Database

Complete database schema documentation for the UserState API backend system.

Table of Contents

1. Database Architecture
2. PostgreSQL Database
3. MongoDB Database
4. Entity Relationships
5. Database Setup
6. Migrations
7. Common Queries
8. Backup and Maintenance
9. Performance Optimization

Database Architecture

Overview

The UserState API uses a **dual-database architecture** to optimize different data access patterns:

- **PostgreSQL** (via Sequelize ORM) - Primary database for transactional data
- **MongoDB** (via Mongoose) - Document store for log data

Why Dual Database?

Database	Use Case	Reason
PostgreSQL	User data, transactions, balances, addresses	ACID compliance, relational integrity, complex queries

Database	Use Case	Reason
MongoDB	Payment logs (Xendit)	Flexible schema, high write throughput, log aggregation

Connection Configuration

PostgreSQL Connection:

```
// config/db.js
exports.userstate = {
    HOST: process.env.PS_HOST,
    USER: process.env.PS_USER,
    PASSWORD: process.env.PS_PASSWORD,
    DB: process.env.PS_DB,
    DIALECT: process.env.PS_DIALECT,
    PORT: process.env.PS_PORT,
    POOL: {
        max: parseInt(process.env.PS_POOL_MAX)
    }
}
```

MongoDB Connection:

```
// config/db.js
exports.db_mongo = {
    url: process.env.MONGO_DB
};
```

PostgreSQL Database

The PostgreSQL database contains **16 tables** managing core application data.

Table Structure Overview

```
PostgreSQL Tables
├── Authentication & User Management
```

```
│   ├── user_login          (Primary authentication)
│   ├── user_info           (User profile details)
│   └── user_balance        (Wallet and points)
│
├── Address Management
│   └── address_list        (User shipping addresses)
│
├── Financial Operations
│   ├── user_transaction    (Transaction history)
│   ├── point_mutation      (Point transactions)
│   ├── voucher_redemption  (Voucher usage tracking)
│   ├── bank_account        (User bank details)
│   ├── bank_channel        (Available payment channels)
│   └── payment_type        (Payment method types)
│
├── Referral System
│   └── referral_history    (Referral tracking)
│
├── Communication
│   ├── fcm                 (Push notification tokens)
│   └── marketing_offer     (Promotional messages)
│
├── System Management
│   ├── api_key             (Service authentication)
│   ├── otp_tracker         (OTP management)
│   ├── counter             (ID generation counter)
│   └── deactivate_reason   (Account deactivation logs)
```

PostgreSQL Tables - Detailed Schema

1. user_login

Purpose: Stores user authentication credentials and account status.

Table Name: `user_login`

Column	Type	Constraints	Description
username	STRING	PRIMARY KEY	Unique username identifier
pwd	STRING	NOT NULL	Hashed password (bcrypt)
disabled	BOOLEAN	NOT NULL, DEFAULT false	Account disabled flag
token	STRING	NULL	Temporary token storage
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `username`

Model File: `models/user_login.model.js`

Sample Data:

```
INSERT INTO user_login (username, pwd, disabled)
VALUES ('john_doe', '$2b$10$...hashedpassword...', false);
```

---

2. user_info

Purpose: Stores detailed user profile information.

Table Name: `user_info`

Column	Type	Constraints	Description
id	INTEGER	AUTO INCREMENT	Auto-generated ID
username	STRING	PRIMARY KEY, FOREIGN KEY	Reference to user_login
email	STRING	UNIQUE, NOT NULL	User email address
first_name	STRING	NOT NULL	User's first name
last_name	STRING	NOT NULL	User's last name
profile_img	STRING	NULL	Path to profile image
gender	STRING	NULL	User gender
date_of_birth	DATE	NULL	User's date of birth
phone_number	STRING	NULL	Contact phone number

Column	Type	Constraints	Description
agent	STRING	NULL	Agent flag/identifier
agent_id	STRING	NULL	Agent ID if applicable
referral_code	STRING	UNIQUE	User's unique referral code
referral_applied_status	STRING	NULL	Referral application status
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `username`
- Unique: `email`, `referral_code`

Foreign Keys:

- `username` → `user_login.username`

Model File: `models/user_info.model.js`

Sample Data:

```
INSERT INTO user_info (username, email, first_name, last_name, referral_code)
VALUES ('john_doe', 'john@example.com', 'John', 'Doe', 'JOHN123');
```

---

3. user_balance

Purpose: Tracks user wallet balance and points.

Table Name: `user_balance`

Column	Type	Constraints	Description
username	STRING	PRIMARY KEY, FOREIGN KEY	Reference to user_login
balance	INTEGER	NOT NULL	Current balance in smallest currency unit
onapps_point	STRING	NULL	Application points
in_transaction	INTEGER	NOT NULL	Amount currently in pending transactions
updated_at	TIMESTAMP	AUTO	Last update timestamp

Note: This table has `createdAt: false` configuration - no creation timestamp.

Indexes:

- Primary Key: `username`

Foreign Keys:

- `username` → `user_login.username`

Model File: `models/user_balance.model.js`

Sample Data:

```
INSERT INTO user_balance (username, balance, onapps_point, in_transaction)
VALUES ('john_doe', 100000, '50', 0);
```

Balance Rules:

- Balance stored in smallest unit (e.g., cents for USD, smallest currency unit)
- `in_transaction` represents locked funds during pending transactions
- Available balance = `balance - in_transaction`

---

4. address_list

Purpose: Stores multiple shipping addresses per user.

Table Name: `address_list`

Column	Type	Constraints	Description
address_id	BIGINT	PRIMARY KEY, AUTO INCREMENT	Unique address identifier
recipient	STRING	NOT NULL	Recipient name
phone	STRING	NOT NULL	Contact phone number
category	STRING	NULL	Address category/label
address_input	STRING	NOT NULL	Raw address input
province	STRING	NOT NULL	Province/state
district	STRING	NOT NULL	District
city	STRING	NOT NULL	City
urban	STRING	NOT NULL	Urban/sub-district

Column	Type	Constraints	Description
post_id	STRING	NOT NULL	Postal code
osas_log_id	INTEGER	NOT NULL	OSAS system log reference
username	STRING	NOT NULL, FOREIGN KEY	Reference to user_login
primary_address	BOOLEAN	NOT NULL	Is primary address flag
address	JSONB	NOT NULL	Structured address data (JSON)
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `address_id`
- Index on: `username`

Foreign Keys:

- `username` → `user_login.username`

Model File: `models/address_list.model.js`

Sample Address JSON:

```
{
  "formatted": "Jl. Example No. 123, Jakarta Pusat, DKI Jakarta 10110",
  "coordinates": {
    "lat": -6.200000,
    "lng": 106.816666
  }
}
```

---

5. user_transaction

Purpose: Records all payment transactions.

Table Name: `user_transaction`

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO INCREMENT	Unique transaction ID

Column	Type	Constraints	Description
username	STRING	NOT NULL, FOREIGN KEY	Reference to user_login
type	STRING	NOT NULL	Transaction type (TOPUP, PAYMENT, etc.)
amount	INTEGER	NOT NULL	Transaction amount
xendit_invoice	STRING	NULL	Xendit invoice ID
invoice	STRING	NOT NULL	Internal invoice number
status	STRING	NOT NULL	Transaction status
src	STRING	NULL	Transaction source
xendit_url	STRING	NULL	Xendit payment URL
invoice_onmarket	STRING	NULL	OnMarket invoice reference
expired_at	DATE	NULL	Payment expiration date
freight_charge	INTEGER	NULL	Shipping cost
insurance_amount	INTEGER	NULL	Insurance amount
onbooking_code	STRING	NULL	Booking code
amount_2	STRING	NULL	Secondary amount field
commission_onmarket	INTEGER	NULL	OnMarket commission
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `id`
- Index on: `username`, `invoice`, `xendit_invoice`

Foreign Keys:

- `username` → `user_login.username`

Transaction Status Values:

- `PENDING` - Awaiting payment
- `PAID` - Payment completed
- `EXPIRED` - Payment deadline passed
- `CANCELLED` - Transaction cancelled
- `FAILED` - Payment failed

Model File: `models/user_transaction.model.js`

---

6. point_mutation

Purpose: Tracks point earning and redemption transactions.

Table Name: `point_mutation`

Column	Type	Constraints	Description
id	INTEGER	PRIMARY KEY, AUTO INCREMENT	Unique mutation ID
username	STRING	NOT NULL, FOREIGN KEY	Reference to user_login
type	ENUM	NOT NULL	'EARN' or 'REDEEM'
amount	NUMERIC	NOT NULL	Point amount
reference_number	STRING	NOT NULL	Transaction reference
src	STRING	NOT NULL	Point source/reason
status	ENUM	NOT NULL	'PENDING' or 'COMPLETE'
account_name	STRING	NULL	Associated account name
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `id`
- Index on: `username`, `reference_number`

Foreign Keys:

- `username` → `user_login.username`

Point Sources (src field):

- `REFERRAL` - From referral bonus
- `SIGNUP_BONUS` - New user bonus
- `TRANSACTION` - From completed transactions
- `PROMO` - Promotional points
- `MANUAL` - Manual adjustment

Model File: `models/point_mutation.model.js`

---

7. referral_history

Purpose: Tracks referral relationships between users.

Table Name: `referral_history`

Column	Type	Constraints	Description
id	INTEGER	PRIMARY KEY, AUTO INCREMENT	Unique referral record ID
username	STRING	NOT NULL, FOREIGN KEY	Referred user (new user)
referral_code	STRING	NOT NULL	Referral code used
submitted_by	STRING	NOT NULL, FOREIGN KEY	Referrer username
status	STRING	NOT NULL	Referral status
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `id`
- Index on: `username`, `submitted_by`

Foreign Keys:

- `username` → `user_info.username`
- `submitted_by` → `user_info.username`

Status Values:

- `PENDING` - Referral registered but not qualified
- `COMPLETED` - Referral qualified, bonus awarded
- `EXPIRED` - Referral expired without qualifying

Model File: `models/referral_history.model.js`

---

8. voucher_redemption

Purpose: Tracks voucher code usage.

Table Name: `voucher_redemption`

Column	Type	Constraints	Description
id	INTEGER	PRIMARY KEY, AUTO INCREMENT	Unique redemption ID

Column	Type	Constraints	Description
username	STRING	NOT NULL, FOREIGN KEY	Reference to user_login
code	STRING	NOT NULL	Voucher code redeemed
status	STRING	NOT NULL	Redemption status
is_onapps	BOOLEAN	NULL	Is OnApps voucher
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `id`
- Index on: `username`, `code`

Foreign Keys:

- `username` → `user_login.username`

Model File: `models/voucher_redemption.model.js`

---

9. fcm (Firebase Cloud Messaging)

Purpose: Stores FCM device tokens for push notifications.

Table Name: `fcm`

Column	Type	Constraints	Description
fcm_id	STRING	PRIMARY KEY	FCM device token
enabled	BOOLEAN	NOT NULL, DEFAULT true	Token enabled/disabled
username	STRING	NULL, FOREIGN KEY	Reference to user_login
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `fcm_id`
- Index on: `username`

Foreign Keys:

- `username` → `user_login.username`

Model File: `models/fcm.model.js`

Notes:

- One user can have multiple FCM tokens (multiple devices)
- Tokens are automatically refreshed by Firebase SDK
- Disabled tokens are kept for analytics but not used for sending

---

10. marketing_offer

Purpose: Stores marketing messages/offers sent to users.

Table Name: `marketing_offer`

Column	Type	Constraints	Description
msg_id	INTEGER	PRIMARY KEY, AUTO INCREMENT	Unique message ID
username	STRING	NOT NULL, FOREIGN KEY	Reference to user_login
name	STRING	NOT NULL	Sender/campaign name
telp	STRING	NOT NULL	Contact telephone
message	STRING	NOT NULL	Message content
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `msg_id`
- Index on: `username`

Foreign Keys:

- `username` → `user_login.username`

Model File: `models/marketing_offer.model.js`

---

11. api_key

Purpose: Manages API keys for service-to-service authentication.

Table Name: `api_key`

Column	Type	Constraints	Description
key	STRING	PRIMARY KEY	Hashed API key
holder	STRING	NOT NULL	Service/holder identifier
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `key`

Model File: `models/api_key.model.js`

Security Note:

- Keys are hashed using bcrypt before storage
- Used with HTTP Basic Authentication for internal services
- Format: `Authorization: Basic base64(holder:apikey)`

---

12. otp_tracker

Purpose: Tracks OTP generation and usage.

Table Name: `otp_tracker`

Column	Type	Constraints	Description
id	INTEGER	PRIMARY KEY, AUTO INCREMENT	Unique OTP record ID
username	STRING	NOT NULL	Username/identifier
otp	STRING	NOT NULL	OTP code (hashed or encrypted)
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `id`
- Index on: `username`

Model File: `models/otp_tracker.model.js`

OTP Rules:

- OTP length: 6 digits
- Validity: 5 minutes (configurable)
- Max attempts: 5 (configurable)

---

13. bank_account

Purpose: Stores user bank account information for payouts.

Table Name: `bank_account`

Column	Type	Constraints	Description
id	INTEGER	PRIMARY KEY	Bank account ID
username	STRING	NOT NULL, FOREIGN KEY	Reference to user_login
channel	INTEGER	NOT NULL	Bank channel ID reference

Note: No timestamps configured for this table.

Indexes:

- Primary Key: `id`
- Index on: `username`

Foreign Keys:

- `username` → `user_login.username`
- `channel` → `bank_channel.id`

Model File: `models/bank_account.model.js`

---

14. bank_channel

Purpose: Defines available banking channels for payments/payouts.

Table Name: `bank_channel`

Column	Type	Constraints	Description
id	INTEGER	PRIMARY KEY	Unique channel ID
code	STRING	NOT NULL	Bank code (e.g., 'BCA', 'MANDIRI')
type	STRING	NOT NULL	Channel type

Column	Type	Constraints	Description
currency	STRING	NOT NULL	Supported currency
description	TEXT	NULL	Channel description
alias	TEXT	NULL	Alternative names/aliases

Note: No timestamps configured for this table.

Indexes:

- Primary Key: `id`
- Index on: `code`

Model File: `models/bank_channel.model.js`

Common Bank Codes:

- `BCA` - Bank Central Asia
- `MANDIRI` - Bank Mandiri
- `BNI` - Bank Negara Indonesia
- `BRI` - Bank Rakyat Indonesia

---

15. payment_type

Purpose: Defines available payment types/methods.

Table Name: `payment_types`

Column	Type	Constraints	Description
id	INTEGER	PRIMARY KEY, AUTO INCREMENT	Unique payment type ID
name	STRING	NOT NULL	Payment type name
status	BOOLEAN	NOT NULL	Active/inactive status
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `id`

Model File: `models/payment_type.model.js`

Payment Types:

- `CREDIT_CARD`
- `BANK_TRANSFER`
- `E_WALLET`
- `RETAIL_OUTLET`
- `QR_CODE`

---

16. counter

Purpose: Generates sequential IDs for various entity types.

Table Name: `counter`

Column	Type	Constraints	Description
type	STRING	PRIMARY KEY	Counter type identifier
counter	INTEGER	NOT NULL	Current counter value
month	INTEGER	NOT NULL	Month for counter reset

Note: No timestamps configured for this table.

Indexes:

- Primary Key: `type`

Model File: `models/counter.model.js`

Counter Types:

- `INVOICE` - Invoice number generation
- `TRANSACTION` - Transaction ID generation
- `REFERRAL` - Referral code generation

---

17. deactivate_reason

Purpose: Logs reasons when users deactivate their accounts.

Table Name: `deactivate_reason`

Column	Type	Constraints	Description

id	INTEGER	PRIMARY KEY, AUTO INCREMENT	Unique record ID
username	STRING	NOT NULL, FOREIGN KEY	Reference to user_login
reason	STRING	NOT NULL	Deactivation reason
created_at	TIMESTAMP	AUTO	Record creation timestamp
updated_at	TIMESTAMP	AUTO	Last update timestamp

Indexes:

- Primary Key: `id`
- Index on: `username`

Foreign Keys:

- `username` → `user_login.username`

Model File: `models/deactivate_reason.model.js`

MongoDB Database

MongoDB is used for high-throughput log storage, particularly for payment gateway logs.

Collection Structure

```
MongoDB Collections
├── xendit_checkout_logs    (Payment checkout logs)
└── xendit_payout_logs      (Payout transaction logs)
```

---

1. xendit_checkout_logs

Purpose: Stores detailed logs of Xendit checkout/invoice operations.

Collection Name: `xendit_checkout_logs`

Schema:

```
{
  id: String,               // Xendit checkout ID
  created_at: Date,         // Checkout creation date
  platform: String,         // Platform identifier
  status: String,           // Checkout status
  data: String,             // Serialized checkout data
  createdAt: Date,          // MongoDB document creation (auto)
  updatedAt: Date           // MongoDB document update (auto)
}
```

Indexes:

- Automatic: `_id`
- Custom: `id`, `status`, `created_at`

Model File: `models/xendit_checkout_logs.model.js`

Status Values:

- `PENDING` - Checkout created
- `PAID` - Payment received
- `EXPIRED` - Checkout expired
- `FAILED` - Payment failed

Sample Document:


```
{
  "_id": "507f1f77bcf86cd799439011",
  "id": "inv_123abc456def",
  "created_at": "2024-01-15T10:30:00.000Z",
  "platform": "web",
  "status": "PAID",
  "data": "{\"amount\":100000,\"payment_method\":\"OVO\"}",
  "createdAt": "2024-01-15T10:30:00.123Z",
  "updatedAt": "2024-01-15T10:35:00.456Z"
}
```


---

2. xendit_payout_logs

Purpose: Stores detailed logs of Xendit payout operations.

Collection Name: `xendit_payout_logs`

Schema:

```
{
  id: String,                    // Xendit payout ID
  created_at: Date,              // Payout creation date
  estimated_arrival_time: Date,  // Estimated completion time
  platform: String,              // Platform identifier
  status: String,                // Payout status
  data: String,                  // Serialized payout data
  createdAt: Date,               // MongoDB document creation (auto)
  updatedAt: Date                // MongoDB document update (auto)
}
```

Indexes:

- Automatic: `_id`
- Custom: `id`, `status`, `created_at`

Model File: `models/xendit_payout_logs.model.js`

Status Values:

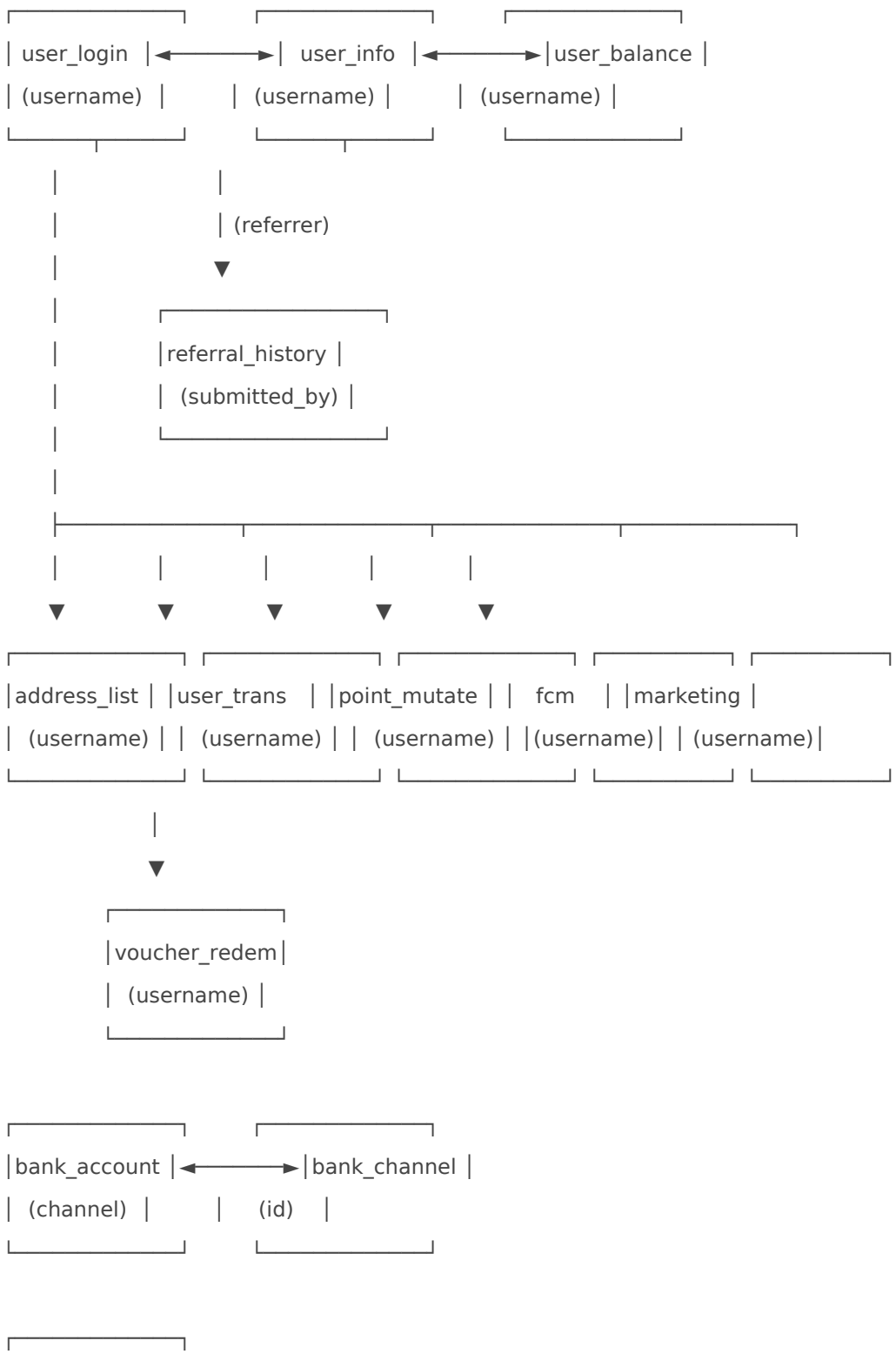
- `PENDING` - Payout requested
- `PROCESSING` - Being processed
- `COMPLETED` - Payout successful
- `FAILED` - Payout failed
- `CANCELLED` - Payout cancelled

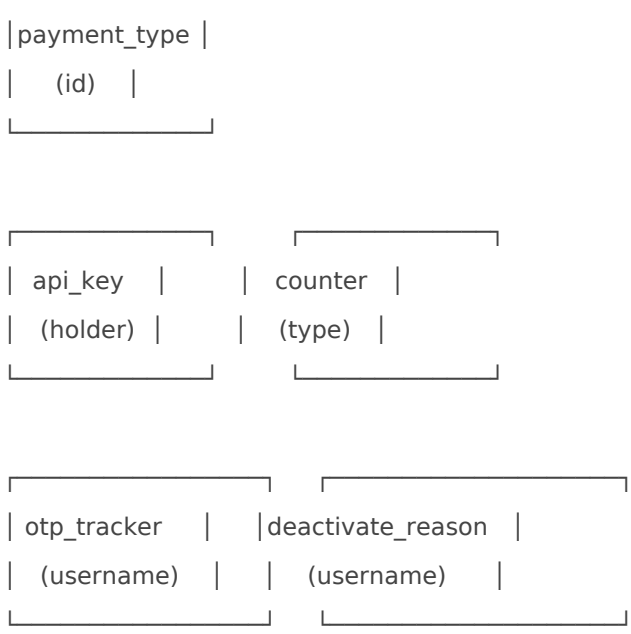
Sample Document:

```
{
  "_id": "507f191e810c19729de860ea",
  "id": "disb_789xyz",
  "created_at": "2024-01-15T11:00:00.000Z",
  "estimated_arrival_time": "2024-01-15T15:00:00.000Z",
  "platform": "api",
  "status": "COMPLETED",
  "data": "{\"amount\":50000,\"bank_code\":\"BCA\"}",
  "createdAt": "2024-01-15T11:00:00.123Z",
  "updatedAt": "2024-01-15T14:30:00.789Z"
}
```

Entity Relationships

Relationship Diagram (ASCII)





Defined Relationships in Code

From `models/index.js`:

```
// user_login → user_info (One-to-One)
db.user_login.hasOne(db.user_info, {
  foreignKey: "username",
  sourceKey: "username",
});

// user_info → user_login (Belongs-to)
db.user_info.belongsTo(db.user_login, {
  foreignKey: "username",
  sourceKey: "username",
});

// user_info → user_balance (Belongs-to)
db.user_info.belongsTo(db.user_balance, {
  foreignKey: "username",
  sourceKey: "username",
});

// user_login → marketing_offer (One-to-Many)
db.user_login.hasMany(db.marketing_offer, {
  foreignKey: "username",
```

```
  sourceKey: "username",
});

// user_login → fcm (One-to-Many)
db.user_login.hasMany(db.fcm, {
  foreignKey: "username",
  sourceKey: "username",
});

// user_login → user_balance (One-to-One)
db.user_login.hasOne(db.user_balance, {
  foreignKey: "username",
  sourceKey: "username",
});

// user_login → user_transaction (One-to-Many)
db.user_login.hasMany(db.user_transaction, {
  foreignKey: "username",
  sourceKey: "username",
});

// user_login → voucher_redemption (One-to-Many)
db.user_login.hasMany(db.voucher_redemption, {
  foreignKey: "username",
  sourceKey: "username",
});

// user_login → address_list (One-to-Many)
db.user_login.hasMany(db.address_list, {
  foreignKey: "username",
  sourceKey: "username",
});

// referral_history → user_info (referrer)
db.referral_history.belongsTo(db.user_info, {
  foreignKey: "submitted_by",
  sourceKey: "username",
  as: "referral_submit_by",
});
```

Database Setup

Prerequisites

- PostgreSQL 12+ installed
- MongoDB 4.4+ installed (optional, for payment logs)
- Node.js 14+ with npm

Step 1: Create Databases

PostgreSQL:

```
# Connect to PostgreSQL
psql -U postgres

# Create database
CREATE DATABASE userstate;

# Create user (if needed)
CREATE USER userstate_user WITH PASSWORD 'your_password';

# Grant privileges
GRANT ALL PRIVILEGES ON DATABASE userstate TO userstate_user;

# Exit
\q
```

MongoDB:

```
# MongoDB creates databases automatically on first use
# Just ensure MongoDB service is running
sudo systemctl start mongodb
# or
mongod --dbpath /path/to/data
```

Step 2: Configure Environment Variables

Create `.env` file:

```
# PostgreSQL
PS_HOST=localhost
PS_USER=userstate_user
PS_PASSWORD=your_password
PS_DB=userstate
PS_DIALECT=postgres
PS_PORT=5432
PS_POOL_MAX=50

# MongoDB
MONGO_DB=mongodb://localhost:27017/userstate
```

Step 3: Initialize Tables

Option 1: Auto-sync (Development)

Uncomment lines in `index.js`:

```
(async () => {
  await db.user_login.sync({ alter: true });
  await db.point_mutation.sync({ alter: true });
  await db.referral_history.sync({ alter: true });
  await db.user_info.sync({ alter: true });
  await db.user_balance.sync({ alter: true });
  // Add other tables...
})();
```

Option 2: Manual SQL (Production)

Run SQL scripts in order:

```
-- 1. Create user_login table
CREATE TABLE user_login (
    username VARCHAR(255) PRIMARY KEY,
    pwd VARCHAR(255) NOT NULL,
```

```
    disabled BOOLEAN NOT NULL DEFAULT false,
    token VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 2. Create user_info table
CREATE TABLE user_info (
    id SERIAL,
    username VARCHAR(255) PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    profile_img VARCHAR(255),
    gender VARCHAR(50),
    date_of_birth DATE,
    phone_number VARCHAR(50),
    agent VARCHAR(50),
    agent_id VARCHAR(50),
    referral_code VARCHAR(50) UNIQUE,
    referral_applied_status VARCHAR(50),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (username) REFERENCES user_login(username)
);

-- Continue with other tables...
```

Step 4: Seed Initial Data

API Keys for Internal Services:

```
INSERT INTO api_key (holder, key)
VALUES ('internal-service', '$2b$10$hashed_api_key_here');
```

Payment Types:

```
INSERT INTO payment_types (name, status) VALUES
('CREDIT_CARD', true),
```

```
('BANK_TRANSFER', true),
('E_WALLET', true),
('RETAIL_OUTLET', true);
```

Bank Channels:

```
INSERT INTO bank_channel (id, code, type, currency) VALUES
(1, 'BCA', 'BANK', 'IDR'),
(2, 'MANDIRI', 'BANK', 'IDR'),
(3, 'BNI', 'BANK', 'IDR');
```

Migrations

Migration Strategy

The application uses Sequelize ORM with the following migration approach:

Development Environment:

- Use `sync({ alter: true })` to auto-update schema
- Quick iteration without migration files

Production Environment:

- Use Sequelize migrations for controlled changes
- Version controlled schema changes
- Rollback capability

Creating a Migration

```
# Install Sequelize CLI
npm install --save-dev sequelize-cli

# Initialize Sequelize
npx sequelize-cli init

# Create migration
```

```
npx sequelize-cli migration:generate --name add-column-to-user-info
```

Example Migration

File: `migrations/20240115000000-add-column-to-user-info.js`

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.addColumn('user_info', 'new_field', {
      type: Sequelize.STRING,
      allowNull: true,
    });
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.removeColumn('user_info', 'new_field');
  }
};
```

Running Migrations

```
# Run all pending migrations
npx sequelize-cli db:migrate

# Rollback last migration
npx sequelize-cli db:migrate:undo

# Rollback all migrations
npx sequelize-cli db:migrate:undo:all
```

Common Queries

User Management

Get user with profile and balance:

```
SELECT
    ul.username,
    ui.email,
    ui.first_name,
    ui.last_name,
    ub.balance,
    ub.onapps_point
FROM user_login ul
JOIN user_info ui ON ul.username = ui.username
LEFT JOIN user_balance ub ON ul.username = ub.username
WHERE ul.username = 'john_doe';
```

Find users by email:

```
SELECT username, email, first_name, last_name
FROM user_info
WHERE email ILIKE '%@example.com';
```

Transaction Queries

Get user transaction history:

```
SELECT
    id,
    type,
    amount,
    status,
    created_at
FROM user_transaction
WHERE username = 'john_doe'
ORDER BY created_at DESC
LIMIT 10;
```

Sum transactions by status:

```
SELECT
    status,
    COUNT(*) as count,
    SUM(amount) as total_amount
```

```
FROM user_transaction
WHERE username = 'john_doe'
GROUP BY status;
```

Point System

Get point mutation history:

```
SELECT
    type,
    amount,
    src,
    status,
    created_at
FROM point_mutation
WHERE username = 'john_doe'
ORDER BY created_at DESC;
```

Calculate total points earned:

```
SELECT
    SUM(CASE WHEN type = 'EARN' THEN amount ELSE 0 END) as earned,
    SUM(CASE WHEN type = 'REDEEM' THEN amount ELSE 0 END) as redeemed,
    SUM(CASE WHEN type = 'EARN' THEN amount ELSE -amount END) as balance
FROM point_mutation
WHERE username = 'john_doe' AND status = 'COMPLETE';
```

Referral Analytics

Get referral statistics:

```
SELECT
    submitted_by as referrer,
    COUNT(*) as total_referrals,
    SUM(CASE WHEN status = 'COMPLETED' THEN 1 ELSE 0 END) as completed,
    SUM(CASE WHEN status = 'PENDING' THEN 1 ELSE 0 END) as pending
FROM referral_history
GROUP BY submitted_by
ORDER BY total_referrals DESC;
```

Find top referrers:

```
SELECT
    rh.submitted_by,
    ui.first_name,
    ui.last_name,
    COUNT(*) as referral_count
FROM referral_history rh
JOIN user_info ui ON rh.submitted_by = ui.username
WHERE rh.status = 'COMPLETED'
GROUP BY rh.submitted_by, ui.first_name, ui.last_name
ORDER BY referral_count DESC
LIMIT 10;
```

Payment Queries

Active payment channels:

```
SELECT
    id,
    code,
    type,
    currency,
    description
FROM bank_channel
WHERE currency = 'IDR'
ORDER BY code;
```

User voucher redemptions:

```
SELECT
    code,
    status,
    created_at
FROM voucher_redemption
WHERE username = 'john_doe'
ORDER BY created_at DESC;
```

MongoDB Queries

Get recent checkout logs:

```
db.xendit_checkout_logs.find({
  status: "PAID",
  created_at: {
    $gte: ISODate("2024-01-01T00:00:00Z")
  }
}).sort({ created_at: -1 }).limit(10);
```

Aggregate checkout stats:

```
db.xendit_checkout_logs.aggregate([
  {
    $group: {
      _id: "$status",
      count: { $sum: 1 },
      total: { $sum: { $toInt: "$amount" } }
    }
  }
]);
```

Backup and Maintenance

PostgreSQL Backup

Full Database Backup:

```
# Backup
pg_dump -U userstate_user -h localhost userstate > backup_$(date +%Y%m%d).sql

# Restore
psql -U userstate_user -h localhost userstate < backup_20240115.sql
```

Specific Table Backup:

```
pg_dump -U userstate_user -h localhost -t user_login -t user_info userstate > users_backup.sql
```

Automated Daily Backup (cron):

```
# Add to crontab
0 2 * * * /usr/bin/pg_dump -U userstate_user userstate > /backups/db_$(date +\%Y\%m\%d).sql
```

MongoDB Backup

Full Backup:

```
# Backup
mongodump --db=userstate --out=/backups/mongo_$(date +%Y%m%d)

# Restore
mongorestore --db=userstate /backups/mongo_20240115/userstate
```

Specific Collection:

```
mongodump --db=userstate --collection=xendit_checkout_logs --out=/backups/
```

Maintenance Tasks

Vacuum PostgreSQL (recommended weekly):

```
VACUUM ANALYZE user_transaction;
VACUUM ANALYZE point_mutation;
VACUUM ANALYZE referral_history;
```

Reindex:

```
REINDEX TABLE user_info;
REINDEX TABLE user_transaction;
```

Clean old logs (MongoDB):

```
// Delete logs older than 90 days
db.xendit_checkout_logs.deleteMany({
  created_at: {
    $lt: new Date(Date.now() - 90*24*60*60*1000)
  }
});
```

Performance Optimization

Indexing Recommendations

High-Priority Indexes:

```
-- User lookups
CREATE INDEX idx_user_info_email ON user_info(email);
CREATE INDEX idx_user_info_referral_code ON user_info(referral_code);

-- Transaction queries
CREATE INDEX idx_user_transaction_username ON user_transaction(username);
CREATE INDEX idx_user_transaction_status ON user_transaction(status);
CREATE INDEX idx_user_transaction_created_at ON user_transaction(created_at);

-- Point queries
CREATE INDEX idx_point_mutation_username ON point_mutation(username);
CREATE INDEX idx_point_mutation_type ON point_mutation(type);

-- Referral queries
CREATE INDEX idx_referral_history_submitted_by ON referral_history(submitted_by);
CREATE INDEX idx_referral_history_status ON referral_history(status);
```

Composite Indexes:

```
-- Common filter combinations
CREATE INDEX idx_transaction_user_status ON user_transaction(username, status);
CREATE INDEX idx_transaction_user_created ON user_transaction(username, created_at DESC);
CREATE INDEX idx_point_user_type_status ON point_mutation(username, type, status);
```

Query Optimization

Use EXPLAIN to analyze queries:

```
EXPLAIN ANALYZE
SELECT * FROM user_transaction
WHERE username = 'john_doe' AND status = 'PAID';
```

Avoid SELECT :

```
-- Bad
SELECT * FROM user_info WHERE username = 'john_doe';

-- Good
SELECT username, email, first_name, last_name
FROM user_info WHERE username = 'john_doe';
```

Connection Pooling

Already configured in `config/db.js`:

```
POOL: {
    max: parseInt(process.env.PS_POOL_MAX) // Default: 50
}
```

Recommended Pool Sizes:

- Development: 5-10
- Production (small): 20-30
- Production (large): 50-100

MongoDB Performance

Create Indexes:

```
db.xendit_checkout_logs.createIndex({ id: 1 });
db.xendit_checkout_logs.createIndex({ status: 1 });
db.xendit_checkout_logs.createIndex({ created_at: -1 });

db.xendit_payout_logs.createIndex({ id: 1 });
db.xendit_payout_logs.createIndex({ status: 1 });
db.xendit_payout_logs.createIndex({ created_at: -1 });
```

Compound Indexes:

```
db.xendit_checkout_logs.createIndex({
  status: 1,
  created_at: -1
```

```
});
```

Data Integrity Rules

Constraints

1. **Username Uniqueness**: Username is unique across `user_login`
2. **Email Uniqueness**: Email is unique in `user_info`
3. **Referral Code Uniqueness**: Each user has a unique referral code
4. **Balance Integrity**: Balance cannot be negative (enforced in application logic)
5. **Transaction Atomicity**: Balance updates and transaction records must be atomic

Cascading Rules

On User Deletion (if implemented):

- Soft delete: Set `disabled = true` in `user_login`
- Hard delete: CASCADE to related tables (not recommended for financial data)

Data Validation

At Database Level:

- NOT NULL constraints on critical fields
- UNIQUE constraints on username, email, referral_code
- CHECK constraints (can be added for balance >= 0)

At Application Level (middleware/validation):

- Email format validation
- Phone number format validation
- Password strength requirements
- Amount range validation

Troubleshooting

Common Issues

Connection Refused:

```
# Check PostgreSQL status
sudo systemctl status postgresql

# Check MongoDB status
sudo systemctl status mongod

# Verify connection settings in .env
```

Table Already Exists:

```
# Drop and recreate (development only!)
DROP TABLE IF EXISTS table_name CASCADE;
```

Foreign Key Violations:

```
-- Find orphaned records
SELECT * FROM user_info ui
LEFT JOIN user_login ul ON ui.username = ul.username
WHERE ul.username IS NULL;
```

Performance Issues:

```
-- Check table sizes
SELECT
    schemaname,
    tablename,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) AS size
FROM pg_tables
WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC;

-- Check index usage
SELECT
    schemaname,
    tablename,
    indexname,
```

```
    idx_scan
FROM pg_stat_user_indexes
ORDER BY idx_scan ASC;
```

Security Best Practices

1. **Password Storage**: Always use bcrypt/argon2, never plain text
2. **SQL Injection**: Use parameterized queries (Sequelize handles this)
3. **Sensitive Data**: Encrypt PII data at rest
4. **Access Control**: Use principle of least privilege for database users
5. **Audit Logging**: Log all sensitive data access
6. **Backup Encryption**: Encrypt backup files
7. **Connection Security**: Use SSL/TLS for database connections

---

Conclusion

This document provides complete database documentation for the UserState API. For application-level documentation, see:

- DOCUMENTATION.md - Complete API documentation
- API_REFERENCE.md - Quick API reference
- QUICKSTART.md - Setup guide

Last Updated: February 2024
Version: 1.0.0
Database Version: PostgreSQL 12+, MongoDB 4.4+

Revision #1
Created 10 February 2026 03:35:50 by ondeliveloper
Updated 10 February 2026 03:36:21 by ondeliveloper