Build a production-ready, end-to-end web app where users log in, set a present location and destination (airports), and see ticket prices computed from fuel prices + distance using multiple graph path algorithms (Dijkstra/A*). The map must visualize airports as nodes and the chosen path as highlighted edges. Prices and availability update in real time via Socket.IO. MongoDB stores users, airports, routes, bookings, and dynamic price factors. Include an admin dashboard to view customer data, bookings, and tweak global price factors (e.g., fuel price).

Tech stack & standards (must use):

- **Frontend:** React + Vite + TypeScript, Tailwind CSS, React Router, React Query, **Leaflet** for maps, **socket.io-client**.
- **Backend:** Node.js + Express + TypeScript, **Socket.IO**, **Mongoose** (MongoDB), JWT auth, bcrypt, Zod (validation), dotenv, helmet, cors, morgan, winston logger, rate limiting.
- **Algorithms:** Haversine distance; Graph library (custom) with **Dijkstra** and **A*** (heuristic = haversine to destination). Toggle algorithm via UI.
- **DB:** MongoDB with indexes. Seed script creates ~10 airports + route edges.
- **Ops:** Docker Compose (api + mongo), ESLint + Prettier, Jest/Vitest tests (unit for algorithms, integration for routes), GitHub-style project structure.
- **UX requirements:** Mobile-friendly, dark/light, map shows markers + polylines, live price updates on fuel/route changes, clear loading/error states.
- **Security:** JWT (access + refresh), password hashing, input validation, rate limits on auth & pricing endpoints, CORS configured.
- **Docs:** README with setup/run, .env.example, API docs (OpenAPI/Swagger).

Features (must implement):

1. **Auth:** Register/Login/Logout, refresh token rotation.
2. **Search & Price:** User selects source/destination airports → backend computes candidate paths using chosen algorithm → pricing engine returns per-path fare breakdown (base + distance + fuel + taxes + demand factor). User sees options and selects a flight/route.
3. **Map:** Leaflet map shows all airports (markers) and highlights the computed route polyline. Clicking a node shows airport meta.

4. **Real-time:** Socket.IO events for:
    - `price:update` (emitted when admin changes fuel price)
    - `route:recomputed` (when a route recalculation happens)
    - `booking:created`
5. **Booking flow:** After price selection, create booking → show ticket summary (PNR), store in MongoDB. My Bookings page.
6. **Admin dashboard:** Manage airports, routes, global price factors (fuel price), see real-time bookings and customers.
7. **Tests:**
    - Unit: haversine, Dijkstra, A* return expected paths on small graphs.
    - Integration: `/api/price/quote` and `/api/route/compute` validate payloads & return correct shape.
8. **Seed data:** `seed.ts` with airports (code, name, lat, lon) and edges (from, to, distance or derived). Example: DEL, BOM, BLR, HYD, MAA, CCU, PNQ, GOI, AMD, COK.

API (minimum):

- `POST /api/auth/register`, `POST /api/auth/login`, `POST /api/auth/refresh`, `POST /api/auth/logout`
- `GET /api/airports` (list), `GET /api/airports/:code`
- `POST /api/route/compute` body: `{from, to, algo: "dijkstra" | "astar"}` → returns `{path:[codes], segments:[{from,to,distance}], totalDistance}`
- `POST /api/price/quote` body: `{path:[codes], pax:int}` → returns array of offers: `{fareBreakdown, totalFare, currency, eta}`

- `POST /api/bookings` body: `{offerId, userNotes?}` → returns `{bookingId, pnr}`

- **Admin:** `GET/POST /api/admin/pricing/fuel`, `POST /api/admin/airports`, `POST /api/admin/routes`

Pricing model (implement):

```
distance_km = sum(segmentDistance_km)
fuel_cost = distance_km * avg_burn_l_per_km * fuel_price_per_litre
base = 1500 INR
ops_fee = 0.08 * (base + fuel_cost)
taxes = 0.18 * (base + fuel_cost + ops_fee)
demand_factor = clamp(0.9 .. 1.5) based on current load (booked seats / capacity)
total = round_to_nearest_10( (base + fuel_cost + ops_fee + taxes) * demand_factor )
```

- `avg_burn_l_per_km` set per aircraft type (seed with defaults).

- Provide at least 3 fare classes with different rules (Saver/Standard/Flex) as separate offers.

Frontend pages (must have):

- `/login`, `/register`

- `/search` (select source/destination, choose algorithm, show results & map)

- `/offers` (price options with fare breakdown)

- `/bookings` (my bookings)

- `/admin` (airports/routes/fuel price, live activity via Socket.IO)

Deliverables:

- Full code, clean UI, Swagger docs, seed data, tests pass, Dockerized, ready to run with `docker compose up`.

How the whole system fits together (step-by-step)

1) Project structure

```
/client (React + TS + Vite + Tailwind)
  src/
    main.tsx, App.tsx, routes/*
    pages/{Login,Register,Search,Offers,Bookings,Admin}.tsx
    components/{Map,AirportSearch,OfferCard,PriceBreakdown,Header}.tsx
    lib/api.ts (axios + interceptors), lib/socket.ts
    types/*
/server (Node + TS + Express + Socket.IO + Mongoose)
  src/
    index.ts (http + socket)
    app.ts (express config)
    config/{env.ts, logger.ts}
    middleware/{auth.ts, validate.ts, error.ts, rateLimit.ts}
    models/{User.ts, Airport.ts, RouteEdge.ts, Booking.ts,
PriceConfig.ts}
    services/{graph.ts, pricing.ts, booking.ts, auth.ts}
    controllers/{authCtrl.ts, routeCtrl.ts, priceCtrl.ts,
bookingCtrl.ts, adminCtrl.ts}
    routes/{auth.ts, route.ts, price.ts, booking.ts, admin.ts}
    utils/{haversine.ts, pnr.ts}
    sockets/{events.ts}
    seed/seed.ts
  prisma/ (not used) | scripts/
docker-compose.yml
.env.example
README.md
```

2) MongoDB data models (with key indexes)

- **User**
    - `_id`, `email` (unique, index), `passwordHash`, `name`, `role: "user" | "admin"`, `createdAt`
- **Airport**
    - `_id`, `code` (unique, index), `name`, `lat`, `lon`, `city`, `country`
    - Index: `{ code: 1 }`, and 2dsphere on `[lon, lat]` if you want geospatial queries.
- **RouteEdge**
    - `_id`, `from` (Airport.code), `to` (Airport.code), `distanceKm` (precomputed), `active: boolean`
    - Index: `{ from: 1, to: 1 }`
- **PriceConfig**
    - singleton `{ fuelPricePerLitre, defaultBurnLPerKm, taxRate, feeRate, baseFare }`
- **Booking**
    - `_id`, `pnr`, `userId`, `path:[Airport.code]`, `fareBreakdown`, `total`, `createdAt`
    - Index: `{ userId: 1, createdAt: -1 }`

3) Graph + algorithms

- Build a directed graph from `RouteEdge` documents.
- **Dijkstra**: minimize total distance.
- **A***: priority = `g(n) + h(n)` where `h(n)` is haversine distance from `n` to destination (admissible).

- Return: `path`, `segments`, `totalDistance`.
- Keep algorithm code in `services/graph.ts` with pure functions and unit tests.

Haversine utility (TypeScript):

```
export function haversineKm(a:{lat:number;lon:number},
b:{lat:number;lon:number}) {
  const R = 6371;
  const dLat = (b.lat - a.lat) * Math.PI/180;
  const dLon = (b.lon - a.lon) * Math.PI/180;
  const lat1 = a.lat * Math.PI/180, lat2 = b.lat * Math.PI/180;
  const s = Math.sin(dLat/2)**2 +
Math.cos(lat1)*Math.cos(lat2)*Math.sin(dLon/2)**2;
  return 2*R*Math.asin(Math.sqrt(s));
}
```

4) Pricing engine

- Inputs: `path`, `segments`, `pax`, `PriceConfig`.
- Steps:
    1. Sum `distanceKm`.
    2. `fuel_cost = distanceKm * burn * fuelPrice`.
    3. `ops_fee = feeRate * (base + fuel_cost)`.
    4. `taxes = taxRate * (base + fuel_cost + ops_fee)`.
    5. `demand_factor` from current occupancy on that path (seed static capacity; later, infer from recent bookings).
    6. Return 3 offers (Saver/Standard/Flex) with modifier multipliers like 0.95, 1.0, 1.15.
- Emit `price:update` via sockets when admin changes fuel price.

5) REST endpoints (example contracts)

- **Compute route**
  - `POST /api/route/compute`
  - Body: `{ from: "DEL", to: "BOM", algo: "dijkstra" | "astar" }`
  - 200: `{ path: ["DEL","BOM"], segments: [{from,to,distanceKm}], totalDistance: number }`
- **Price quote**
  - `POST /api/price/quote`
  - Body: `{ path: ["DEL","BOM"], pax: 1 }`
  - 200: `[{ offerId, class: "Saver"|"Standard"|"Flex", fareBreakdown, totalFare, currency: "INR" }]`
- **Booking**
  - `POST /api/bookings` (auth required)
  - Body: `{ offerId }`
  - 200: `{ bookingId, pnr, total }`

6) Socket.IO events (server ↔ client)

- Server emits:
  - `price:update` → `{ fuelPricePerLitre }`
  - `route:recomputed` → `{ from, to, algo, path, totalDistance }`
  - `booking:created` → `{ pnr, userName, path, total }`
- Client emits:

- `route:request` → `{ from, to, algo }` (server computes & broadcasts)
- `subscribe:admin` (join admin room for dashboards)

7) Frontend UX flow

1. **Auth pages** with form validation (Zod) → receive tokens; axios interceptor attaches JWT.
2. **Search page**
   - Airport autocomplete (type→filter local list from `/api/airports`).
   - Algo dropdown (Dijkstra/A*).
   - Button “Compute Route & Prices”.
   - Map (Leaflet): show markers for all airports, draw polyline for returned `segments`.
   - Right panel lists offers with fare breakdown; selecting one goes to “Confirm”.
3. **Offers page** shows 3+ offers (cards), “Book” button → POST booking.
4. **Bookings page** lists user bookings.
5. **Admin page**
   - Update fuel price (number input) → PUT → emits `price:update`.
   - CRUD airports/routes (simple forms + tables).
   - Live activity feed via `booking:created`.

8) Leaflet map essentials (frontend)

- Use OSM tiles: `https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png`
- Markers: airports; on selection, pan/zoom.

- Polyline: build from airport lat/lon along path order.

Map snippet (React):

```
import { MapContainer, TileLayer, Marker, Popup, Polyline } from
'react-leaflet';

export default function RouteMap({ airports, path }) {
  const points = path.map(c => {
    const a = airports.find(x => x.code === c)!;
    return [a.lat, a.lon] as [number, number];
  });
  return (
    <MapContainer center={[22.97, 79.59]} zoom={5} className="h-96
rounded-2xl">
      <TileLayer
url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png" />
      {airports.map(a => (
        <Marker key={a.code} position={[a.lat, a.lon]}>
          <Popup>{a.code} — {a.name}</Popup>
        </Marker>
      ))}
      {points.length > 1 && <Polyline positions={points} />}
    </MapContainer>
  );
}
```

9) Auth & security highlights

- Passwords: bcrypt hash with salt (12 rounds+).
- JWT: short-lived access (15m) + refresh (7d) with rotation & blacklist table if desired.
- Rate limit: `/api/auth/login` & `/api/price/quote`.
- Input validation: Zod schemas in controllers.
- Helmet + CORS configured for client origin.

10) Example backend snippets

Socket.IO bootstrap (`index.ts`):

```
const app = express();
const server = http.createServer(app);
const io = new Server(server, { cors: { origin:
process.env.CLIENT_ORIGIN, credentials: true } });

io.on('connection', socket => {
  socket.on('route:request', async ({ from, to, algo }) => {
    const result = await computeRoute(from, to, algo);
    io.emit('route:recomputed', { from, to, algo, ...result });
  });
});
export { io, server };
```

Route compute controller:

```
export const computeRouteHandler = async (req: Request, res: Response)
=> {
  const { from, to, algo } = routeSchema.parse(req.body);
  const result = await graphService.computeRoute(from, to, algo);
  res.json(result);
};
```

Pricing service (sketch):

```
export function priceQuote(path: string[], pax: number, cfg:
PriceCfg): Offer[] {
  const totalKm = totalDistanceForPath(path);
  const fuelCost = totalKm * cfg.defaultBurnLPerKm *
cfg.fuelPricePerLitre;
  const base = cfg.baseFare;
  const ops = cfg.feeRate * (base + fuelCost);
  const taxes = cfg.taxRate * (base + fuelCost + ops);
  const demand = demandFactorForPath(path); // 0.9..1.5
  const core = (base + fuelCost + ops + taxes) * demand;
```

```
  const classes = [
    { name: 'Saver', mul: 0.95 },
    { name: 'Standard', mul: 1.0 },
    { name: 'Flex', mul: 1.15 },
  ];
  return classes.map((c) => ({
    offerId: nanoid(),
    class: c.name,
    fareBreakdown: { base, fuelCost, ops, taxes, demand },
    totalFare: round10(core * c.mul) * pax,
    currency: 'INR'
  }));
}
```

11) Seed data (example airports)

(You can add more; these are enough to demo.)

```
[
  {"code":"DEL","name":"Indira Gandhi
Intl","lat":28.556,"lon":77.100},
  {"code":"BOM","name":"Chhatrapati
Shivaji","lat":19.089,"lon":72.865},
  {"code":"BLR","name":"Kempegowda","lat":13.198,"lon":77.706},
  {"code":"HYD","name":"Rajiv Gandhi","lat":17.24,"lon":78.43},
  {"code":"MAA","name":"Chennai Intl","lat":12.99,"lon":80.17},
  {"code":"CCU","name":"Netaji Subhas
Chandra","lat":22.65,"lon":88.44},
  {"code":"PNQ","name":"Pune","lat":18.58,"lon":73.92},
  {"code":"GOI","name":"Goa","lat":15.38,"lon":73.83},
  {"code":"AMD","name":"Ahmedabad","lat":23.07,"lon":72.63},
  {"code":"COK","name":"Cochin Intl","lat":10.15,"lon":76.40}
]
```

For `RouteEdge`, connect sensible pairs (e.g., DEL↔BOM, DEL↔BLR, BOM↔GOI, BLR↔MAA, HYD↔MAA, CCU↔DEL, CCU↔MAA, PNQ↔BOM, AMD↔BOM, COK↔MAA). Precompute `distanceKm` via haversine.

12) Running locally (Docker)

- `docker-compose.yml` services:
    - `api` (Node 20-alpine), `mongo` (latest), optional `mongo-express`.

`server/.env.example`:

```
 PORT=4000
MONGO_URI=mongodb://mongo:27017/airline
JWT_SECRET=change-me
JWT_REFRESH_SECRET=change-me-too
CLIENT_ORIGIN=http://localhost:5173
```

-
- Commands:
    - `docker compose up --build`
    - `pnpm --filter server seed` (or `npm run seed`)
    - Open client at `http://localhost:5173`

13) Testing checklist

- **Algorithms:** small fixed graph → Dijkstra and A* return the same shortest path; A* expands fewer nodes.
- **API:** invalid codes rejected; missing auth on bookings blocked; prices recompute when fuel changes.
- **Sockets:** admin fuel change triggers `price:update` and UI badge “prices refreshed”.

14) Stretch ideas

- Airline schedules & time-dependent edges (flight time vs. price).
- Multi-criteria shortest path (Pareto frontier: price vs. duration).

- Caching popular routes.
- Payments sandbox (Stripe test mode) with webhooks.
- Geospatial MongoDB queries to auto-suggest nearest airport by user geolocation.