**Windsurf.AI**

**Windsurf rules:**

**1) do-not-use-double-ampersand**

- Don't use double ampersand (&&) in terminal

**2) windsurf-rules-for-react-fastapi-postgresql**

#Windsurf AI Rules: React + TypeScript (Vite), FastAPI, PostgreSQL, Tailwind CSS

##Code Style & Structure

- Use functional, declarative programming (avoid classes).

- Separate concerns: UI, state, business logic, and API interactions.

- Follow DRY principles; extract reusable utilities and components.

- Prefer RORO pattern: Receive an Object, Return an Object.

- Use feature-based folder structure on both frontend and backend.

- write responsive design.

---

##Project Structure

### Frontend (`vite + react`)

```

src/

components/

pages/

hooks/

context/

utils/

types/

assets/

```

### Backend (`fastapi`)

```

app/

routers/

services/

models/

schemas/

db/

core/

middleware/

```

---

##Naming Conventions

- camelCase for variables/functions (TS).

- PascalCase for React components/types/interfaces.

- snake\_case for Python variables, files, and database objects.

- UPPER\_CASE for constants and environment variables.

- Folder names: lowercase with dashes (e.g., `user-profile`).

---

##Syntax & Formatting

- Format TypeScript using Prettier.

- Format Python using Black or Ruff.

- Use optional chaining (`?.`) and nullish coalescing (`??`) in TypeScript.

- Use one-liners for simple `if` statements.

- Avoid excessive nesting; use early returns.

---

##TypeScript Best Practices

- Enable strict mode in `tsconfig.json`.

- Use `interface` for props and models; use `type` for unions.

- Avoid `any`; use specific types or `unknown`.

- Use readonly and utility types (`Partial`, `Pick`, `Record`) when applicable.

- Always type props, API responses, and hooks.

---

### Styling & UI

- Use Tailwind CSS for styling.

- Use Shadcn UI for components.

- Use utility-first styling throughout the app.

- Use responsive design with `sm:`, `md:`, `lg:`, etc.

- Implement dark mode with `dark:` modifier.

- Use `@apply` sparingly for reused classes.

- Keep classNames readable; break long chains into multiple lines if needed.

---

##FastAPI Guidelines

- Use `async def` for all route handlers.

- Organize routes by feature using routers (e.g., `user\_router`, `auth\_router`).

- Use `Depends()` for services and dependency injection.

- Validate all inputs and outputs using Pydantic v2 `BaseModel`.

- Prefer lifespan context managers over `@app.on\_event`.

- Always check for an virtual environment before creating an new one.

---

##PostgreSQL Guidelines

- Use SQLAlchemy 2.0 (sync or async).

- Use snake\_case for table and column names.

- Index foreign keys and frequently queried columns.

- Avoid raw SQL unless necessary; parameterize all queries.

---

##API Design

- Use RESTful route naming (`/users/{id}/posts`, `/orders/{order\_id}`).

- Return typed responses using Pydantic models.

- Implement pagination and filtering in list endpoints.

- Use `status\_code` and raise `HTTPException` for errors.

- Automatically document APIs using FastAPI’s OpenAPI.

---

##Performance Optimization

- Lazy load routes/components in React with `React.lazy` + `Suspense`.

- Use `useMemo`, `useCallback` to prevent re-renders.

- Optimize images using `vite-imagetools` or external CDN.

- Use async DB calls and avoid blocking I/O in FastAPI.

- Add indexes for common queries in PostgreSQL.

---

##Security

- Sanitize user inputs both client and server-side.

- Use HTTPS in production with secure cookies & CSRF tokens.

- Restrict CORS in FastAPI.

- Store secrets in `.env`, and never commit `.env`.

- Escape all raw SQL or use parameterized queries.

- Use `react-helmet` or meta headers to protect frontend.

---

##Internationalization (i18n)

- Use `react-i18next` with JSON-based translation files.

- Parse `Accept-Language` header in FastAPI for locale-aware responses.

- Support RTL layouts and dynamic font switching.

- Ensure all text is scaleable and not hardcoded.

---

##State Management (Frontend)

- Use `useState` and `useReducer` for local state.

- Use `React Context` for lightweight global state.

- Use `Zustand` or `React Query` for async/server state.

- Avoid overusing Redux unless necessary.

---

##Error Handling

- Use `ErrorBoundary` in React for global error catching.

- Use try-catch in async functions.

- Return typed `HTTPException` from FastAPI with helpful messages.

- Log errors using services like Sentry or LogRocket.

- Display fallback UI or toast notifications on failure.

---

##API Integration (Frontend)

- Use `fetch` or `axios` wrapped in a `useApi` hook.

- Use `AbortController` to cancel unused requests.

- Handle loading, success, and error states explicitly.

- Centralize API base URL and endpoints in one config file.

---

##Dev & Deployment

- Use `.env.local`, `.env.production` for Vite environments.

- Set up CORS and secure headers properly.

- Deploy via Vercel (frontend) and Railway/Fly.io/Render (backend).

- Use Nginx or Caddy as a reverse proxy in production.

---

##Accessibility

- Use semantic HTML (`<button>`, `<label>`, `<input>`).

- Add `aria-\*` attributes and `role` when needed.

- Ensure all interactive elements are keyboard-navigable.

- Respect system theme (light/dark) and font scaling.

---

##Documentation

- Auto-generate API docs with FastAPI’s Swagger UI.

- Comment all public functions and exports in TypeScript.

- Maintain a `README.md` with setup instructions and `.env` variables.

- Use `Typedoc` (optional) for frontend type documentation.

---

##Key Rules Summary

1. Use \*\*Vite\*\* for fast builds, HMR, and optimized output.

2. Use \*\*FastAPI + Pydantic\*\* for declarative, typed APIs.

3. Manage global state wisely: minimal context, prefer API cache layers.

4. Follow \*\*utility-first\*\* styling with Tailwind CSS.

5. Enforce \*\*code quality\*\* via Prettier, Ruff, and Black.

6. Follow \*\*responsive, accessible, and secure\*\* design patterns.

7. Monitor performance and UX with appropriate dev tools.

8. Use \*\*OpenAPI\*\* and \*\*typed errors\*\* for a stable developer experience.

**Windsurf Memories:**

**1) DB Schema: Open Questions & Recommendations:**

Open Questions

- Appraisal.appraisee\_id duplicated in input; confirm single field (FK to employees.emp\_id).

- appraisal\_setting\_id: which table does this reference? Need schema for settings.

- appraisal\_ranges.start\_date/end\_date are ints: are these month or day offsets? Consider naming (e.g., start\_month\_offset) or store only label and compute actual dates into Appraisals.start\_date/end\_date.

- Goals vs AppraisalGoals: per-goal self/appraiser comments/ratings currently in Goals (global). Should be per appraisal. Move these fields to AppraisalGoals (and possibly rename to appraisal\_goals) or create appraisal\_goal\_evaluations.

- goal\_important vs goal\_importance: confirm canonical name and allowed values (High/Medium/Low).

- Categories are multi-select for templates: model as separate Categories table + join table (goal\_template\_categories).

- Rating scale bounds: 1–5 or 1–10? Integers only or allow decimals?

- Enforce total weightage = 100% per appraisal: implement via DB trigger/constraint strategy.

- Reviewer: overall comments/rating only (no per-goal)? Confirm.

Recommendations

- Use snake\_case for all identifiers; add created\_at/updated\_at (timestamptz) and created\_by where useful.

- Use PostgreSQL enums: appraisal\_status (draft, submitted, self\_assessment, appraiser\_evaluation, reviewer\_evaluation, complete); role\_level or role as enum.

- Add unique/partial index to prevent overlapping active appraisals for same appraisee/type/range/period.

- Index FKs and frequently filtered columns (status, appraisee\_id, appraiser\_id, reviewer\_id).

- Snapshot goal attributes into appraisal\_goals at assignment time to avoid template drift (store template\_id reference too).

- Set sensible constraints: rating smallint check (between bounds), weightage smallint 0–100, non-null where appropriate.

- Define FK ON DELETE/UPDATE policies (e.g., reporting\_manager SET NULL).

**2) Database Schema: Employees, Appraisals, Goals, Types & Ranges:**

Relational Schema (initial)

i) Employees (Employee)

- emp\_id: int (PK)

- emp\_name: string

- emp\_email: string

- emp\_department: string

- emp\_roles: string // e.g., Intern, Fresher, Developer, Team Lead, Manager, VP, CEO

- emp\_roles\_level: int // e.g., 1..7

- emp\_reporting\_manager: FK -> Employee(emp\_id)

- emp\_status: boolean

ii) Appraisals (Appraisal)

- appraisal\_id: int (PK)

- appraisal\_setting\_id: int

- appraisee\_id: int (FK -> Employee.emp\_id)

- appraiser\_id: int (FK -> Employee.emp\_id)

- reviewer\_id: int (FK -> Employee.emp\_id)

- appraisal\_type\_id: int (FK -> appraisal\_types.id)

- appraisal\_type\_range\_id: int (FK -> appraisal\_ranges.id)

- start\_date: date

- end\_date: date

- status: string // Draft, Submitted, Appraisee Self Assessment, Appraiser Evaluation, Reviewer Evaluation, Complete

- appraiser\_overall\_comments: string

- appraiser\_overall\_rating: int

- reviewer\_overall\_comments: string

- reviewer\_overall\_rating: int

iii) AppraisalGoals

- id: serial (PK)

- appraisal\_id: int (FK -> Appraisal.appraisal\_id)

- goal\_id: int (FK -> Goals.goal\_id)

iv) appraisal\_types

- id: integer (PK, autoincrement)

- name: varchar(100) not null // e.g., Annual, Half-yearly, Quarterly, Tri-annual, Project-end, Annual-Probation

- has\_range: boolean default false

v) appraisal\_ranges

- id: integer (PK, autoincrement)

- appraisal\_type\_id: integer not null (FK -> appraisal\_types.id)

- name: varchar(50) not null // e.g., "1st", "2nd", "3rd", "4th"

- start\_date: int

- end\_date: int

vi) Goals\_Template

- temp\_id: int (PK)

- temp\_title: string

- temp\_description: string

- temp\_performance\_factor: string

- temp\_category: string

- temp\_importance: string // High/Medium/Low

- temp\_weightage: int // percentage

vii) Goals

- goal\_id: int (PK)

- goal\_template\_id: int (FK -> Goals\_Template.temp\_id)

- goal\_title: string

- goal\_performance\_factor: string

- goal\_category: string

- goal\_important: string

- goal\_weightage: int

- self\_comment: string

- self\_rating: int

- appraiser\_comment: string

- appraiser\_rating: int

**3) Appraisal Workflow & Status Transitions**

i) Actors

- Appraiser: Manager or higher; creates appraisals and manages goal templates.

- Appraisee: Employee.

- Reviewer: Final reviewer (Manager level or higher level).

ii) Workflow & Status Transitions

1) Draft: Appraiser creates appraisal (sets appraisee, appraiser, reviewer; adds goals from goal template or by modifying goal template). Status = Draft.

2) Submitted (aka Waiting Acknowledgement): Appraiser sends to appraisee for acknowledgement. Status = Submitted.

3) Appraisee Self Assessment: After acknowledgement, appraisee fills self assessment comments and ratings per goal; on submit fields become read-only; Status -> Appraiser Evaluation.

4) Appraiser Evaluation: Appraiser reviews self assessment; enters per-goal appraiser comments and ratings; adds overall appraiser comment and overall appraiser rating; on submit fields become read-only; Status -> Reviewer Evaluation.

5) Reviewer Evaluation: Reviewer reviews per-goal comments/ratings and appraiser overall; enters reviewer overall comments and reviewer overall rating; on submit fields become read-only; Status -> Complete.

6) Complete: Final State

**4) Permissions Matrix by Role & Status**

i) Appraisee

- Draft: No access (cannot view or edit goals; no actions).

- Submitted: Can view goals; can acknowledge; cannot edit goals.

- Appraisee Self Assessment: Can view goals; can perform self assessment (comments, rating) per goal.

- Appraiser Evaluation: Can view goals; self assessment becomes read-only; cannot see appraiser/reviewer comments.

- Reviewer Evaluation: Can view goals; self assessment read-only; cannot see appraiser/reviewer comments.

- Complete: Can view goals; self/appraiser/reviewer evaluations all read-only; can view all comments and ratings.

ii) Appraiser

- Draft: Can view and edit goals.

- Submitted: Can view goals.

- Appraisee Self Assessment: Can view goals; cannot edit; waits for self assessment.

- Appraiser Evaluation: Can view goals; self assessment read-only; can perform appraiser evaluation (per-goal comments/ratings + overall comments/rating).

- Reviewer Evaluation: Can view goals; self assessment and appraiser evaluation read-only.

- Complete: Can view goals; all evaluations read-only; can view all comments.

iii) Reviewer

- Draft to Appraiser Eval: No access.

- Reviewer Evaluation: Can view goals; self assessment and appraiser evaluation read-only; can perform reviewer evaluation (overall comments/rating).

- Complete: Can view goals; all evaluations read-only; can view all comments.

**5) Security & Visibility**

- Strict security enforced across roles and stages.

- Appraisee visibility by status:

- Draft: No access.

- Submitted: Can view goals; can acknowledge; cannot edit goals or see appraiser/reviewer comments.

- Appraisee Self Assessment: Can view goals; can edit self assessment; cannot see appraiser/reviewer comments.

- Appraiser Evaluation: Can view goals; self assessment is read-only; cannot see reviewer comments.

- Reviewer Evaluation: Can view goals; self and appraiser evaluations are read-only;

- Complete: Can view goals; self/appraiser/reviewer evaluations are read-only; can view all comments/ratings.

- Appraiser and Reviewer have permissions per the Permissions Matrix memory.

**6) Validation & UI Behavior**

- Total weightage across all goals for an appraisal must equal 100%.

- Self Assessment status: Under each goal, show self assessment inputs (comments, rating). Goals are read-only.

- Appraiser Evaluation status: Show appraiser evaluation inputs (comments, rating) below self assessment. Goals and appraisee self assessment fields become read-only. Show Appraiser Overall Comments and Appraiser Overall Rating at the end.

- Reviewer Evaluation status: Show Reviewer Overall Comments and Reviewer Overall Rating below the appraiser overall section. Inputs become read-only after submission at each stage.

**7) Data Model: Goal Template & Appraisal Form**

i) Goal Template

- Fields: Category (multi-select dropdown), Performance Factor, Description, Importance (High/Medium/Low), Weightage (%).

ii) Appraisal Form

- Fields: Appraisee Name/Email, Appraiser Name/Email, Reviewer Name/Email, Appraisal Type (Annual, Half-yearly, Quarterly, Project-end, Tri-annual, Annual-Probation). Period auto-populated.

- Settings: Global appraisal start date and end date.

- Type-specific Ranges: Half-yearly (1st, 2nd), Tri-annual (1st, 2nd, 3rd), Quarterly (1st, 2nd, 3rd, 4th). When a range is chosen, start/end dates are calculated from current date and configured settings.

**Windsurf workflow:**

**1) generate-er-diagram**

Analyze all database models in the `models/` folder of this project.

1. Identify all entities (tables) and their attributes, including primary keys, foreign keys, and data types.

2. Detect relationships between models (one-to-one, one-to-many, many-to-many).

3. Generate an ER diagram using the \*\*dbdiagram.io DSL\*\* syntax with:

- `Table` declarations for each entity

- Field names and their types

- Primary keys and foreign keys explicitly marked

- Relationship definitions using `Ref:` clauses

Ensure:

- The output is clean and copy-pasteable into https://dbdiagram.io for visualization

- All foreign key references include correct source and target fields

- Relationships use the correct direction and notation (e.g., `Ref: orders.user\_id > users.id`)

- Only use information from the actual codebase — do not guess or assume

Group related entities for readability and include comments to identify logical modules if applicable.