

# Wallet App — Roadmap técnico por módulos (Backend + Frontend)

**Objetivo:** Documentar el plan de construcción del sistema (API Express + MySQL y Frontend React) con **módulos**, **endpoints detallados**, **validaciones** y **tareas de frontend**. Todo alineado al esquema SQL ya importado.

## Convenciones generales

- **Base URL API (prod):** `https://wallet-api-production-2e8a.up.railway.app/api`
- **Base URL API (dev local):** `http://localhost:4000/api`
- **Autenticación inicial:** sin login; para scope usamos header `x-user-id` y validamos pertenencia de recursos.
- **Formato dinero:** centavos (`BIGINT`).
- **Fechas:** `YYYY-MM-DD`.
- **Validación:** `zod` en backend (inputs) y opcionalmente en frontend (formularios y respuestas).
- **DB:** MySQL 8. Tablas: `users`, `accounts`, `categories`, `pay_periods`, `planned_payments`, `transactions`, `saving_entries`, `saving_goals`, `saving_entry_goals`.

## Estructura de rutas en backend (un archivo por módulo)

Carpeta `src/routes/` :- `health.ts` → `/api/health`, `/api/db-ping` - `users.ts` → `/api/users/*` - `accounts.ts` → `/api/accounts/*` - `categories.ts` → `/api/categories/*` - `payPeriods.ts` → `/api/pay-periods/*` - `transactions.ts` → `/api/transactions/*` - `planned.ts` → `/api/planned/*` - `savings.ts` → `/api/savings/*` - `goals.ts` → `/api/savings/goals/*` y vínculos `saving_entry_goals` - `summary.ts` → `/api/summary/*`

**server.ts** (registro de rutas):

```
import express from 'express'
import cors from 'cors'
import { health } from './routes/health'
import { router as users } from './routes/users'
import { router as accounts } from './routes/accounts'
import { router as categories } from './routes/categories'
import { router as payPeriods } from './routes/payPeriods'
import { router as transactions } from './routes/transactions'
import { router as planned } from './routes/planned'
import { router as savings } from './routes/savings'
import { router as goals } from './routes/goals'
import { router as summary } from './routes/summary'

const app = express()
app.use(cors())
app.use(express.json())
```

```

app.use('/api/health', health)
app.use('/api/users', users)
app.use('/api/accounts', accounts)
app.use('/api/categories', categories)
app.use('/api/pay-periods', payPeriods)
app.use('/api/transactions', transactions)
app.use('/api/planned', planned)
app.use('/api/savings', savings)
app.use('/api/savings/goals', goals)
app.use('/api/summary', summary)

```

**Convención de nombres de archivos:** singular del dominio en lowerCamel + `.ts` (`users.ts`, `accounts.ts`, etc.). **Convención de rutas:** plural en kebab-case a nivel HTTP (`/api/pay-periods`, `/api/saving-entries` si separas), manteniendo consistencia con el documento.

---

## 0) Health & Infra

### Backend

- GET `/api/health` → `{ ok: true, service: 'wallet-api', ts }`
- GET `/api/db-ping` → valida conexión MySQL.

### Frontend

- Endpoint check en `src/lib/api.ts`.
  - Banner en UI si el backend no responde.
- 

## 1) Users (usuarios)

**Tabla:** `users(id, name, email, created_at)`

### Endpoints

1. **POST** `/api/users`
2. **Body:** `{ name: string(min 2), email?: string|null }`
3. **201:** `{ id: number }`
4. **400:** validación fallida
5. **409:** email ya existe (si se envía)
6. **SQL:** `INSERT INTO users (name,email) VALUES (?, ?)`
7. **GET** `/api/users/:id`
8. **200:** `{ id, name, email, created_at }`
9. **404:** no existe

10. **SQL:** `SELECT id,name,email,created_at FROM users WHERE id=?`

11. (Opcional) **GET /api/users**

12. **200:** `Array<User>`

13. **SQL:** `SELECT id,name,email,created_at FROM users ORDER BY id DESC`

## Frontend

- **Pantallas:** Onboarding/Settings → Crear usuario, Seleccionar usuario activo.
- **Estado:** `useAuthStore` (solo `activeUserId` por ahora) + persistencia en `localStorage`.
- **API:** `createUser`, `getUser`, `listUsers` (opcional).

---

## 2) Accounts (cuentas)

**Tabla:** `accounts(id, user_id, name, type[cash|bank|credit|savings], currency, is_active, created_at)`

### Endpoints

1. **GET /api/accounts/user/:userId**

2. **200:** `Array<{ id, user_id, name, type, currency, is_active, created_at }>`

3. **SQL:** `SELECT ... FROM accounts WHERE user_id=? ORDER BY created_at DESC`

4. **POST /api/accounts**

5. **Body:** `{ user_id: number, name: string, type: 'cash'|'bank'|'credit'|'savings', currency?: 'AUD'|string, is_active?: boolean }`

6. **201:** `{ id: number }`

7. **400:** validación

8. **404:** `user_id` no existe

9. **SQL:** `INSERT INTO accounts (user_id,name,type,currency,is_active) VALUES (?, ?, ?, ?, ?)`

10. (Opcional) **PATCH /api/accounts/:id** (activar/desactivar, renombrar)

11. **Body:** `{ name?, is_active?, currency? }`

12. **200:** `{ updated: true }`

13. **404:** no existe / no pertenece al user

## Frontend

- **Pantalla:** Cuentas (lista, crear, activar/desactivar).
- **API:** `listAccountsByUser(userId)`, `createAccount`, `updateAccount`.
- **Estado:** `useWalletStore.accounts`.

### 3) Categories (categorías)

**Tabla:** `categories(id, user_id|null, name, kind[income|expense|transfer|adjustment])`

#### Endpoints

1. **GET** `/api/categories` (*query opcional* `user_id`)
2. **200:** `Array<{ id, user_id, name, kind }>` (globales `user_id=NULL` + personales)
3. **SQL:** `SELECT ... FROM categories WHERE user_id IS NULL OR user_id=?`
4. **POST** `/api/categories`
5. **Body:** `{ user_id?: number|null, name: string, kind: 'income'|'expense'|'transfer'|'adjustment' }`
6. **201:** `{ id: number }`
7. **400:** validación
8. **409:** `UNIQUE (user_id,name)` violada

#### Frontend

- **Uso:** Selects de categoría en formularios.
- **Pantalla (opcional):** gestión de categorías personales.
- **API:** `listCategories(userId)`, `createCategory`.

### 4) Pay Periods (quincenas)

**Tabla:** `pay_periods(id, user_id, pay_date, gross_income_cents, note, created_at)`  
con **UNIQUE** (`user_id, pay_date`)

#### Endpoints

1. **GET** `/api/pay-periods/user/:userId`
2. **200:** `Array<{ id,user_id,pay_date,gross_income_cents,note }>`
3. **SQL:** `SELECT ... WHERE user_id=? ORDER BY pay_date DESC`
4. **POST** `/api/pay-periods` (*upsert*)
5. **Body:** `{ user_id, pay_date, gross_income_cents?: number, note?: string }`
6. **201/200:** `{ id | upsertedId }`
7. **SQL:** `INSERT ... ON DUPLICATE KEY UPDATE gross_income_cents=VALUES(...), note=VALUES(...)`

#### Frontend

- **Pantalla:** Quincenas (lista + crear/editar ingreso bruto y nota).
- **API:** `listPayPeriods(userId)`, `upsertPayPeriod`.

- Estado: `useWalletStore.payPeriods`.

## 5) Transactions (transacciones)

**Tabla:** `transactions(id, user_id, pay_period_id?, account_id, category_id?, type[income|expense|transfer|adjustment], amount_cents, description?, txn_date, planned_payment_id?, counterparty_user_id?, created_at)`

**Reglas** - `amount_cents` con **signo:** `income/adjustment` que **entran** `>0`; `expense/transfer` que **salen** `<0`. - Validar pertenencia de `account_id` al `user_id`.

### Endpoints

1. **GET** `/api/transactions/user/:userId` con filtros
2. **Query:** `from?`, `to?`, `pay_period_id?`, `limit?`, `offset?`
3. **200:** `Array<Transaction>`
4. **SQL:** índices `idx_txn_user_date`, `idx_txn_payperiod`.

5. **POST** `/api/transactions`

6. **Body:**

```
{
  "user_id": 1,
  "pay_period_id": 5,
  "account_id": 1,
  "category_id": 3,
  "type": "income|expense|transfer|adjustment",
  "amount_cents": 12345,
  "description": "string?",
  "txn_date": "YYYY-MM-DD",
  "planned_payment_id": null,
  "counterparty_user_id": 2
}
```

7. **201:** `{ id }`
8. **400:** validación / signo incorrecto
9. **404:** `user/account/category` inexistentes
10. (Opcional) **DELETE** `/api/transactions/:id`
11. **200:** `{ deleted: true }` (o usa soft-delete en el futuro)

### Frontend

- **Pantalla:** Lista de transacciones con filtros (rango de fechas, quincena, cuenta, categoría, tipo) + Form de creación.

- **API:** `listTransactions(params)`, `createTransaction`, `deleteTransaction` (opcional).
- **Estado:** `useWalletStore.transactions`.

## 6) Planned Payments (pagos planificados / reservas)

**Tabla:** `planned_payments(id, user_id, account_id?, description, amount_cents, due_date, auto_debit, status['planned'|'executed'|'canceled'], linked_txn_id?, created_at)`

### Endpoints

1. **GET** `/api/planned/user/:userId`
2. **Query:** `from?`, `to?`, `status?`
3. **200:** `Array<PlannedPayment>`
4. **POST** `/api/planned`
5. **Body:** `{ user_id, account_id?, description, amount_cents, due_date, auto_debit?: boolean }`
6. **201:** `{ id }`
7. **PATCH** `/api/planned/:id/execute`
8. **Body:** `{ txn_date: 'YYYY-MM-DD', account_id?: number, category_id?: number, description?: string }`
9. **200:** `{ executed: true, txn_id }`
10. **Efecto:** crea `transactions` `type='expense'` (monto **negativo**) y actualiza `planned_payments.status='executed'` + `linked_txn_id`.
11. (Opcional) **PATCH** `/api/planned/:id/cancel` → `{ canceled: true }`

### Frontend

- **Pantalla:** Lista de planificados + filtros + botón **Ejecutar/Cancelar**.
- **API:** `listPlanned`, `createPlanned`, `executePlanned`, `cancelPlanned`.
- **Estado:** `useWalletStore.planned`.

## 7) Savings (entradas de ahorro)

**Tabla:** `saving_entries(id, user_id, pay_period_id?, account_id, amount_cents, entry_date, note?, created_at)`

## Endpoints

1. **GET** `/api/savings/entries/user/:userId`
2. **Query:** `pay_period_id?`, `from?`, `to?`
3. **200:** `Array<SavingEntry>`
4. **POST** `/api/savings/entries`
5. **Body:** `{ user_id, pay_period_id?, account_id, amount_cents(>0 depósito | <0 retiro), entry_date, note? }`
6. **201:** `{ id }`

## Frontend

- **Pantalla:** Registro de aportes a ahorro (por quincena) + historial.
- **API:** `listSavingEntries`, `createSavingEntry`.
- **Estado:** `useWalletStore.savings`.

---

## 8) Saving Goals (metas) + vínculo entrada→meta

**Tablas:** `saving_goals(id, user_id, name, target_amount_cents, target_date, created_at)` y `saving_entry_goals(saving_entry_id, goal_id)`

### Endpoints (goals)

1. **GET** `/api/savings/goals/user/:userId`
2. **200:** `Array<{ id,name,target_amount_cents,target_date, created_at, saved_cents, remaining_cents }>`
3. **SQL:** `LEFT JOIN saving_entry_goals + saving_entries para saved_cents`.
4. **POST** `/api/savings/goals`
5. **Body:** `{ user_id, name, target_amount_cents, target_date }`
6. **201:** `{ id }`
7. (Opcional) **DELETE** `/api/savings/goals/:id` → `{ deleted: true }` (si no hay dependencias)

### Endpoints (link entry→goal)

1. **POST** `/api/savings/entries/:entryId/assign-goal/:goalId`
2. **200:** `{ linked: true }`
3. **DELETE** `/api/savings/entries/:entryId/assign-goal/:goalId`
4. **200:** `{ unlinked: true }`

## Frontend

- **Pantalla:** Metas (crear, ver progreso); desde cada aporte, **Asignar a meta**.

- **API:** `listGoalsWithProgress`, `createGoal`, `assignEntryToGoal`, `unassignEntryToGoal`.
  - **Extras UI:** cálculo de **aporte mínimo por quincena** hasta `target_date`.
- 

## 9) Summary (resumen por quincena)

**Objetivo:** derivar métricas de una quincena: ingresos+ajustes, gastos, reservas, ahorros, leftover.

### Endpoint

- **GET** `/api/summary/pay-period/:id`
- **200:**

```
{
  "pay_period_id": 5,
  "pay_date": "2025-10-07",
  "income_in_cents": 0,
  "expenses_out_cents": 0,
  "reserved_planned_cents": 0,
  "savings_out_cents": 0,
  "leftover_cents": 0
}
```

- **SQL guía:** usar la vista `v_pay_period_summary` para income/expenses/savings y sumar `reserved_planned_cents` con subquery a `planned_payments` en el rango `[pay_date, pay_date+14d)`.

### Frontend

- **Dashboard:** tarjeta por quincena con métricas + tendencia.
  - **API:** `getPayPeriodSummary(id)`; batch para lista.
- 

## 10) Export / Import (backups)

### Endpoints

- **GET** `/api/export/user/:userId` → JSON con: `accounts`, `categories`, `pay_periods`, `transactions`, `planned_payments`, `saving_entries`, `saving_goals`, `saving_entry_goals`.
- **POST** `/api/import`
- **Body:** `{ dryRun?: boolean, data: {...} }`
- **200:** `{ inserted, updated, skipped }`

### Frontend

- **Settings:** botones **Exportar JSON** / **Importar JSON**.
-



## 11) Seguridad (fase posterior)

- **Ahora:** `x-user-id` + validación de pertenencia.
  - **Luego:** Auth con JWT (registro/login), middlewares, password hashing, refresh tokens.
- 

## 12) Orden de construcción sugerido

1) **Users** → crear/obtener, setear `activeUser` en FE. 2) **Accounts** → crear/listar por usuario. 3) **Categories** → listar global + crear personales. 4) **Pay Periods** → upsert/listar. 5) **Transactions** → crear/listar con filtros. 6) **Planned Payments** → crear/listar/ejecutar. 7) **Savings** → entradas; **Saving Goals** → metas + progreso; enlace entry→goal. 8) **Summary** → endpoint de resumen por quincena. 9) **Export/Import**.

---

## 13) Validaciones y errores comunes (por módulo)

- **Users:** `name` min 2; `email` válido y único si se envía.
  - **Accounts:** `type` ∈ {cash,bank,credit,savings}; `currency` 3 letras; `user_id` existe.
  - **Categories:** `kind` ∈ {income,expense,transfer,adjustment}; (`user_id`,`name`) único.
  - **Pay Periods:** (`user_id`,`pay_date`) único; `gross_income_cents` ≥ 0.
  - **Transactions:** signo de `amount_cents` vs `type`; `account_id` pertenece a `user_id`.
  - **Planned:** `amount_cents` > 0 (monto a reservar); `due_date` válida; `execute` crea `transaction` negativa.
  - **Savings:** `amount_cents` > 0 depósito / < 0 retiro; `account_id` (savings recomendado).
  - **Goals:** `target_amount_cents` > 0; `target_date` ≥ hoy.
- 

## 14) Frontend: capas y utilidades

- `src/lib/api.ts`: fetchers por módulo (tipados con `zod` si quieres). Usa `BASE_URL` configurable por entorno (`prod / dev`). Ej.: `const BASE_URL = import.meta.env.VITE_API_BASE ?? 'https://wallet-api-production-2e8a.up.railway.app/api'`.
- `zustand`: stores por dominio (`useUserStore`, `useAccountStore`, etc.) o un store global con slices.
- **Componentes:** formularios con `react-hook-form` + `zod`; tablas con `@tanstack/react-table`.
- **UX:** toasts, loaders por request, confirm dialogs en `execute/cancel`.

### Design System — estilo "iOS 26.01 Liquid" minimalista

**Principios:** superficies fluidas, bordes suaves, profundidad sutil, tipografía limpia. - **Color base:** escala de grises fría + un acento (azul o verde) pálido. - Fondo app: `#0b0d10` (dark) o `#f7f8fa` (light). - Superficies (cards/nav): translucidez (usa `bg-white/60` dark:`bg-neutral-900/60`), `blur` (`backdrop-blur-md`). - Acento: `#4da3ff` (azul líquido) o `#22d3ee` (cian). - **Radiuses:** `rounded-2xl` para cards/botones; `rounded-full` en chips. - **Sombras:** suaves y difusas (`shadow-lg/20`, `shadow-[0_8px_30px_rgba(0,0,0,0.12)]`). - **Tipografía:** `text-[15px]` base, títulos

`text-2xl/3xl` con `font-semibold`. - **Motion**: transiciones `ease-out`, `duration-200/300` y micro-interacciones (hover/press) con `scale-95`. - **Componentes clave**: - **Navbar** translúcida con blur. - **Cards** con gradientes sutiles (`bg-gradient-to-b from-white/70 to-white/40`). - **Buttons** con relleno líquido (`bg-gradient-to-r` y `shadow-inner`). - **Charts** minimalistas (sin grid fuerte; líneas suaves).

**Tokens Tailwind sugeridos** (v4): utilidades directas sin config: `backdrop-blur-md`, `bg-white/60`, `dark:bg-neutral-900/60`, `rounded-2xl`, `shadow`, `transition`, `ease-out`, `duration-200`.

---

## 15) Próximos entregables

- Esqueletos de rutas Express por módulo (archivos `routes/*.ts`).
- DTOs `zod` por endpoint (inputs/outputs) y controladores de DB con queries.
- Colección de Postman / `.http` para pruebas rápidas.
- Boilerplate de `src/lib/api.ts` en frontend con ejemplos de uso.