Courtify Platform Structure

Tech Stack Justification

Front-End (React + Vite or HTML/CSS/JS build):

- Component-based UI for filters, slot search, and confirmation flows; fast builds and solid Core Web Vitals for SEO/discoverability.
- Smooth client-side updates without full page reloads for a better booking user experience.

Back-End (Node.js + Express):

- Lightweight REST services for /services and /bookings, with easy validation and error handling.
- Non-blocking I/O suits small transactional loads typical of booking MVPs.

Database (MongoDB Atlas – Free Tier):

- Stores services and bookings as JSON-like documents, which is a natural fit for Node and React.
- Cloud-hosted persistence with easy scaling and zero local operations burden.

External API (live):

- Pulls availability, maps, distance, or other relevant data to enrich results.
  Implements graceful handling for timeouts, rate limits, and empty responses.

Fit to GC1 business concept:

- Courtify needs real-time slot discovery and reliable booking storage; this stack delivers a deployable, low-cost solution today and scales toward authentication, payments, and analytics later.

Hosting & Deployment Plan

- Front-End: Vercel or Netlify (free tier) with auto-deploy from Git. Global CDN and HTTPS for performance and SEO optimization.
- Back-End API: Render or Railway (free tier) with a public HTTPS endpoint consumed by the front-end.
- Database: MongoDB Atlas free tier for persistent cloud-based data storage accessible through the backend connection string.
- Access: Final live URLs will be placed in the README.md and submission PDF. Environment variables managed through each hosting platform's dashboard.

<u>Data Flow Description</u>

Browse & Filter

1. The user loads Courtify, and the front-end sends a GET /services request with filter parameters such as location, indoor/outdoor, date range, and time.
2. The back-end queries the database and/or external API to assemble the availability payload.
3. The API returns JSON, and the UI renders service cards with dates, times, and prices.

Book a Slot

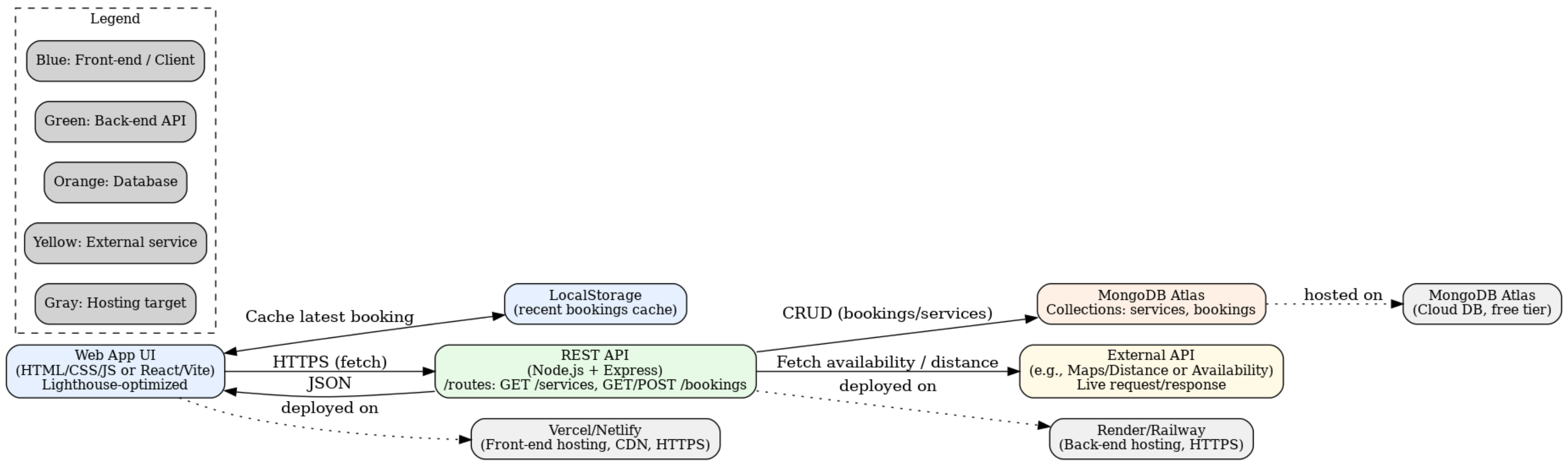
4. The user selects a slot, and the front-end sends a POST /bookings request with { serviceId, date, time, contact? }.
5. The back-end validates the payload and, on success, saves the record to MongoDB Atlas (collection: bookings).
6. The API returns confirmation JSON { bookingId, status, details }, and the UI shows a confirmation screen.

View Booking History

7. On app revisit, the front-end sends a GET /bookings request to fetch stored records.
8. The UI displays booking history, and LocalStorage can cache the latest booking for quick display.

<u>Error Handling</u>

- If /services or /bookings fail, the UI shows friendly fallback messages such as “Failed to load services (server offline?)” or “Error loading bookings.”
- The API standardizes errors as { error: string, code: number } for consistent UI messaging.
- Rate limiting and input validation help mitigate abuse, and CORS allowlists restrict requests from unapproved origins.


Legend
Blue: Front-end / Client
Green: Back-end API
Orange: Database
Yellow: External service
Gray: Hosting target
Web App UI (HTML/CSS/JS or React/Vite) Lighthouse-optimized
Cache latest booking
LocalStorage (recent bookings cache)
HTTPS (fetch)
JSON
REST API (Node.js + Express) /routes: GET /services, GET/POST /bookings
deployed on
Vercel/Netlify (Front-end hosting, CDN, HTTPS)
CRUD (bookings/services)
MongoDB Atlas Collections: services, bookings
hosted on
MongoDB Atlas (Cloud DB, free tier)
Fetch availability / distance
External API (e.g., Maps/Distance or Availability) Live request/response
deployed on
Render/Railway (Back-end hosting, HTTPS)