Architecture Philosophy

The Clean Architecture Approach

Each service follows a layered architecture inspired by Clean Architecture and Hexagonal Architecture principles. Let's understand why this matters.

What are the layers?

Looking at services/users/src/app.ts (lines 1-222) and services/wallet/src/app.ts (lines 1-240), we can see:

1. **HTTP Layer** (Routes): Handles web requests
2. **Application Layer** (Use cases): Orchestrates business logic
3. **Domain Layer** (Business rules): Pure business logic like balance calculations
4. **Infrastructure Layer** (Plugins): Database, JWT, rate limiting

Why this layering?

Testability: Pure domain logic like computeBalanceMinor has zero dependencies. It's just math:

```
export function computeBalanceMinor(transactions: Transaction
[]): number {
    return transactions.reduce((sum, tx) => {
        const amount = Number.isFinite(tx.amountMinor) ? tx.a
mountMinor : 0;
        if (tx.type === "credit") {
            return sum + amount;
        }
        return sum - amount;
    }, 0);
}
```

This function can be tested without a database, without HTTP, without anything. That's the power of clean architecture.

Flexibility: Want to swap Postgres for MongoDB? You only change the infrastructure layer. The business logic remains untouched.

The Twelve-Factor App Principles

The project follows 12-factor app principles:

Configuration via Environment: See services/users/src/config.ts - all configuration comes from environment variables, never hardcoded.

Backing services as attached resources: Databases and RabbitMQ are treated as attached resources via URLs.

Stateless processes: No in-memory session state. JWT tokens carry authentication.

Getting Started

Prerequisites

From the root README.md (lines 6-9):

- Bun 1.1.38 (the JavaScript runtime)
- Docker + Docker Compose

One-Command Startup

The package.json defines a simple command:

```
bun run up
```

What does this do?

Looking at package.json (lines 11-12), it expands to:

```
docker compose up -d --build users-migrate wallet-migrate use
rs wallet users-worker wallet-worker rabbitmq users-db wallet
-db docs
```

This single command:

1. Starts PostgreSQL databases for users and wallet
2. Starts RabbitMQ message broker
3. Runs database migrations
4. Starts both HTTP services
5. Starts background worker processes
6. Starts a unified documentation server

Why this approach?

Developer experience! One command gets you from zero to a fully running system. This is documented in the Architecture Decision D009 about avoiding vendor lock-in.

The Docker Compose Configuration

The docker-compose.yml (lines 1-151) orchestrates everything. Let's break it down:

Database Services

```
users-db:
  image: postgres:16
  environment:
    POSTGRES_DB: users
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  ports:
    - "5433:5432"
```

What: Each service gets its own PostgreSQL database.

Why: The database-per-service pattern ensures service independence. If the wallet database goes down, users can still register.

How: Different host ports (5433 vs 5434) allow both databases to run simultaneously.

The Message Broker

```
rabbitmq:
  image: rabbitmq:3-management
  ports:
    - "5672:5672"    # AMQP protocol
    - "15672:15672"  # Management UI
```

What: RabbitMQ handles all async communication between services.

Why: From Decision D001 in docs/architecture/decisions.md, async provisioning decouples services and tolerates downtime.

How: Services publish/consume messages. The management UI (http://localhost:15672) lets you inspect queues.

The Users Service

What Does It Do?

The Users service handles three core responsibilities:

1. User registration
2. User authentication (login)
3. Profile retrieval

Let's explore each deeply.

Service Structure

The Users service lives in services/users/ with this structure:

```
services/users/
├── src/
│   ├── app.ts             # Fastify app configuration
│   ├── server.ts          # HTTP server entry point
│   ├── worker.ts          # Background publisher
│   ├── config.ts          # Environment configuration
│   ├── routes/
│   │   ├── auth.ts        # Registration & login
│   │   └── me.ts          # Profile endpoint
│   └── plugins/
│       ├── db.ts          # Postgres connection
│       ├── jwt.ts         # JWT authentication
│       └── rateLimit.ts # Rate limiting
└── migrations/            # SQL schema definitions
```

The Application Setup

Let's understand services/users/src/app.ts:

Request ID Tracing

```
genReqId: (req) => {
  const header = req.headers["x-request-id"];
  if (typeof header === "string" && header.trim() !== "") {
    return header;
  }
  return randomUUID();
}
```

What: Every request gets a unique ID.

Why: When debugging async flows across services, you can trace a single user action through logs. This is part of Decision D008 on observability.

How: If a client sends x-request-id, we use it. Otherwise, generate one. The worker passes this ID in events so the Wallet service can correlate logs.

Structured Logging

```
logger: {
  level: config.logLevel,
  base: { service: config.serviceName },
  redact: {
    paths: ["req.headers.authorization", "req.headers.cooki
e"],
    remove: true,
  },
}
```

What: JSON-structured logs with Pino.

Why: Machine-readable logs are essential for production. Tools like ELK or Datadog can ingest these directly.

How: The redact configuration ensures sensitive data (JWT tokens, passwords) never appears in logs. This is a security best practice.

User Registration Flow

The registration endpoint is in services/users/src/routes/auth.ts (lines 40-127). Let's walk through it step by step.

Input Validation

```
schema: {
  body: {
    type: "object",
    required: ["firstName", "lastName", "email", "password"],
    properties: {
      firstName: { type: "string", minLength: 1 },
      lastName: { type: "string", minLength: 1 },
      email: { type: "string", format: "email" },
      password: { type: "string", minLength: 8 },
    },
  },
}
```

What: Fastify validates the request body automatically.

Why: Input validation at the boundary prevents bad data from entering your system. The 8-character password minimum is a basic security requirement.

How: Fastify uses JSON Schema. If validation fails, it returns 400 automatically before your handler runs.

Password Hashing

```
const passwordHash = await bcrypt.hash(password, 10);
```

What: Bcrypt hashes the password with a cost factor of 10.

Why: **Never store plaintext passwords**. Bcrypt is designed to be slow (prevents brute force) and includes automatic salting.

How: The cost factor of 10 means 2^10 iterations. This takes ~100ms, making brute forcing extremely expensive.

The Database Transaction

This is where it gets interesting. Look at auth.ts (lines 73-96):

```
const client = await app.db.connect();
try {
  await client.query("BEGIN");

  const result = await client.query<UserRow>(
    `INSERT INTO users (id, first_name, last_name, email, password_hash)
     VALUES ($1, $2, $3, $4, $5)
     RETURNING id, first_name, last_name, email, password_hash, created_at`,
    [id, firstName, lastName, email, passwordHash],
  );

  const user = result.rows[0];

  await client.query(
    `INSERT INTO outbox (id, type, payload_json)
     VALUES ($1, $2, $3)`,
    [randomUUID(), "UserRegistered", { userId: user.id, requestId: req.id }],
  );

  await client.query("COMMIT");
  // ... success response
} catch (error) {
  await client.query("ROLLBACK");
  // ... error handling
}
```

What: A single database transaction that creates the user AND writes to the outbox table.

Why: This is the **Outbox Pattern** (Decision D002). Both operations must succeed or both must fail. You can never have a user without an outbox entry.

How:

1. BEGIN starts a transaction
2. Insert user (this might fail if email exists)
3. Insert outbox entry
4. COMMIT makes both permanent atomically
5. If anything fails, ROLLBACK undoes everything

This guarantees that every registered user will eventually get a wallet, even if RabbitMQ is down during registration.

Error Handling

```
catch (error) {
  await client.query("ROLLBACK");
  const err = error as { code?: string };
  if (err.code === "23505") {  // Unique violation
    reply.code(409).send({
      code: "EMAIL_EXISTS",
      message: "Email already registered",
    });
    return;
  }
  throw error;
}
```

What: Postgres error code 23505 means unique constraint violation.

Why: The users table at services/users/migrations/001_create_users.sql has email TEXT NOT NULL UNIQUE. We give the user a helpful error instead of "internal server error".

How: Catch the specific error code, return 409 Conflict. All other errors bubble up and Fastify returns 500.

The Outbox Publisher Worker

Now let's understand how events get published. The worker.ts (lines 1-120) runs as a separate process.

Why a Separate Worker?

From Decision D002:

Use an outbox table and a publisher worker; mark outbox rows as published only after broker confirms.

The Problem: If you publish directly to RabbitMQ after inserting the user:

- What if RabbitMQ is down? The user is created but no wallet is provisioned.
- What if your app crashes after user creation but before publishing?

The Solution: Write to an outbox table in the same transaction. A separate worker polls this table and publishes reliably.

The Publishing Logic

Let's trace through publishBatch in services/users/src/worker.ts (lines 18-64):

```
await client.query("BEGIN");

const result = await client.query<OutboxRow>(
  `SELECT id, type, payload_json
   FROM outbox
   WHERE status = 'pending'
   ORDER BY created_at
   LIMIT 20
   FOR UPDATE SKIP LOCKED`,
);
```

What: Select up to 20 pending outbox entries, locking them for this worker.

Why: FOR UPDATE SKIP LOCKED is crucial for **concurrent workers**. Multiple worker instances can run; each grabs different rows without blocking each other.

How: Postgres locks the selected rows. SKIP LOCKED means "skip rows that another worker already locked".

Publishing with Confirms

```
channel.publish(
  EXCHANGE,
  routingKey,
  Buffer.from(JSON.stringify(row.payload_json)),
  {
    persistent: true,
    contentType: "application/json",
    messageId: row.id,
    headers: buildHeaders(row.payload_json),
  },
);

await channel.waitForConfirms();

await client.query(
  "UPDATE outbox SET status = 'published', published_at = NOW
() WHERE id = $1",
  [row.id],
);
```

What: Publish the message and wait for RabbitMQ to confirm receipt before marking as published.

Why: Without confirmation, the message might be lost in transit. With confirmation, we know RabbitMQ persisted it.

How: waitForConfirms() blocks until RabbitMQ acknowledges. Only then do we update the database status.

The Polling Loop

```
setInterval(() => {
  publishBatch(pool, channel)
    .then((count) => {
      if (count > 0) {
        console.log(`[outbox] published ${count} message(s)
`);
      }
    })
    .catch((error) => {
      console.error("[outbox] publish failed", error);
    });
}, interval);
```

What: Every second (configurable), try to publish pending messages.

Why: This provides **at-least-once delivery**. If the worker crashes mid-batch, unpublished messages remain in the outbox and get retried.

How: The status column tracks state. Failed publishes leave the status as 'pending', so they're picked up in the next poll.

User Login

The login endpoint in auth.ts (lines 129-178) is simpler but has important security considerations:

```
const result = await app.db.query<UserRow>(
  "SELECT id, first_name, last_name, email, password_hash, created_at FROM users WHERE email = $1",
  [email],
);
const user = result.rows[0];
if (!user) {
  reply.code(401).send({
    code: "INVALID_CREDENTIALS",
    message: "Invalid credentials",
  });
  return;
}

const match = await bcrypt.compare(password, user.password_hash);
if (!match) {
  reply.code(401).send({
    code: "INVALID_CREDENTIALS",
    message: "Invalid credentials",
  });
  return;
}
```

What: Look up user by email, verify password hash.

Why: The generic "Invalid credentials" message for both "user not found" and "wrong password" prevents email enumeration attacks.

How: `bcrypt.compare` safely compares the provided password against the stored hash.

The Wallet Service

What Does It Do?

The Wallet service manages:

1. Wallet provisioning (via event consumption)
2. Transaction creation (credits and debits)
3. Balance calculation
4. Transaction history

The Critical Design Decision: Eventual Consistency

From Decision D003:

Return `503 Service Unavailable` with `Retry-After: 2` until the wallet exists.

The Challenge: Wallets are provisioned asynchronously. A user can register and immediately try to use their wallet before it's ready.

The Solution: The walletReady plugin at services/wallet/src/plugins/walletReady.ts (lines 1-30) checks every request:

```
app.decorate("walletReady", async (req, reply) => {
  const userId = req.user?.sub;
  if (!userId) {
    return reply
      .code(401)
      .send({ code: "UNAUTHORIZED", message: "Unauthorized"
});
  }

  const result = await app.db.query(
    "SELECT 1 FROM wallets WHERE user_id = $1",
    [userId],
  );
  if (result.rowCount === 0) {
    reply.header("Retry-After", "2");
    return reply.code(503).send({
      code: "WALLET_PROVISIONING",
      message: "Wallet is being provisioned. Retry shortly.",
    });
  }
});
```

What: Every wallet endpoint calls this middleware.

Why: HTTP 503 with `Retry-After` is semantically correct for temporary unavailability. Clients know to retry.

How: A simple database check. If no wallet row exists, return 503. The `Retry-After: 2` header tells clients to wait 2 seconds.

This is used in transactions.ts as a preHandler:

```
preHandler: [app.authenticate, app.walletReady]
```

The Wallet Provisioning Consumer

The worker.ts (lines 1-145) consumes `UserRegistered` events:

Queue Topology Setup

```
await channel.assertQueue(QUEUE, {
  durable: true,
  deadLetterExchange: DLX,
  deadLetterRoutingKey: DLQ,
});
await channel.bindQueue(QUEUE, EXCHANGE, ROUTING_KEY);
```

What: Create a durable queue bound to `user.registered` routing key.

Why: `durable: true` means messages survive broker restarts. The `deadLetterExchange` ensures failed messages go to a dead-letter queue for inspection.

How: RabbitMQ's topic exchange routes messages based on routing keys. Our queue listens for `user.registered`.

The Retry Ladder

This is one of the most sophisticated parts. From Decision D005:

TTL retry queues (10s → 30s → 120s) + DLQ

Look at ensureTopology in services/wallet/src/worker.ts (lines 22-52):

```
const retryQueues = [
  { name: "wallet.provision.retry.10s", ttl: retryTtls[0] },
  { name: "wallet.provision.retry.30s", ttl: retryTtls[1] },
  { name: "wallet.provision.retry.120s", ttl: retryTtls[2] },
];

for (const retry of retryQueues) {
  await channel.assertQueue(retry.name, {
    durable: true,
    messageTtl: retry.ttl,
    deadLetterExchange: EXCHANGE,
    deadLetterRoutingKey: ROUTING_KEY,
  });
}
```

What: Three queues with increasing TTLs. Messages "expire" from these queues back to the main queue.

Why: Transient failures (database temporarily down) should be retried with backoff. This prevents retry storms.

How:

1. Main queue fails → route to 10s retry queue
2. After 10s, message automatically dead-letters back to main queue (retry #1)
3. Fails again → route to 30s retry queue
4. After 30s, retry #2
5. Fails again → route to 120s retry queue
6. After 120s, retry #3
7. Fails again → DLQ (give up)

The retry logic in services/wallet/src/consumer/retry.ts determines which queue:

```
export function nextRetryQueue(attempts: number): string | nu
ll {
    if (attempts <= 0) return "wallet.provision.retry.10s";
    if (attempts === 1) return "wallet.provision.retry.30s";
    if (attempts === 2) return "wallet.provision.retry.120s";
    return null;
}
```

The Consumption Handler

Let's trace through the consume callback in services/wallet/src/worker.ts (lines 84-135):

```
const payload = parsePayload(message);
if (!payload) {
  console.warn("[consumer] invalid payload, sending to DLQ");
  channel.nack(message, false, false);
  return;
}
```

What: Parse and validate the message payload.

Why: Defense in depth. Bad payloads (malformed JSON, wrong schema) should never crash the worker.

How: `nack(message, false, false)` means "negative acknowledgement, don't requeue, don't requeue-multiple". This sends directly to DLQ.

Idempotent Wallet Creation

```
await pool.query(
  `INSERT INTO wallets (id, user_id)
   VALUES ($1, $2)
   ON CONFLICT (user_id) DO NOTHING`,
  [randomUUID(), payload.userId],
);
```

What: Insert wallet, but ignore if it already exists.

Why: From Decision D004, events are **at-least-once**. You might receive `UserRegistered` multiple times for the same user.

How: The `wallets.user_id` column has a `UNIQUE` constraint. `ON CONFLICT DO NOTHING` makes this operation idempotent.

Retry on Transient Failure

```
catch (error) {
  const attempts = readDeathCount(message.properties.headers
?? {});
  const retryQueue = nextRetryQueue(attempts);
  if (retryQueue) {
    channel.sendToQueue(retryQueue, message.content, {
      contentType: "application/json",
      persistent: true,
      headers: message.properties.headers,
    });
    console.warn(`[consumer] retry ${attempts + 1} for user
${payload.userId} -> ${retryQueue}`);
    channel.ack(message);
  } else {
    console.error(`[consumer] giving up for user ${payload.us
erId}; to DLQ`);
    channel.nack(message, false, false);
  }
}
```

What: On database errors, route to appropriate retry queue based on attempt count.

Why: The database might be temporarily unavailable. Retrying with backoff gives it time to recover.

How: `readDeathCount` examines the `x-death` header that RabbitMQ adds on each failure. After 3 retries, `nextRetryQueue` returns null, and we give up.

Transaction Creation

The transactions route in services/wallet/src/routes/transactions.ts handles credits and debits. This has several layers of sophistication.

Idempotency Keys

From the OpenAPI schema:

```
headers: {
  type: "object",
  required: ["idempotency-key"],
  properties: {
    "idempotency-key": { type: "string", minLength: 1, maxLen
gth: 128 },
  },
}
```

What: Clients must provide an `Idempotency-Key` header.

Why: Network requests can be retried. Without idempotency, a retry could charge a user twice.

How: The key is stored in the transactions table. If the same key is used again, we return the existing transaction.

The idempotency check:

```
const existing = await client.query<{ id: string }>(
  `SELECT id
   FROM transactions
   WHERE user_id = $1 AND idempotency_key = $2 AND type = $3
`,
  [userId, idempotencyKey, type],
);
if (existing.rowCount && existing.rows[0]) {
  await client.query("COMMIT");
  reply.code(200).send({ id: existing.rows[0].id, status: "re
corded" });
  return;
}
```

What: Before creating a transaction, check if one exists with this idempotency key.

Why: Safe retries. The client gets the same transaction ID whether this is the first request or a retry.

How: Query by user_id + idempotency_key + type. If found, return it. Note the 200 status (not 201) indicates this is a replay.

Debit Validation

Debits require a balance check:

```
if (type === "debit") {
  const sums = await client.query<TxSumRow>(
    `SELECT type, COALESCE(SUM(amount_minor), 0) AS amount_mi
nor
     FROM transactions
     WHERE user_id = $1
     GROUP BY type`,
    [userId],
  );
  const balance = computeBalanceMinor(
    sums.rows.map((row) => ({
      type: row.type,
      amountMinor: Number(row.amount_minor),
    })),
  );

  if (!canDebit(balance, amountMinor)) {
    await client.query("ROLLBACK");
    reply.code(409).send({
      code: "INSUFFICIENT_FUNDS",
      message: "Insufficient balance",
    });
    return;
  }
}
```

What: Compute current balance and ensure it doesn't go negative.

Why: Business rule: users can't spend money they don't have.

How:

1. Query sum of credits and debits
2. Call computeBalanceMinor (pure function)
3. Call canDebit to check if allowed
4. If insufficient funds, rollback and return 409

This happens **inside the transaction** that will create the debit, ensuring consistency.

Balance Calculation Model

From Decision D006:

Compute balance from transactions; amounts stored in amountMinor integers.

Let's understand the domain model in services/wallet/src/domain/balance.ts:

```
export type Transaction = {
    type: "credit" | "debit";
    amountMinor: number;
};

export function computeBalanceMinor(transactions: Transaction[]): number {
    return transactions.reduce((sum, tx) => {
        const amount = Number.isFinite(tx.amountMinor) ? tx.amountMinor : 0;
        if (tx.type === "credit") {
            return sum + amount;
        }
        return sum - amount;
    }, 0);
}
```

What: Balance is the sum of all transactions: credits add, debits subtract.

Why: This is a **ledger model**. It's simple, correct, and auditable. Every transaction is immutable.

How: Store amounts as integers (e.g., 100 cents instead of $1.00) to avoid floating-point errors. This is standard in financial systems.

The canDebit function enforces the non-negative invariant:

```
export function canDebit(balanceMinor: number, amountMinor: number): boolean {
    if (!Number.isFinite(balanceMinor) || !Number.isFinite(amountMinor)) {
        return false;
    }
    if (amountMinor < 0) {
        return false;
    }
    return balanceMinor - amountMinor >= 0;
}
```

What: Validation logic for debits.

Why: Guard against invalid inputs (NaN, negative amounts) and ensure balance stays non-negative.

How: Pure function with no dependencies. Easily testable.

Transaction Pagination

The transactions list endpoint uses cursor-based pagination:

```
const cursor =
  typeof req.query.cursor === "string" ? req.query.cursor : u
ndefined;

const params: Array<string | number> = [userId, limit];
const cursorClause = cursor ? "AND created_at < $3" : "";
if (cursor) {
  params.push(cursor);
}

const result = await app.db.query<TxRow>(
  `SELECT id, type, amount_minor, description, created_at
   FROM transactions
   WHERE user_id = $1 ${cursorClause}
   ORDER BY created_at DESC
   LIMIT $2`,
  params,
);

const nextCursor = result.rows.at(-1)?.created_at.toISOString
();
return { items, nextCursor };
```

What: Cursor-based pagination using `created_at` timestamp.

Why: Offset-based pagination (`LIMIT X OFFSET Y`) performs poorly on large datasets. Cursor-based pagination is O(log n) with proper indexes.

How:

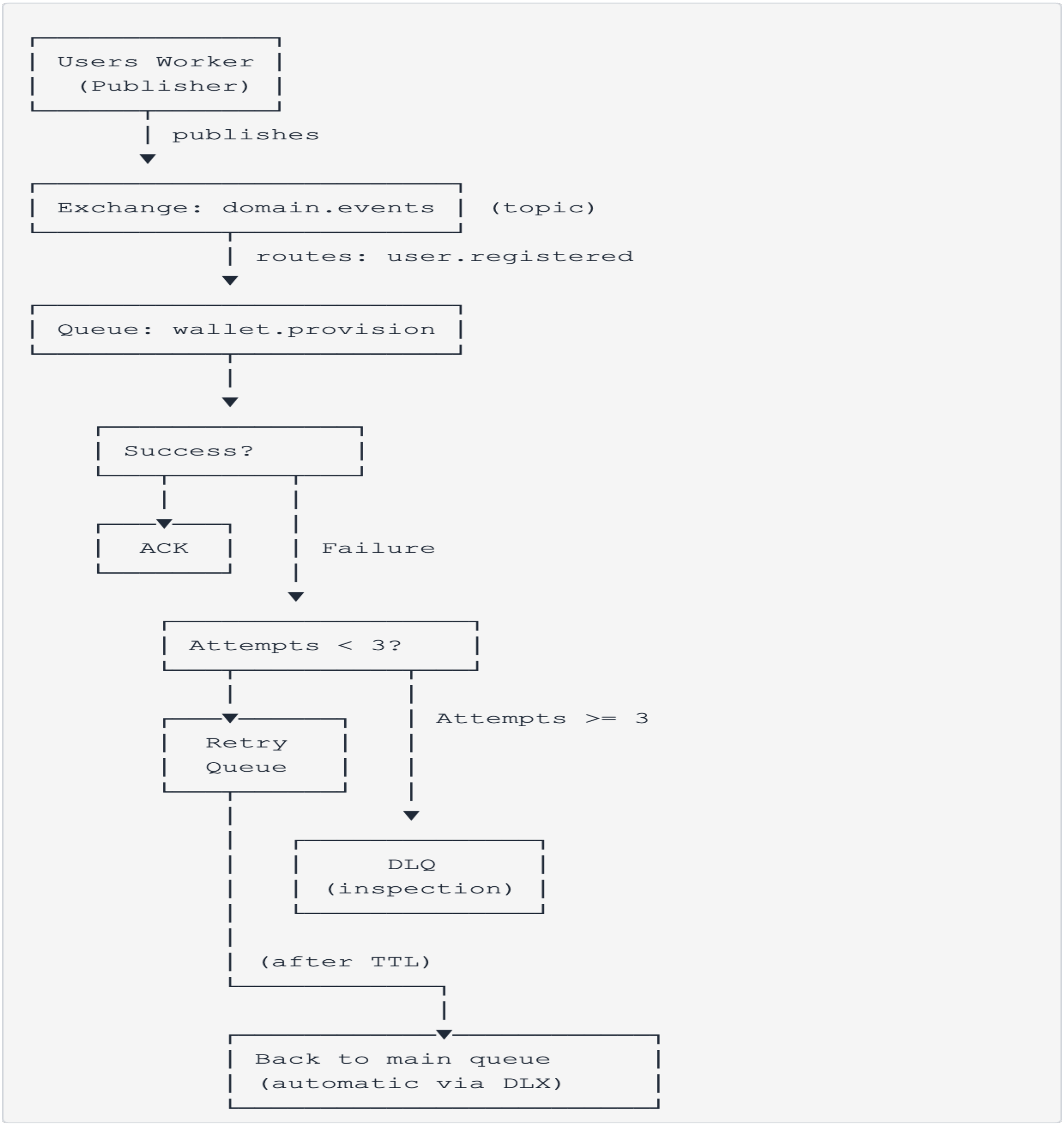
1. First request: no cursor, get latest transactions
2. Response includes `nextCursor` (the last transaction's timestamp)
3. Next request: include cursor, query `WHERE created_at < cursor`
4. Repeat until no more results

Limitation: From IMPROVEMENTS.md, this can skip/duplicate transactions if multiple have the same timestamp. The improvement suggestion is to use `(created_at, id)` composite cursor.

The Messaging Infrastructure

RabbitMQ Topology

Let's visualize the complete message flow:



Understanding Dead Letter Exchanges

The topology setup configures DLX:

```
await channel.assertQueue(QUEUE, {
  durable: true,
  deadLetterExchange: DLX,
  deadLetterRoutingKey: DLQ,
});
```

What: The main queue has a dead letter exchange (DLX) configured.

Why: When a message is negatively acknowledged (`nack`) without requeue, it goes to the DLX instead of being lost.

How: RabbitMQ automatically routes the message to the `domain.events.dlx` exchange, which routes to the `wallet.provision.dlq` queue.

Retry Queue Mechanics

The retry queues use a clever trick:

```
await channel.assertQueue(retry.name, {
  durable: true,
  messageTtl: retry.ttl,
  deadLetterExchange: EXCHANGE,
  deadLetterRoutingKey: ROUTING_KEY,
});
```

What: These queues have NO consumer. They exist solely for delayed retry.

Why: RabbitMQ doesn't have built-in delayed/scheduled message delivery. This pattern simulates it.

How:

1. Message sent to wallet.provision.retry.10s
2. No consumer, so it sits there
3. After messageTtl expires (10 seconds)
4. RabbitMQ dead-letters it to EXCHANGE with routing key user.registered
5. This routes back to the main wallet.provision queue
6. Consumer tries again

This is a standard RabbitMQ pattern for retry with backoff.

Database Design

The Users Database

The users database has three migrations:

Migration 001: Users Table

From services/users/migrations/001_create_users.sql:

```
CREATE TABLE IF NOT EXISTS users (
  id UUID PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  email TEXT NOT NULL UNIQUE,
  password_hash TEXT NOT NULL,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

What: Core user table.

Why:

- UUID as primary key: globally unique, no coordination needed between services

- email UNIQUE: business invariant enforced at database level
- password_hash: never store plaintext passwords
- TIMESTAMPTZ: timezone-aware timestamps

How: CREATE TABLE IF NOT EXISTS makes migrations idempotent.

Migration 002: Outbox Table

From services/users/migrations/002_create_outbox.sql:

```
CREATE TABLE IF NOT EXISTS outbox (
  id UUID PRIMARY KEY,
  type TEXT NOT NULL,
  payload_json JSONB NOT NULL,
  status TEXT NOT NULL DEFAULT 'pending',
  attempts INT NOT NULL DEFAULT 0,
  last_error TEXT,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  published_at TIMESTAMPTZ
);

CREATE INDEX IF NOT EXISTS outbox_status_idx ON outbox(status);
```

What: The outbox pattern implementation.

Why:

- status column: tracks lifecycle (pending → published/failed)
- attempts and last_error: debugging failed publishes
- payload_json JSONB: PostgreSQL's JSON type, allows flexible event schemas

How: The index on status makes WHERE status = 'pending' queries fast.

Migration 003: Name Split

Migration 003_split_name.sql splits a single `name` column into `first_name` and `last_name`.

What: Schema evolution example.

Why: Demonstrates how to handle schema changes in a running system.

How:

1. Add new columns
2. Migrate data
3. Drop old column

This is **irreversible** migration, which is fine for a demo but production might need reversible migrations.

The Wallet Database

Migration 001: Core Tables

From services/wallet/migrations/001_create_wallets.sql:

```
CREATE TABLE IF NOT EXISTS wallets (
  id UUID PRIMARY KEY,
  user_id UUID NOT NULL UNIQUE,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE TABLE IF NOT EXISTS transactions (
  id UUID PRIMARY KEY,
  user_id UUID NOT NULL,
  type TEXT NOT NULL CHECK (type IN ('credit', 'debit')),
  amount_minor INTEGER NOT NULL CHECK (amount_minor >= 0),
  description TEXT,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE INDEX transactions_user_id_idx ON transactions(user_i
d);
```

What: Wallets and transactions tables.

Why:

- `wallets.user_id UNIQUE`: one wallet per user
- `CHECK (type IN ('credit', 'debit'))`: database-level enum
- `CHECK (amount_minor >= 0)`: can't create negative-amount transactions
- `amount_minor INTEGER`: store cents as integers

How: The index on `user_id` makes balance calculations fast.

Migration 002: Idempotency

From services/wallet/migrations/002_add_idempotency_key.sql:

```
ALTER TABLE transactions ADD COLUMN idempotency_key TEXT;
CREATE UNIQUE INDEX transactions_idempotency_key_idx
  ON transactions(user_id, idempotency_key, type);
```

What: Add idempotency key support.

Why: Enables safe retries of transaction creation.

How: The unique index on `(user_id, idempotency_key, type)` enforces uniqueness at the database level.

Security & Authentication

JWT Authentication Flow

Both services use JWT (JSON Web Tokens) for authentication. Let's understand the JWT plugin:

```
app.register(jwt, {
  secret: config.jwtPrivateKey,
  verify: {
    extractToken: (request) => {
      const auth = request.headers.authorization;
      if (!auth) return null;
      const match = /^Bearer (.+)$/.exec(auth);
      return match ? match[1] : null;
    },
  },
});
```

What: Configure JWT verification.

Why: Stateless authentication. The token contains all necessary information; no server-side session needed.

How: Extract token from Authorization: Bearer <token> header.

The authenticate decorator validates tokens on protected routes:

```
app.decorate("authenticate", async (req, reply) => {
  try {
    await req.jwtVerify();
  } catch (error) {
    reply
      .code(401)
      .send({ code: "UNAUTHORIZED", message: "Unauthorized"
});
  }
});
```

What: Fastify decorator that validates JWT.

Why: Reusable authentication logic used by all protected routes.

How: jwtVerify() checks signature, expiration, and extracts the payload into req.user.

Token Creation

When users log in, a token is created:

```
const accessToken = app.jwt.sign(
  { sub: user.id },
  { expiresIn: accessTokenTtl },
);
```

What: Create a JWT with the user's ID as the sub (subject) claim.

Why: The sub claim is a standard JWT claim. It identifies who the token is for.

How:

- Signed with `JWT_PRIVATE_KEY` (HMAC-SHA256)
- Expires in 1 hour
- Contains only the user ID (no sensitive data)

Authorization via Subject Matching

From Decision D007:

Self-only access: `sub` is the userId for all wallet actions.

This is implemented in wallet routes:

```
preHandler: [app.authenticate, app.walletReady]
```

Then in the handler:

```
const userId = req.user.sub;
```

What: Use the authenticated user's ID from the JWT.

Why: Users can only access their own wallet. No horizontal privilege escalation.

How: The JWT's `sub` claim is trusted (because we verified the signature). We use it for all database queries.

Rate Limiting

The rateLimit plugin protects against abuse:

```
export const rateLimitConfig = {
    register: {
        max: 5,
        timeWindow: "1m",
    },
    login: {
        max: 10,
        timeWindow: "1m",
    },
    transactions: {
        max: 30,
        timeWindow: "1m",
    },
};
```

What: Different limits for different endpoints.

Why: From Decision D010, rate limiting prevents abuse and brute force attacks.

How: Applied per-route via Fastify's config:

```
config: { rateLimit: rateLimitConfig.register }
```

When exceeded, returns 429 Too Many Requests.

Limitation: From IMPROVEMENTS.md, in-memory limits don't share across instances. Production should use Redis.

Observability & Operations

Health Endpoints

Both services implement health checks:

```
app.get("/health/live", () => ({ status: "ok" }));

app.get("/health/ready", async () => {
  const checks = await Promise.all([
    tcpCheck("database", config.databaseUrl),
    tcpCheck("rabbitmq", config.rabbitUrl),
  ]);
  const ok = checks.every((c) => c.ok);
  return { status: ok ? "ok" : "unavailable", checks };
});
```

What: Liveness and readiness probes.

Why: Kubernetes-compatible health checks.

How:

- `/health/live`: Always returns 200 if the process is running
- `/health/ready`: Checks dependencies (database, RabbitMQ) via TCP connections

The tcpCheck function attempts to connect:

```
const socket = net.createConnection({ host: url.hostname, por
t });
const timeout = setTimeout(() => {
  socket.destroy();
  resolve({ name, ok: false, details: "timeout" });
}, 300);
```

What: 300ms timeout TCP connection attempt.

Why: Quick check without full protocol handshake.

How: If connection succeeds, the dependency is reachable.

Structured Logging

From Decision D008:

Structured JSON logs (pino), `x-request-id`, `/health/live`, `/health/ready`.

The logger configuration:

```
logger: {
  level: config.logLevel,
  base: { service: config.serviceName },
  redact: {
    paths: ["req.headers.authorization", "req.headers.cooki
e"],
    remove: true,
  },
}
```

What: Pino logger with sensitive data redaction.

Why: JSON logs are machine-readable. Redaction prevents leaking credentials.

How: Pino intercepts Fastify's internal logging and structures it.

Example log entry:

```
{
  "level": 30,
  "service": "users",
  "req": {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "method": "POST",
    "url": "/v1/auth/register"
  },
  "msg": "request completed"
}
```

OpenAPI Documentation

Both services expose Swagger UI:

```
app.register(swagger, {
  openapi: {
    info: {
      title: "Users Service",
      version: "1.0.0",
    },
    servers: [{ url: `http://localhost:${config.port}` }],
    tags: [
      { name: "auth", description: "Authentication" },
      { name: "users", description: "User profile" },
    ],
  },
});

app.register(swaggerUi, {
  routePrefix: "/docs",
  staticCSP: true,
});
```

What: Interactive API documentation.

Why: Self-documenting APIs. Developers can test endpoints directly from the browser.

How:

- Access at `http://localhost:3002/docs`
- OpenAPI spec at `http://localhost:3002/openapi.json`
- The consolidated docs service merges both service specs

Architecture Decisions

Let's explore the key architectural decisions documented in docs/architecture/decisions.md.

D001: Async Provisioning via RabbitMQ

The Decision: Use RabbitMQ event-driven provisioning.

Why Async?

1. **Decoupling**: Users service doesn't need to know about Wallet service
2. **Resilience**: If Wallet is down during registration, the user can still register
3. **Scalability**: Event-driven systems scale horizontally easily

Why RabbitMQ?

From Decision D001:

Decouples services, tolerates wallet downtime, aligns with queue requirement, and demonstrates resilience patterns.

Alternatives considered:

- **Synchronous REST**: Simple but creates tight coupling
- **Kafka**: Overkill for this scale; more complex ops

Trade-offs:

- Services can scale independently
- Wallet outages don't block registration
- lEventual consistency complexity
- lAdditional infrastructure (RabbitMQ)

D002: Outbox Pattern

The Problem: What if the event publish fails after user creation?

```
1. Start transaction
2. Insert user ✅
3. Commit ✅
4. Publish to RabbitMQ ❌ (RabbitMQ is down!)
5. User exists but no wallet will ever be created!
```

The Solution: From Decision D002:

Use an outbox table and a publisher worker.

```
1. Start transaction
2. Insert user ✅
3. Insert outbox entry ✅
4. Commit ✅ (atomic!)
5. Worker polls outbox
6. Worker publishes to RabbitMQ
7. Worker marks as published
```

Trade-offs:

- Guaranteed event delivery

- No events lost
- LAdditional complexity (outbox table + worker)
- LSmall delay in event delivery

D003: Wallet Readiness Gating

The Challenge: Async provisioning means wallets aren't instantly available.

Options Considered:

1. **Lazy Creation**: Create wallet on first wallet API call
   - LSide effects on GET requests are confusing
   - LRace conditions if multiple requests arrive simultaneously
2. **409 Conflict**: Return "wallet doesn't exist"
   - LNot semantically correct (it will exist soon)
   - LClient doesn't know when to retry
3. **503 Service Unavailable** with `Retry-After`: **Chosen**
   - Semantically correct for temporary unavailability
   - Standard HTTP retry semantics
   - Client knows when to retry (2 seconds)

From Decision D003:

Return `503 Service Unavailable` with `Retry-After: 2` until the wallet exists.

Implementation: The walletReady plugin applied to ALL wallet endpoints.

D005: Retry Ladder + DLQ

The Problem: Consumer failures come in two flavors:

1. **Transient**: Database temporarily down, network glitch
   - Should retry with backoff
2. **Permanent**: Invalid payload, programming bug

- Should give up and alert humans

The Solution: From Decision D005:

TTL retry queues (10s → 30s → 120s) + DLQ.

Why Progressive Backoff?

Imagine the database is down. Without backoff:

- Consumer tries message → fails → retries immediately → fails → retries immediately
- This is a "retry storm" that wastes CPU and network

With backoff:

- Try 1: Immediate (might be transient)
- Try 2: After 10s (give it a moment)
- Try 3: After 30s (maybe it's restarting)
- Try 4: After 120s (seriously wait)
- Give up: After 4 attempts, something is fundamentally broken

Why DLQ?

Dead Letter Queue (DLQ) is where messages go to die. It's for human inspection:

- Invalid payloads
- Persistent bugs
- Unexpected error conditions

An operator can inspect the DLQ, fix the issue, and manually reprocess if needed.

D006: Balance as Ledger

The Decision: Compute balance from transactions rather than storing a balance column.

From Decision D006:

Compute balance from transactions; amounts stored in amountMinor integers.

Why?

1. **Correctness**: The transaction log is the source of truth
2. **Auditability**: Every cent is traceable
3. **Simplicity**: No balance update logic that could have bugs

Trade-offs:

- Simple mental model
- Easy to audit
- LO(n) to compute balance
- LNo database-level concurrency control

From IMPROVEMENTS.md:

At scale, add a cached balance column + DB transaction locking for concurrent debits.

Production Evolution: The improvement suggests:

```
ALTER TABLE wallets ADD COLUMN balance_minor INTEGER DEFAULT
0;
CREATE UNIQUE INDEX ON transactions(user_id, idempotency_ke
y);

BEGIN;
  UPDATE wallets SET balance_minor = balance_minor - $amount
WHERE user_id = $1 AND balance_minor >= $amount;
  INSERT INTO transactions ...;
COMMIT;
```

This adds a balance cache but keeps the transaction log as source of truth.

D007: Self-Only Authorization

The Decision: No admin role; users can only access their own data.

From Decision D007:

Self-only access: `sub` is the userId for all wallet actions.

Why?

Security Principle: Least privilege. Most users don't need admin access.

Implementation: Extract `userId` from JWT's `sub` claim and use it for all queries.

```
const userId = req.user.sub;
const result = await app.db.query(
  "SELECT * FROM transactions WHERE user_id = $1",
  [userId]
);
```

What This Prevents: Horizontal privilege escalation. User A can't access User B's wallet by changing request parameters.

What This Doesn't Support: Admin operations (support inspecting wallets, etc.)

From Decision D011:

Proposed: Add admin role (JWT claim) + admin-only endpoints.

D008: Minimal Observability

The Decision: From Decision D008:

Structured JSON logs (pino), `x-request-id`, `/health/live`, `/health/ready`.

Why Bother?

Async systems are hard to debug. When a user reports "my wallet wasn't created", how do you trace it?

Request ID Tracing:

1. User calls `POST /v1/auth/register` → generates request ID `abc-123`
2. Users service logs: `[abc-123] User created: user-456`
3. Outbox entry includes: `{ requestId: "abc-123" }`
4. Worker publishes event with header: `x-request-id: abc-123`
5. Wallet consumer logs: `[abc-123] provisioned wallet for user-456`

Now you can search logs for `abc-123` and see the entire flow across services!

Health Checks:

Kubernetes (or any orchestrator) needs to know:

- **Liveness**: Is the process alive? (restart if not)
- **Readiness**: Is it ready to serve traffic? (don't send traffic if not)

The `/health/ready` endpoint checks dependencies so a pod won't receive traffic if its database is down.

D009: Changelog Automation

The Decision: Use `standard-version` for changelog generation.

From Decision D009:

`standard-version` is CI-provider-agnostic.

Why?

Conventional Commits + automated changelog:

- `feat: add admin role` → minor version bump + feature entry in changelog
- `fix: prevent negative balance` → patch version bump + bugfix entry
- `BREAKING CHANGE:` → major version bump

Run `bun run release`:

```
$ standard-version
✔ bumping version in package.json from 1.0.0 to 1.1.0
✔ outputting changes to CHANGELOG.md
✔ committing package.json and CHANGELOG.md
✔ tagging release v1.1.0
ℹ Run `git push --follow-tags origin main` to publish
```

Why Not release-please?

`release-please` is GitHub-specific. `standard-version` works with any Git provider (GitLab, Bitbucket, local).

D010: Rate Limiting

The Decision: Use `@fastify/rate-limit`.

From Decision D010:

Use `@fastify/rate-limit` with per-route limits and an env toggle.

Why?

Public endpoints (register, login) are attack vectors:

- Brute force password guessing
- Account enumeration
- DDoS

Implementation: Different limits per endpoint:

- Register: 5 requests/minute (creating accounts is expensive)
- Login: 10 requests/minute (legitimate users might mistype)
- Transactions: 30 requests/minute (more frequent in normal use)

Production Note: From IMPROVEMENTS.md:

In-memory limits won't be shared across instances; document Redis/gateway approach for production.

For multi-instance deployments, use Redis as shared rate limit store.

Future Improvements

The IMPROVEMENTS.md file documents 10 improvement areas. Let's highlight a few.

Improvement #4: Idempotency Key Payload Validation

The Problem: From IMPROVEMENTS.md:

Current idempotency behavior returns an existing transaction when the same `Idempotency-Key` is provided, but it does not validate that the request payload matches.

Scenario:

```
Request 1: POST /transactions/credit
  Idempotency-Key: abc-123
  { amountMinor: 1000 }
  → Creates transaction T1 for $10.00

Request 2: POST /transactions/credit  (retry with bug!)
  Idempotency-Key: abc-123
  { amountMinor: 500 }
  → Returns transaction T1 (for $10.00!)
```

The client thinks they credited $5.00 but actually credited $10.00.

The Solution: Hash the request payload and store it with the idempotency key:

```
const payloadHash = sha256(JSON.stringify(canonicalize(bod
y)));

const existing = await client.query(
  "SELECT id, payload_hash FROM transactions WHERE idempotenc
y_key = $1",
  [idempotencyKey]
);

if (existing.rows[0]) {
  if (existing.rows[0].payload_hash !== payloadHash) {
    return reply.code(409).send({
      code: "IDEMPOTENCY_KEY_MISMATCH",
      message: "Idempotency key used with different payload"
    });
  }
  return reply.send({ id: existing.rows[0].id });
}
```

Improvement #5: Migrations Outside Docker

The Problem: From IMPROVEMENTS.md:

services/users/src/migrate.sh references /app/..., so bun --filter ... run migrate won't work when running locally.

Current Limitation: The migration scripts hardcode Docker paths:

```
#!/bin/bash
psql $DATABASE_URL -f /app/migrations/001_create_users.sql
```

This works in Docker (where code is mounted at /app) but fails locally.

The Solution: Use paths relative to the script:

```
#!/bin/bash
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
MIGRATIONS_DIR="$SCRIPT_DIR/../migrations"
psql $DATABASE_URL -f "$MIGRATIONS_DIR/001_create_users.sql"
```

Now it works both in Docker and locally.

Improvement #7: Database Indexes for Hot Paths

The Problem: From IMPROVEMENTS.md:

Listing uses WHERE user_id = $1 ORDER BY created_at DESC LIMIT .... Current index is transactions(user_id) only.

Performance Impact: Without a composite index, Postgres:

1. Uses index to find all transactions for user
2. Sorts them by created_at (sort operation)

3. Returns top N

With composite index on `(user_id, created_at DESC)`:

1. Index is already sorted
2. Return top N directly (no sort!)

The Solution:

```
CREATE INDEX transactions_user_id_created_at_idx
  ON transactions(user_id, created_at DESC);
```

For large wallets (thousands of transactions), this is the difference between 5ms and 500ms.

Improvement #10: Transaction Pagination Correctness

The Problem: From IMPROVEMENTS.md:

`/v1/transactions` paginates by `created_at` only. If multiple rows share the same timestamp, pagination can skip/duplicate items.

Scenario: Three transactions created in the same millisecond:

```
id: tx-001, created_at: 2024-01-01 12:00:00.123
id: tx-002, created_at: 2024-01-01 12:00:00.123
id: tx-003, created_at: 2024-01-01 12:00:00.123
```

Page 1 (limit=2):

```
SELECT * FROM transactions
WHERE user_id = $1
ORDER BY created_at DESC
LIMIT 2
```

Returns `tx-003, tx-002`. Cursor: `2024-01-01 12:00:00.123`

Page 2:

```
SELECT * FROM transactions
WHERE user_id = $1
  AND created_at < '2024-01-01 12:00:00.123'
ORDER BY created_at DESC
LIMIT 2
```

Returns nothing! We've skipped `tx-001` because it has the same timestamp.

The Solution: Composite cursor `(created_at, id)`:

```
const cursor = req.query.cursor
  ? JSON.parse(Buffer.from(req.query.cursor, 'base64').toStri
ng())
  : null;

const params = [userId, limit];
let whereClause = "";
if (cursor) {
  whereClause = "AND (created_at, id) < ($3, $4)";
  params.push(cursor.created_at, cursor.id);
}

const result = await db.query(`
  SELECT * FROM transactions
  WHERE user_id = $1 ${whereClause}
  ORDER BY created_at DESC, id DESC
  LIMIT $2
`, params);

const nextCursor = result.rows.at(-1)
  ? Buffer.from(JSON.stringify({
      created_at: result.rows.at(-1).created_at,
      id: result.rows.at(-1).id
    })).toString('base64')
  : null;
```

Now pagination is correct even with duplicate timestamps.

Conclusion

This project demonstrates production-ready microservices architecture with:

Event-Driven Communication: Services communicate via RabbitMQ, enabling loose coupling

Reliability Patterns: Outbox pattern, retry ladders, dead-letter queues ensure robustness

Clean Architecture: Layered design with pure domain logic, testable and maintainable

Security: JWT authentication, authorization, password hashing, rate limiting

Observability: Structured logging, request tracing, health checks

Developer Experience: One-command startup, comprehensive documentation, automated changelog

The Architecture Decisions document (docs/architecture/decisions.md) captures the "why" behind each choice, and IMPROVEMENTS.md provides a roadmap for production-hardening.

Key Takeaways

1. **Async is powerful but complex**: Event-driven architecture enables scalability and resilience, but you pay for it with eventual consistency and debugging complexity.
2. **Patterns matter**: The Outbox pattern, retry ladders, and idempotency keys aren't academic exercises—they're battle-tested solutions to real distributed systems problems.
3. **Layer your architecture**: Separating HTTP → Application → Domain → Infrastructure makes testing easier and changes safer.
4. **Observability is not optional**: In distributed systems, you can't debug without structured logs and request tracing.
5. **Trade-offs are explicit**: Every design decision involves trade-offs. Documenting them in ADRs helps future you (and your team) understand why things are the way they are.

Further Reading

- **Outbox Pattern**: https://microservices.io/patterns/data/transactional-outbox.html
- **RabbitMQ Reliability**: https://www.rabbitmq.com/reliability.html
- **JWT Best Practices**: https://tools.ietf.org/html/rfc8725
- **Clean Architecture**: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- **Twelve-Factor App**: https://12factor.net/

Congratulations! You now understand every architectural decision, pattern, and line of code in this project. Use this knowledge to build your own production-ready microservices!⋯