Chainflow V1 Documentation

Project Overview

Chainflow V1 is a robustness-oriented Web3 backend infrastructure designed to prioritize correctness, reliability, and maintainability over scalability in its initial iteration. At its core, the system serves as a secure and deterministic platform for managing encrypted wallets and processing asynchronous blockchain transactions across multiple chains. This approach ensures that all operations are executed in a predictable manner, minimizing risks associated with state inconsistencies or unexpected behaviors.

Key functionalities include:

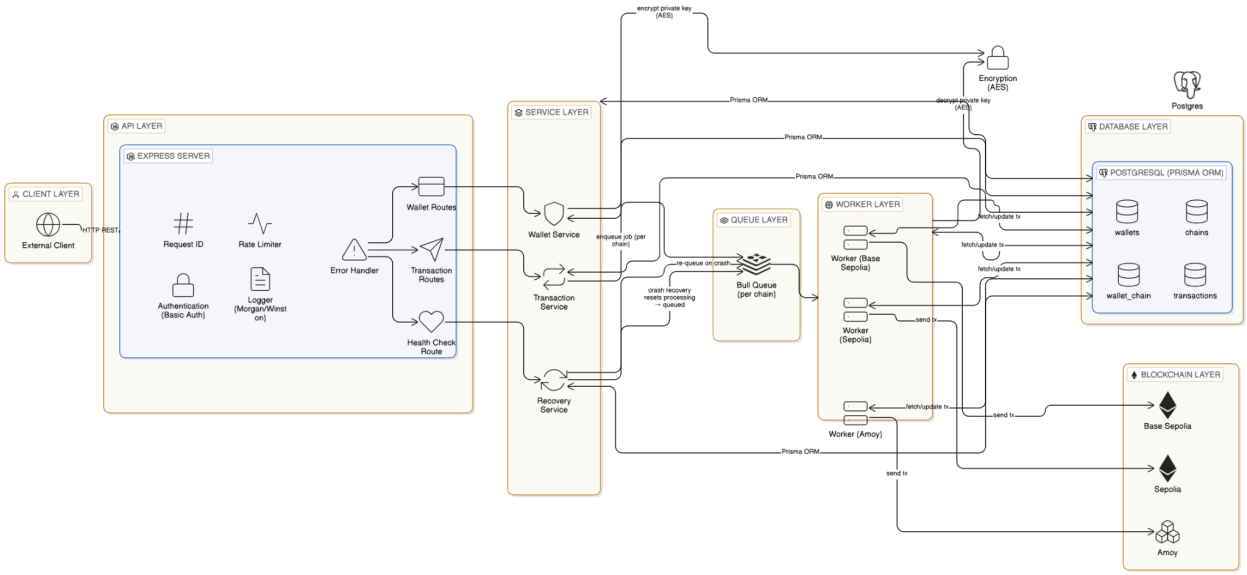
- **Encrypted Wallet Management**: Securely stores and handles wallets, with support for associating wallets to specific blockchain chains.
- **Per-Chain Wallet Enablement**: Allows wallets to be enabled on a per-chain basis, providing flexibility for multi-chain operations.
- **Asynchronous Transaction Handling**: Accepts transaction requests, queues them for execution, and manages their lifecycle without blocking the client.
- **Per-Chain Queuing**: Utilizes dedicated queues for each blockchain chain to isolate and organize transaction processing.
- **Transaction Lifecycle Management**: Tracks the state of transactions from initiation to completion or failure, ensuring transparency and auditability.
- **Crash Recovery Mechanisms**: Implements safeguards to recover gracefully from system crashes, preserving data integrity.

Priorities in V1 emphasize:

- **Deterministic Behavior**: All processes are designed to produce consistent outcomes given the same inputs, reducing non-determinism in blockchain interactions.
- **State Correctness**: Rigorous validation and state transitions ensure that the system's internal state accurately reflects reality.
- **Clean Layering**: The architecture is modular, with clear separation of concerns to facilitate maintenance and debugging.
- **Observability**: Extensive logging and structured responses enable easy monitoring and troubleshooting.
- **Infrastructure Clarity**: Uses well-established tools like Postgres and Redis with configurations focused on durability and simplicity.

Scalability features, such as parallel processing or advanced retry mechanisms, are intentionally deferred to future versions to maintain focus on foundational reliability.

Architecture Overview



The system follows a layered architecture to promote separation of concerns and ease of development. Requests flow sequentially through the layers, ensuring that each component handles a specific aspect of the operation. This design prevents tight coupling and allows for independent testing and scaling of individual layers.

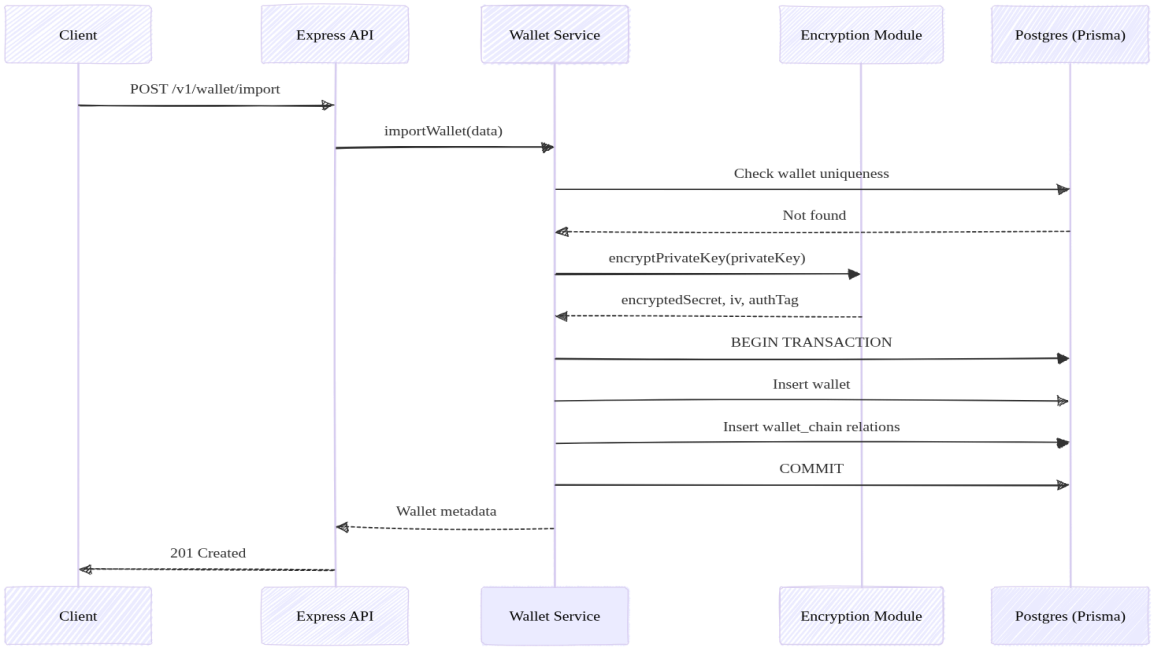
High-level flow:

- **Client** → Initiates requests via HTTP.
- **Express API Layer** → Handles incoming requests, validation, and response formatting.
- **Service Layer** → Encapsulates business logic, including validations and transaction preparation.
- **Database (Postgres via Prisma)** → Persists data using an ORM for type-safe interactions.
- **Queue Layer (Bull + Redis)** → Manages job queuing with durability features.
- **Worker Layer** → Executes queued jobs, interacting with blockchain RPCs.
- **Blockchain RPC** → External interface for signing and broadcasting transactions.

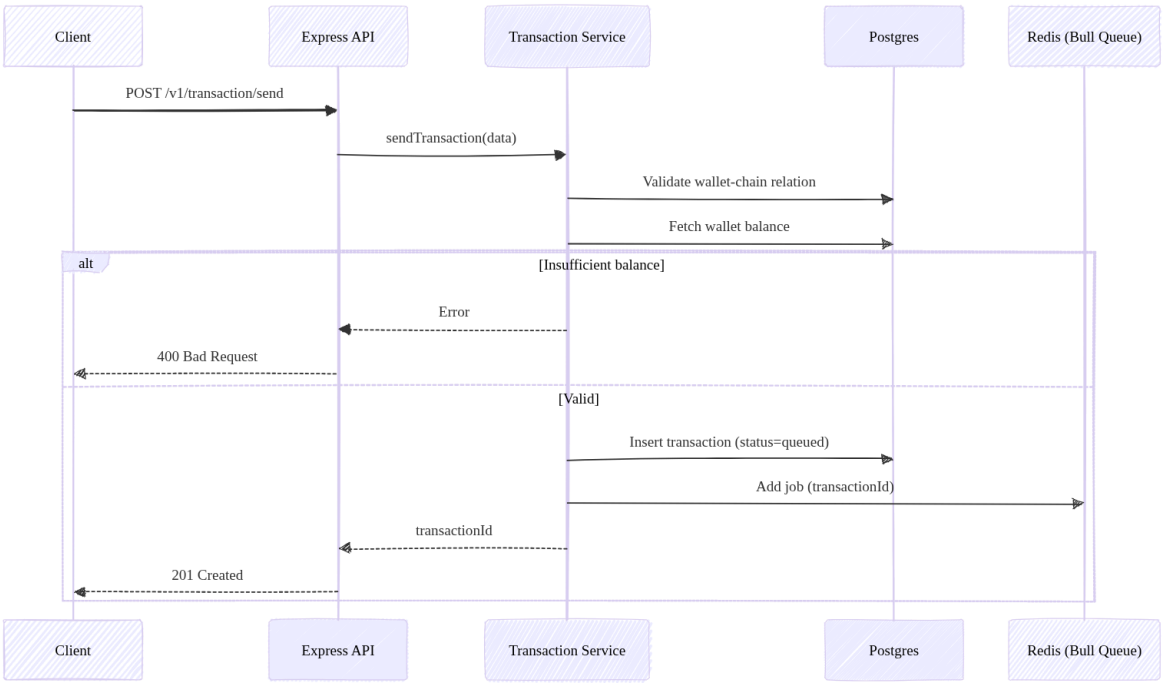
This unidirectional flow ensures that errors are propagated appropriately and that the system remains resilient to failures at any layer.

Sequence diagrams

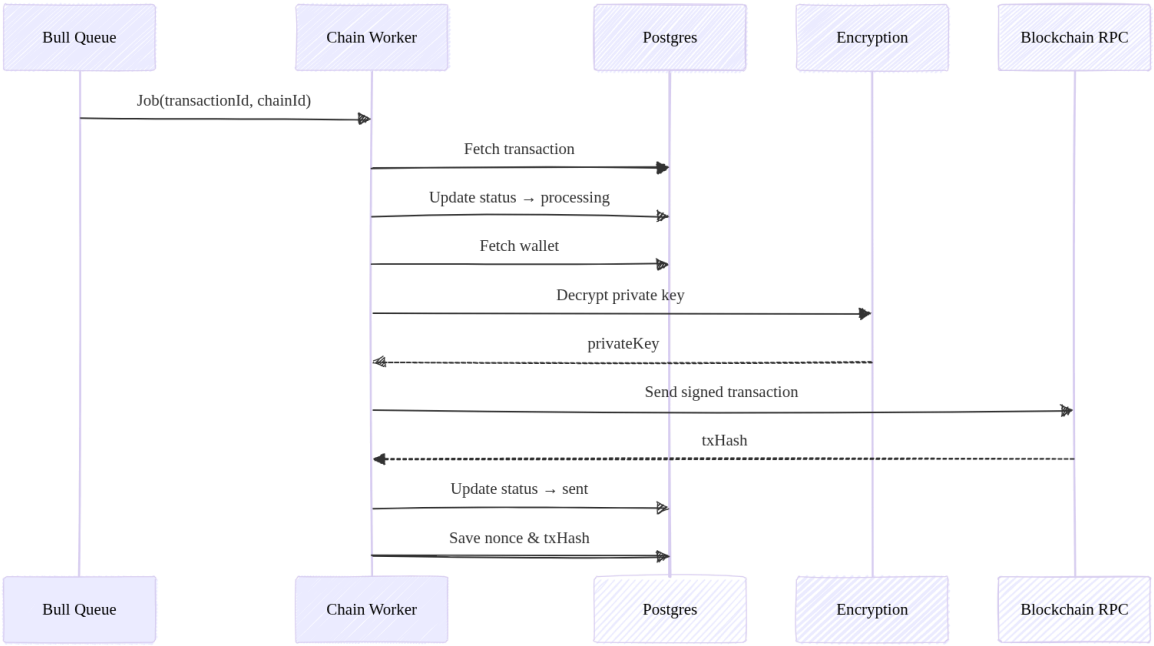
1. Import Wallet



2. Transaction Submission Flow (API → Queue)



3.Worker Processing Flow



Layers Breakdown

1. API Layer

The API layer serves as the entry point for all client interactions, built on Express.js for efficient request handling. It focuses on ensuring that requests are properly validated and logged before passing control to deeper layers.

Responsibilities:

- **Request Validation**: Checks for the presence and format of required parameters to prevent invalid data from propagating.
- **Lifecycle Logging**: Records the start, processing, and end of each request for auditing and performance monitoring.
- **Service Invocation**: Calls the appropriate service method to perform the core operation.
- **Structured Responses**: Returns JSON responses in a consistent format, including success data or error details.
- **Error Propagation**: Forwards any exceptions to a centralized error-handling middleware, which standardizes error responses (e.g., HTTP status codes and messages).

This layer acts as a gatekeeper, enhancing security and usability by providing clear, predictable interfaces.

2. Service Layer

The service layer houses the application's business logic, bridging the API and persistence layers. It ensures that operations align with domain rules, such as wallet permissions and balance sufficiency.

Responsibilities:

- **Business Logic Execution**: Orchestrates the steps needed for operations like wallet imports or transaction submissions.
- **Wallet-Chain Validation**: Verifies that the requested wallet is enabled for the specified chain, preventing unauthorized actions.
- **Balance Checks**: Queries the blockchain (via RPC) to confirm sufficient funds before proceeding, avoiding failed transactions due to insufficient balance.
- **Transaction Construction**: Builds the transaction payload, including calldata for contract interactions if applicable.
- **Database Writes**: Inserts initial transaction records into the database with a "queued" status.
- **Job Enqueuing**: Adds the job to the appropriate chain-specific queue for asynchronous processing.

By centralizing logic here, the system maintains consistency and makes it easier to update rules without affecting other layers.

3. Queue Layer

The queue layer leverages Bull (a Redis-based queue manager) to handle asynchronous tasks reliably. Each blockchain chain has its own dedicated queue, isolating workloads and preventing cross-chain interference.

Key Features:

- **Job Contents**: Minimalist design; each job includes only the `transactionId` and `chainId` to reduce payload size and complexity.
- **Durability**: Redis is configured with Append-Only File (AOF) persistence, ensuring that queued jobs survive restarts or crashes.

This setup guarantees that transactions are processed in order, with built-in mechanisms for job uniqueness (using `transactionId` as the job ID).

4. Worker Layer

Workers are responsible for executing queued jobs, interacting directly with blockchain networks. In V1, each chain has a single worker with concurrency set to 1, enforcing sequential processing.

Responsibilities:

- **Just-in-Time Nonce Fetching**: Retrieves the latest nonce from the blockchain right before signing to avoid nonce conflicts.
- **Deterministic State Transitions**: Updates transaction status in a predictable sequence (e.g., from "queued" to "processing").
- **Wallet Decryption**: Securely decrypts the wallet's private key for signing.
- **Transaction Signing and Broadcasting**: Signs the transaction and sends it via RPC.
- **Database Updates**: Records the outcome (e.g., "sent" or "failed") in the database.
- **Key Management**: Clears decrypted keys from memory immediately after use to enhance security.

This layer's focus on single-threaded execution per chain ensures correctness in V1, though it limits throughput.

5. Recovery Layer

The recovery layer provides crash-safety by resetting incomplete operations during system startup.

Behavior:

- Scans the database for transactions in "processing" status.
- Resets them to "queued" for re-processing.

This simple yet effective mechanism ensures idempotency and prevents data loss, aligning with V1's correctness goals.

Wallet Management APIs

These endpoints handle wallet lifecycle operations, ensuring secure import, retrieval, and configuration.

POST /v1/wallet/import

Description: Imports a new wallet by encrypting and storing its private key, optionally enabling it for specific chains.

Request Body Example:

```
{
  "name": "primary-wallet",
  "publicAddress": "0xABC...",
  "privateKey": "0xPRIVATEKEY...",
  "chainIds": [80002, 11155111]
}
```

Behavior:

- Validates uniqueness of the public address.
- Encrypts the private key using a secure algorithm.
- Stores wallet metadata in the database.
- If `chainIds` are provided, creates associations in a junction table.
- Wraps operations in a database transaction for atomicity.

Response Example:

```
{
  "result": {
    "id": "uuid",
    "name": "primary-wallet",
    "publicAddress": "0xABC...",
    "chainIds": [80002],
    "status": "active",
    "createdAt": "...",
    "updatedAt": "..."
  }
```

```
}
```

GET /v1/wallet/:publicAddress

Description: Retrieves metadata for a wallet by its public address.

Response Example:

```
{
  "name": "primary-wallet",
  "publicAddress": "0xABC...",
  "status": "active",
  "createdAt": "...",
  "updatedAt": "..."
}
```

PATCH /v1/wallet/:publicAddress/status

Description: Updates the status of a wallet (e.g., to disable it).

Request Body Example:

```
{
  "status": "disabled"
}
```

PATCH /v1/wallet/:walletId/chains

Description: Enables additional chains for an existing wallet.

Request Body Example:

```
{
  "chainIds": [80002, 97]
}
```

Behavior:

- Inserts new rows in the wallet-chain junction table.
- Skips existing associations to avoid duplicates.

Transaction System

These APIs facilitate submitting and querying asynchronous transactions.

POST /v1/transaction/send

Description: Submits a transaction for asynchronous execution, supporting transfers or contract interactions.

Request Body Examples:

Transfer:

```
 {
  "walletId": "uuid",
  "chainId": 80002,
  "toAddress": "0xReceiver",
  "value": "0.01"
}
```

Contract Interaction:

```
{
  "walletId": "uuid",
  "chainId": 80002,
  "toAddress": "0xContract",
  "value": "0",
```

```
  "functionSignature": "function set(uint256)",
  "args": [55]
}
```

Flow:

- Validates all required fields.
- Confirms the wallet is enabled for the chain.
- Fetches and verifies wallet balance against the requested value.
- Constructs calldata for contracts if needed.
- Inserts a new transaction record in "queued" status.
- Enqueues the job in the chain-specific queue.

Response Example:

```
{
  "transactionId": "uuid",
  "status": "queued",
  "walletId": "uuid",
  "chainId": 80002,
  "createdAt": "..."
}
```

GET /v1/transaction/:id

Description: Retrieves the current status and details of a transaction.

Response Example:

```
{
  "transactionId": "uuid",
  "status": "sent",
  "walletId": "uuid",
  "chainId": 80002,
  "createdAt": "...",
  "updatedAt": "..."
```

```
}
```

Transaction State Machine

Transactions progress through a finite state machine to track their lifecycle deterministically:

- **queued**: Initial state after submission and enqueuing.
- **processing**: Worker has started execution.
- **sent**: Transaction successfully signed and broadcasted.
- **failed**: An error occurred during processing.

V1 omits advanced states like "mined," retries, or confirmation polling to keep the system simple and focused on core correctness.

Nonce Handling (V1 Model)

In V1, nonce management is simplified due to single-concurrency workers per chain:

- Nonces are fetched just-in-time before signing.
- No dedicated nonce manager is used.
- Sequential execution ensures nonces are used in order, avoiding gaps or duplicates.

This model guarantees safety and determinism but may become a bottleneck in scaled versions.

Error Handling Strategy

Errors are managed centrally to ensure consistent behavior:

- Services throw descriptive errors.
- Controllers pass them to middleware for standardized HTTP responses.
- Workers update the transaction to "failed" in the database before re-throwing.
- No automatic retries in V1; failures require manual intervention.

This approach prioritizes transparency, allowing users to inspect and resubmit as needed.

Infrastructure

Redis

- Deployed in Docker for portability.
- AOF persistence enabled for queue durability.
- Used exclusively for queuing, ensuring jobs persist across restarts.

Postgres

- Accessed via Prisma ORM for schema management and type safety.
- Supports transactions, e.g., during wallet imports for atomic operations.

Reliability Guarantees (V1)

Chainflow V1 provides the following assurances:

- **Crash-Safe Recovery**: Startup logic resets "processing" transactions to "queued."
- **No Double-Processing**: Enforced by concurrency=1 and unique job IDs.
- **Duplicate Prevention**: Job IDs match transaction IDs.
- **Validation Enforcement**: Wallet-chain relationships and balances are checked pre-enqueue.
- **Overall Integrity**: Focus on deterministic, observable operations minimizes runtime surprises.

This foundation sets the stage for future enhancements while ensuring a solid, trustworthy base.