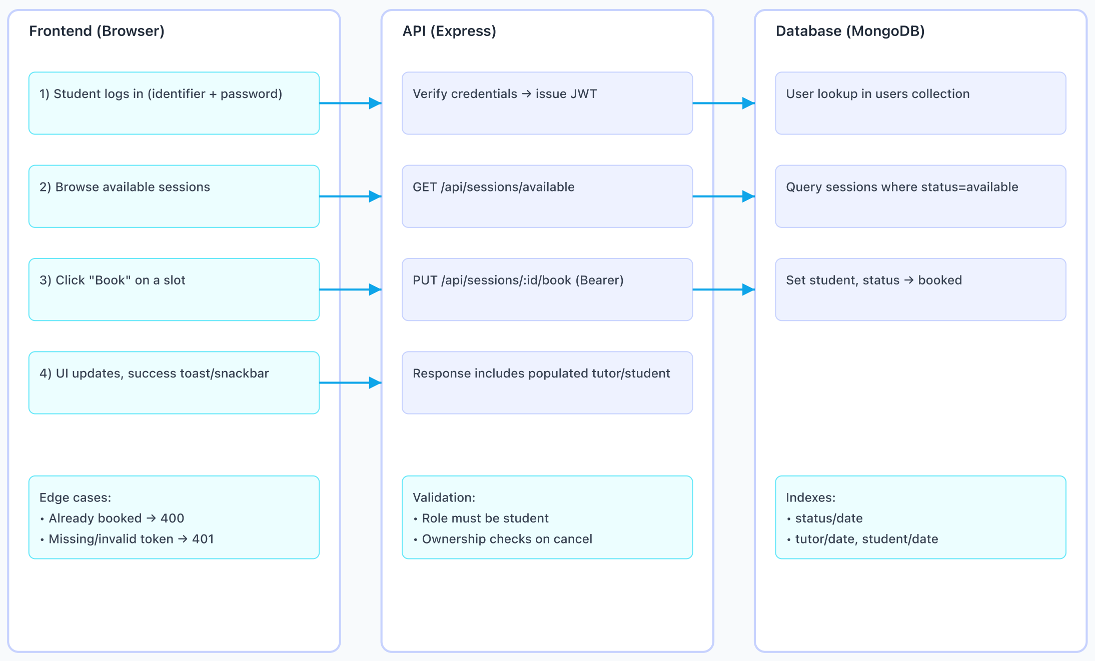
18. Diagrams

Embed preview of `architecture.svg` when rendered on Git platforms:


Tutoring App – High-level Architecture
Frontend (Vue 3 + Vite + Vuetify)
• SPA, Vue Router, Pinia
• Axios API client
• Cookies for JWT
• Views: Sessions, Availability, Admin
Auth flow (JWT):
1) Login → API (identifier + password)
2) API verifies → issues JWT
3) Frontend stores token (cookie)
4) Subsequent requests with Bearer
API (Node.js + Express)
/api/auth – register, login, me
/api/users – admin CRUD, roles
/api/sessions – CRUD, book/cancel
/api/availability – tutor slots
Middleware: helmet, cors, morgan
Auth: protect + authorize(roles)
Database (MongoDB + Mongoose)
Collections:
• users { role, password(hash) }
• sessions { date, times, status }
• availability { recurring/specific }
Indexes: tutor/date, status/date

Additional: booking flow

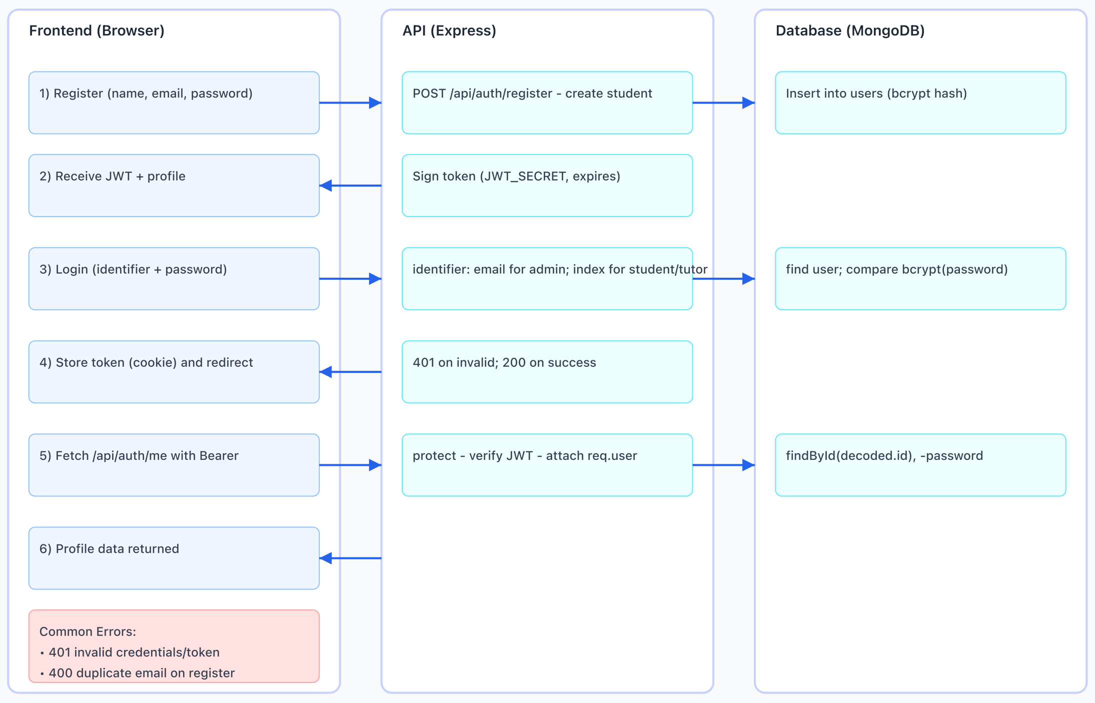
Booking Flow – Student books a session



Frontend (Browser)
API (Express)
Database (MongoDB)
1) Student logs in (identifier + password)
Verify credentials → issue JWT
User lookup in users collection
2) Browse available sessions
GET /api/sessions/available
Query sessions where status=available
3) Click "Book" on a slot
PUT /api/sessions/:id/book (Bearer)
Set student, status → booked
4) UI updates, success toast/snackbar
Response includes populated tutor/student
Edge cases:
• Already booked → 400
• Missing/invalid token → 401
Validation:
• Role must be student
• Ownership checks on cancel
Indexes:
• status/date
• tutor/date, student/date

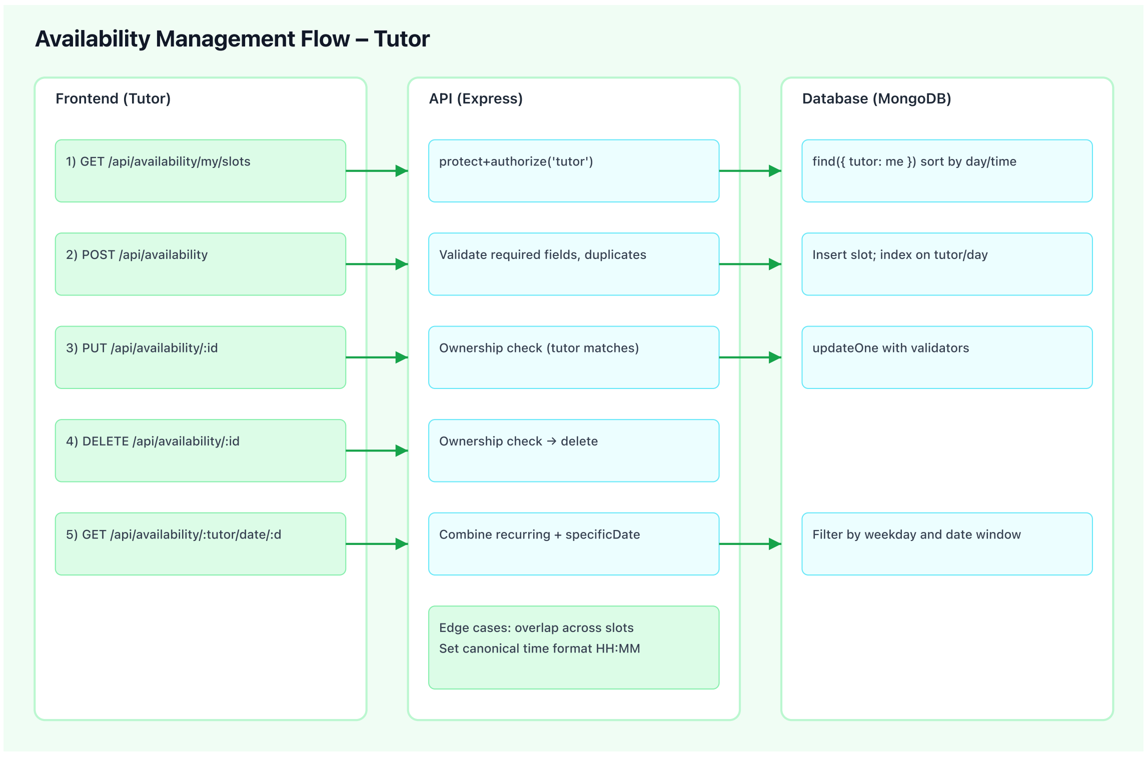
Authentication

Authentication Flow - Register and Login

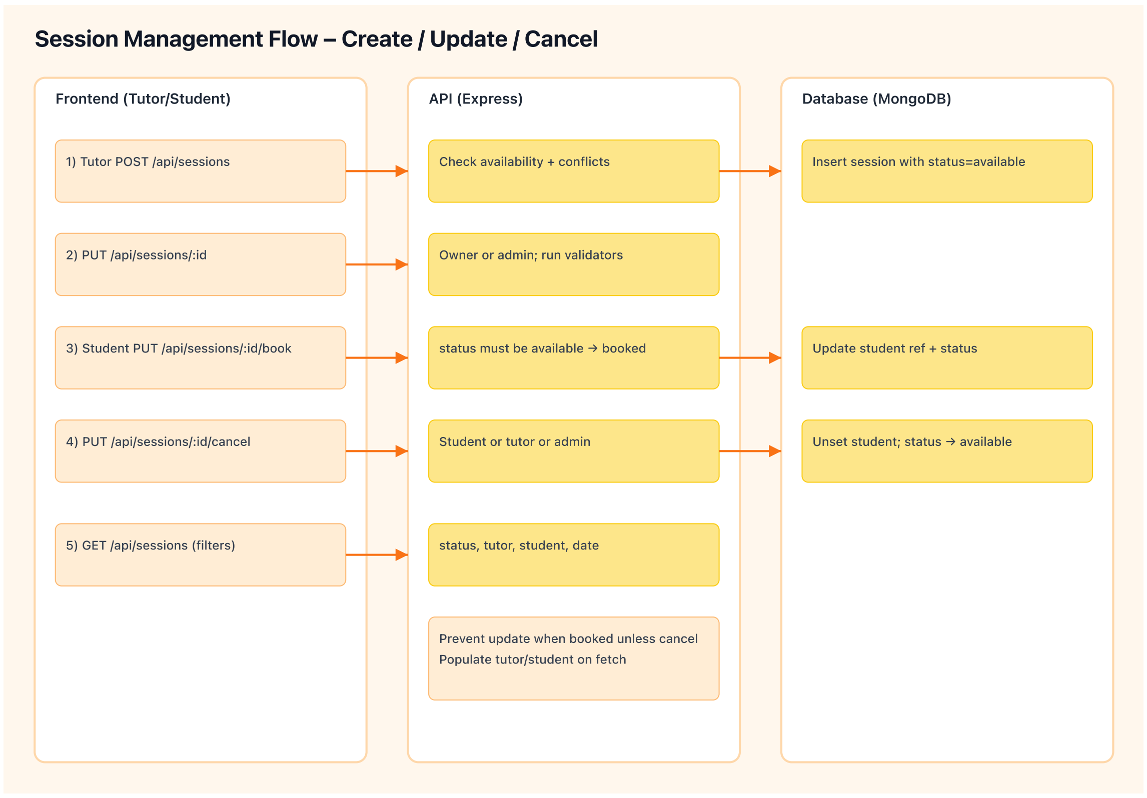


Frontend (Browser)
API (Express)
Database (MongoDB)
1) Register (name, email, password)
POST /api/auth/register - create student
Insert into users (bcrypt hash)
2) Receive JWT + profile
Sign token (JWT_SECRET, expires)
3) Login (identifier + password)
identifier: email for admin; index for student/tutor
find user; compare bcrypt(password)
4) Store token (cookie) and redirect
401 on invalid; 200 on success
5) Fetch /api/auth/me with Bearer
protect - verify JWT - attach req.user
findById(decoded.id), -password
6) Profile data returned
Common Errors:
• 401 invalid credentials/token
• 400 duplicate email on register

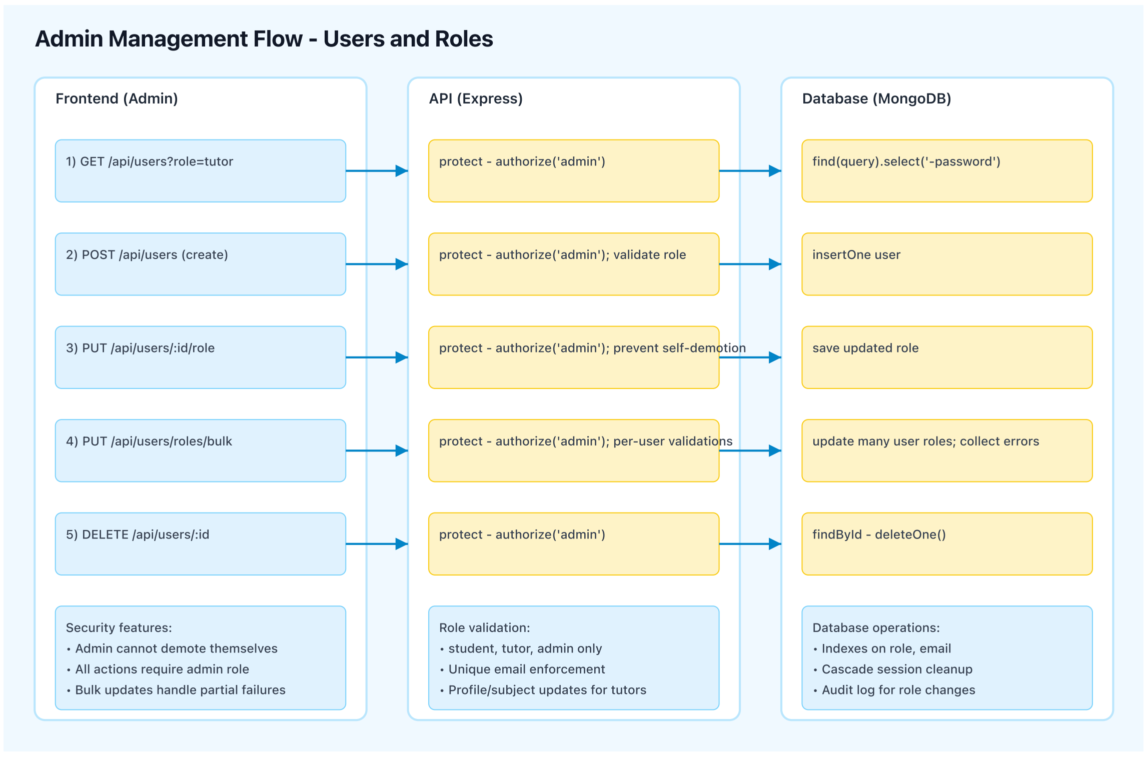
Availability management


Availability Management Flow – Tutor
Frontend (Tutor)
1) GET /api/availability/my/slots
2) POST /api/availability
3) PUT /api/availability/:id
4) DELETE /api/availability/:id
5) GET /api/availability/:tutor/date/:d
API (Express)
protect+authorize('tutor')
Validate required fields, duplicates
Ownership check (tutor matches)
Ownership check → delete
Combine recurring + specificDate
Edge cases: overlap across slots
Set canonical time format HH:MM
Database (MongoDB)
find({ tutor: me }) sort by day/time
Insert slot; index on tutor/day
updateOne with validators
Filter by weekday and date window

Session lifecycle


Session Management Flow – Create / Update / Cancel
Frontend (Tutor/Student)
1) Tutor POST /api/sessions
2) PUT /api/sessions/:id
3) Student PUT /api/sessions/:id/book
4) PUT /api/sessions/:id/cancel
5) GET /api/sessions (filters)
API (Express)
Check availability + conflicts
Owner or admin; run validators
status must be available → booked
Student or tutor or admin
status, tutor, student, date
Prevent update when booked unless cancel
Populate tutor/student on fetch
Database (MongoDB)
Insert session with status=available
Update student ref + status
Unset student; status → available

Admin users & roles


Admin Management Flow - Users and Roles
Frontend (Admin)
API (Express)
Database (MongoDB)
1) GET /api/users?role=tutor
protect - authorize('admin')
find(query).select('-password')
2) POST /api/users (create)
protect - authorize('admin'); validate role
insertOne user
3) PUT /api/users/:id/role
protect - authorize('admin'); prevent self-demotion
save updated role
4) PUT /api/users/roles/bulk
protect - authorize('admin'); per-user validations
update many user roles; collect errors
5) DELETE /api/users/:id
protect - authorize('admin')
findById - deleteOne()
Security features:
• Admin cannot demote themselves
• All actions require admin role
• Bulk updates handle partial failures
Role validation:
• student, tutor, admin only
• Unique email enforcement
• Profile/subject updates for tutors
Database operations:
• Indexes on role, email
• Cascade session cleanup
• Audit log for role changes