Employee Directory with Search

A simple, full-featured **Employee Directory** project with fast search and filters. This README explains the app structure, how to run it locally, search behavior (client & server), API endpoints, deployment tips, and how to extend the project.

Table of Contents

Features

- List employees with profile photo, name, role, department, location, email, phone.
- Instant search (name, role, department, skills).
- Filters: department, location, role.
- Sort: name, join date, role.
- Pagination / infinite scroll for large datasets.
- Responsive UI (desktop & mobile).
- Optional server-side full-text search and indexing for large orgs.

Tech Stack (example)

- Frontend: React + Tailwind CSS
- State management: React Context / hooks
- Backend (optional): Node.js + Express or Firebase / Supabase
- Database: PostgreSQL (with `pg_trgm` for fuzzy search) or MongoDB
- Search: simple SQL `ILIKE` / full-text or Elasticsearch for very large teams
- Tests: Jest + React Testing Library

Project Structure (example)

```
employee-directory/
├─ client/                # React app
│  ├─ src/
│  │  ├─ components/
│  │  │  ├─ EmployeeCard.jsx
│  │  │  ├─ SearchBar.jsx
│  │  │  └─ FiltersPanel.jsx
│  │  ├─ pages/
│  │  └─ context/
│  └─ public/
├─ server/                # Optional Express API
│  ├─ routes/
│  ├─ controllers/
│  └─ db/
├─ mock-data/             # sample JSON for seeding
└─ README.md
```

Getting Started

Prerequisites

- Node.js >= 16
- npm or yarn
- (Optional) PostgreSQL / MongoDB if using server and persistent storage

Install

From repository root:

```
# install client deps
cd client
npm install

# if using server
cd ../server
npm install
```

Run (development)

Frontend (client-only mode with mock data):

```
cd client
npm start
```

With backend (server + client):

```
# start server
cd server
npm run dev

# in new shell start client
cd ../client
npm start
```

Build (production)

```
cd client
npm run build
# serve `client/build` from your preferred static host or from Express
```

Sample Data

Store a `mock-data/employees.json` file (used by client in dev or to seed the DB):

```
[
  {
    "id": "1",
    "name": "Aisha Patel",
    "role": "Frontend Engineer",
```

```
    "department": "Engineering",
    "location": "Bengaluru, India",
    "email": "aisha.patel@example.com",
    "phone": "+91-98xxxxxxx",
    "skills": ["React", "TypeScript", "Tailwind"],
    "joinedAt": "2022-06-12",
    "avatar": "/avatars/aisha.jpg"
  }
]
```

Usage / Search Behavior

Client-side Search

- Best for small datasets (< 1,000 items).
- Implementation:
- Load JSON into client state on mount.
- Keep a search input with `onChange` that updates a `query` state.
- Use `useMemo` to compute filtered results: filter by `name`, `role`, `department`, `skills`.
- Provide highlighting on matches (e.g., wrap matched substring in `<mark>`).

Server-side Search & Pagination

- Required for larger orgs or when you want to avoid shipping full DB to clients.
- Endpoint example: `GET /api/employees?q=react&page=2&perPage=25&department=Engineering&sort=name`.
- Server should:
- Sanitize inputs.
- Support pagination (offset/limit or cursor-based).
- Use database-level indexes (GIN, trigram) for fast searches.

Debounce, Throttling & UX

- Debounce user input (e.g., 250–400ms) to reduce repeated requests.
- Show loading spinner when fetching server results.
- Support keyboard navigation of search results and accessible ARIA attributes.

Fuzzy / Advanced Search

- For fuzzy matching use PostgreSQL `pg_trgm` or integrate a search engine (Elasticsearch, Typesense, MeiliSearch).
- Offer filters for exact fields and full-text search for `skills` and `bio`.

API Endpoints (example)

```
GET /api/employees?
q=<query>&department=<dept>&location=<loc>&page=<n>&perPage=<m>&sort=<field>
GET /api/employees/:id
POST /api/employees         # create
PUT /api/employees/:id     # update
DELETE /api/employees/:id  # delete
```

Responses should return `{ data: [...], meta: { total, page, perPage } }`.

UI / Component Notes

- SearchBar component:
- Controlled input, clear button, optional voice input.
- Shows recent queries / suggestions.
- EmployeeCard:
- Compact layout with avatar, name, title, and quick-actions (email, call, view profile).
- FiltersPanel:
- Multi-select for departments and locations, with counts.
- Accessibility:
- Use semantic HTML, aria attributes, and ensure keyboard focus states.

Testing

- Unit test components with Jest + React Testing Library.
- Test search logic separately (pure functions). Example tests:
- `filters by name`.
- `applies department filter`.
- `pagination returns correct slice`.
- E2E tests: Playwright or Cypress to test full search behavior and filtering.

Deployment

- Client: Vercel, Netlify, or static host from build folder.
- Server: Heroku, Railway, Render, or self-host on a VM.
- Database: Managed Postgres (Supabase, Railway) or MongoDB Atlas.
- For heavy search traffic, consider Typesense / MeiliSearch or hosted Elasticsearch.

Contributing

1. Fork the repo
2. Create a feature branch `feat/<description>`
3. Run tests and linters
4. Open a PR describing changes

License

MIT © Your Name

If you'd like, I can also:

- Add a ready-to-run React + Express starter with the search already wired up.
- Produce example component code (SearchBar, EmployeeCard) and server sample (Express routes + Postgres queries).