nap-serv Design Overview

This document consolidates the key design details of the *nap-serv* application so that a contractor can understand the system architecture, data model, business logic, API design and development conventions necessary to build and extend the system.

1. System Architecture & Tenancy Model

nap-serv is a multi-tenant project costing and accounting platform built on Node/Express and PostgreSQL. Each customer (tenant) is isolated in its own PostgreSQL schema. All tables include a `tenant_id` column, and there are no cross-tenant constraints. The application dynamically switches the current schema using the pg-schemata library.

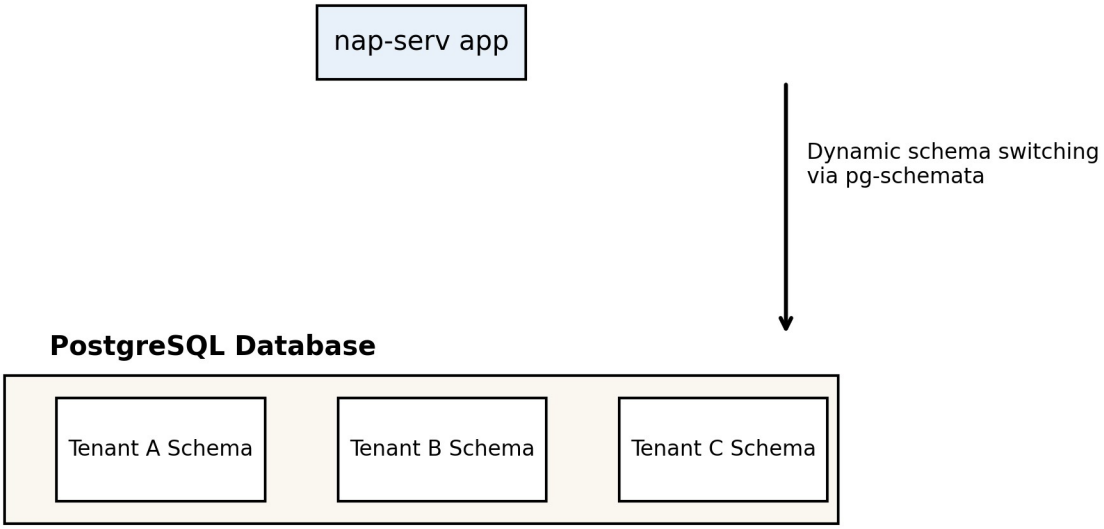

Figure 1: Tenants are isolated within their own PostgreSQL schemas. The application uses pg-schemata to set the current schema at runtime.

The platform is modular: distinct services handle projects and units, budgeting and cost tracking, general ledger (GL), accounts payable (AP), accounts receivable (AR), inter-company transactions and consolidated reporting. A common authentication layer issues short-lived JWT access tokens and longer-lived refresh tokens stored in secure cookies.

2. Data Model

The core entities form a hierarchical structure that supports detailed budgeting and actual cost tracking:

- **Projects** group related work.
- **Units** are subdivisions of a project.
- **Unit budgets** hold approved budget versions for each unit.
- **Activities** capture planned work categories.
- **Cost lines** represent planned spend per activity and vendor.
- **Actual costs** record real-world spend; they cannot exceed the approved budget unless a change order or tolerance applies.
- **Change orders** capture approved adjustments to budgets.

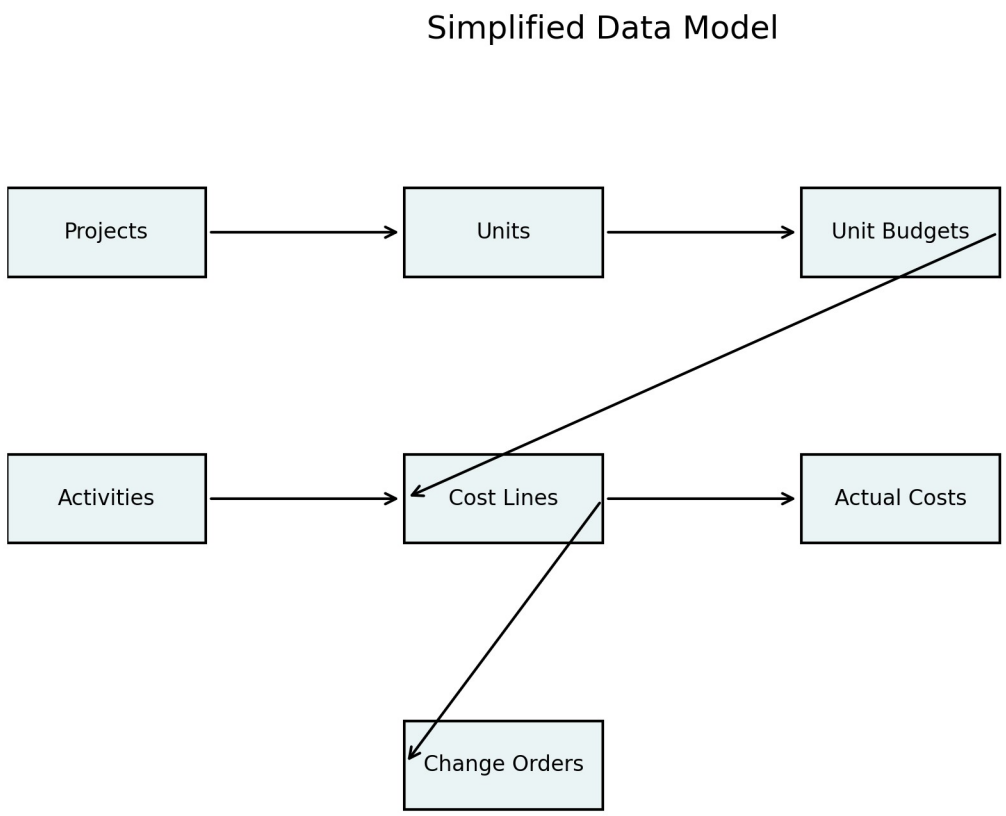

Figure 2: Simplified data model showing relationships between projects, units, budgets, activities and costs.

Additional tables include *journal_entries* for the general ledger, *ap_invoices* and *ar_invoices* for payables and receivables, and *intercompany_transactions* for services performed between companies within a tenant.

3. Business Logic & Workflows

Budgeting & Cost Tracking

Budgets are versioned and must be approved before any actuals can post. Cost lines define planned amounts per activity and vendor. Actual costs default to a pending state when created and go through an approval workflow. Change orders adjust budgets and require approval.

The posting engine creates balanced journal entries for cost lines, actual costs and change orders. AP invoices incorporate cost line items and update vendor balances. AR invoices support revenue recognition based on activity completion. Inter-company transactions automatically generate mirrored revenue and expense entries with elimination tags for consolidated reporting.

Authentication & Authorization

Users authenticate via email/password. The server issues a 15-minute access token and a 7-day refresh token, both stored as HTTP-only cookies. Front-end code refreshes the session periodically and logs the user out after inactivity. Middleware verifies the token, injects the user and tenant into the request context and enforces role-based permissions.

API Design & Query Parameters

RESTful endpoints are versioned under `/api/v1`. Modules mount their routers statically (e.g., `/v1/tenants`, `/v1/projects`). List endpoints accept common query parameters such as `limit`, `offset`, `orderBy`, `columnWhitelist` and `includeDeactivated`. Cursor pagination uses `cursor.<field>` query parameters. A `/where` endpoint accepts an array of conditions supporting `$like`, `$ilike`, `$from`, `$to` and `$in` operators. All values are parameterized to prevent SQL injection.

4. Development Conventions

nap-serv follows a repository pattern: models are defined with columns and constraints, controllers extend a base class and are exported as both classes and default instances, and

API routers aggregate sub-routers. Tables are defined using the pg-schemata typedefs with column sets for inserts and updates. Schemas use UUIDv7 primary keys and always include `id` and `tenant_id` columns.

Contribution guidelines specify a fork-branch-commit-PR workflow, naming conventions (plural nouns for collections), ESLint/Prettier styles and Conventional Commits. Tests mock models via constructor injection and avoid global `jest.mock`; custom unit tests cover business rules and the posting engine.

5. Roadmap & Future Work

The project's development is structured into phases. Earlier phases delivered core infrastructure, costing, and GL/AP/AR scaffolding. Upcoming phases will complete advanced costing and accounting logic (Phase 5), consolidated reporting (Phase 6) and security/roles (Phase 7). Future enhancements include project scheduling, AI-driven forecasting, multi-currency support and external integrations.

6. Summary

This consolidated document is intended to guide a contractor in understanding and implementing the nap-serv platform. It outlines how tenants are isolated, how data is structured and how modules interact; it explains the business workflows for budgeting, costing, accounting and inter-company transactions; it describes the API and authentication patterns; and it highlights the coding conventions and roadmap guiding development.