Okay, this is a solid project idea with clear roles and functionalities. Let's break down a professional approach and structure.

**I. Core Concepts & Considerations**

1. **Currency Pairs & Rates:**

* You'll need to define buy and sell rates for each pair from the *system's perspective*.
* Example: EUR/USD
* System Buys EUR (Sells USD): Customer gives EUR, gets USD. Rate: e.g., 1 EUR = 1.08 USD.
* System Sells EUR (Buys USD): Customer gives USD, gets EUR. Rate: e.g., 1 EUR = 1.10 USD.
* The difference (1.10 - 1.08 = 0.02 USD per EUR) is the system's spread/potential profit margin.
* You'll have these pairs:
* EUR/USD (and USD/EUR implicitly, it's the inverse rate with spread)
* RWF/USD (and USD/RWF)
* EUR/RWF (and RWF/EUR)
* Admin updates these rates. It's crucial to store a history of rates with timestamps.

1. **Profit Calculation:**

* **Per Transaction:** This is the most granular. For each exchange, the profit can be calculated based on the spread of the rates used.
* Example: Teller sells 100 EUR to customer for 110 USD. If the system's "buy" rate for EUR (its own cost basis) was 1.08 USD, the profit on this transaction could be considered (110 USD received) - (100 EUR \* 1.08 USD/EUR) = 110 - 108 = 2 USD.
* *Simplification for your system:* Since the Admin sets both buy and sell rates, the "profit" on a transaction where the system *sells* currency X (and receives currency Y) is: (Amount of Y received) - (Amount of X sold \* System's Buy Rate for X in terms of Y).
* **Daily/Monthly Profit:** Sum of profits from all transactions within the period. This should ideally be viewable in a base currency (e.g., USD equivalent).

1. **Teller Cash Management:**

* **Initial Float:** Admin assigns specific amounts of *each currency* to the teller (e.g., 5000 EUR, 10000 USD, 2,000,000 RWF).
* **End of Day (EOD) Reporting:**
* Teller declares their final holdings of *each currency*.
* System should be able to calculate the *expected* EOD holdings based on initial float and all transactions.
* Expected\_End\_Currency\_X = Initial\_Currency\_X + Total\_Currency\_X\_Received\_In\_Transactions - Total\_Currency\_X\_Paid\_Out\_In\_Transactions
* Any discrepancy between teller's declared EOD and system's expected EOD needs to be highlighted.

1. **Security & Auditability:**

* Authentication (JWT is good) and Authorization (role-based access control).
* All rate changes, cash assignments, and transactions must be logged with user, timestamp, and details.

**II. Project Structure (Monorepo or Separate Repos)**

A monorepo (e.g., using Nx or Turborepo, or just Yarn/NPM workspaces) can simplify shared types and configurations, but separate repos are also fine.

**A. Backend (NestJS)**

/backend

├── src

│ ├── auth/ # Authentication (login, JWT, guards)

│ │ ├── auth.controller.ts

│ │ ├── auth.service.ts

│ │ ├── auth.module.ts

│ │ ├── dto/

│ │ └── strategies/ # (jwt.strategy.ts)

│ │ └── guards/ # (jwt-auth.guard.ts, roles.guard.ts)

│ │

│ ├── users/ # User management (Admin, Teller)

│ │ ├── users.controller.ts

│ │ ├── users.service.ts

│ │ ├── users.module.ts

│ │ ├── dto/

│ │ └── entities/user.entity.ts

│ │

│ ├── currencies/ # Managing currency definitions (EUR, USD, RWF)

│ │ ├── currencies.controller.ts

│ │ ├── currencies.service.ts

│ │ ├── currencies.module.ts

│ │ ├── dto/

│ │ └── entities/currency.entity.ts

│ │

│ ├── rates/ # Exchange rate management

│ │ ├── rates.controller.ts # Admin sets rates here

│ │ ├── rates.service.ts

│ │ ├── rates.module.ts

│ │ ├── dto/

│ │ └── entities/exchange-rate.entity.ts (from\_currency, to\_currency, buy\_rate, sell\_rate, effective\_at, set\_by\_admin\_id)

│ │

│ ├── transactions/ # Handling actual exchanges

│ │ ├── transactions.controller.ts # Teller performs exchanges

│ │ ├── transactions.service.ts

│ │ ├── transactions.module.ts

│ │ ├── dto/

│ │ └── entities/transaction.entity.ts (teller\_id, from\_currency, from\_amount, to\_currency, to\_amount, rate\_used, profit, timestamp)

│ │

│ ├── teller-sessions/ # Managing teller's daily cash

│ │ ├── teller-sessions.controller.ts # Admin assigns cash, Teller submits EOD

│ │ ├── teller-sessions.service.ts

│ │ ├── teller-sessions.module.ts

│ │ ├── dto/

│ │ └── entities/

│ │ ├── teller-session.entity.ts (teller\_id, start\_time, end\_time, status)

│ │ └── teller-session-balance.entity.ts (session\_id, currency\_id, initial\_amount, final\_amount\_reported, final\_amount\_calculated)

│ │

│ ├── reporting/ # Profit reports, teller activity

│ │ ├── reporting.controller.ts

│ │ ├── reporting.service.ts

│ │ ├── reporting.module.ts

│ │ └── dto/ (for report outputs)

│ │

│ ├── shared/ # Common utilities, enums, interfaces, base entities

│ │ ├── enums/ (user-role.enum.ts, currency-code.enum.ts)

│ │ ├── decorators/ (roles.decorator.ts)

│ │ └── ...

│ │

│ ├── app.module.ts

│ ├── app.controller.ts

│ ├── app.service.ts

│ └── main.ts

│

├── test/ # E2E and unit tests

├── .env # Environment variables

├── ormconfig.json (or TypeORM config in app.module)

├── package.json

└── tsconfig.json

**B. Frontend (ReactJS - using Vite or Create React App)**

/frontend

├── public/

├── src

│ ├── App.tsx

│ ├── main.tsx (or index.tsx)

│ ├── assets/

│ ├── components/ # Reusable UI components

│ │ ├── common/ # (Button, Input, Modal, Table, Navbar, Sidebar)

│ │ ├── admin/ # (UserForm, RateForm, ReportChart)

│ │ └── teller/ # (ExchangeForm, EODReportForm)

│ │

│ ├── pages/ # Route-level components

│ │ ├── auth/

│ │ │ └── LoginPage.tsx

│ │ ├── admin/

│ │ │ ├── AdminDashboardPage.tsx

│ │ │ ├── TellerManagementPage.tsx

│ │ │ ├── RateManagementPage.tsx

│ │ │ ├── CashAssignmentPage.tsx

│ │ │ ├── DailyReportsPage.tsx

│ │ │ └── MonthlyReportsPage.tsx

│ │ └── teller/

│ │ ├── TellerDashboardPage.tsx # Main exchange interface

│ │ └── EndOfDayReportPage.tsx

│ │

│ ├── layouts/ # (AdminLayout.tsx, TellerLayout.tsx, AuthLayout.tsx)

│ │

│ ├── services/ # API call functions (e.g., using Axios or fetch)

│ │ ├── auth.service.ts

│ │ ├── userService.ts

│ │ ├── rateService.ts

│ │ └── ...

│ │

│ ├── hooks/ # Custom React hooks

│ │ └── useAuth.ts

│ │

│ ├── store/ # State management (Zustand, Redux Toolkit, or Context API)

│ │ ├── authStore.ts

│ │ └── ...

│ │

│ ├── routes/ # Routing configuration (using React Router)

│ │ ├── ProtectedRoute.tsx

│ │ └── index.tsx

│ │

│ ├── types/ # TypeScript interfaces/types (can be shared with backend if using monorepo)

│ │ ├── user.types.ts

│ │ ├── rate.types.ts

│ │ └── ...

│ │

│ └── utils/ # Utility functions (formatting, validation)

│

├── .env

├── package.json

├── tsconfig.json

└── vite.config.ts (if using Vite)

IGNORE\_WHEN\_COPYING\_START

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).

IGNORE\_WHEN\_COPYING\_END

**III. Database Schema (Example with PostgreSQL)**

1. **users**

* id (UUID, PK)
* username (VARCHAR, UNIQUE, NOT NULL)
* password\_hash (VARCHAR, NOT NULL)
* full\_name (VARCHAR)
* role (ENUM('admin', 'teller'), NOT NULL)
* is\_active (BOOLEAN, DEFAULT TRUE)
* created\_at (TIMESTAMPZ, DEFAULT NOW())
* updated\_at (TIMESTAMPZ, DEFAULT NOW())

1. **currencies** (Could be seeded or managed if you plan to add more)

* id (SERIAL, PK)
* code (VARCHAR(3), UNIQUE, NOT NULL) e.g., "USD", "EUR", "RWF"
* name (VARCHAR, NOT NULL) e.g., "US Dollar", "Euro", "Rwandan Franc"
* symbol (VARCHAR(5))

1. **exchange\_rates** (Stores history of rates)

* id (UUID, PK)
* from\_currency\_id (INTEGER, FK to currencies.id)
* to\_currency\_id (INTEGER, FK to currencies.id)
* admin\_sell\_rate (NUMERIC(18,8), NOT NULL) -- Rate at which system SELLS from\_currency FOR to\_currency
* admin\_buy\_rate (NUMERIC(18,8), NOT NULL) -- Rate at which system BUYS from\_currency WITH to\_currency
* effective\_from\_timestamp (TIMESTAMPZ, NOT NULL, DEFAULT NOW())
* set\_by\_admin\_id (UUID, FK to users.id)
* created\_at (TIMESTAMPZ, DEFAULT NOW())
* *Constraint:* UNIQUE (from\_currency\_id, to\_currency\_id, effective\_from\_timestamp)
* *Note:* When a teller needs a rate, they query for the latest effective\_from\_timestamp <= NOW() for the given pair.

1. **teller\_sessions**

* id (UUID, PK)
* teller\_id (UUID, FK to users.id)
* start\_timestamp (TIMESTAMPZ, DEFAULT NOW())
* end\_timestamp (TIMESTAMPZ, NULLABLE)
* status (ENUM('open', 'closed', 'reconciled'), DEFAULT 'open')
* assigned\_by\_admin\_id (UUID, FK to users.id)
* notes (TEXT, NULLABLE)

1. **teller\_session\_balances** (Initial and final cash for a session)

* id (UUID, PK)
* teller\_session\_id (UUID, FK to teller\_sessions.id)
* currency\_id (INTEGER, FK to currencies.id)
* initial\_amount (NUMERIC(18,2), NOT NULL)
* final\_amount\_reported\_by\_teller (NUMERIC(18,2), NULLABLE)
* final\_amount\_calculated\_by\_system (NUMERIC(18,2), NULLABLE) -- Calculated after session close
* discrepancy (NUMERIC(18,2), NULLABLE)
* *Constraint:* UNIQUE (teller\_session\_id, currency\_id)

1. **transactions**

* id (UUID, PK)
* teller\_session\_id (UUID, FK to teller\_sessions.id)
* teller\_id (UUID, FK to users.id)
* customer\_gives\_currency\_id (INTEGER, FK to currencies.id)
* customer\_gives\_amount (NUMERIC(18,2), NOT NULL)
* customer\_receives\_currency\_id (INTEGER, FK to currencies.id)
* customer\_receives\_amount (NUMERIC(18,2), NOT NULL)
* rate\_snapshot\_id (UUID, FK to exchange\_rates.id, optional, or store rate values directly)
* *Alternatively and perhaps better:*
* system\_sell\_rate\_used (NUMERIC(18,8)) -- If system sold customer\_receives\_currency
* system\_buy\_rate\_used (NUMERIC(18,8)) -- If system bought customer\_gives\_currency
* profit\_base\_currency (VARCHAR(3), DEFAULT 'USD') -- e.g., USD
* profit\_amount (NUMERIC(18,2), NOT NULL)
* transaction\_timestamp (TIMESTAMPZ, DEFAULT NOW())

**IV. Workflow Highlights**

1. **Admin: Add Teller:** Creates a user with 'teller' role.
2. **Admin: Update Rates:** Sets new admin\_sell\_rate and admin\_buy\_rate for currency pairs. This creates a new exchange\_rates record.
3. **Admin: Assign Morning Cash:**

* Creates a teller\_sessions record for a specific teller.
* Creates teller\_session\_balances records for that session, one for each currency with initial\_amount.

1. **Teller: Perform Exchange:**

* Selects "Customer Gives" currency/amount and "Customer Receives" currency.
* System fetches the current active rate (latest exchange\_rates for the pair).
* If customer gives EUR for USD: System uses its admin\_buy\_rate for EUR/USD.
* If customer gives USD for EUR: System uses its admin\_sell\_rate for EUR/USD (or inverse of admin\_buy\_rate for USD/EUR).
* System calculates customer\_receives\_amount.
* Teller confirms. A transactions record is created.
* Profit is calculated for this transaction and stored.
* Teller's virtual cash balance (in the system, not yet the EOD) is updated.

1. **Teller: End of Day Report:**

* Teller inputs final\_amount\_reported\_by\_teller for each currency in teller\_session\_balances.
* Teller "closes" their session. teller\_sessions.status -> 'closed'.

1. **Admin/System: Reconciliation & Reporting:**

* System calculates final\_amount\_calculated\_by\_system for the closed session based on initial\_amount and all transactions for that session.
* Calculates discrepancy.
* Admin can view daily/monthly profit reports (summing profit\_amount from transactions).
* Admin can view teller EOD reports and discrepancies.

**V. Professional Practices / Other Ways**

1. **API Design:** Use RESTful principles. Consider OpenAPI/Swagger for documentation (NestJS has good integration).
2. **DTOs (Data Transfer Objects):** Use DTOs for request/response validation and shaping in NestJS (class-validator, class-transformer).
3. **Error Handling:** Consistent error responses from the API.
4. **Logging:** Implement comprehensive logging (e.g., Winston) for auditing and debugging.
5. **Testing:**

* **Backend:** Unit tests (Jest), Integration tests (with a test database).
* **Frontend:** Unit tests (Jest/React Testing Library), E2E tests (Cypress/Playwright).

1. **Configuration:** Use .env files for environment-specific configurations.
2. **Security:**

* HTTPS for all communication.
* Regularly update dependencies.
* Input validation on both frontend and backend.
* Protect against common web vulnerabilities (OWASP Top 10).
* Rate limiting on sensitive endpoints.

1. **Code Quality:** ESLint, Prettier for consistent code style.
2. **Database:**

* Use migrations (e.g., TypeORM migrations) for schema changes.
* Consider indexing frequently queried columns.
* Transactions for operations that need atomicity (e.g., creating a transaction and updating balances).

1. **State Management (React):**

* For global state like user auth, rates: Zustand, Redux Toolkit, or Jotai.
* For local component state: useState, useReducer.
* React Query (TanStack Query) is excellent for managing server state (fetching, caching, updating API data).

1. **Real-time (Optional Enhancement):** If rates change frequently and need to be pushed to tellers instantly, consider WebSockets (NestJS Gateways). For most cases, tellers fetching rates on new transaction initiation is fine.

**VI. Phased Development Approach**

1. **MVP (Minimum Viable Product):**

* Admin login, add teller.
* Admin sets rates for EUR/USD.
* Teller login.
* Teller performs EUR/USD exchange.
* Basic daily profit calculation (sum of transaction profits).
* No complex cash assignment/EOD yet, just focus on the exchange.

1. **Phase 2:**

* Full currency pair support (RWF/USD, EUR/RWF).
* Admin cash assignment to tellers.
* Teller EOD reporting (manual input).
* System calculation of expected EOD and discrepancy.

1. **Phase 3:**

* Detailed reporting (monthly profit, teller performance).
* UI/UX refinements.
* Enhanced security measures.
* Historical rate views.

This detailed breakdown should give you a strong foundation. Remember to start simple, build incrementally, and test thoroughly. Good luck!