API Documentation

Purpose: Comprehensive documentation of the REST/GraphQL API that serves as the interface between frontend and backend systems.

Key Contents:

- API design principles and standards
- Authentication mechanisms and security protocols
- Complete endpoint documentation with request/response examples
- Data model specifications and relationships
- Error codes and handling procedures
- Performance considerations and optimizations

Introduction & Standards

Overview

Our API serves as the primary interface between the frontend application and the blockchain infrastructure. It provides a RESTful interface that handles business logic, smart contract interactions, and data management.

API Design Principles

1. RESTful Resource Naming

- Use nouns for resource names
- Apply consistent plural forms for collections
- Use kebab-case for URLs
- Nest resources logically

? Good Examples	? Bad Examples
GET /transactions	GET /get-all-transactions
GET /users/{userId}/portfolios	GET /userPortfolio
GET /smart-contracts/events	GET /smart_contract_events

2. HTTP Methods

Use standard HTTP methods consistently:

Method	Description
`GET`	Retrieve resources
`POST`	Create new resources
`PUT`	Update existing resources (full update)
`PATCH`	Partial resource updates
`DELETE`	Remove resources

3. Response Formats

All API responses follow this standard structure:

```
{
  "status": "success" | "error",
  "data": {
    // Response payload
  },
  "metadata": {
    "timestamp": "ISO-8601 timestamp",
```

```
    "version": "API version",
    "pagination": {
      "page": 1,
      "limit": 20,
      "total": 100
    }
  },
  "errors": [
    {
      "code": "ERROR_CODE",
      "message": "Human readable message",
      "field": "field_name"
    }
  ]
}
```

4. Status Codes

Code	Description	Usage
200	OK	Successful GET, PUT, PATCH
201	Created	Successful POST
204	No Content	Successful DELETE
400	Bad Request	Invalid input
401	Unauthorized	Missing/invalid authentication
403	Forbidden	Authenticated but unauthorized
404	Not Found	Resource doesn't exist
429	Too Many Requests	Rate limit exceeded
500	Internal Server Error	Server-side error

5. Versioning

- Version prefix in URL: `/v1/resources`
- Major version changes only
- Maintain backwards compatibility within versions
- Support at most 2 versions simultaneously

6. Query Parameters

Pagination

```
GET /transactions?page=2&limit=20
```

Filtering

```
GET /transactions?status=pending&type=deposit
```

Sorting

```
GET /transactions?sort=timestamp:desc
```

Authentication & Security

Any code or data structure shown below is sample code, and sample data structures provided as best practice. As we start implementing this in our project, I expect the engineering team to update this page.

Authentication Flow

API Key Authentication

```
interface ApiKey {
  id: string;
  key: string; // Hashed
  roles: string[];
  permissions: Permission[];
  metadata: {
    createdAt: Date;
    lastUsed: Date;
    createdBy: string;
    environment: string;
  };
}

const authenticate = async (req: Request, res: Response, next: NextFunction) => {
  const apiKey = req.headers['x-api-key'];

  if (!apiKey) {
    throw new AuthError('API key required');
  }

  const keyDetails = await validateApiKey(apiKey);
  req.auth = {
    keyId: keyDetails.id,
    roles: keyDetails.roles,
    permissions: keyDetails.permissions
  };

  next();
};
```

Wallet Authentication

```
interface WalletAuth {
  address: string;
  nonce: string;
  signature: string;
  timestamp: number;
}

const verifyWalletSignature = async (req: Request, res: Response, next: NextFunction) => {
  const { address, nonce, signature, timestamp } = req.body;

  // Verify timestamp is recent
  if (Date.now() - timestamp > 5 * 60 * 1000) {
    throw new AuthError('Signature expired');
  }

  // Verify signature
  const message = `${nonce}:${timestamp}`;
  const recoveredAddress = ethers.utils.verifyMessage(message, signature);

  if (recoveredAddress.toLowerCase() !== address.toLowerCase()) {
    throw new AuthError('Invalid signature');
  }

  req.wallet = { address };
  next();
};
```

Authorization

Role-Based Access Control

```
interface Permission {
  resource: string;
  action: 'CREATE' | 'READ' | 'UPDATE' | 'DELETE';
  conditions?: Record<string, any>;
}

const ROLES = {
  ADMIN: {
    permissions: [
```

```
      { resource: '*', action: '*' }
    ]
  },
  VAULT_CREATOR: {
    permissions: [
      { resource: 'vault', action: 'CREATE' },
      { resource: 'vault', action: 'READ' }
    ]
  },
  VOTER: {
    permissions: [
      { resource: 'proposal', action: 'READ' },
      { resource: 'vote', action: 'CREATE' },
      {
        resource: 'vault',
        action: 'READ',
        conditions: { status: 'ACTIVE' }
      }
    ]
  }
};

const checkPermission = (required: Permission) => {
  return (req: Request, res: Response, next: NextFunction) => {
    const hasPermission = req.auth.permissions.some(p =>
      matchPermission(p, required)
    );

    if (!hasPermission) {
      throw new AuthError('Insufficient permissions');
    }

    next();
  };
};
```

Blockchain Security

Transaction Signing

```
interface SignedTransaction {
  hash: string;
  signature: string;
  signers: string[];
  threshold: number;
}

const validateTransaction = async (tx: SignedTransaction) => {
  // Verify sufficient signers
  if (tx.signers.length < tx.threshold) {
    throw new SecurityError('Insufficient signatures');
  }

  // Verify all signatures
  for (const signer of tx.signers) {
    const isValid = await verifySignature(tx.hash, signer);
    if (!isValid) {
      throw new SecurityError('Invalid signature');
    }
  }

  return true;
};
```

Smart Contract Interaction

```
const executeSecureContractCall = async (
  contract: ethers.Contract,
  method: string,
  params: any[],
  options: SecurityOptions
) => {
  // Validate parameters
  const validated = await validateContractParams(method, params);

  // Simulate transaction
  const simulation = await contract.callStatic[method](...validated);

  // Check security constraints
  if (!meetsSecurityConstraints(simulation, options)) {
```

```
    throw new SecurityError('Security constraints not met');
  }

  // Execute transaction
  return await contract[method](...validated);
};
```

Rate Limiting

Implementation

```
import rateLimit from 'express-rate-limit';
import RedisStore from 'rate-limit-redis';

const rateLimiter = rateLimit({
  store: new RedisStore({
    client: redisClient,
    prefix: 'rate-limit:'
  }),
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: (req) => {
    // Adjust limits based on authentication level
    if (req.auth?.roles.includes('ADMIN')) {
      return 1000;
    }
    return 100;
  },
  keyGenerator: (req) => {
    return `${req.auth?.keyId || req.ip}`;
  }
});
```

Security Headers

Configuration

```
import helmet from 'helmet';

app.use(helmet({
  contentSecurityPolicy: {
```

```
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'"],
      styleSrc: ["'self'"],
      imgSrc: ["'self'"],
      connectSrc: ["'self'", process.env.BLOCKCHAIN_RPC],
    }
  },
  referrerPolicy: { policy: 'same-origin' },
  hsts: {
    maxAge: 31536000,
    includeSubDomains: true,
    preload: true
  }
}));
```

Data Encryption

Sensitive Data Handling

```
interface EncryptedData {
  data: string;
  iv: string;
  tag: string;
}

const encryptSensitiveData = async (
  data: any,
  key: Buffer
): Promise<EncryptedData> => {
  const iv = crypto.randomBytes(12);
  const cipher = crypto.createCipheriv('aes-256-gcm', key, iv);

  const encrypted = Buffer.concat([
    cipher.update(JSON.stringify(data)),
    cipher.final()
  ]);

  return {
    data: encrypted.toString('base64'),
```

```
    iv: iv.toString('base64'),
    tag: cipher.getAuthTag().toString('base64')
  };
};
```

Security Monitoring

Audit Logging

```
interface AuditLog {
  action: string;
  actor: {
    id: string;
    type: 'API_KEY' | 'WALLET';
  };
  resource: {
    type: string;
    id: string;
  };
  metadata: Record<string, any>;
  timestamp: Date;
}

const auditLogger = {
  log: async (audit: AuditLog) => {
    await Promise.all([
      // Store in database
      db.collection('audit_logs').insertOne(audit),

      // Send to logging service
      logger.info('Security audit', { audit }),

      // Alert on suspicious activity
      detectSuspiciousActivity(audit)
    ]);
  }
};
```

Security Checklist

Authentication

- [ ] Implement API key rotation
- [ ] Set up wallet signature verification
- [ ] Configure token expiration
- [ ] Implement rate limiting

Authorization

- [ ] Define role permissions
- [ ] Implement access control
- [ ] Set up resource policies
- [ ] Configure audit logging

Blockchain

- [ ] Set up transaction validation
- [ ] Implement multi-sig requirements
- [ ] Configure contract interaction limits
- [ ] Set up transaction monitoring

Data Security

- [ ] Implement encryption at rest
- [ ] Set up secure key management
- [ ] Configure data backup
- [ ] Implement data sanitization

Infrastructure

- [ ] Configure security headers
- [ ] Set up DDOS protection

- [ ] Implement IP filtering
- [ ] Configure SSL/TLS

Monitoring

- [ ] Set up activity logging
- [ ] Configure alerts
- [ ] Implement anomaly detection
- [ ] Set up incident response

Data Models & Relationships

Overview

This document outlines the updated core data models in the L4VA system. Our revised architecture now supports dynamic vault management and a dual-level asset classification. This design not only covers standard use cases (e.g., works of art, real estate, gold bullion, and equity) but also enables the registration of custom vault and asset types as new real-world asset classes emerge.

Core Models

1. Vault

The primary entity representing a tokenized asset vault. The vault model now supports both fixed types (e.g., PUBLIC, PRIVATE, SEMI_PRIVATE) and dynamically registered types. It also allows for an open set of supported asset types, which can be expanded via a configuration registry or admin endpoint.

```
interface Vault {
  id: string;                  // Unique identifier
  contractAddress: string;       // On-chain vault contract address
  // VaultType can be one of the predefined values or a dynamically registered type.
  type: VaultType;               // e.g., PRIVATE | PUBLIC | SEMI_PRIVATE | custom types
  status: VaultStatus;           // DRAFT | ACTIVE | LOCKED | TERMINATED

  // Asset Configuration: supports dynamic asset types.
  // Defaults include NFT (for unique assets) and CNT (for fractional assets).
  assetTypes: AssetType[];
  assetWhitelist: string[];     // Allowed asset addresses
  contributorWhitelist: string[]; // Allowed contributor addresses

  // Windows Configuration
  assetWindow: {
    startTime: Date;
    duration: string;           // Format: "DD:HH:MM"
    status: WindowStatus;
  };

  investmentWindow: {
    startTime: Date;
    duration: string;           // Format: "DD:HH:MM"
    status: WindowStatus;
    valuationType: 'FIXED' | 'LBE';
  };
```

```
  // Fractionalization Settings
  fractionalization: {
    percentage: number;         // XX.XX%
    tokenSupply: number;
    tokenDecimals: number;      // 1-9
    tokenAddress: string;       // Fractional token contract address
  };

  // Investment Settings
  investment: {
    reserve: number;            // XX.XX%
    liquidityPool: number;      // XX.XX%
  };

  // Termination Settings
  termination: {
    type: 'DAO' | 'PROGRAMMED';
    fdp: number;                // Floor price deviation percentage
  };

  // Metadata: provides room for additional dynamic configuration.
  createdBy: string;            // Admin wallet address
  createdAt: Date;
  updatedAt: Date;
  metadata: Record<string, any>; // Additional configurable fields (e.g., custom vault parameters)
}

type WindowStatus = 'PENDING' | 'ACTIVE' | 'COMPLETED' | 'FAILED';
```

2. Asset

Represents assets held within vaults. This model introduces a dual-level classification: a primary tokenization method (e.g., NFT for unique items, CNT for fractional assets) and a secondary categorization (via metadata) to define asset-specific classes such as art, real estate, commodities, or equity.

```
interface Asset {
  id: string;
  vaultId: string;
  // AssetType may be one of the default types (e.g., NFT, CNT) or a dynamically registered type.
  type: AssetType;
```

```
  contractAddress: string;
  tokenId?: string;             // For NFTs
  quantity: number;             // For CNTs

  // Valuation
  floorPrice: number;           // For NFTs
  dexPrice: number;             // For CNTs
  lastValuation: Date;

  // Status
  status: AssetStatus;          // PENDING | LOCKED | RELEASED
  lockedAt?: Date;
  releasedAt?: Date;

  // Metadata now includes an additional "category" field.
  // This field can capture asset-specific classifications (e.g., art, real_estate, commodity, equity)
  metadata: {
    name: string;
    description: string;
    imageUrl: string;
    category?: string;
    attributes: Record<string, any>;
  };

  // Tracking
  addedBy: string;              // Wallet address
  addedAt: Date;
  updatedAt: Date;
}

type AssetStatus = 'PENDING' | 'LOCKED' | 'RELEASED';
```

3. Proposal

Governance proposals remain largely unchanged, but note that proposals may reference dynamically registered asset types or custom governance rules based on asset categories.

```
interface Proposal {
  id: string;
  vaultId: string;
  type: ProposalType;         // ASSET_SALE | BUY | STAKE | LIQUIDATE
```

```
// Phases
votingPhase: {
  duration: string;        // Format: "DD:HH:MM"
  startTime: Date;
  endTime: Date;
  status: PhaseStatus;
  quorum: number;          // Required participation percentage
};

lockPhase: {
  duration: string;
  startTime: Date;
  endTime: Date;
  status: PhaseStatus;
};

executionPhase: {
  duration: string;
  startTime: Date;
  endTime: Date;
  status: PhaseStatus;
  requiredCosigners: number;
};

// Proposal Details: can target assets of any supported type.
settings: {
  assets: string[];        // Affected asset IDs
  effects: ProposalEffect[];
};

// Results
votes: {
  approve: number;
  reject: number;
  totalVoted: number;
  uniqueVoters: number;
};

// Status
```

```
  status: ProposalStatus;    // DRAFT | ACTIVE | PASSED | FAILED | EXECUTED
  result?: ProposalResult;

  // Metadata
  createdBy: string;         // Proposer wallet address
  createdAt: Date;
  updatedAt: Date;
}

type PhaseStatus = 'PENDING' | 'ACTIVE' | 'COMPLETED' | 'FAILED';
type ProposalStatus = 'DRAFT' | 'ACTIVE' | 'PASSED' | 'FAILED' | 'EXECUTED';
```

4. Vote

Individual votes on proposals used to execute governance decisions. This model remains unchanged.

```
interface Vote {
  id: string;
  proposalId: string;
  wallet: string;          // Voter's wallet address
  decision: 'APPROVE' | 'REJECT';
  amount: number;          // Amount of FTs staked in vote

  // Status
  status: VoteStatus;      // CAST | CONFIRMED | REVOKED
  confirmationTx?: string; // Blockchain transaction hash

  // Timing
  castAt: Date;
  confirmedAt?: Date;
  revokedAt?: Date;
}
```

5. Stake

Represents staked fractional tokens that contribute to voting power. This model remains similar but supports dynamic governance processes tied to various asset classes.

```
interface Stake {
  id: string;
  vaultId: string;
  wallet: string;          // Staker's wallet address
```

```
  amount: number;         // Staked amount

  // Status
  status: StakeStatus;    // ACTIVE | LOCKED | RELEASED
  lockedUntil?: Date;     // For time-locked stakes

  // Voting Power
  votingPower: number;
  usedPower: number;      // Power used in active votes

  // Tracking
  createdAt: Date;
  updatedAt: Date;
  transactions: {
    stakeHash: string;
    unstakeHash?: string;
  };
}
```

Relationships

Entity Relationships Diagram

```
  Vault ||--o{ Asset : contains
  Vault ||--o{ Proposal : has
  Vault ||--o{ Stake : holds
  Proposal ||--o{ Vote : receives
  Stake ||--o{ Vote : powers
```

Key Relationships

1. **Vault ? Assets**
   - One-to-many relationship
   - A vault can contain multiple assets; each asset belongs to a single vault.
   - Relationship constraints are enforced by assetTypes and assetWhitelist.
2. **Vault ? Proposals**
   - One-to-many relationship
   - Proposals are tied to a single vault and may leverage asset category-specific rules.
3. **Proposal ? Votes**
   - One-to-many relationship

- Votes are associated with specific proposals, with vote weight determined by staked tokens.

4. **Stake ? Votes**
   - One-to-many relationship
   - Stakes power votes and help track voting power relative to the amount staked.

Validation Rules

1. Vault Validation

```
const vaultValidation = {
  assetWindow: {
    minDuration: "01:00:00",
    maxDuration: "30:00:00"
  },
  investmentWindow: {
    minDuration: "24:00:00",
    maxDuration: "168:00:00"
  },
  fractionalization: {
    minPercentage: 1,
    maxPercentage: 100,
    minDecimals: 1,
    maxDecimals: 9
  },
  investment: {
    minReserve: 5,
    maxReserve: 50,
    minLiquidity: 1,
    maxLiquidity: 20
  }
};
```

2. Asset Validation

```
const assetValidation = {
  nft: {
    maxPerVault: 100
  },
  cnt: {
    minQuantity: 1,
```

```
    maxQuantity: 1000000
  },
  valuation: {
    maxAge: "24:00:00"
  }
};
```

3. Proposal Validation

```
const proposalValidation = {
  voting: {
    minDuration: "24:00:00",
    maxDuration: "168:00:00",
    minQuorum: 10,
    maxQuorum: 100
  },
  lock: {
    minDuration: "12:00:00",
    maxDuration: "48:00:00"
  },
  execution: {
    minDuration: "24:00:00",
    maxDuration: "72:00:00"
  }
};
```

Implementation Guidelines

1. Database Indexes

```
// Vault Indexes
{
  "contractAddress": 1,
  "status": 1,
  "type": 1,
  "createdAt": -1
}

// Asset Indexes
{
```

```
  "vaultId": 1,
  "contractAddress": 1,
  "status": 1
}

// Proposal Indexes
{
  "vaultId": 1,
  "status": 1,
  "votingPhase.endTime": 1
}

// Vote Indexes
{
  "proposalId": 1,
  "wallet": 1,
  "status": 1
}

// Stake Indexes
{
  "vaultId": 1,
  "wallet": 1,
  "status": 1
}
```

2. Cascade Behaviors

```
// Delete cascade rules
const cascadeRules = {
  vault: {
    onDelete: ['assets', 'proposals', 'stakes']
  },
  proposal: {
    onDelete: ['votes']
  }
};
```

3. Data Integrity

```
interface DataIntegrityChecks {
  validateStakeBalance: () => Promise<void>;
  validateVotingPower: () => Promise<void>;
  validateProposalStatus: () => Promise<void>;
  validateAssetLocks: () => Promise<void>;
}
```

Best Practices

1. Data Access

- Use the repositories pattern.
- Implement a caching strategy.
- Handle concurrent updates.
- Maintain audit logs.

2. Validation

- Validate at the model level.
- Enforce business rules.
- Check relationships.
- Verify blockchain state.

3. Performance

- Use appropriate indexes.
- Implement pagination.
- Optimize queries.
- Monitor database load.

4. Security

- Encrypt sensitive data.
- Validate permissions.
- Audit data access.
- Handle PII properly.

Note: I recently updated this document to reflect a move toward a dynamic, extensible framework as detailed in our Vault Formation and Data Models & Relationships guidelines, ensuring our platform remains agile in representing any real-world asset.

Error Handling & Logging

Error Types and Handling

Standard Error Types

```
enum ErrorType {
  VALIDATION_ERROR = 'VALIDATION_ERROR',
  AUTHENTICATION_ERROR = 'AUTHENTICATION_ERROR',
  AUTHORIZATION_ERROR = 'AUTHORIZATION_ERROR',
  BLOCKCHAIN_ERROR = 'BLOCKCHAIN_ERROR',
  BUSINESS_LOGIC_ERROR = 'BUSINESS_LOGIC_ERROR',
  EXTERNAL_SERVICE_ERROR = 'EXTERNAL_SERVICE_ERROR',
  SYSTEM_ERROR = 'SYSTEM_ERROR'
}

interface BaseError extends Error {
  type: ErrorType;
  code: string;
  details?: Record<string, any>;
  timestamp: string;
  correlationId: string;
}

class VaultError extends BaseError {
  constructor(type: ErrorType, code: string, message: string, details?: Record<string, any>) {
    super(message);
    this.type = type;
    this.code = code;
    this.details = details;
    this.timestamp = new Date().toISOString();
    this.correlationId = getCurrentCorrelationId();
  }
}
```

Error Implementation

```
// Error throwing
async function createVault(params: VaultParams) {
  try {
```

```
    await validateVaultParams(params);
  } catch (error) {
    throw new VaultError(
      ErrorType.VALIDATION_ERROR,
      'INVALID_VAULT_PARAMS',
      'Invalid vault parameters provided',
      { params, validationErrors: error.details }
    );
  }
}

// Error handling middleware
app.use((error: Error, req: Request, res: Response, next: NextFunction) => {
  if (error instanceof VaultError) {
    logger.error('Vault operation failed', {
      type: error.type,
      code: error.code,
      message: error.message,
      details: error.details,
      correlationId: error.correlationId
    });

    return res.status(getHttpStatus(error.type)).json({
      status: 'error',
      error: {
        code: error.code,
        message: error.message,
        correlationId: error.correlationId
      }
    });
  }
  next(error);
});
```

Logging Implementation

Logger Configuration

```
import winston from 'winston';
import { ElasticsearchTransport } from 'winston-elasticsearch';
```

```
const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  defaultMeta: { service: 'vault-service' },
  transports: [
    new winston.transports.Console(),
    new ElasticsearchTransport({
      level: 'info',
      clientOpts: {
        node: process.env.ELASTICSEARCH_URL,
        auth: {
          username: process.env.ELASTICSEARCH_USERNAME,
          password: process.env.ELASTICSEARCH_PASSWORD
        }
      },
      index: 'vault-logs-${process.env.NODE_ENV}-%{YYYY.MM.DD}',
      mappingTemplate: {
        index_patterns: ['vault-logs-*'],
        settings: {
          number_of_shards: 1,
          number_of_replicas: 1
        },
        mappings: {
          properties: {
            '@timestamp': { type: 'date' },
            level: { type: 'keyword' },
            message: { type: 'text' },
            correlationId: { type: 'keyword' },
            type: { type: 'keyword' },
            code: { type: 'keyword' },
            details: { type: 'object' }
          }
        }
      }
    })
  ]
```

```
});
```

Structured Logging

```
// Request logging middleware
app.use((req: Request, res: Response, next: NextFunction) => {
  const correlationId = generateCorrelationId();
  setCurrentCorrelationId(correlationId);

  logger.info('Incoming request', {
    method: req.method,
    path: req.path,
    correlationId,
    ip: req.ip,
    userAgent: req.get('user-agent')
  });

  next();
});

// Business operation logging
async function executeProposal(proposalId: string) {
  logger.info('Starting proposal execution', {
    proposalId,
    correlationId: getCurrentCorrelationId()
  });

  try {
    const result = await performExecution(proposalId);

    logger.info('Proposal execution completed', {
      proposalId,
      result,
      correlationId: getCurrentCorrelationId()
    });

    return result;
  } catch (error) {
    logger.error('Proposal execution failed', {
      proposalId,
```

```
      error: error.message,
      stack: error.stack,
      correlationId: getCurrentCorrelationId()
    });
    throw error;
  }
}
```

ELK Stack Configuration

Logstash Pipeline

```
input {
  beats {
    port => 5044
  }
}

filter {
  json {
    source => "message"
  }

  date {
    match => [ "@timestamp", "ISO8601" ]
  }

  if [type] == "BLOCKCHAIN_ERROR" {
    grok {
      match => { "message" => "%{GREEDYDATA:transaction_hash}" }
    }
  }
}

output {
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    index => "vault-logs-%{+YYYY.MM.dd}"
    document_type => "_doc"
  }
```

```
}
```

ElasticSearch Index Template

```
{
  "index_patterns": ["vault-logs-*"],
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1,
    "index.refresh_interval": "5s"
  },
  "mappings": {
    "properties": {
      "@timestamp": { "type": "date" },
      "service": { "type": "keyword" },
      "level": { "type": "keyword" },
      "correlationId": { "type": "keyword" },
      "type": { "type": "keyword" },
      "code": { "type": "keyword" },
      "message": { "type": "text" },
      "details": {
        "type": "object",
        "dynamic": true
      },
      "trace": {
        "properties": {
          "stack": { "type": "text" },
          "method": { "type": "keyword" },
          "line": { "type": "integer" }
        }
      },
      "request": {
        "properties": {
          "method": { "type": "keyword" },
          "path": { "type": "keyword" },
          "ip": { "type": "ip" },
          "userAgent": { "type": "text" }
        }
      }
    }
```

```
  }
}
```

Kibana Error Monitoring Dashboard

```
{
  "title": "Vault Error Monitoring",
  "panels": [
    {
      "type": "visualization",
      "title": "Errors by Type",
      "visualization": {
        "type": "pie",
        "aggs": [
          { "type": "count", "schema": "metric" },
          { "type": "terms", "field": "type", "schema": "segment" }
        ]
      }
    },
    {
      "type": "visualization",
      "title": "Error Timeline",
      "visualization": {
        "type": "line",
        "aggs": [
          { "type": "count", "schema": "metric" },
          { "type": "date_histogram", "field": "@timestamp", "schema": "segment" }
        ]
      }
    }
  ]
}
```

Implementation Guidelines

Best Practices

1. Error Handling
- Use custom error classes
- Include correlation IDs
- Add contextual details
- Handle errors at boundaries

- Never expose internal errors

2. Logging
- Use structured logging
- Include correlation IDs
- Log at appropriate levels
- Avoid sensitive data
- Add business context

3. Monitoring
- Set up alerting
- Monitor error rates
- Track response times
- Watch resource usage
- Set up dashboards

4. Security
- Sanitize logged data
- Encrypt sensitive logs
- Implement log retention
- Control log access
- Monitor suspicious patterns

Getting Started

1. Install Dependencies

```
npm install winston winston-elasticsearch @elastic/elasticsearch
```

2. Configure Environment

```
# .env file
ELASTICSEARCH_URL=http://localhost:9200
ELASTICSEARCH_USERNAME=elastic
ELASTICSEARCH_PASSWORD=changeme
LOG_LEVEL=info
```

3. Set Up ELK Stack

```
docker-compose up -d elasticsearch logstash kibana
```

4. Initialize Templates

```
curl -X PUT "localhost:9200/_template/vault-logs" -H "Content-Type: application/json" -d @template.json
```

5. Verify Setup

```
# Test logging
logger.info('Test log entry');
```

```
# Check Elasticsearch
curl -X GET "localhost:9200/vault-logs-*/_search"
```

API Versioning & Changes

Introduction

This guide outlines our approach to API versioning, managing changes, and supporting clients through API evolution. We use semantic versioning (major.minor.patch) to communicate the scope and impact of changes while ensuring a smooth transition between versions.

Version Control Strategy

Our versioning strategy ensures backward compatibility while enabling API evolution. We maintain clear upgrade paths and support multiple versions simultaneously to accommodate different client needs.

Versioning Schema

Defines how we structure and identify different versions of the API using semantic versioning.

```
interface APIVersion {
  major: number;    // Breaking changes
  minor: number;    // Backward-compatible features
  patch: number;    // Bug fixes
  status: 'alpha' | 'beta' | 'stable' | 'deprecated';
}

// Example: v2.1.3
const VERSION_PATTERN = /^v(\d+)\.(\d+)\.(\d+)$/;
```

URI Versioning

Implements version routing in API endpoints through URL-based versioning.

```
// Base URL structure
const API_URL = 'https://api.example.com/v{major}/{resource}';

// Version routing middleware
const versionRouter = (req: Request, res: Response, next: NextFunction) => {
  const version = req.path.split('/')[1]; // e.g., 'v2'
  const [major] = version.match(VERSION_PATTERN) || [];

  req.apiVersion = {
    major: parseInt(major),
    implementation: getVersionImplementation(major)
  };
```

```
  next();
};
```

Change Classifications

We classify API changes based on their impact on existing clients, which determines version increments and communication strategies.

1. Breaking Changes

Changes that require client updates and trigger a major version increment.

```
interface BreakingChange {
  type: 'BREAKING';
  changes: {
    removedFields?: string[];
    modifiedFields?: {
      field: string;
      oldType: string;
      newType: string;
    }[];
    removedEndpoints?: string[];
    authenticationChanges?: string[];
  };
  migrationGuide: string;
  effectiveDate: Date;
}
```

2. Non-Breaking Changes

Backward-compatible additions resulting in minor version increments.

```
interface NonBreakingChange {
  type: 'NON_BREAKING';
  changes: {
    addedFields?: string[];
    addedEndpoints?: string[];
    optionalParameters?: string[];
    enhancedResponses?: string[];
  };
  announcementDate: Date;
```

```
}
```

3. Bug Fixes

Corrections that don't affect the API contract, resulting in patch version increments.

```
interface BugFix {
  type: 'BUG_FIX';
  fixes: {
    description: string;
    affectedEndpoints: string[];
    resolution: string;
  }[];
  deploymentDate: Date;
}
```

Version Implementation

Manages different API versions within the codebase while maintaining clean separation between versions.

Version Manager

```
class APIVersionManager {
  private versions: Map<number, Implementation>;

  constructor() {
    this.versions = new Map();
    this.initializeVersions();
  }

  private initializeVersions() {
    this.versions.set(1, new V1Implementation());
    this.versions.set(2, new V2Implementation());
  }

  public getImplementation(version: number): Implementation {
    const impl = this.versions.get(version);
    if (!impl) {
      throw new Error(`Version ${version} not supported`);
    }
    return impl;
```

```
  }
}
```

Change Management Process

1. Planning Phase

Structured approach to planning and assessing API changes.

```
interface ChangeProposal {
  type: 'BREAKING' | 'NON_BREAKING' | 'BUG_FIX';
  description: string;
  justification: string;
  impact: {
    clients: string[];
    endpoints: string[];
    estimatedEffort: string;
  };
  timeline: {
    developmentStart: Date;
    betaRelease: Date;
    stableRelease: Date;
    deprecationDate?: Date;
  };
}
```

2. Communication Strategy

Clear communication of changes to API clients.

```
interface VersionAnnouncement {
  version: string;
  type: 'NEW_VERSION' | 'DEPRECATION' | 'END_OF_LIFE';
  details: {
    summary: string;
    changes: string[];
    migrationGuide?: string;
    timeline: {
      announcementDate: Date;
      effectiveDate: Date;
      endOfSupportDate?: Date;
```

```
    };
  };
  distributionChannels: string[];
}
```

Deprecation Strategy

1. Deprecation Timeline

```
interface DeprecationSchedule {
  version: string;
  announceDate: Date;
  deprecationDate: Date;
  endOfLifeDate: Date;
  alternativeVersion: string;
  migrationDeadline: Date;
}

const DEPRECATION_POLICY = {
  minimumNoticePeroid: 180, // days
  supportWindow: 365, // days
  maxVersionsSupported: 2
};
```

2. Deprecation Notices

Implementation of standardized deprecation warnings.

```
app.use((req, res, next) => {
  const version = req.apiVersion;

  if (isDeprecated(version)) {
    res.setHeader('Deprecation', 'true');
    res.setHeader('Sunset', getEndOfLifeDate(version));
    res.setHeader('Link', `<${getUpgradeGuideUrl(version)}>; rel="deprecation"`);
  }

  next();
});
```

Monitoring and Analytics

Track API version usage to inform deprecation decisions.

```
interface VersionMetrics {
  version: string;
  requests: number;
  uniqueClients: number;
  errorRate: number;
  avgResponseTime: number;
  deprecationStatus: 'active' | 'deprecated' | 'sunset';
}

const trackVersionUsage = async (req: Request, res: Response, next: NextFunction) => {
  const startTime = Date.now();
  const version = req.apiVersion;

  res.on('finish', () => {
    const duration = Date.now() - startTime;
    metrics.recordVersionUsage({
      version,
      duration,
      status: res.statusCode,
      clientId: req.auth?.clientId
    });
  });

  next();
};
```

Best Practices

1. **Version Selection**
   - Use semantic versioning consistently
   - Make breaking changes only in major versions
   - Maintain at least one previous major version

2. **Change Management**
   - Plan changes well in advance
   - Provide detailed migration guides
   - Use feature flags for gradual rollouts

3. **Communication**
   - Announce changes early
   - Provide clear upgrade paths

- Maintain comprehensive documentation

4. **Support**
   - Offer migration tools
   - Provide support during transitions
   - Monitor version usage patterns

5. **Maintenance**
   - Regular deprecation reviews
   - Clean up deprecated features
   - Archive old documentation

Implementation Checklist

- [ ] Define versioning strategy
- [ ] Set up version routing
- [ ] Implement version manager
- [ ] Create deprecation policy
- [ ] Configure monitoring
- [ ] Prepare communication templates
- [ ] Document upgrade paths
- [ ] Set up automated testing

API Endpoints

Overview

This documentation covers the REST API endpoints required to implement vault creation, asset management, and governance functionality for the L4VA protocol. The API enables:

1. **Vault Formation**
   - Creation of Private, Public, and Semi-Private vaults
   - Asset contribution during timed windows
   - Investment handling with Fixed/LBE options
2. **Governance**
   - Proposal creation and management
   - Voting mechanisms
   - Multi-signature execution flows
   - Threshold validations
3. **Wallet Integration**
   - Wallet connection and session management
   - Vault association and role management

Getting Started

1. Obtain API credentials
2. Review authentication requirements
3. Test endpoints in development environment
4. Implement error handling
5. Add real-time updates using WebSocket endpoints

Environment URLs

- Development: https://api-dev.l4va.example.com
- Staging: https://api-staging.l4va.example.com
- Production: https://api.l4va.example.com

Authentication

All endpoints require API key authentication using the `X-API-Key` header.

Response Format

All endpoints follow a consistent response format:

```
{
  "status": "success|error",
  "data": {}, // Response payload
  "error": {} // Present only on errors
}
```

Blockchain Integration

- All state-changing operations return transaction hashes
- Wallet signatures required for sensitive operations
- Real-time updates available via WebSocket connections

Postman Collection

TBD: Provide a Postman Collection

API Endpoints

Vault Formation

Create Vault

Creates a new vault with specified configuration.

Endpoint

POST /api/v1/vaults

Sample Request


```
{
 "vault_name": "Example Vault",
 "vault_type": "public",
 "privacy_type": "semi-private",
 "admin_user": {
  "wallet_address": "addr1q9example123",
  "email": "admin@example.com"
 },
 "asset_settings": {
  "allowed_asset_types": ["NFT", "CNT"],
  "policy_ids": ["policy12345", "policy67890"],
  "valuation_type": "LBE",
  "floor_price_percentage": 90,
  "max_assets": 100
 },
 "investment_settings": {
  "investment_window_duration": "48h",  // ISO-like duration
  "investment_start_time": "2024-11-24T10:00:00Z",  // Optional; defaults to vault creation time
  "minimum_investment_reserve": 10.0,
  "ft_supply": 100000,
  "ft_token_decimals": 6,
  "lp_percentage": 10
 },
 "governance_settings": {
  "creation_threshold": 5,
  "start_threshold": 10,
  "vote_threshold": 50,
```

```
    "execution_threshold": 60,
    "cosigning_threshold": 3
  }
}
```


Sample Response (201: Created)


```
{
  "vault_id": "vault123",
  "vault_name": "Example Vault",
  "transaction": {
    "tx_hash": "b26f8c9a0d1a4a9b8b12f9f9a8c1234567e9d0f1c234567890abcdef",
    "status": "confirmed",
    "block_height": 1234567
  },
  "investment_window": {
    "duration": "48h",
    "start_time": "2024-11-24T10:00:00Z",
    "end_time": "2024-11-26T10:00:00Z"
  },
  "status": "created",
  "created_at": "2024-11-23T10:00:00Z"
}
```


Add Assets to Vault

Add assets during the asset window period.

HTTP Request

```
POST /api/v1/vaults/{vaultId}/assets
```

Sample Request


```
{
  "policy_id": "policy12345",
  "asset_name": "ExampleAsset",
  "valuation_method": "floor_price",
  "metadata": {
    "creator": "CreatorAddress",
    "legal_proof": "https://proof.example.com/doc.pdf"
```

```
  }
}
```


Sample Response (201: Created)


```
{
  "vault_id": "vault123",
  "asset_id": "asset456",
  "transaction": {
    "tx_hash": "a92f81e3b69c4d12b34c567890fabcde1234567890abcdef56789abc",
    "status": "confirmed",
    "block_height": 1234570
  },
  "status": "added",
  "valuation": {
    "method": "floor_price",
    "value": 1000000
  },
  "created_at": "2024-11-23T11:00:00Z"
}
```


Remove Asset

Remove assets during the asset window period. Will fail if attempted before or after the window period.

Endpoint

```
DELETE /api/v1/vaults/{vaultId}/assets/{assetId}
```

Sample Response


```
{
  "status": "success",
  "data": {
    "assetId": "asset_123",
    "removalStatus": "COMPLETED",
    "transactionHash": "tx_hash789...",
    "updatedValuation": {
      "total": "140000",
      "timestamp": "2024-12-01T01:35:00Z"
    }
```

```
  }
}
```

Get Vault Valuation

Calculate current vault valuation based on assets.

HTTP Request

```
GET /api/v1/vaults/{vaultId}/valuation
```

Sample Response


```
{
  "status": "success",
  "data": {
    "valuation": {
      "total": "150000",
      "breakdown": {
        "nftValue": "120000",
        "cntValue": "30000"
      },
      "assetCounts": {
        "nfts": 2,
        "cnts": 1
      },
      "timestamp": "2024-12-01T01:30:00Z"
    }
  }
}
```


List Vault Assets

List all assets in the vault.

HTTP Request

```
GET /api/v1/vaults/{vaultId}/assets
```

Sample Response

```
{
  "status": "success",
  "data": {
    "assets": [{
      "assetId": "asset_123",
      "type": "NFT",
      "contractAddress": "addr_nft123...",
      "tokenId": "42",
      "addedAt": "2024-12-01T01:00:00Z",
      "status": "LOCKED",
      "currentValue": "90000"
    }],
    "pagination": {
      "page": 1,
      "limit": 20,
      "total": 45
    }
  }
}
```


Update Vault AllowList

Updates the list of wallets allowed to participate in the vault, and their type.

HTTP Request

```
PATCH /api/v1/vaults/{vaultId}/allowlist
```

Sample Request

```
{
  "type": "ASSET|CONTRIBUTOR|INVESTOR",
  "operation": "ADD|REMOVE",
  "addresses": ["addr1...", "addr2..."]
}
```

Sample Response


```
{
  "status": "success",
  "data": {
```

```
    "updatedAllowList": {
      "type": "ASSET",
      "count": 24,
      "lastUpdated": "2024-12-01T02:00:00Z"
    }
  }
}
```

Vaut Performance Metrics

Returns some vault performance metrics.

HTTP Request

```
GET /api/v1/vaults/{vaultId}/metrics
```

Sample Response

```
{
  "status": "success",
  "data": {
    "valuation": {
      "initial": "100000",
      "current": "150000",
      "change": "50.00"
    },
    "participation": {
      "uniqueVoters": 45,
      "averageQuorum": "68.00",
      "proposalCount": 12
    },
    "timeline": {
      "created": "2024-11-20T10:00:00Z",
      "locked": "2024-12-01T02:00:00Z",
      "age": "11d 16h"
    }
  }
}
```

Vault Activity

Returns a vault activity log.

HTTP Request

```
GET /api/v1/vaults/{vaultId}/activity
```

Sample Response

```
{
  "status": "success",
  "data": {
    "activities": [{
      "type": "ASSET_ADDED|PROPOSAL_CREATED|VOTE_CAST",
      "timestamp": "2024-12-01T01:00:00Z",
      "actor": "addr_user123...",
      "details": {},
      "transactionHash": "tx_hash123..."
    }],
    "pagination": {
      "page": 1,
      "limit": 20,
      "total": 156
    }
  }
}
```

Update Vault Settings

Lets you update modifiable vault settings.

Sample Request

```
{
  "investorAllowList": {
    "enabled": true,
    "addresses": ["addr1..."]
  },
  "valuationType": "LBE",
  "termination": {
    "fdp": "12.00"
  }
}
```

This would return the standard Success/Fail response object.

API Endpoints

Governance

Proposal Management

Create a new governance proposal.

HTTP Request

```
POST /api/v1/vaults/{vaultId}/proposals
```

Sample Request

```
{
  "proposer": "user_wallet123",
  "proposal_details": {
    "title": "Liquidate Asset A",
    "description": "Sell asset A for 90% of its floor price.",
    "action_type": "sell_asset",
    "affected_assets": [
      {
        "policy_id": "policy12345",
        "asset_name": "ExampleAsset"
      }
    ]
  },
  "governance_thresholds": {
    "creation_threshold": 5,
    "start_threshold": 10,
    "vote_threshold": 50,
    "execution_threshold": 60
  }
}
```

Sample Response (201: Created)

```
{
  "vault_id": "vault123",
  "proposal_id": "proposal789",
  "transaction": {
```

```
    "tx_hash": "de1234abc567890def1234567890abcdef1234567890abcdef12345678",
    "status": "confirmed",
    "block_height": 1234575
  },
  "status": "created",
  "thresholds_met": {
    "creation_threshold": true
  },
  "created_at": "2024-11-23T12:00:00Z"
}
```


Submit Vote

Submit a vote on a proposal.

HTTP Request

```
POST /api/v1/proposals/{proposalId}/votes
```

Sample Request


```
{
  "voter": {
    "wallet_address": "addr1q9voter123",
    "staked_ft_amount": 1000
  },
  "vote": "yes"
}
```


Sample Response (200: OK)


```
{
  "vault_id": "vault123",
  "proposal_id": "proposal789",
  "voter": "addr1q9voter123",
  "transaction": {
    "tx_hash": "ab567890abcdef1234567890abcdef1234567890abcdef1234567890",
    "status": "confirmed",
    "block_height": 1234580
  },
  "vote": "yes",
```

```
  "staked_ft_amount": 1000,
  "status": "vote_recorded",
  "updated_at": "2024-11-23T13:00:00Z"
}
```


Execute Proposal

Execute an approved proposal.

Endpoint

```
POST /api/v1/proposals/{proposalId}/execute
```

Sample Request


```
{
  "executing_user": "user_wallet123",
  "cosigners": [
    "wallet_cosigner1",
    "wallet_cosigner2"
  ],
  "action": {
    "type": "sell_asset",
    "details": {
      "policy_id": "policy12345",
      "asset_name": "ExampleAsset",
      "price": 900000
    }
  }
}
```


Sample Response (200: OK)


```
{
  "vault_id": "vault123",
  "proposal_id": "proposal789",
  "transaction": {
    "tx_hash": "cd1234567890abcdef1234567890abcdef1234567890abcdef123456",
    "status": "confirmed",
    "block_height": 1234585
  },
```

```
  "status": "executed",
  "executed_by": "user_wallet123",
  "cosigners": [
    "wallet_cosigner1",
    "wallet_cosigner2"
  ],
  "action": {
    "type": "sell_asset",
    "details": {
      "policy_id": "policy12345",
      "asset_name": "ExampleAsset",
      "price": 900000
    }
  },
  "executed_at": "2024-11-23T14:00:00Z"
}
```

Wallet Integration

About CIP-08 and CIP-30

This guide is a walkthrough on how to implement the *message signing* described in CIP-08 in order to authenticate users on the web with just their CIP-30-compatible wallet app:

https://developers.cardano.org/docs/integrate-cardano/user-wallet-authentication/

However while it is useful to study the above, in order to simplify and standardize our platform, we're going to use a multi-chain wrapper library instead (see next section).

Using Weld

We're going to use this library to integrate with Cardano wallets:

https://github.com/Cardano-Forge/weld

Weld lets you manage wallet connections across multiple blockchains using a single intuitive interface.

How Authentication works

In a Web3 app using wallets like **Nami** or **Vespr**, authentication typically works through **wallet-based signature verification**. Here's a simplified flow:

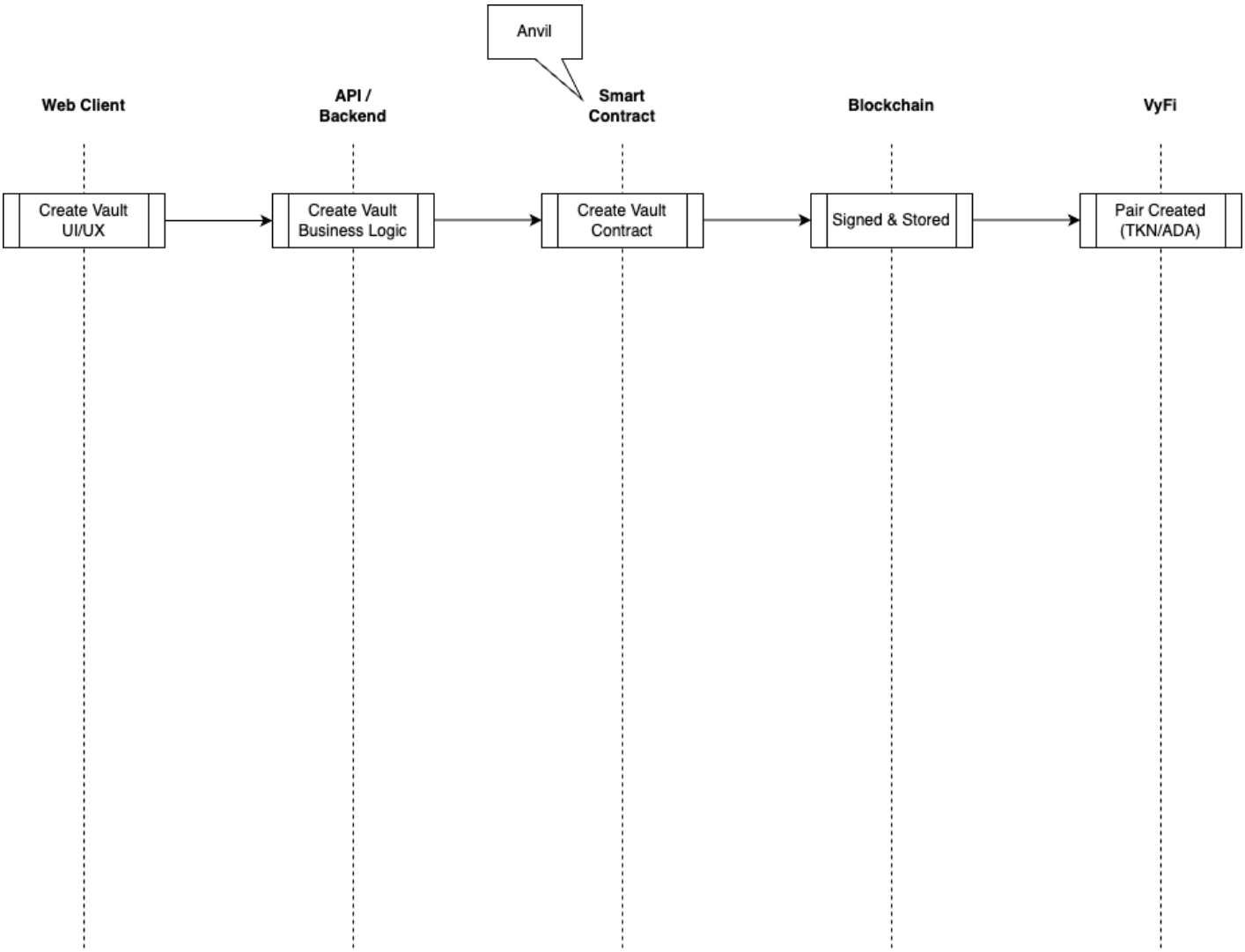
1. **Connect Wallet**: The user connects their wallet to the Web3 app. The wallet extension (like Nami or Vespr) interfaces with the app to allow interactions.
2. **Generate Nonce**: The app generates a unique, random string (nonce) and sends it to the wallet for the user to sign. This ensures that each authentication request is unique and prevents replay attacks.
3. **Sign Nonce**: The user signs the nonce using their private key in the wallet. This signature doesn't expose the private key but proves ownership of the wallet.
4. **Verify Signature**: The app receives the signed nonce and verifies it using the user's public key (derived from their wallet address). If the signature is valid, it confirms the user controls the wallet.
5. **Authenticate User**: Once verified, the app logs in the user and associates their wallet address with their session or profile. The wallet address often serves as the unique user identifier.
6. **Session Management**: The app can use cookies, tokens (like JWTs), or smart contract events to manage sessions while interacting with the blockchain.

This method ensures secure, decentralized authentication without traditional usernames or passwords. Wallets like Nami and Vespr streamline this process by providing user-friendly interfaces for signing and verifying data.

We'll be testing with both those wallets, and the screenshots you're going to see in the documentation will be from one of those two wallets (and especially on mobile phones, with Vespr).

Swim Lane Diagram

This diagram explains how this API backend interacts with the blockchain and the L4VA Smart Contracts



Performance Optimization

Overview

This guide outlines strategies and implementations for optimizing the L4VA API's performance across multiple layers: API, Database, Caching, and Blockchain interactions.

API Layer Optimization

1. Request-Response Optimization

```
interface ResponseOptimization {
  // Field selection
  fields?: string[];          // Selected fields to return
  expand?: string[];          // Related data to include
  version?: string;           // Response format version
}

// Implementation
const optimizeResponse = (data: any, options: ResponseOptimization) => {
  const optimized = options.fields
    ? pickFields(data, options.fields)
    : data;

  if (options.expand) {
    await expandRelations(optimized, options.expand);
  }

  return optimized;
};

// Usage example
app.get('/api/v1/vaults/:id', async (req, res) => {
  const vault = await VaultService.findById(req.params.id);
  const optimized = await optimizeResponse(vault, {
    fields: ['id', 'status', 'assets'],
    expand: ['activeProposals']
  });
  res.json(optimized);
});
```

2. Request Batching

```
interface BatchRequest {
  id: string;
  method: string;
  path: string;
  body?: any;
}

const batchHandler = async (requests: BatchRequest[]) => {
  return Promise.all(requests.map(async (request) => {
    try {
      const result = await router.handle(request);
      return {
        id: request.id,
        status: 'success',
        data: result
      };
    } catch (error) {
      return {
        id: request.id,
        status: 'error',
        error: error.message
      };
    }
  }));
};
```

3. Rate Limiting with Redis

```
class RateLimiter {
  private redis: Redis;

  async checkLimit(key: string, limit: number, window: number): Promise<boolean> {
    const multi = this.redis.multi();
    const now = Date.now();

    multi.zremrangebyscore(key, 0, now - window);
    multi.zadd(key, now, `${now}`);
    multi.zcard(key);
```

```
    const [,, count] = await multi.exec();
    return count < limit;
  }
}
```

Database Optimization

1. Query Optimization

```
// Optimized query builder
class QueryBuilder {
  private query: any = {};
  private options: QueryOptions = {};

  // Index-aware filtering
  addFilter(field: string, value: any) {
    if (this.hasIndex(field)) {
      this.query[field] = value;
    } else {
      this.options.postProcess = true;
    }
  }

  // Efficient pagination
  setPagination(page: number, limit: number) {
    this.options.skip = (page - 1) * limit;
    this.options.limit = limit;
    this.options.sort = { _id: 1 }; // Index-based sorting
  }

  // Selective field projection
  selectFields(fields: string[]) {
    this.options.projection = fields.reduce((acc, field) => {
      acc[field] = 1;
      return acc;
    }, {});
  }
}
```

2. Aggregation Pipeline Optimization

```
const optimizedAggregation = [
  // Early filtering
  {
    $match: {
      status: 'ACTIVE',
      'assetWindow.endTime': { $gt: new Date() }
    }
  },

  // Limit fields early
  {
    $project: {
      id: 1,
      status: 1,
      assets: 1
    }
  },

  // Use index for sorting
  {
    $sort: {
      'assetWindow.endTime': 1
    }
  },

  // Paginate results
  {
    $skip: skip
  },
  {
    $limit: limit
  }
];
```

3. Indexing Strategy

```
interface IndexStrategy {
  // Compound indexes for common queries
```

```
  compoundIndexes: {
    vault_status_type: { status: 1, type: 1 },
    proposal_vault_status: { vaultId: 1, status: 1 },
    asset_contract_token: { contractAddress: 1, tokenId: 1 }
  };

  // Text indexes for search
  textIndexes: {
    vault_search: { name: 'text', description: 'text' }
  };

  // Partial indexes for active records
  partialIndexes: {
    active_vaults: {
      index: { status: 1 },
      filter: { status: 'ACTIVE' }
    }
  };
}
```

Caching Layer

1. Multi-Level Caching

```
class CacheManager {
  private memoryCache: Map<string, any>;
  private redis: Redis;

  async get(key: string, fetchFn: () => Promise<any>) {
    // Check memory cache
    if (this.memoryCache.has(key)) {
      return this.memoryCache.get(key);
    }

    // Check Redis cache
    const redisValue = await this.redis.get(key);
    if (redisValue) {
      this.memoryCache.set(key, redisValue);
      return redisValue;
    }
```

```
    // Fetch and cache
    const value = await fetchFn();
    await this.set(key, value);
    return value;
  }

  async set(key: string, value: any) {
    this.memoryCache.set(key, value);
    await this.redis.set(key, value);
  }
}
```

2. Cache Invalidation Strategy

```
interface CacheInvalidation {
  patterns: {
    vault: 'vault:*',
    proposal: 'proposal:*',
    asset: 'asset:*'
  };

  dependencies: {
    vault: ['asset', 'proposal'],
    proposal: ['vote'],
    asset: ['valuation']
  };
}

class CacheInvalidator {
  async invalidate(type: string, id: string) {
    const pattern = this.patterns[type];
    const keys = await this.redis.keys(pattern);

    // Invalidate direct cache
    await this.redis.del(keys);

    // Invalidate dependencies
    for (const depType of this.dependencies[type]) {
      await this.invalidate(depType, id);
```

```
    }
  }
}
```

Blockchain Optimization

1. Transaction Batching

```
class TransactionBatcher {
  private queue: Transaction[] = [];
  private batchSize: number = 10;
  private batchTimeout: number = 5000;

  async addTransaction(tx: Transaction) {
    this.queue.push(tx);

    if (this.queue.length >= this.batchSize) {
      await this.processBatch();
    }
  }

  private async processBatch() {
    const batch = this.queue.splice(0, this.batchSize);
    const multicall = await this.createMulticall(batch);
    return await this.sendTransaction(multicall);
  }
}
```

2. Event Processing Optimization

```
class EventProcessor {
  private lastProcessedBlock: number;
  private batchSize: number = 1000;

  async processEvents(startBlock: number, endBlock: number) {
    for (let block = startBlock; block <= endBlock; block += this.batchSize) {
      const events = await this.fetchEvents(block, block + this.batchSize);
      await this.processEventBatch(events);
    }
  }
```

```
  private async processEventBatch(events: Event[]) {
    // Group events by type
    const grouped = groupBy(events, 'eventType');

    // Process each type in parallel
    await Promise.all(
      Object.entries(grouped).map(([type, events]) =>
        this.processEventType(type, events)
      )
    );
  }
}
```

Load Testing and Monitoring

1. Load Testing Configuration

```
interface LoadTest {
  scenarios: {
    name: string;
    weight: number;
    flow: RequestFlow[];
  }[];

  thresholds: {
    http_req_duration: ['p(95)<500'],
    http_reqs: ['rate>100'],
    errors: ['rate<0.1']
  };

  stages: {
    duration: string;
    target: number;
  }[];
}
```

2. Performance Monitoring

```
interface PerformanceMetrics {
  // API metrics
  api: {
    responseTime: Histogram;
    requestRate: Counter;
    errorRate: Counter;
  };

  // Database metrics
  db: {
    queryTime: Histogram;
    connectionPool: Gauge;
    activeQueries: Gauge;
  };

  // Cache metrics
  cache: {
    hitRate: Gauge;
    missRate: Gauge;
    evictionRate: Counter;
  };

  // Blockchain metrics
  blockchain: {
    transactionTime: Histogram;
    gasUsage: Histogram;
    nodeLatency: Gauge;
  };
}
```

Best Practices

1. **API Optimization**
   - Implement field selection
   - Use request batching
   - Enable compression
   - Optimize payload size

2. **Database Optimization**
   - Create efficient indexes
   - Optimize query patterns

- Use connection pooling
- Implement sharding strategy

3. **Caching Strategy**
   - Implement multi-level caching
   - Use appropriate TTLs
   - Handle cache invalidation
   - Monitor cache hit rates

4. **Blockchain Optimization**
   - Batch transactions
   - Implement retry strategies
   - Optimize gas usage
   - Cache blockchain data

5. **Monitoring and Alerting**
   - Track key metrics
   - Set up alerts
   - Monitor resource usage
   - Analyze performance trends

Implementation Checklist

- ☐ Configure response optimization
- ☐ Implement query optimization
- ☐ Set up caching layer
- ☐ Configure blockchain batching
- ☐ Implement monitoring
- ☐ Set up load testing
- ☐ Document optimization strategies
- ☐ Train team on best practices