ChatGPT

Summary of Design Documents in nap-serv and pg-schemata

Overview

The two repositories contain a wide range of design documents. The `nap-serv` docs describe a multi-tenant project-costing system built on Node/Express and PostgreSQL. They cover architectural decisions, detailed module specifications (costing, accounting, inventory and SKU matching), coding conventions, API routing, authentication, testing practices and long-term roadmaps. The `pg-schemata` docs focus on best practices for the underlying Postgres schema layer and the release process for the associated NPM package.

---

nap-serv design docs

Architecture & tenancy

- **Tenant isolation.** Each customer (tenant) is stored in its own Postgres schema; queries always include a `tenant_id` column, and middleware injects the tenant context into requests [1]. The `Accounting Module Design` summary further emphasises strict intra-tenant isolation: all accounting data is scoped by `tenant_id` and there are no cross-tenant constraints, ensuring data remains isolated [2].
- **Project/unit hierarchy.** Projects are subdivided into units, unit budgets, activities, cost lines and actual costs [3]. This hierarchy supports fine-grained budgeting and cost tracking down to individual activities or change orders [4]. All IDs use UUIDv7 and include a `tenant_id` for isolation [5].

Costing & accounting modules

- **Budgeting & cost tracking.** Budgets are versioned per unit and must be approved before any actuals can post [6]. Cost lines represent planned amounts per activity and vendor; actual costs track real-world spend and cannot exceed the approved budget unless a change order or tolerance percentage allows it [7]. Change orders capture approved adjustments to budgets and include workflow statuses (`pending`, `approved` or `rejected`) [8]. The **cost-tracking** overview explains how projects, categories, activity budgets and cost lines connect, highlighting foreign-key relationships and sample SQL for variance reporting [9].
- **Actual cost logic.** Only project managers or automated imports can create actual cost entries. Default behaviors include auto-generated IDs, budget enforcement and pending approval status [10]. An approval flow governs how pending costs become approved or rejected, with audit fields and tolerance rules [11].
- **Accounting (General Ledger).** The accounting roadmap and task list describe an engine that automatically generates journal entries from cost lines, change orders, AP invoices, AR invoices and

inter-company transactions. It enforces balanced debits and credits and validates that postings occur within open fiscal periods [12].

- **Accounts Payable (AP) and Receivable (AR).** AP includes invoice approval workflows, cost-line integration, GL posting and vendor balance updates [13]. AR supports invoice creation, revenue recognition based on activity completion and posting of debit/credit entries to the ledger [14].
- **Intercompany & consolidation.** The system supports companies within a tenant engaging in services for each other. Intercompany transactions auto-generate mirrored revenue and expense entries and include elimination tagging to facilitate consolidated reporting [15]. Later phases of the roadmap plan to generate consolidated P&L, balance sheets and elimination reports across companies [16].

System features and roadmap

- **Feature overview.** The `FEATURES.md` document provides a design summary covering tenant isolation, activity codes, budgeting/actuals, billing models, GL, AP/AR, intercompany transactions, consolidation, chart of accounts, roles/permissions and reporting [1] [17]. It distinguishes fixed-price, cost-plus and milestone billing and introduces **billing units**, abstract events to which costs and revenue are mapped [18]. The MVP includes tenant management, project costing, GL/AP, basic AR, intercompany logic, consolidated reporting and role-based security [19]. Post-MVP enhancements mention project scheduling, AI forecasting, change orders, retainage, multi-currency and external integrations [20].
- **Development roadmap.** `ROADMAP.md` defines seven phases: (1) core infrastructure and tenant isolation, (2) project costing engine, (3) GL/AP/AR scaffolding, (4) intercompany accounting, (5) consolidated reporting, (6) security & roles and (7) QA/hardening [21] [22]. A refactored `ROADMAPv2.md` notes that phases 1–4 are complete and lists remaining tasks for phase 5 (costing & accounting logic), phase 6 (consolidated reporting), phase 7 (security) and phase 8 (beta release) [23]. It also includes an operational checklist and best-practice table showing that static module loading and routing have replaced deprecated dynamic loading [24].
- **Phase 5 tasks.** The phase-5 documents list specific tasks for activities, accounting, AP and AR modules. For activities, budget status must be enforced, remaining budget fields auto-calculated and change orders integrated [25]. Accounting tasks include auto-posting journal entries, balanced transactions and linking entries to source documents [26]. AP tasks involve invoice workflows, account mapping and vendor balance updates [13], while AR tasks cover invoice creation, revenue recognition, postings and payment application [27]. Shared logic requires a status engine, mapping tables and middleware to enforce fiscal rules [28].

API design & query parameters

- **Static routing.** `static-api-routing.md` shows how Express routers statically mount versioned module routes. The main `apiRoutes.js` mounts tenant and activities routers under `/api` [29], and module routers mount sub-routers under paths like `/v1/tenants` and `/v1/nap-users` [30]. Tables map HTTP verbs to CRUD operations for each resource [31].
- **Query parameter guidelines.** The frontend guide lists standard query parameters (`limit`, `offset`, `orderBy`, `columnWhitelist`, `includeDeactivated`) for list endpoints [32]. It explains cursor pagination using `cursor.<field>` [33], advanced condition-based filtering via `/where` [34], an endpoint for archived records [35] and notes for developers such as using `encodeURIComponent` and not mixing offset and cursor pagination [36].

- **OpenAPI and reference docs.** The OpenAPI YAML defines query parameters accepted by the base controller: `limit`, `orderBy`, `includeDeactivated`, `columnWhitelist`, JSON `conditions` arrays and `cursor.{field}` values for cursor pagination [37]. The `/where` and `/archived` paths support `offset`, `limit`, `joinType` and field filters [38]. A companion reference summarises the allowed parameters and provides examples of simple, filtered and cursor-based requests [39] [40].

Coding conventions, testing and developer workflow

- **Contribution guidelines.** Contributors are asked to clone the repo, install dependencies and configure environment variables before running `npm start` [41]. Bugs and feature requests should include clear descriptions and reproduction steps [42], while pull requests must follow a fork→branch→commit→PR workflow and include tests [43]. The document prescribes repository naming (plural for collections, singular for conceptual modules) and file naming conventions for repositories, controllers and routes [44], enforces ESLint and Prettier styles [45] and adopts Conventional Commits (`feat`, `fix`, etc.) [46].
- **Coding conventions.** A separate document elaborates on the repository pattern, API setup, model definitions, controller patterns, routing and schema definitions. Repository objects export model classes using descriptive keys [47], API routers aggregate sub-routers under versioned paths [48] and controllers extend a base controller and are exported both as a default instance and a class for testing [49]. Table schemas are defined using `pg-schemata` typedefs with column and constraint definitions [50].
- **Test setup.** Controller files should export both the class and its instance so tests can import the class directly [51]. Constructors accept optional model injections for isolation [52], and tests should mock models before instantiating controllers [53]. The guide recommends avoiding `jest.mock` of the database, using factory helpers to create controllers, resetting mocks between tests, and writing custom unit tests for business logic [54] [55].

Authentication & authorization

- **Client and server requirements.** The frontend (React) needs the user's role for UI configuration and should automatically log out after 15 minutes of inactivity [56]. The backend authenticates via email and password and expects `email`, `user_name`, `tenant_code`, `role` and schema to be present in middleware [57].
- **JWT structure.** A 15-minute access token and a 7-day refresh token are stored in secure, HTTP-only cookies [58]. An inactivity timeout is enforced by the frontend calling `/auth/refresh` periodically; if no request occurs, the user must reauthenticate [59].
- **Backend middleware & endpoints.** The middleware verifies the token and attaches user and tenant context to the request [60]. `/auth/user` returns safe user profile details for UI rendering [61]. A typical flow: user logs in, frontend fetches user info, periodically refreshes the session and logs out after inactivity [62]. A summary table highlights token storage, expiry and safe exposure of user data [63].

Additional nap-serv topics

- **Phase II activities summary.** The Phase II context explains the PERN stack, use of `pg-schemata` as ORM, UUIDv7 primary keys and multi-tenant architecture [5]. It lists the modules—categories, activities, cost lines, activity budgets and actuals—and links to profitability views [64]. Project

management involves projects tied to clients and addresses [65]. Next steps include scaffolding repositories, controllers and APIs for projects and budgets and writing tests [66].
- **Schema refactoring.** As the project evolved, a refactor plan outlines which schemas to keep (clients and cost lines), which to remove (activity budgets, actuals and project budgets) and which to refactor (projects, activities and profitability view) [67] [68]. New schemas introduced include units, unit budgets, actual costs and change orders [69].
- **SKU matching & onboarding.** A design document proposes automatic matching of vendor SKUs to catalog SKUs using vector embeddings. Models provide functions to find embeddings and similar records [70] and to insert and query matches [71]. Matching logic compares vendor embeddings to catalog embeddings with a configurable threshold and auto-inserts high-confidence matches while returning low-confidence matches for user review [72]. A mermaid flowchart illustrates the end-to-end onboarding workflow: import vendor spreadsheets, validate data, generate embeddings, match to catalog SKUs, review matches and store confirmed matches; separate flows handle price updates, new/deprecated SKUs, maintenance and bid preparation [73].
- **Query model & OOP comparison.** An extensive guide compares object-oriented features in C++ and JavaScript, explaining class syntax, inheritance (`extends` vs. `:`), static methods, memory management, function overloading and private members [74] [75]. Summary notes highlight that JavaScript has class syntax and private fields but remains prototype-based and lacks true function overloading [76].

---

pg-schemata design docs

Design and best practices

- **Shared connection pool & tenant schemas.** The library uses a single `pg-promise` connection pool and switches schemas at runtime, avoiding the complexity of multiple pools [77]. Each tenant has its own Postgres schema; models call `.setSchema()` after login to target the correct tenant [78].
- **Model schema definitions.** Models are defined as structured JavaScript objects with separate `columns` and `constraints` sections [79]. UUIDs are used for primary keys and tenant IDs, generated with `gen_random_uuid()` [80].
- **ColumnSets and CRUD helpers.** The `TableModel` builds separate `insertColumnSet` and `updateColumnSet` to handle immutable and mutable fields, making inserts and updates safer [81]. It implements common read methods (`findById`, `findAll`) that automatically qualify tables with the current schema [82].
- **Immutable fields & auto-creation.** The framework can mark columns immutable to prevent accidental updates [83]. On tenant signup, schemas and tables are created programmatically [84].
- **Design principles.** Final principles emphasise a single connection pool, dynamic schema switching, structured model schemas, ubiquitous UUIDs, early construction of ColumnSets, dynamic read methods, careful handling of immutable fields and auto-generated SQL for migrations [85].

Release process

- **GitHub release.** To release a new version, the developer updates the `dev` branch, merges it into `main` with a release commit, bumps the version using `npm version`, then pushes `main` and tags to GitHub [86]. A GitHub release can be drafted using the new tag and release notes [87].

- **npm publish.** After verifying login, run `npm publish` to publish the stable version, optionally using `--tag beta` for beta releases [88]. The document includes commands for managing npm dist-tags and notes on how to deploy documentation to GitHub Pages [89].
- **Post-release sync.** After releasing, merge `main` back into `dev` to carry version bumps forward and push the updated `dev` branch [90].

QueryModel conditions

- **Structured conditions.** The `QueryModel.findWhere()` method accepts an array of condition objects that map cleanly to SQL `WHERE` clauses. Basic equality is expressed as `{ column: value }` [91]; `null` checks use `{ column: null }` to produce `IS NULL` [92].
- **Operators.** Supported operators include `$like` and `$ilike` for pattern matching [93], `$from` and `$to` for range queries [94], and `$in` for membership tests [95]. Nested `$or` and `$and` groups allow complex boolean logic [96]. Only these operator keys are supported; unsupported keys throw an error [97].
- **Notes & future operators.** The document stresses that all values are parameterized to prevent SQL injection and hints at potential future operators like `$not`, `$gt`, `$lt`, `$between` and `$notIn` [98].

Conclusion

The `nap-serv` design documents outline a robust, multi-tenant accounting platform with comprehensive budgeting, costing, accounting, inter-company transactions and reporting features. They provide detailed development roadmaps, coding conventions, testing practices, authentication strategies and specialized modules (e.g., SKU matching). The `pg-schemata` design docs complement this by prescribing best practices for Postgres schema management and model definitions, ensuring that the underlying data layer remains scalable, extensible and easy to maintain. Together, these documents offer a holistic view of building a SaaS-grade project accounting system using Node/Express and PostgreSQL.

---

[1] [15] [17] [18] [19] [20] raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/FEATURES.md

[2] raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/Accounting_Module_Design_Summary.md

[3] [4] [6] raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/budgeting_cost_tracking_spec.md

[5] [64] [65] [66] raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/phase2_activities_summary.md

[7] [10] [11] raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/actual_costs_business_logic.md

[8] raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/change_order_lines_business_logic.md

9 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/cost-tracking.md

12 13 14 25 26 27 28 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/Phase%205%20ROADMAP.md

16 21 22 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/ROADMAP.md

23 24 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/ROADMAPv2.md

29 30 31 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/static-api-routing.md

32 33 34 35 36 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/frontend-query-guide.md

37 38 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/query-parameters-openapi.yaml

39 40 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/query-parameters-reference.md

41 42 43 44 45 46 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/CONTRIBUTING.md

47 48 49 50 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/coding_conventions.md

51 52 53 54 55 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/test-setup.md

56 57 58 59 60 61 62 63 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/nap-auth-strategy.md

67 68 69 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/schema_refactor_plan.md

70 71 72 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/sku_matching_design.md

73 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/sku_onboarding_and_bidding_process.md

74 75 76 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/nap-serv/main/design_docs/cpp-vs-javascript-oop-comparison.md

77 78 79 80 81 82 83 84 85 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/pg-schemata/main/design_docs/DesignAndBestPractices.md

86 87 88 89 90 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/pg-schemata/main/design_docs/RELEASE.md

91 92 93 94 95 96 97 98 raw.githubusercontent.com
https://raw.githubusercontent.com/silverstone-i/pg-schemata/main/design_docs/querymodel-conditions.md