Transaction Processing Simulator — Cheat Sheet

What Is This Project?

A backend service that simulates real-world transaction processing workflows. Built with TypeScript, Node.js, Express, and MongoDB.

Your elevator pitch:

> "I built a transaction processing simulator based on my 7 years of experience in payments operations at Navy Federal. I'm transitioning into application engineering, and I wanted to demonstrate that I understand not just the operations side, but the technical systems behind it. It includes a rules engine for fraud detection and compliance, JWT authentication, webhook notifications, and batch processing. I built it with TypeScript and Node.js."

How a Request Flows Through the System

```
User makes request (e.g., POST /api/transactions)
        ↓
index.ts — middleware runs (logging, rate limiting)
        ↓
routes/index.ts — directs to transactionRoutes.ts
        ↓
transactionRoutes.ts — matches endpoint, calls controller
        ↓
controller — validates request, calls service
        ↓
TransactionService.ts — business logic, calls RulesEngine
        ↓
RulesEngine.ts — evaluates rules, returns allow/deny
        ↓
Response sent back to user
        ↓
WebhookService.ts — notifies external systems (async)
```

How to explain this verbally:

> "When a user makes a request, it hits the server where middleware runs — that handles logging and rate limiting. Then it gets routed to the transaction routes, which matches the endpoint and calls the controller. The controller validates the request and calls the service. The TransactionService handles the business logic and calls the Rules Engine, which evaluates the rules and returns allow or deny. The response gets sent back to the user, and the WebhookService notifies any external systems."

Core Files Explained

index.ts (The Front Door)

- Creates the Express server
- Sets up middleware (logging, rate limiting, JSON parsing)
- Mounts all routes
- Handles errors and 404s

Key point: Rate limiting protects against brute force attacks. Login endpoints have stricter limits than regular API calls.

---

routes/index.ts (Traffic Director)

- Directs requests to the right handler based on URL path
- `/api/auth` → authRoutes
- `/api/transactions` → transactionRoutes
- `/api/rules` → rulesRoutes
- `/api/webhooks` → webhookRoutes
- `/api/batch` → batchRoutes

Key point: Specific routes (like `/stats`) must come before dynamic routes (like `/:id`) or they won't match correctly.

TransactionService.ts (Transaction Lifecycle)

Statuses:

```
PENDING → VALIDATING → PROCESSING → COMPLETED
  ↓      ↓         ↓         ↓
CANCELLED  FAILED      FAILED     REFUNDED
          ↓
       PENDING (retry, max 3 attempts)
```

Key methods:

Method	What It Does
createTransaction()	Validates input, creates with PENDING status
processTransaction()	Runs rules engine, simulates processing
retryTransaction()	Only for FAILED, max 3 retries
cancelTransaction()	Only before COMPLETED
refundTransaction()	Only after COMPLETED

Key point: Status transitions are enforced — you can't skip steps or go backwards illegally.

Interview-ready explanation for cancel vs refund:

> "Canceling a transaction means the authorization falls off — the funds never left the account, they were just on hold. Refunding means the customer gets their money back from funds that already left their account. In the code, cancelTransaction only works before COMPLETED status, and refundTransaction only works after COMPLETED."

RulesEngine.ts (Business Rules)

Rule Actions:

Action	What Happens	Real-World Example
ALLOW	Transaction proceeds	Normal transaction
DENY	Transaction blocked	Exceeds $100k limit
FLAG	Proceeds but marked for review	Unusual pattern
REQUIRE_APPROVAL	Needs supervisor approval	Large transfer

Operators:

- `GT` (greater than), `LT` (less than), `EQ` (equals)
- `IN` (in list), `NOT_IN` (not in list)
- `AND` / `OR` for composite rules

Default Rules:

1. Block transactions over $100,000
2. Flag transactions over $10,000
3. Reject transactions below $0.01
4. Only allow USD, EUR, GBP currencies

Key point: Rules are evaluated by priority order. If `stopOnFirstDeny` is true, processing stops at the first DENY.

Interview-ready explanation:

> "In payment operations, every transaction goes through compliance and fraud checks. The rules engine automates this — it evaluates things like amount thresholds, currency validation, and flags suspicious patterns. For example, there's a rule that blocks any transaction over $100,000 and another that flags anything over $10,000 for review. These mirror the kind of checks I've worked with at Navy Federal."

AuthService.ts (Security)

How login works:

1. User sends email + password
2. Server verifies credentials (password checked against hash)
3. Server generates JWT token with user info
4. User includes token in all future requests
5. Server verifies token signature on each request

Key concepts:

Concept	Why It Matters
Password hashing (bcrypt)	Never store plain passwords — protects users if database is breached
JWT token	Proves identity without re-entering password every request
Roles (admin/operator/user)	Different access levels for different users
Generic error messages	"Invalid email or password" — don't reveal which part was wrong

Key point: The token is like an ID badge. You show it with every request so the server knows who you are, what role you have, and that you already proved your identity.

WebhookService.ts (External Notifications)

What it does: When events happen (transaction completed, failed, etc.), automatically notifies subscribed external systems.

Events available:

- transaction.created, transaction.processing, transaction.completed
- transaction.failed, transaction.cancelled, transaction.refunded
- rule.triggered, error.critical

How subscription works: External system provides:

- URL (where to send notifications)
- Events (which ones they care about)
- Secret (for HMAC signature verification)

Key concepts:

Concept	What It Does
HMAC signature	Proves the webhook is authentic, not faked
Exponential backoff	Retry delays: 2s → 4s → 8s (don't hammer a struggling system)
Logging	Every attempt logged for audit trail

Interview-ready explanation:

> "Webhooks let external systems subscribe to events in my application. For example, a fraud monitoring system could register their URL and say 'notify me whenever a transaction fails.' When that event happens, my system sends them a POST request with the transaction details. It includes an HMAC signature so they can verify it's really from us, and it retries with exponential backoff if their system is temporarily down."

Technical Terms You Should Know

Term	Simple Explanation
API	A way for systems to talk to each other over the internet
Endpoint	A specific URL that does a specific thing (e.g., `/api/transactions`)

Term	Simple Explanation
Middleware	Code that runs on every request before it reaches its destination
JWT	JSON Web Token — a signed "permission slip" proving who you are
Hashing	One-way scrambling — you can verify a match but can't reverse it
HMAC	A signature using a shared secret to prove authenticity
Rate limiting	Restricting how many requests someone can make in a time period
Webhook	A way to push notifications to external systems when events happen
Exponential backoff	Retry delays that double each time (2s, 4s, 8s...)
Separation of concerns	Each file has one job — easier to debug, modify, collaborate

Common Interview Questions & Answers

"Tell me about this project."

> "I built a transaction processing simulator based on my experience working in payments operations at Navy Federal. I'm transitioning into application engineering, and I wanted to demonstrate that I understand not just the operations side, but the technical systems behind it. The simulator includes a rules engine for fraud detection and compliance, JWT authentication, webhook notifications, and batch processing. I built it with TypeScript and Node.js."

"Walk me through what happens when a user creates a transaction."

> "When a user makes a request, it hits the server where middleware runs — that handles logging and rate limiting. Then it gets routed to the transaction routes, which matches the endpoint and calls the controller. The controller validates the request and calls the service. The TransactionService handles the business logic and calls the Rules Engine, which evaluates the rules and returns allow or deny. The response gets sent back to the user, and the WebhookService notifies any external systems."

"What happens if a transaction fails? Can it be retried?"

> "If a transaction fails, it can be retried — but only if the status is FAILED, and there's a maximum of 3 retries. After that, it stops. The limit prevents infinite loops — if something is genuinely broken, you don't want the system hammering away forever. It could be a temporary network issue that clears up on retry, or it could be a real problem that needs someone to investigate."

"Why did you use TypeScript instead of JavaScript?"

> "TypeScript catches errors at compile time instead of runtime. So if I pass the wrong type of data to a function, I find out immediately while coding — not when a user hits the bug in production. For a financial system, that extra safety matters."

"Why separate files for routes, controllers, and services?"

> "I separated the code into routes, controllers, and services so each file has one job. Routes handle directing traffic, controllers handle validation and HTTP responses, and services handle the business logic. This makes it easier to debug — if transactions are failing, I know exactly where to look. It also means I can change the business logic without touching the routing, and multiple people could work on different parts without conflicts."

"Why include random failures in the simulator?"

> "The simulator includes random failures because in production, transactions don't always succeed — you get network timeouts, external bank outages, system errors. By simulating failures, I can verify that retry logic works correctly, error messages are clear, and the system fails gracefully instead of crashing. It's better to catch these issues in testing than discover your error handling is broken when real members are affected."

"Did you face any challenges building this?"

> "The main challenge was that I scaffolded the project with AI tools, which got me a working codebase fast — but I didn't fully understand every piece at first. I could have just left it there, but that wouldn't help me in interviews or on the job. So I went through each file, traced how requests flow through the system, and connected the code to patterns I already knew from my 7 years in payments operations. Now I can explain every component and extend it if needed. It taught me that using AI tools is fine, but you have to actually learn what they produce."

"What would you add if you had more time?"

> "I'd add two things. First, a frontend dashboard to visualize transactions in real-time — see what's pending, what's failed, spot bottlenecks. Second, settlement windows. Right now, transactions complete individually, but in real payment systems, completed transactions get batched and settled at end-of-day cutoff times. At Navy Federal, we have specific settlement windows — transactions after the cutoff roll to the next day. Adding that would make the simulator more realistic."

Tech Stack Summary

Technology	Purpose
TypeScript	Type-safe JavaScript — catches errors at compile time

Technology	Purpose
Node.js	JavaScript runtime for backend
Express	Web framework for handling HTTP requests
MongoDB	Database (optional — falls back to in-memory)
JWT	Authentication tokens
bcrypt	Password hashing
Axios	Making HTTP requests (for webhooks)
Jest	Testing framework
Swagger	API documentation

Your Honest Story

> "I scaffolded this project using AI tools to accelerate development, then studied the codebase to understand the architecture and business logic. The design decisions — the rules engine, transaction lifecycle, webhook system — are based on patterns I've worked with for 7 years in payments operations at Navy Federal. I can explain every component, debug issues, and extend the functionality."

This is a strength, not a weakness. Senior engineers use AI tools. What matters is that you understand what was built.