ChatGPT

Comprehensive Plan to Complete the Accounting Module

1. User Roles, Permissions & Security Enhancements

- **Define Complete Role Hierarchy:** Expand the current role setup to cover all necessary accounting roles. The system already seeds roles like **Admin, Clerk, Manager, Auditor** for each organization [1]. Ensure each role has appropriate permissions:
- *Admin:* Full access to everything (already configured to get all permissions by default [2]).
- *Clerk:* Basic create/edit access (e.g. all CRUD on transactions but no approvals).
- *Manager:* Approval rights for journals and period closing (e.g. can approve or reject journal entries, can close/reopen periods).
- *Auditor:* Read-only access to all accounting data (view permissions only).
- **Map Permissions to Actions:** For every key action in accounting, ensure a corresponding permission exists and is checked. The **Permission** model supports granular actions (`view`, `add`, `change`, `delete`, plus custom ones like `approve`, `submit`, `post` etc. [3]). For example, define permissions such as **submit_journal**, **approve_journal**, **post_journal**, **reverse_journal** and assign them to the relevant roles [4]. Use the existing permission framework (e.g. `PermissionUtils.has_permission`) to enforce these in views – many views already use `PermissionRequiredMixin` with a `(module, entity, action)` tuple [5] [6], so extend this to all new actions (like posting or approval).
- **Enforce Permissions in UI & API:** Audit all accounting views, templates, and APIs to ensure they respect permissions:
- Protect critical views with the `PermissionRequiredMixin` or `@require_permission` decorator (for function-based views) so unauthorized users see a 403 page [7] [5]. For example, the journal entry view already requires `('accounting','journal','add')` for new entries [8] – implement similar checks for actions like posting (require `post_journal` perm) or approving (require `approve_journal`).
- In templates, utilize the `has_permission` template filter to hide or disable buttons that the user shouldn't use [9]. For instance, show **"Post"** or **"Approve"** buttons only if `user|has_permission:"accounting_journal_post"` or `"accounting_journal_approve"` returns true.
- **User-Specific Overrides:** Leverage the `UserPermission` model for edge cases (grant or revoke specific permissions per user) [10]. Extend `PermissionUtils.get_user_permissions` to include these overrides (currently it only gathers role perms [11] [12]). This allows fine-tuning access (e.g. temporarily revoke a permission from a user or grant an extra one without creating a new role).
- **Account Security & MFA:** Take advantage of the fields in `CustomUser` for security (e.g. `failed_login_attempts`, `locked_until`, `mfa_enabled`) [13]. Implement policies like account lockout after a number of failed attempts and optional **Multi-Factor Authentication** for logins. This is beyond accounting-specific, but it strengthens overall security for accessing the accounting module.

- **Audit Logging of Changes:** Activate the `AuditLog` model to record critical changes [14]. For each create/update/delete action on accounting data (like journals, fiscal periods, COA), log an entry with who did it and what changed:
- Use Django signals or override model save/delete to create `AuditLog` records, storing old vs new values in the JSON field `changes` [14].
- Include context like IP address and timestamps (fields are available in the model). This ensures an audit trail for compliance – a requirement of GAAP/IFRS compliance and Nepal's regulations (e.g., immutable records for audit).
- **Data Access Segmentation:** Ensure that all queries filter by the active organization (multi-tenant safety). The mixins (e.g. `UserOrganizationMixin`) already filter data by `organization` [15]; verify all new views/services use them so users only see their organization's financial data. This prevents any cross-org data leaks, aligning with security best practices.

2. Journal Entry Workflow & Voucher Management

- **Complete Journal Lifecycle:** Implement the full **voucher (journal entry) workflow** so that journals move through statuses from Draft to Posted:
- **Draft & Submit:** When a journal entry is saved initially, mark it as **draft**. Provide a "Submit for Approval" action that changes status to **awaiting_approval** [16]. Only users with submit rights (e.g. the Accountant role) have this ability [17].
- **Approval Process:** Users in managerial roles can review and **approve** or **reject** submitted journals. On approval, set status to **approved** [18]. Rejection sends it back to draft or a rejected state; log the decision (maybe via AuditLog). Use permissions like `can_approve_journal` and `can_reject_journal` to gate these actions [4]. For example, a Manager role would have "Can approve journal" permission and see an **Approve** button, whereas a Clerk would not.
- **Posting to Ledger:** Implement **posting** of approved journals to finalize them. Posting should:
  - Change status to **posted** and record `posted_by` and `posted_at` [19].
  - Generate the corresponding ledger entries (in GeneralLedger table) and update account balances. The code should create a **double-entry**: for each `JournalLine`, add debit/credit entries to the GL. Ensure debits equal credits before allowing post (the `imbalance` property can be used to check balance [20]).
  - Mark the journal as locked/readonly after posting (e.g. set `is_locked=True` to prevent further edits [21]). In the UI, a posted voucher should display as read-only.
- **Reversal and Corrections:** Provide a way to reverse posted journals instead of deleting them (to maintain audit trail). For example, a **"Reverse Journal"** action can create a new journal with inverted debit/credit amounts and link it to the original (set original's status to **reversed** and `is_reversal=True` on the new one [16] [22]). Only allow users with a `can_reverse_journal` permission to do this [23].
- **Modification Controls:** Enforce that once a journal is posted (or awaiting approval), it cannot be modified unless properly reverted:
  - Disable editing of approved/posted entries in the UI.
  - If a draft needs changes, allow editing or deleting by users with `modify_journal` permission [17]. The system already seeds a "modify_journal" perm for the Accounting role [24] – use that to control who can edit draft journals or unpost a journal in exceptional cases.
- **Journal Numbering Configuration:** Finalize the auto-numbering for vouchers. The system uses `JournalType` with prefixes and sequence numbers to generate journal codes (e.g. "GJ" for General Journal) [25]. Complete this by:

- Ensuring each new journal gets a unique sequential number per JournalType (the provided `get_next_journal_number()` locks the row and increments sequence safely [26] [27]).
- If **fiscal_year_prefix** is true, prefix the number with the fiscal year code to restart numbering each year [28]. This meets common practice of unique numbering per fiscal year.
- Possibly allow configuring number padding length (currently it's fixed to 3 digits with `.zfill(3)` in the code [29]). If needed, make this padding configurable (e.g. 4 digits for up to 9999 entries).
- **Fiscal Period Management:** Leverage the **FiscalYear** and **AccountingPeriod** models to enforce date restrictions:
- Ensure that journals can only be posted to an **open** accounting period. If a period is closed, prevent posting and inform the user (the Journal model has a `period` foreign key; use its status). Provide a permission-based action to **reopen periods** if needed (the system defines a `can_reopen_period` permission for this [30]).
- Automate period closing: develop a service or command to close all periods at year-end and maybe create opening balances for the new year. Only users with appropriate rights can run year-end close.
- When closing a fiscal year, set one of the year's periods as final and mark FiscalYear.status = "closed", and perhaps mark the next year as current. The **FiscalYear** model already enforces that only one fiscal year per org can be current [31] – ensure the UI/logic to toggle `is_current` and `is_default` is in place with validation (the model clean() covers it [31]).
- **General Ledger Integration:** Make sure posting a journal updates the **GeneralLedger** entries and account balances:
- The system likely has a `GeneralLedger` model to store finalized entries (so reports don't have to sum journals each time). On posting, create GL records for each line. Include which **Ledger** it belongs to if using parallel ledgers (see IFRS section below).
- Update each affected account's `current_balance` and `reconciled_balance` in ChartOfAccount [32]. Consider wrapping posting in a database transaction so that all or nothing is saved (to maintain consistency between Journal, JournalLine, and account balances).
- If multi-currency journals are allowed, convert amounts to base currency for GL. Currently, Journal has `currency_code` and `exchange_rate` [33] – use these to compute functional currency amounts. (In future multi-ledger design, this moves to JournalLine, but for now ensure at least one currency conversion).
- **Recurring and Template Journals:** Implement functionality for `is_recurring` journals if needed. This could include:
- Marking certain journal entries as recurring templates (perhaps stored with `is_recurring=True` [34] and a recurrence rule). Develop a job or management command to generate instances of those journals at the specified intervals.
- Alternatively, allow users to duplicate an existing journal entry (to reuse as a template), which can save time for regular transactions like monthly depreciation or payroll entries.
- **Voucher Configuration & UDFs:** Leverage the **VoucherModeConfig** and **VoucherUDFConfig** to allow custom fields on vouchers:
- The seed data creates a default "Nepal Standard Voucher" config with certain settings and a default UI schema [35] [36]. Build on this by exposing a UI for administrators to edit voucher configurations (already accessible via VoucherModeConfig views [37] [38]).
- Ensure that user-defined fields (UDFs) can be added via the config and are included in the journal entry form. The mechanism `_inject_udfs_into_schema` is in place to add UDFs to forms dynamically [39]. Test that this works so non-technical users can add custom fields (e.g., a field for "Cheque Number" on payment vouchers) without code changes.
- **Data Validation & Concurrency:** Finalize all validations for journal entries:

- Ensure **debits equal credits** before allowing save or post (perhaps enforce at form validation level and in `JournalValidationService` if present).
- Enforce that no two journal lines have the same line number in a journal, and line numbering is sequential. (The model has `line_number` field – auto-set these or validate on save).
- Use database **transactions** for multi-step operations like posting (the code already wraps number generation in a transaction [40]; extend similar protections to posting logic).
- Implement optimistic locking where needed – e.g., the Journal model has a `rowversion` field [41], possibly to detect concurrent modifications. Use this field to prevent overlapping edits: if an update is attempted with an old rowversion, reject it with an error to the user to avoid lost updates.

3. Financial Reporting & Analytics

- **Implement Standard Reports:** The system should deliver a suite of accounting reports with filters, drill-down, and export. According to the design, the following reports are planned and partially implemented [42]:
- **General Ledger Report:** List all transactions for each account within a date range, with opening and closing balances. This is already outlined with filtering by account and date [43] [44]. Complete it by ensuring it sums running totals per account and allows drilling into transaction details (e.g., each journal line can link to the journal voucher page).
- **Trial Balance:** Show each account's debit, credit, and net balance as of a certain date [45] [46]. Ensure it pulls all active accounts and correctly reflects posted transactions up to the cutoff date. Provide an "as of" date filter (already in use) and include both opened and closed periods before that date.
- **Profit & Loss Statement (Income Statement):** Aggregate revenues and expenses over a period [47] [48]. Use the account classifications (income vs expense) to group accounts. The default chart of accounts groups like *Revenue*, *Other Income*, *Cost of Goods Sold*, *Operating Expenses*, etc., which align to P&L sections [49] [50]. Ensure the report calculates subtotals and net profit correctly. Allow filtering by fiscal year or custom date range.
- **Balance Sheet:** Summarize assets, liabilities, and equity at a point in time. Use the AccountType classifications (current vs non-current) and balance sheet categories seeded [51] [52] to group accounts on the report. The report should pull the ending balances as of the report date for each account and compute totals for Assets, Liabilities, Equity (and verify Assets = Liabilities + Equity for balance).
- **Cash Flow Statement:** Support at least a basic cash flow report (likely indirect method using P&L and balance sheet changes). If not fully automated, mark as a future enhancement; otherwise, use movement in cash accounts and certain non-cash account changes to derive cash flow. Provide it for completeness, as IFRS requires a cash flow statement.
- **A/R Aging Report:** The template for Accounts Receivable Aging is present [53], indicating this report is expected. If an AR module (invoices/customers) exists, implement this by grouping receivables by age buckets (30/60/90+ days overdue). If no AR module yet, this can be integrated once sales invoices are tracked. Ensure the report fetches data via appropriate queries (possibly from an AR ledger or invoice table).
- **Drill-Down & Interactivity:** Enable users to navigate from summary reports into details:
- On **Trial Balance**, each account line could link to the General Ledger report filtered for that account (e.g., clicking on "Cash" account opens the GL detail of Cash transactions).
- On **Financial Statements (P&L, Balance Sheet)**, allow expanding groups to see constituent accounts. For example, on the Balance Sheet, clicking "Current Assets" could list the accounts under

it (Cash, Receivables, etc.) with their balances. This can be done in the template via collapsible sections or separate views.

- In the **General Ledger report**, each journal entry line should link to the actual **Journal detail** (voucher) page. This provides a true drill-down from a report number to the originating transaction form for verification.
- **Stored Procedure Data Fetching:** For complex or large data sets, consider moving report queries into PostgreSQL stored procedures:
- Identify reports that may benefit from SPs for performance (e.g., Trial Balance or Aging reports that aggregate lots of data). You can create SQL functions or views in the database to compute these, then call them via Django (using raw SQL or a database function call).
- For instance, a stored procedure for Trial Balance could accept an `as_of_date` and compute each account's balance with a single SQL query, rather than iterating in Python. The question explicitly asks for reports to be fetched from stored procedures, so design the interface where the `ReportService` in Python calls `SELECT * FROM generate_trial_balance(%s)` etc.
- Maintain flexibility by still allowing filtering parameters (dates, account, etc.) to pass into the SP. This approach mirrors SSRS where a report is often backed by a stored procedure for heavy lifting.
- **Export to Excel/PDF/CSV:** The reporting module should allow non-technical users to easily export data. The code already includes a **ReportExportService** that supports CSV, Excel, and PDF generation [54] [55]. Ensure this is fully integrated:
- Provide **Export** buttons on each report page (GL, TB, P&L, etc.) to download the report in the desired format. The `ReportExportView` (if implemented) or similar should handle a request like `?format=excel` or `format=pdf` and use `ReportExportService` to produce the file [56].
- Fine-tune the export formatting: The code uses openpyxl for Excel with styling for headers and totals [57] [58], and likely WeasyPrint/ReportLab for PDF. Verify that the output is nicely formatted (proper column widths, bold headers, etc. already handled in code [59]). For PDF, create a template or use HTML-to-PDF so the print layout is professional (include company name, report title, date, etc. – some of this is already written to the CSV/Excel as headers [60]).
- Ensure all fields (including any custom fields or multi-currency data) are included in exports. Also, test that large data exports (like a full GL for a year) are efficient (might need streaming responses or limiting date ranges to avoid timeouts).
- **Non-Technical Custom Report Design:** Aim to empower power users to design custom reports similarly to SSRS (to an extent):
- Implement a **Report Definition** model/table where admin users can define new reports by providing a stored procedure name or an SQL query along with metadata (report name, description, expected parameters). Then have a generic frontend to input parameters and display the results in a table or chart.
- For instance, a user could register a SP like `sp_income_by_customer(year int)` as a new report "Income by Customer". The system would present a form for the `year` parameter, execute the SP, and render the result as an HTML table which can then be exported. This approach lets advanced users add reports without modifying code.
- Allow some format customization: maybe let them provide simple templates for how to display the data (e.g., group by some field or highlight certain values), though full SSRS-style drag-and-drop designers are out of scope. Instead, focus on making all data accessible and exportable, and possibly allow HTML editing for headers/footers.
- Use the existing framework for exports on these custom reports too, so any user-created report automatically gains CSV/PDF/Excel download buttons.

- **Performance and Pagination:** Some reports (like General Ledger with thousands of entries) might need pagination or on-demand loading. Implement pagination or lazy-loading in the web view for extremely large reports, and use database indexes (ensure key fields like dates, account IDs are indexed on the GL table). The code already defines indexes (e.g., on LoginLog, and likely on ledger tables) – verify indexes on JournalLine or GL for date and account [61] and add if missing, to speed up reporting queries.

4. Printing & Template Customization

- **Voucher Printouts:** Provide printable voucher documents for journal entries and other accounting transactions. Many organizations need a physical or PDF **Journal Voucher** print for each entry (with signature lines, etc.). Implement a template (HTML/CSS or PDF) for journal vouchers that includes:
- Header with company name, voucher number, date, journal type, and reference.
- Table of line items (account code, account name, debit, credit, description per line).
- Totals and an automatic "Approved by / Prepared by" signature section.
- Ensure this printout aligns with local standards (e.g., voucher format common in Nepal).
- Use WeasyPrint or ReportLab (as hinted in the code) to generate a PDF of this template on demand. The user should be able to click **"Print Voucher"** on a posted journal and get a PDF download/ preview.
- **Invoice and Billing Prints:** If the ERP includes sales/purchase modules (even if minimal now), design invoice templates compliant with national e-billing. For example, a **Tax Invoice** print template should have:
- Seller and buyer details (addresses, tax IDs like PAN/VAT numbers).
- Invoice number and date, due date for payments.
- Line items with description, quantity, rate, amount, tax breakdown.
- VAT summary (taxable amount, VAT amount) and a gross total in words.
- Any government-mandated fields (for Nepal's e-billing, include things like "IRDN" if applicable, and a QR code as required by IRD guidelines).
- The print layout must be precise to be IRD-approved – likely A4 size, with certain font sizes. Use the national format as a guide.
- **Template Customization by Users:** Allow non-developers to adjust print templates to some extent:
- One approach is to use a template engine (Django templates or Jinja) for documents and let advanced users upload a custom template file or edit through the UI (with proper security). For instance, store a default template for invoices and let users insert their logo or change labels without altering code.
- Provide documentation on how to tweak these templates and perhaps a preview function. If direct editing is risky, at least allow uploading a company logo and setting header/footer texts (like "Accounts Department, XYZ Corp") via settings, then merge those into the template.
- Ensure that any changes maintain compliance (e.g., users shouldn't be able to remove mandatory fields on an invoice).
- In the future, a visual template designer could be considered, but initially even small customization options (logo, company info, signature images) will meet needs of non-technical users.
- **Batch Printing and Export:** Allow printing multiple documents in bulk if needed:
- Example: printing all vouchers of a day or all customer invoices in a batch to PDF. This could generate one PDF with multiple documents or a ZIP of PDFs.
- Ensure the printing process is robust (use background tasks if generating many PDFs to avoid web timeouts).

- **Excel Templates for Reports:** In addition to PDF, consider if users want to design Excel report templates (like pivot tables or formatted spreadsheets). A non-technical user might simply rely on the raw Excel export and then adjust it manually. As an enhancement, you could allow uploading an Excel file with macros/pivot that uses a data sheet the system fills – but that might be too advanced. Initially, ensure the raw exports are clean and usable in Excel (the current openpyxl export already applies some formatting [62] [63]).
- **Embed Charts and Visuals:** To mimic some SSRS capabilities, consider adding simple charts (e.g., bar chart of income vs expense, or pie of expenses by category) in the PDF/Excel outputs of reports. This could be optional but can greatly help non-technical users understand data. Leverage libraries like matplotlib or openpyxl chart for Excel, and WeasyPrint for including SVG charts in PDFs.
- **Localization of Formats:** Since printing often involves number and date formats, ensure that these templates format amounts with proper currency symbols and formatting (e.g., `1,23,456.00` vs `1,234,56.00` depending on locale). The default currency is NPR for Nepal; include "Rs" symbol on documents where relevant. Also include a way to switch language if needed (the system might be mostly English, but consider Nepali date or text for local reports if required by law).

5. Default Seed Data & Configuration

- **Chart of Accounts (COA):** Verify that a comprehensive **default chart of accounts** is provided for new installations. The seeding script already creates a structured COA for Nepal [64] [65], including common accounts like Cash, Bank, Receivables, Payables, VAT Payable, Salary Payable, Equity, Revenue, COGS, Operating Expenses, etc. This default COA aligns with NFRS standards and can be used as a starting point. Ensure that:
- Accounts are correctly categorized by type (Asset, Liability, Equity, Income, Expense) using the AccountType model [66] [67]. The seed covers current vs non-current subdivisions (e.g. Current Assets vs Fixed Assets) and operating vs non-operating expenses [68] [69].
- The numbering scheme for accounts is logical (the code uses a 4-digit hierarchical format like 1000 for Assets, 2000 for Liabilities, etc., with subaccounts incrementing by 100 [70] [71]). The seed likely generates codes automatically for these accounts – confirm they appear as expected (e.g., Assets = 1000, Cash in Hand = 1100, Bank = 1200, etc. as children).
- Include any accounts necessary for statutory reporting in Nepal (the seed includes provident fund, CIT, etc. which are local compliance accounts [72]).
- **Accounting Dimensions:** The seed script also creates default **Departments, Projects, Cost Centers** (as indicated by the summary log [73]). These provide additional tracking dimensions. Ensure at least a few sample departments (HR, Sales, etc.), projects, and cost centers are created so users can immediately tag transactions. This meets internal reporting needs.
- **Fiscal Year & Periods:** By default, a Nepali fiscal year is created (e.g., mid-July to mid-July) with 12 periods [74]. Verify the seed uses the Nepali calendar if intended (the script uses Baisakh start i.e. April 14 [75] and likely creates 12 monthly periods). Adjust if needed for businesses that follow calendar year or other fiscal calendars (perhaps make the fiscal year start configurable).
- **Currency Setup:** Seed multiple currencies with proper symbols. The default data already adds NPR, USD, EUR, INR [76]. Confirm NPR is set as the base currency for the organization [77]. In the UI, allow adding more currencies or updating rates. Also, ensure a mechanism to update exchange rates (maybe a simple form or integration with an API in the future) – but at least provide a CurrencyExchangeRate model/table for users to enter rates. (It looks like a `CurrencyExchangeRate` model might exist since forms import it [78]).

- **Tax Configuration:** The seed includes a default **Tax Authority** (Nepal Inland Revenue Department) [79], common **Tax Types** (VAT, TDS, Income Tax) [80], and standard **Tax Codes** [81]. Verify these:
- VAT Standard 13% is set up (the seed shows "VAT Standard Rate – 13%" [81]) along with Zero VAT and Exempt.
- TDS (with typical rates 1%, 5%, 2%, 10% for various categories) is seeded [82].
- Each TaxCode links to the Chart of Account for tracking (the model has sales_account and purchase_account fields [83] – ensure the seed associates VAT payable account with VAT code, etc., if not, add that).
- The system should use these tax codes on transactions (e.g., if there is an invoice or expense booking form). Even within journal entries, if `default_tax_code` is set on an account, auto-apply it for convenience [84].
- Provide a UI to manage tax codes in case rates change (Nepal's VAT rate might change, etc.), and ensure effective_from and to dates can be used for rate history [85].
- **Journal Types and Voucher Settings:** Default **JournalType** entries are created (General Journal, Cash Receipt, Cash Payment, etc.) [86]. These come with unique prefixes (GJ, CR, CP) and their own numbering sequences. Ensure these are in place:
- *General Journal* (GJ) for non-cash adjustments, *Cash Receipt* (CR) for money received, *Cash Payment* (CP) for money paid. If needed, add more types like Bank Payment, Sales Journal, Purchase Journal, etc., depending on the scope of module (sales/purchases might be separate modules, but at least provide the basics).
- Set `requires_approval=True` on types that should have an approval workflow (maybe large Journal entries or specific types can be flagged).
- The seed likely sets `is_system_type=True` for core types [87], and uses `sequence_next` for numbering [88]. After posting a few transactions, verify the auto-number increments correctly.
- **Voucher Mode Configuration:** A default **VoucherModeConfig** named "Nepal Standard Voucher" is provided [35]. This controls how the entry form behaves (layout, mandatory fields, etc.). Ensure this default is marked `is_default=True` so it's used system-wide [89]. The config sets things like `allow_multiple_currencies=True`, `show_tax_details=True` [90] which means the journal entry form will accommodate tax lines and multiple currencies. Test that:
- The journal entry UI indeed shows tax fields or currency dropdown when these flags are on.
- Users can create additional VoucherModeConfig if needed for different entry modes (e.g., a simplified voucher form for certain entries).
- **Initial User/Organization Setup:** The script creates a default organization and ties the superuser to it [91] [92]. Verify that on a fresh installation, after running the default data script or migration, an admin can log in and immediately see the Accounting module with all these defaults (no further manual setup needed). The default org "Nepali Accounting System" with base currency NPR is created [93]; this can be used for demo or changed by the user.
- **Permissions & Entities Seed:** Ensure all **Permission** entries for accounting are seeded so roles can be assigned properly. The default data script creates the Accounting module and a list of entities (FiscalYear, AccountingPeriod, ChartOfAccount, Journal, etc.) with CRUD actions [94] [95]. It then grants Admin all permissions and User role view-only perms [2]. Double-check that every model in accounting has a corresponding Entity and at least view/add/change/delete perms:
- If any models were missed (e.g., if new models like Ledger or ExchangeRate are added later), update the seed to include them in `entities_data`. The placeholder comment "Add other entities..." in earlier setup code [96] suggests a need to include all accounting components, which the unified `create_default_data.py` appears to handle now.
- This comprehensive permissions seeding ensures the security model is robust from the start.

- **Sample Transactions:** Optionally, include some **seed transactions** for demo purposes (the script does create one demo journal entry [97] [98]). This helps new users see how a completed voucher looks. The provided demo (debit an expense and credit cash for NPR 1000 [99] [100]) is a good example. Make sure this only runs if there are no existing journals to avoid duplicate data on re-seed.
- **Validation of Seed Data:** After seeding, run a self-check (the script logs a summary [101]). Ensure the counts of items (accounts, tax codes, etc.) match expectations. This confirms that all default data loaded correctly. For instance, it logs 15 account types, ~50 accounts, a number of tax codes, etc. If any are missing (count is off), debug the seeding process.
- **Documentation for Users:** Provide documentation or an initial setup guide for the accounting module, explaining that a default chart of accounts and settings are already in place (with GAAP/NFRS alignment), and how to modify them. Non-technical users should know they don't have to start from scratch – they can edit the provided accounts and tax rates to fit their company. This will accelerate adoption.

6. Compliance with GAAP, IFRS, NFRS & E-Billing

- **GAAP/IFRS Accounting Standards:** Ensure the accounting module can support both local accounting standards (NFRS, which is Nepal's equivalent of IFRS) and also Generally Accepted Accounting Principles:
- The default account structure and financial statements should satisfy IFRS/NFRS presentation. The seeded AccountTypes already categorize accounts into Current/Non-current and link to balance sheet or P&L sections [66] [67], which is critical for IFRS-compliant statements (e.g., separate current assets from non-current on the Balance Sheet).
- To handle any differences between local tax accounting vs IFRS adjustments, plan for **parallel ledgers**. For example, a company might maintain one ledger for tax/NFRS reporting and another for internal or GAAP reporting. The architecture document suggests introducing a `Ledger` model to allow multiple representations (Actual, Statutory, etc.) [102] [103]. Though not implemented yet, design Phase 2/3 tasks to add this:
    - Create a **Ledger** model (e.g., one for "Local (NFRS)" and one for "IFRS") and tag every Journal or GL entry with which ledger it belongs to. A primary ledger (local books) and a secondary ledger (IFRS adjustments) would let the system generate both NFRS statements and IFRS statements.
    - This might involve removing currency fields from Journal (as the plan indicates) and storing valuations for each ledger separately [104] [105]. While this is a major enhancement, acknowledging it in the plan ensures the system is future-proof for full IFRS compliance.
    - In the interim, the system can still handle IFRS vs GAAP adjustments by manual journal entries (e.g., create adjusting entries at period end). Advise users that any differences (like different depreciation methods) can be booked via journal entries. The reports can then be produced accordingly if those adjustments are included or excluded.
- **Audit and Control:** To comply with auditing standards:
- **Immutable Records:** Once a journal is posted, it should not be deletable. This is already enforced conceptually by using reversal instead of deletion and by locking posted entries. You might also implement database constraints or at least application logic to prevent deletion of posted journals or used accounts. For instance, do not allow deleting an account that has transactions; instead, allow deactivation (`is_active=False`).

- **Audit Trail:** As mentioned, every change should be logged (who changed what and when). Auditors will expect the system to produce audit logs on demand. We should test that our AuditLog captures key events (creation of a new fiscal year, editing an account, posting a journal, etc.) and consider adding a UI for administrators to view these logs.
- **User Access Reviews:** Provide reports or utilities for admins to review which users have which roles/permissions (to facilitate periodic access reviews, a common audit requirement). For example, an export of all UserRoles in the system, showing any privilege assignments.
- **Reconciliation Features:** GAAP/IFRS require periodic reconciliation (e.g., bank reconciliations, subledger vs general ledger reconciliation for AR/AP). While a full bank reconciliation module might be outside scope, ensure the GL can support it (the ChartOfAccount has fields like `last_reconciled_date` and `reconciled_balance` [106], implying a reconciliation feature). As an enhancement, implement a simple bank reconciliation UI for bank accounts (mark transactions as cleared, compare to bank statements, and update those fields).
- **Compliance with Nepal IRD E-Billing:** Address the government requirements for electronic billing systems:
- **Sequential Numbering & Gaps:** The system must guarantee that invoice/voucher numbers are sequential and cannot be duplicated or skipped without record. Our JournalType auto-numbering covers this [26]. Additionally, ensure that if a number is voided (e.g., a journal entry is created and then reversed), the gap is documented (perhaps print a “cancelled voucher” with that number for audit, or at least keep the journal record with a cancelled status so the number is not reused).
- **Data Integrity Controls:** Implement measures to prevent tampering:
    - Once an invoice or voucher is approved/posted, disallow editing critical fields (amount, account, date). If changes are needed, require a formal adjustment entry. This aligns with IRD’s requirement that software should prevent retroactive edits that alter reported data [107].
    - Possibly generate a hash or unique code for each invoice/voucher that can be used to verify it hasn’t been altered (some e-billing systems use digital signatures or QR codes for verification).
- **Mandatory Fields on Invoices:** When printing invoices or sales bills, include all fields required by IRD’s guideline (e.g., buyer’s PAN for B2B transactions, a unique invoice identifier, the company’s IRD approval number for the billing software, etc.). The seeded tax codes and authority info will feed into these (for example, display "Inland Revenue Dept. – VAT 13%" on invoice).
- **E-Invoice Data Export:** Prepare for eventual integration with IRD:
    - The IRD in Nepal might require businesses to upload sales data or use an API. Even if not implemented now, structure the data so that it can be exported easily. For instance, ensure every sales transaction captures customer details and VAT breakdown so you can generate the monthly VAT report or CSV for IRD.
    - Build a feature to export all tax invoices within a date range in a government-specified format (if they have one) – this helps users comply with filing requirements.
    - If possible, integrate an API (some countries have real-time invoice verification APIs). If IRD provides an API endpoint, consider adding a function to automatically transmit each invoice when posted and store the acknowledgment or reference number from the tax authority.
- **Certification Checklists:** Cross-check the system against the IRD’s e-billing software checklist [107]:
    - Does the system maintain logs of all operations? (Yes, via AuditLog and LoginLog).
    - Does it prevent deletion/modification of issued invoices? (Yes, through posting lock and reversal approach).
    - Does it back up data and allow retrieval of any invoice by number? (We should ensure easy search of invoices/journals by number).

    - These kinds of items should be verified to ensure the software could be approved by regulators.
- **Local Accounting Standards (NFRS):** NFRS is largely IFRS-aligned, but ensure compliance with any specific local requirements:
- For example, Nepal mandates that financial statements be presented in Nepali Rupees and maybe with Nepali dates in reports submitted to authorities. Consider adding the ability to output reports in Bikram Sambat (BS) date format, or at least allow an option to display dates in BS alongside AD. This might not be a core feature but could be a value-add for local users.
- If any specific accounts or disclosures are required (like a specific reserve or reporting of a Social Security Fund usage), ensure the COA and reporting can accommodate that (the seed already includes Social Security Fund payable account [108]).
- Keep updated with any changes in local laws (e.g., new taxes or changes in rates) and ensure the system can be easily updated (through configuration rather than code where possible).
- **Quality and Testing:** Finally, to meet all these standards, thorough testing is needed:
- Perform **unit tests and integration tests** for each module (journal posting, permission enforcement, report calculations, etc.) – the system should have test cases (e.g., the repository has Cypress tests for journal entry keyboard shortcuts). Expand tests to cover permission scenarios (user without permission cannot access a view), posting logic (balances update correctly), and report accuracy (trial balance always nets to zero, etc.).
- User acceptance testing with a sample company's data under an auditor's eye would be ideal to ensure nothing is missing for compliance.
- Security testing: attempt common exploits (e.g., URL without permission, editing HTML to enable a disabled button) to ensure the permission backend truly blocks unauthorized actions.

By addressing all the above points, the accounting module will be feature-complete and robust. It will have a fine-grained security model, a full journal workflow, rich reporting with exports, customizable print templates, all necessary seed data (accounts, taxes, currencies, etc.), and compliance with international and local accounting standards [102] . This comprehensive enhancement plan sets the stage for a reliable ERP accounting system that can serve users for months of operations and meet audit requirements confidently. Each bullet point here can be taken as a development **prompt/task** to be implemented and checked off, driving the project toward a successful, compliant accounting module deployment [101] .

---

[1] [2] [25] [35] [36] [49] [50] [51] [52] [64] [65] [66] [67] [68] [69] [72] [73] [74] [75] [76] [77] [79] [80] [81] [82] [86] [89] [90] [91] [92] [93] [94] [95] [97] [98] [99] [100] [101] [108] create_default_data.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/scripts/create_default_data.py

[3] [10] [13] models.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/usermanagement/models.py

[4] [14] [16] [18] [19] [20] [21] [22] [23] [26] [27] [28] [29] [30] [31] [32] [33] [34] [40] [41] [61] [70] [71] [83] [84] [85] [87] [88] [106] models.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/accounting/models.py

[5] [7] [15] views_mixins.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/accounting/views/views_mixins.py

6 8 37 38 39 78 views.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/accounting/views/views.py

9 permission_tags.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/usermanagement/templatetags/permission_tags.py

11 12 utils.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/usermanagement/utils.py

17 24 setup_journal_permissions.py
https://github.com/ritsnep/ERP/blob/a9720c59b33e554df34deb64e5a5ec377bd1fce1/usermanagement/management/commands/setup_journal_permissions.py

42 43 44 45 46 47 48 53 56 report_views.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/accounting/views/report_views.py

54 55 57 58 59 60 62 63 report_export_service.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/accounting/services/report_export_service.py

96 setup_permissions.py
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/usermanagement/management/commands/setup_permissions.py

102 103 104 105 multi_currency_plan.md
https://github.com/ritsnep/HimalytixNew/blob/553f89567301921056340ec78dc95102bcca633c/ERP/accounting/docs/multi_currency_plan.md

107 Compliance Checklist for E-billing Software: What Every Nepali ...
https://www.tiggapp.com/blog-posts/compliance-checklist-for-e-billing-software-what-every-nepali-business-must-know