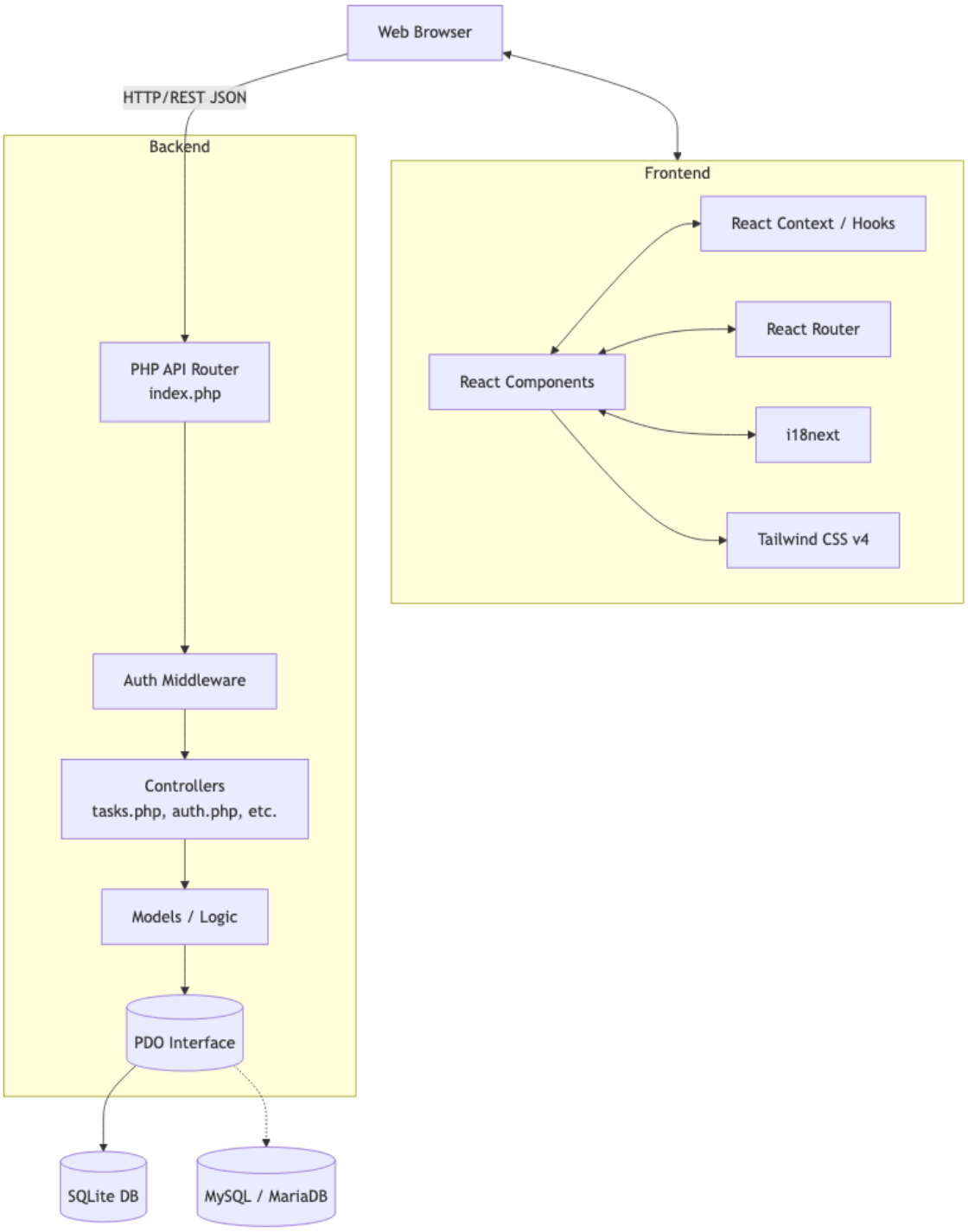
CyberTasker v2.7.0 - Technical Reference

This document provides a highly detailed technical blueprint for CyberTasker, designed for developers and system operators.

1. System Architecture

CyberTasker employs a monolithic but internally decoupled architecture. It uses a modern React frontend hosted statically, communicating with a lightweight vanilla PHP REST-like API.

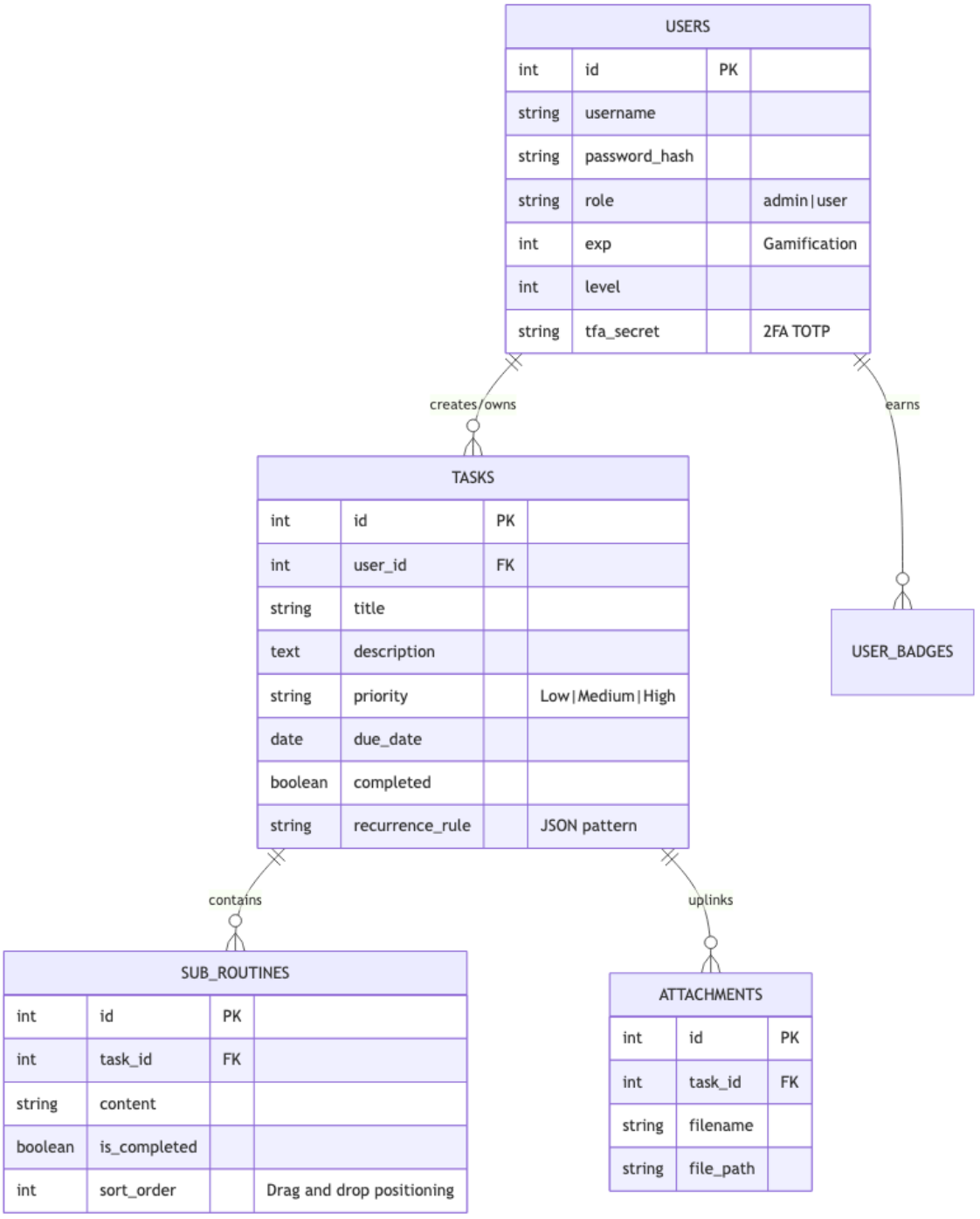


Key Technical Decisions:

- **Vanilla PHP:** No heavyweight frameworks are used on the backend. This guarantees extremely low latency, minimal memory footprint, and trivial deployment on any host supporting standard PHP.
- **Vite/React 19:** Utilized for rapid HMR during development and optimized tree-shaking for production builds.

2. Database Schema (Entities & Relationships)

The relational integrity is maintained by PHP logic on top of the database. Below is the simplified Entity-Relationship Diagram focusing on the primary structures.



3. Routing & API Lifecycle

The backend relies on a custom front-controller pattern configured via `.htaccess`. All incoming API requests are rewritten to `dist/api/index.php`.

1. **Request Ingestion:** `index.php` reads the `?route=` parameter and standardizes the request payload (parsing JSON from `php://input`).
2. **Middleware Authorization:** Before reaching controllers, requests pass through `validate_session()` or JWT-equivalent checks. Mutating endpoints (`POST`, `PUT`, `DELETE`) require rigorous **CSRF token** validation.
3. **Dispatch:** Controller files handle domain logic. For instance, `route=tasks/list` mapped to `GET` will dispatch to `api/tasks.php` -> `get_tasks()`.
4. **Response:** Controllers return normalized JSON structures strictly defined by the API contract.

4. Localization (i18next & Database-Driven)

The frontend's text layer is abstracted using `react-i18next`.

- Translation files (`.json`) are dynamically loaded via `i18next-http-backend`.
- **Database Source of Truth:** As of v2.6.0, the operative's language preference is stored in the `users` table, overwriting browser `localStorage` upon authentication to ensure cross-device consistency.
- A custom python script (`scripts/check_translations.py`) exists in the pipeline to validate synchronization between language keys.

5. Testing Suite (Playwright E2E)

Quality assurance is strictly enforced via **Playwright** End-to-End tests simulating Chromium browsers.

- `tests/e2e/`: Contains complex user-journey scenarios.
- **Notable test paths:** Re-authentication after 2FA toggles, accurately rendering recurrence projections in the Calendar component, and `@dnd-kit` drag-and-drop resilience for sub-routines. The zero-config installer is completely automated via Playwright to guarantee master account creation security.

6. CI/CD Pipeline & Build Orchestration

The project leverages automated pipelines (GitHub Actions based on implementation US-2.3.1) and a streamlined Bash release protocol.

1. **Lint & Unit:** Standard ESLint checks and Vite-based unit tests. Theme CSS bleed is validated via `scripts/check-theme.js`.
2. **Cross-Database E2E Matrix:** A critical step that spins up Docker containers. Playwright tests execute concurrently against both an SQLite instance and a MySQL instance to ensure PDO abstraction remains agnostic and functional.
3. **Bash Release Pipeline (`scripts/release.sh`):** Automates the finalization phase. It runs all translation/theme checks, executes the Playwright E2E suite, increments version numbers in `package.json` and backend headers, builds the Vite `dist/` directory, and concludes by deploying a signed Git tag for the release.