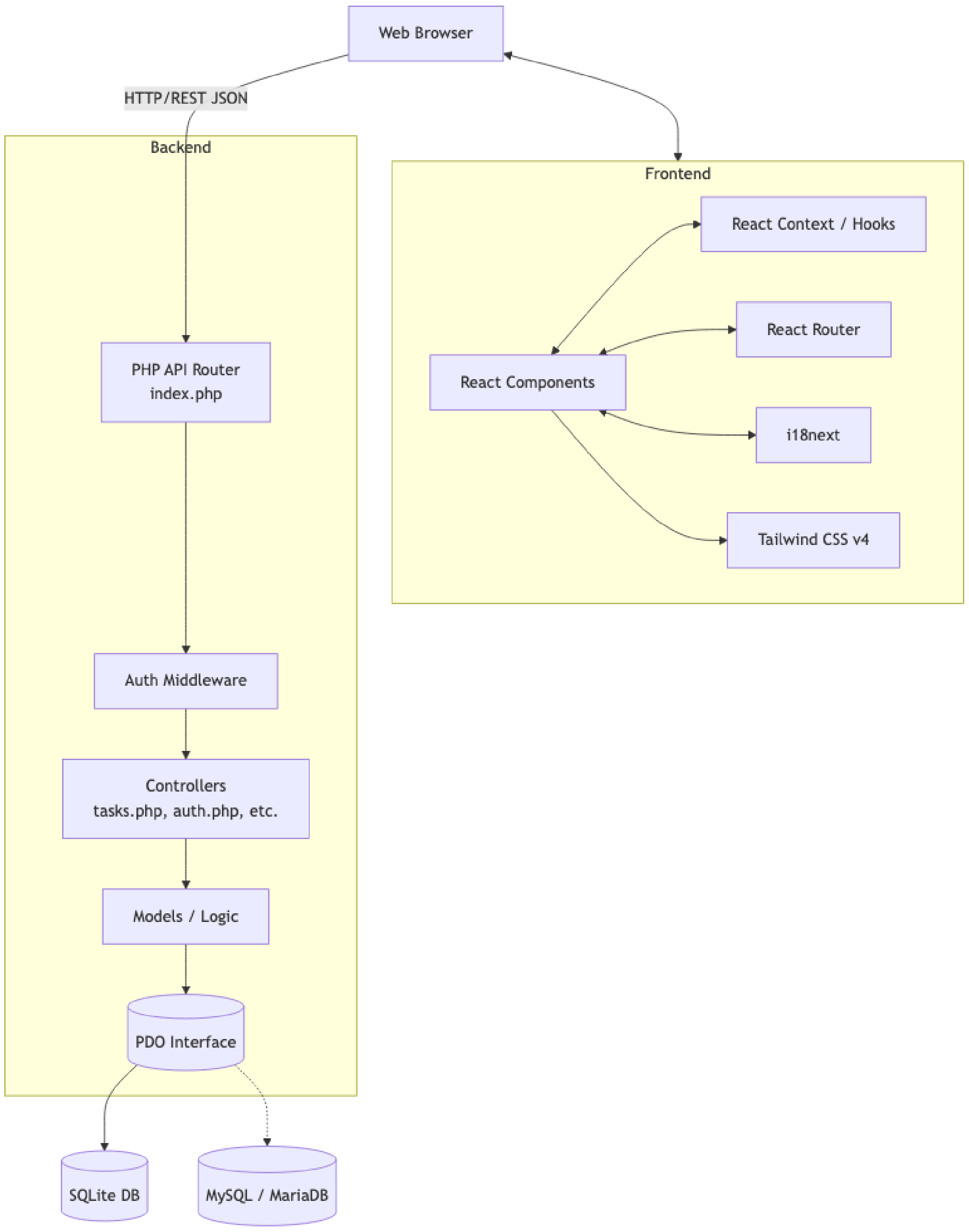
CyberTasker v2.4 - Technical Reference

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.

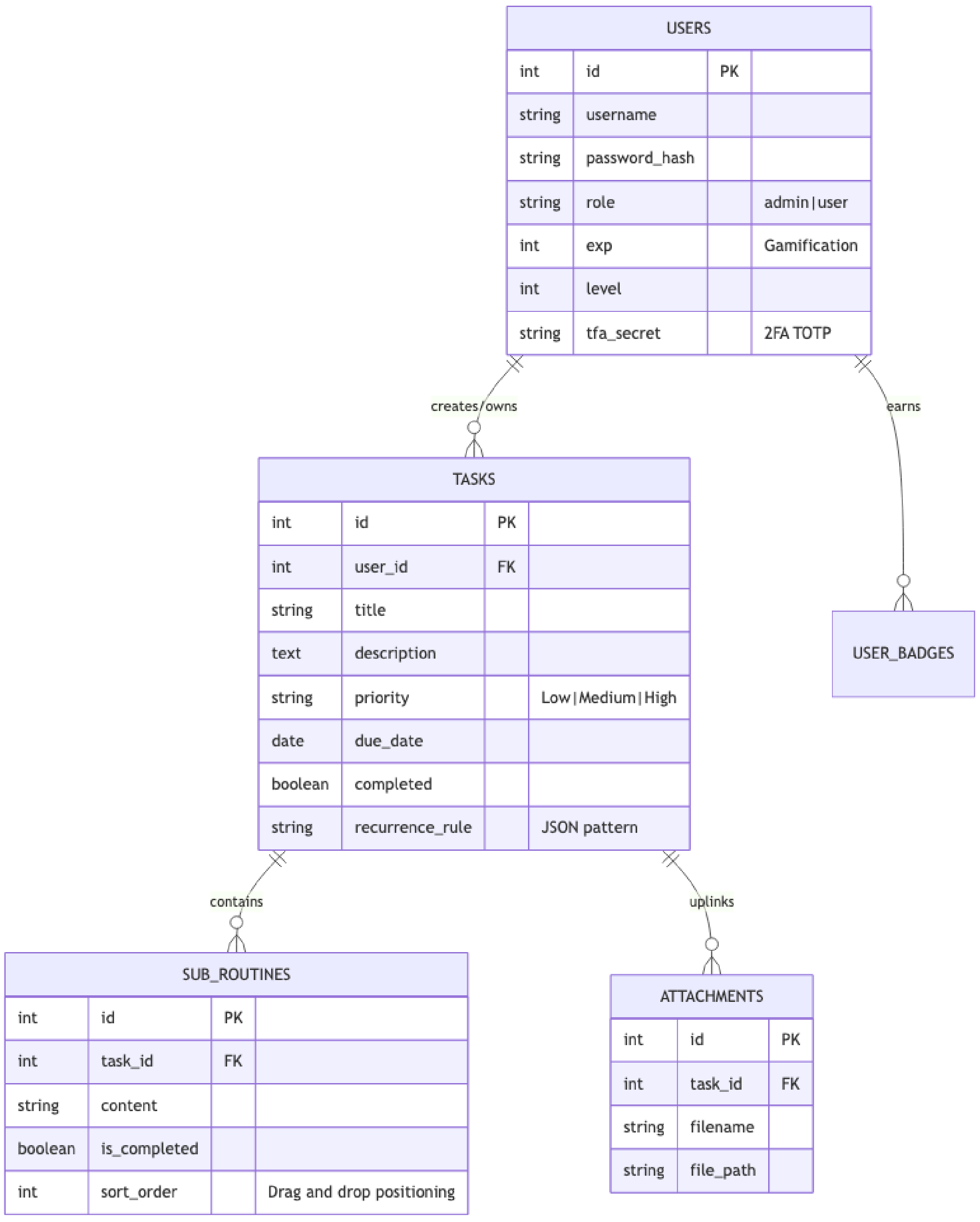


Key Technical Decisions:

- **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.



USERS

int	id	PK	
string	username		
string	password_hash		
string	role		admin|user
int	exp		Gamification
int	level		
string	tfa_secret		2FA TOTP

TASKS

int	id	PK	
int	user_id	FK	
string	title		
text	description		
string	priority		Low|Medium|High
date	due_date		
boolean	completed		
string	recurrence_rule		JSON pattern

SUB_ROUTINES

int	id	PK	
int	task_id	FK	
string	content		
boolean	is_completed		
int	sort_order		Drag and drop positioning

ATTACHMENTS

int	id	PK
int	task_id	FK
string	filename	
string	file_path	

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)

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

- Translation files (`.json`) are dynamically loaded via `i18next-http-backend`.
- 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 drag-and-drop resilience for sub-routines.

6. CI/CD Pipeline & Build Orchestration

The project leverages automated pipelines (GitHub Actions based on implementation US-2.3.1).

1. **Lint & Unit:** Standard ESLint checks and Vite-based unit tests.
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. **Release Bundling:** Upon success, `.htaccess` files are injected into `dist/`, local configuration files are scrubbed, and the final production asset is packaged into a release ZIP.