Authentication Service Documentation

Complete API Reference and Testing Guide

Table of Contents

1 Overview
2 Getting Started
3 Environment Configuration
4 Database Schema
5 API Endpoints
6 Request and Response Formats
7 Authentication Flow
8 Error Handling
9 Local Development Setup
10 Testing with Postman
11 Security Considerations
12 Troubleshooting
13 References

Overview

The Authentication Service is a Node.js-based microservice that provides secure user authentication and authorization capabilities for the virtual card financial system. Built with Express.js and MySQL, the service implements industry-standard JWT (JSON Web Token) authentication with token refresh mechanisms, password hashing with bcryptjs, and comprehensive input validation.

Key Features

User Registration - Secure user account creation with email validation, password strength requirements, and duplicate prevention.

User Login - Credential-based authentication with JWT token generation for subsequent API requests.

Token Management - Access token and refresh token generation with configurable expiration times, enabling secure session management.

Token Refresh - Ability to obtain new access tokens using refresh tokens without requiring users to re-enter credentials.

User Profile Retrieval - Authenticated endpoint to retrieve current user information.

Logout - Endpoint to invalidate user sessions (token-based logout).

Rate Limiting - Protection against brute force attacks through request rate limiting.

Security Middleware - Helmet.js for HTTP header security, CORS for cross-origin requests, and input validation.

Architecture

The service follows a layered architecture pattern:

- **Routes Layer** - Express route handlers for HTTP endpoints
- **Service Layer** - Business logic for authentication operations
- **Repository Layer** - Data access layer for database operations
- **Middleware Layer** - Authentication and security middleware
- **Utility Layer** - Reusable functions for JWT, password, and validation operations

Getting Started

Prerequisites

Before setting up the authentication service, ensure the following software is installed on your system:

- **Node.js** (version 14.0 or higher) - JavaScript runtime
- **npm** (version 6.0 or higher) - Node package manager
- **MySQL** (version 5.7 or higher) - Relational database server
- **Git** - Version control system (optional, for cloning repositories)

Quick Start

Step 1: Extract the Service

Extract the auth-service.zip file to your desired location:

```
unzip auth-service.zip
```

```
cd auth-service
```

Step 2: Install Dependencies

Install all required npm packages:

```
npm install
```

This command reads the package.json file and installs all dependencies listed in the dependencies section.

Step 3: Configure Environment Variables

Create a .env file in the project root directory by copying the example.env file:

```
cp example.env .env
```

Edit the .env file with your database credentials and JWT secret:

```
PORT=3000
```

```
NODE_ENV=development

JWT_SECRET=your-super-secret-jwt-key-change-this-in-production
JWT_ACCESS_EXPIRY=15m
JWT_REFRESH_EXPIRY=7d

RATE_LIMIT_WINDOW_MS=900000
RATE_LIMIT_MAX_REQUESTS=100

DB_HOST=127.0.0.1
DB_PORT=3306
DB_USER=auth_user
DB_PASSWORD=authpwd
DB_NAME=auth_service
```

```
DB_CONN_LIMIT=10
```

Step 4: Ensure MySQL is Running

Verify that MySQL server is running on your system. The service will automatically create the database and tables on first run.

Step 5: Start the Service

Start the authentication service:

```
npm start
```

Or for development with automatic reload on file changes:

```
npm run dev
```

The service will output:

```
Auth service running on port 3000
```

```
Environment: development
Frontend: http://localhost:3000/
API Auth: http://localhost:3000/v1/api/auth
API Users: http://localhost:3000/v1/api/users
```

Environment Configuration

The authentication service uses environment variables for configuration, enabling different settings for development, testing, and production environments.

Configuration Variables

Variable	Default	Description
PORT	3000	Port number on which the service runs
NODE_ENV	development	Environment mode (development, production, test)
JWT_SECRET	default-secret-change-me	Secret key for signing JWT tokens (MUST be changed in production)
JWT_ACCESS_EXPIRY	15m	Expiration time for access tokens (e.g., 15m, 1h, 7d)
JWT_REFRESH_EXPIRY	7d	Expiration time for refresh tokens
RATE_LIMIT_WINDOW_MS	900000	Time window for rate limiting in milliseconds (900000 = 15 minutes)
RATE_LIMIT_MAX_REQUESTS	100	Maximum requests allowed per window
DB_HOST	127.0.0.1	MySQL server hostname or IP address
DB_PORT	3306	MySQL server port
DB_USER	root	MySQL database user
DB_PASSWORD	(empty)	MySQL database password
DB_NAME	auth_service	Database name
DB_CONN_LIMIT	10	Maximum database connection pool size

Production Configuration

For production deployments, ensure the following security measures:

14 **Change JWT_SECRET** - Use a strong, randomly generated secret key (minimum 32 characters)
15 **Set NODE_ENV=production** - Enables production optimizations
16 **Use Strong Database Credentials** - Implement complex passwords and restrict database access
17 **Enable HTTPS** - Deploy behind a reverse proxy (nginx, Apache) with SSL/TLS
18 **Adjust Rate Limiting** - Fine-tune based on expected traffic patterns
19 **Use Environment Variables** - Never commit sensitive credentials to version control

Example Production .env

```
PORT=3000
```

```
NODE_ENV=production

JWT_SECRET=your-generated-random-secret-key-minimum-32-characters-long
JWT_ACCESS_EXPIRY=15m
JWT_REFRESH_EXPIRY=7d

RATE_LIMIT_WINDOW_MS=900000
RATE_LIMIT_MAX_REQUESTS=100

DB_HOST=db.example.com
DB_PORT=3306
DB_USER=auth_user_prod
DB_PASSWORD=strong-production-password
DB_NAME=auth_service_prod
```

```
DB_CONN_LIMIT=20
```

Database Schema

The authentication service uses a single primary table for user data storage. The database is automatically created on service startup.

Users Table

The users table stores all user account information:

```
CREATE TABLE users (

  id CHAR(36) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL,
  name VARCHAR(255) NULL,
  account_status ENUM('active','disabled') NOT NULL DEFAULT 'active',
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Column Definitions

Column	Type	Constraints	Description
id	CHAR(36)	PRIMARY KEY, NOT NULL	Unique user identifier (UUID format)
email	VARCHAR(255)	UNIQUE, NOT NULL	User email address (must be unique)
password_hash	VARCHAR(255)	NOT NULL	Bcrypt hashed password (never store plain text)
name	VARCHAR(255)	NULL	User's full name (optional)
account_status	ENUM('active','disabled')	NOT NULL, DEFAULT 'active'	Account status (active or disabled)
created_at	DATETIME	NOT NULL, DEFAULT CURRENT_TIMESTAMP	Account creation timestamp
updated_at	DATETIME	NOT NULL, DEFAULT CURRENT_TIMESTAMP	Last profile update timestamp

Indexes

The table includes the following indexes for optimal query performance:

- **PRIMARY KEY (id)** - Enables fast user lookup by ID
- **UNIQUE (email)** - Ensures email uniqueness and enables fast lookup by email

API Endpoints

The authentication service exposes the following REST API endpoints. All endpoints are prefixed with /v1/api/.

Authentication Endpoints

1. User Registration

Endpoint: POST /v1/api/auth/register

Description: Creates a new user account with email, password, and optional name.

Request Headers:

```
Content-Type: application/json
```

Request Body:

```
{
  "email": "user@example.com",
  "password": "SecurePassword123!",
  "name": "John Doe"
}
```

Request Body Parameters:

Parameter	Type	Required	Description
email	string	Yes	User email address (must be valid email format)
password	string	Yes	User password (must meet complexity requirements)
name	string	No	User's full name

Password Requirements:

The password must meet the following complexity requirements:

- Minimum 8 characters long
- At least one uppercase letter (A-Z)
- At least one lowercase letter (a-z)
- At least one number (0-9)
- At least one special character (!@#$%^&*)

Example valid password: SecurePass123!

Success Response (201 Created):

```
{
  "status": "success",
  "message": "User registered successfully",
  "data": {
    "user": {
      "id": "550e8400-e29b-41d4-a716-446655440000",
      "email": "user@example.com",
      "name": "John Doe",
      "accountStatus": "active",
      "createdAt": "2025-11-05T10:30:00.000Z",
      "updatedAt": "2025-11-05T10:30:00.000Z"
    },
    "tokens": {
      "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
      "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    }
  },
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Error Responses:

400 Bad Request - Missing Fields:

```
{
  "status": "error",
  "message": "Email and password are required",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

400 Bad Request - Invalid Email:

```
{
  "status": "error",
  "message": "Invalid email address",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

400 Bad Request - Weak Password:

```
{
  "status": "error",
  "message": "Password must be at least 8 characters long, Password must contain at least one uppercase letter, Password must contain at least one special character (!@#$%^&*)",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

400 Bad Request - User Exists:

```
{
  "status": "error",
  "message": "User already exists",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

---

2. User Login

Endpoint: POST /v1/api/auth/login

Description: Authenticates a user with email and password, returning JWT tokens.

Request Headers:

```
Content-Type: application/json
```

Request Body:

```
{
  "email": "user@example.com",
  "password": "SecurePassword123!"
}
```

Request Body Parameters:

Parameter	Type	Required	Description
email	string	Yes	User email address
password	string	Yes	User password

Success Response (200 OK):

```
{
  "status": "success",
  "message": "Login successful",
  "data": {
    "user": {
      "id": "550e8400-e29b-41d4-a716-446655440000",
      "email": "user@example.com",
      "name": "John Doe",
      "accountStatus": "active",
      "createdAt": "2025-11-05T10:30:00.000Z",
      "updatedAt": "2025-11-05T10:30:00.000Z"
    },
    "tokens": {
      "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
      "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    }
  },
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Error Responses:

401 Unauthorized - Invalid Credentials:

```
{
  "status": "error",
  "message": "Invalid credentials",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

401 Unauthorized - Account Disabled:

```
{
  "status": "error",
  "message": "Account disabled",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

400 Bad Request - Missing Fields:

```
{
  "status": "error",
  "message": "Email and password are required",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

---

3. Refresh Token

Endpoint: POST /v1/api/auth/refresh

Description: Generates new access and refresh tokens using a valid refresh token.

Request Headers:

```
Content-Type: application/json
```

Request Body:

```
{
  "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

Request Body Parameters:

Parameter	Type	Required	Description
refreshToken	string	Yes	Valid refresh token from login or previous refresh

Success Response (200 OK):

```
{
  "status": "success",
  "message": "Token refreshed successfully",
  "data": {
    "tokens": {
      "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
      "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    }
  },
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Error Responses:

400 Bad Request - Missing Token:

```json
{
  "status": "error",
  "message": "Refresh token is required",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

401 Unauthorized - Invalid Token:

```json
{
  "status": "error",
  "message": "Invalid or expired token",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

401 Unauthorized - User Not Found:

```json
{
  "status": "error",
  "message": "User not found",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

401 Unauthorized - Account Disabled:

```json
{
  "status": "error",
  "message": "Account disabled",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

4. Logout

Endpoint: POST /v1/api/auth/logout

Description: Invalidates the current user session. Requires authentication.

Request Headers:

```
Content-Type: application/json
Authorization: Bearer <accessToken>
```

Request Body: (empty)

Success Response (200 OK):

```
{
  "status": "success",
  "message": "Logout successful",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Error Responses:

401 Unauthorized - No Token:

```
{
  "status": "error",
  "message": "No token provided",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

401 Unauthorized - Invalid Token:

```
{
  "status": "error",
  "message": "Invalid or expired token",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

User Endpoints

5. Get Current User

Endpoint: GET /v1/api/users/me

Description: Retrieves the current authenticated user's profile information.

Request Headers:

```
Authorization: Bearer <accessToken>
```

Success Response (200 OK):

```
{
  "status": "success",
  "message": "User retrieved successfully",
  "data": {
    "user": {
      "id": "550e8400-e29b-41d4-a716-446655440000",
      "email": "user@example.com",
      "name": "John Doe",
      "accountStatus": "active",
      "createdAt": "2025-11-05T10:30:00.000Z",
      "updatedAt": "2025-11-05T10:30:00.000Z"
    }
  },
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Error Responses:

401 Unauthorized - No Token:

```
{
  "status": "error",
  "message": "No token provided",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

401 Unauthorized - Invalid Token:

```
{
  "status": "error",
  "message": "Invalid or expired token",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

404 Not Found - User Not Found:

```
{
  "status": "error",
  "message": "User not found",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Health Check Endpoint

6. Health Check

Endpoint: GET /health

Description: Returns the health status of the authentication service.

Request Headers: (none required)

Success Response (200 OK):

```json
{
  "status": "ok",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Request and Response Formats

Standard Response Format

All API responses follow a consistent JSON format:

```json
{
  "status": "success|error",
  "message": "Human-readable message",
  "data": {
    // Response-specific data
  },
  "timestamp": "ISO 8601 timestamp"
}
```

Response Fields

Field	Type	Description
status	string	Response status: "success" or "error"
message	string	Human-readable message describing the response
data	object	Response-specific data (may be null for error responses)
timestamp	string	ISO 8601 formatted timestamp of the response

HTTP Status Codes

Status Code	Meaning	Usage
200	OK	Successful GET, POST, or other requests
201	Created	Successful resource creation (registration)
400	Bad Request	Invalid request format or validation failure
401	Unauthorized	Authentication failure or missing token
404	Not Found	Resource not found
429	Too Many Requests	Rate limit exceeded
500	Internal Server Error	Server error

JWT Token Structure

Access tokens and refresh tokens are JWT (JSON Web Tokens) with the following structure:

Header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload:

```
{
  "userId": "550e8400-e29b-41d4-a716-446655440000",
  "email": "user@example.com",
  "iat": 1730703000,
  "exp": 1730703900
}
```

Signature:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload),
secret)
```

Authentication Flow

Registration and Login Flow

The following diagram illustrates the complete authentication flow from registration through token refresh:

Step 1: User Registration

20. Client sends POST request to /v1/api/auth/register with email, password, and name
21. Service validates email format and password complexity
22. Service checks if user already exists
23. Service hashes password using bcryptjs
24. Service creates new user record in database
25. Service generates access and refresh tokens
26. Service returns user data and tokens to client

Step 2: User Login

27. Client sends POST request to /v1/api/auth/login with email and password
28. Service retrieves user by email from database
29. Service verifies password using bcryptjs comparison
30. Service checks if account is active
31. Service generates access and refresh tokens
32. Service returns user data and tokens to client

Step 3: Authenticated Request

33. Client includes access token in Authorization header: Bearer <accessToken>
34. API Gateway/Middleware extracts token from header
35. Middleware verifies token signature using JWT_SECRET
36. Middleware checks token expiration
37. Middleware retrieves user from database to verify account status
38. Middleware attaches user information to request
39. Route handler processes request with authenticated user context

Step 4: Token Refresh

40. Client detects access token expiration (or proactively refreshes)
41. Client sends POST request to /v1/api/auth/refresh with refresh token
42. Service verifies refresh token signature and expiration
43. Service retrieves user from database
44. Service checks if account is active
45. Service generates new access and refresh tokens
46. Service returns new tokens to client
47. Client updates stored tokens and continues with authenticated requests

Token Expiration and Refresh

Access Token - Short-lived token (default 15 minutes) used for API requests. Expires quickly to minimize security risk if token is compromised.

Refresh Token - Long-lived token (default 7 days) used to obtain new access tokens without requiring user re-authentication.

Refresh Flow - When access token expires, client uses refresh token to obtain new tokens without user interaction, providing seamless user experience.

Error Handling

The authentication service implements comprehensive error handling with meaningful error messages and appropriate HTTP status codes.

Common Error Scenarios

Validation Errors (400 Bad Request)

Validation errors occur when request data doesn't meet requirements:

```
{
  "status": "error",
  "message": "Invalid email address",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Common validation errors include:

- Missing required fields (email, password)

- Invalid email format
- Weak password (insufficient complexity)
- Password too short

Authentication Errors (401 Unauthorized)

Authentication errors occur when credentials are invalid or tokens are expired:

```
{
  "status": "error",
  "message": "Invalid credentials",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Common authentication errors include:

- Invalid email or password
- Missing or malformed token
- Expired token
- Invalid token signature
- User account disabled

Resource Errors (404 Not Found)

Resource errors occur when requested resources don't exist:

```
{
  "status": "error",
  "message": "User not found",
  "timestamp": "2025-11-05T10:30:00.000Z"
}
```

Rate Limiting (429 Too Many Requests)

Rate limiting errors occur when request limit is exceeded:

```
{
  "status": "error",
  "message": "Too many requests from this IP, please try again later",
  "timestamp": "2025-11-05T10:30:00.000Z"
```

```
}
```

Server Errors (500 Internal Server Error)

Server errors indicate unexpected failures:

```
{
```

```
  "status": "error",
  "message": "Database error (PROTOCOL_CONNECTION_LOST)",
  "timestamp": "2025-11-05T10:30:00.000Z"
```

```
}
```

Error Response Structure

All error responses follow the standard response format with status set to "error":

```
{
```

```
  "status": "error",
  "message": "Descriptive error message",
  "timestamp": "ISO 8601 timestamp"
```

```
}
```

Local Development Setup

Complete Setup Guide

This section provides step-by-step instructions for setting up the authentication service on your local development machine.

Prerequisites Installation

On macOS (using Homebrew):

```
# Install Node.js and npm
brew install node

# Install MySQL
brew install mysql

# Start MySQL service
brew services start mysql
```

On Ubuntu/Debian:

```
# Update package manager
sudo apt-get update

# Install Node.js and npm
sudo apt-get install nodejs npm

# Install MySQL
sudo apt-get install mysql-server

# Start MySQL service
sudo systemctl start mysql
```

On Windows:

48 Download and install Node.js from https://nodejs.org/
49 Download and install MySQL from https://dev.mysql.com/downloads/mysql/
50 During MySQL installation, configure MySQL Server as a Windows Service

Database Setup

Step 1: Connect to MySQL

```
mysql -u root -p
```

Enter the root password when prompted (default is empty on fresh installations).

Step 2: Create Database User

```
CREATE USER 'auth_user'@'localhost' IDENTIFIED BY 'authpwd';
```

```
GRANT ALL PRIVILEGES ON auth_service.* TO 'auth_user'@'localhost';
FLUSH PRIVILEGES;
```

```
EXIT;
```

Step 3: Verify Connection

```
mysql -u auth_user -p -h 127.0.0.1
```

Enter password "authpwd" when prompted. If successful, you'll see the MySQL prompt.

Service Setup

Step 1: Extract and Navigate

```
unzip auth-service.zip
```

```
cd auth-service
```

Step 2: Install Dependencies

```
npm install
```

This installs all packages listed in package.json:

- express - Web framework
- cors - Cross-origin resource sharing
- helmet - Security headers
- express-rate-limit - Rate limiting
- mysql2 - MySQL driver

- bcryptjs - Password hashing
- jsonwebtoken - JWT handling
- uuid - Unique ID generation
- dotenv - Environment variable management

Step 3: Configure Environment

```
cp example.env .env
```

Edit .env with your settings (default values work for local development):

```
PORT=3000
```

```
NODE_ENV=development

JWT_SECRET=your-super-secret-jwt-key-change-this-in-production
JWT_ACCESS_EXPIRY=15m
JWT_REFRESH_EXPIRY=7d

RATE_LIMIT_WINDOW_MS=900000
RATE_LIMIT_MAX_REQUESTS=100

DB_HOST=127.0.0.1
DB_PORT=3306
DB_USER=auth_user
DB_PASSWORD=authpwd
DB_NAME=auth_service
```

```
DB_CONN_LIMIT=10
```

Step 4: Start the Service

```
npm start
```

Expected output:

```
Auth service running on port 3000
```

```
Environment: development
Frontend: http://localhost:3000/
API Auth: http://localhost:3000/v1/api/auth
API Users: http://localhost:3000/v1/api/users
```

Step 5: Verify Service is Running

Open a new terminal and test the health endpoint:

```
curl http://localhost:3000/health
```

Expected response:

```
{"status":"ok","timestamp":"2025-11-05T10:30:00.000Z"}
```

Development Workflow

Running in Development Mode

For development with automatic restart on file changes, use:

```
npm run dev
```

This requires nodemon to be installed globally or as a dev dependency.

Accessing the Frontend

The service includes static HTML pages for testing:

- Login: http://localhost:3000/index.html
- Register: http://localhost:3000/register.html
- Dashboard: http://localhost:3000/dashboard.html

Viewing Database

To view the database and tables created by the service:

```
mysql -u auth_user -p auth_service
```

```

# View tables
```

```
SHOW TABLES;

# View users table structure
DESCRIBE users;

# View all users
```

```
SELECT * FROM users;
```

Testing with Postman

Postman is a powerful API testing tool that enables comprehensive testing of the authentication service endpoints. This section provides detailed instructions for setting up and using Postman.

Postman Setup

Step 1: Download and Install Postman

Download Postman from https://www.postman.com/downloads/ and install it on your system.

Step 2: Create a New Collection

51 Open Postman
52 Click "Create" button
53 Select "Collection"
54 Name it "Auth Service API"
55 Click "Create"

Step 3: Create Environment Variables

Environment variables allow you to store and reuse values across requests.

56 Click the "Environments" icon (gear) in the top right
57 Click "Create Environment"
58 Name it "Local Development"
59 Add the following variables:

Variable	Initial Value	Current Value
baseUrl	http://localhost:3000	http://localhost:3000

Variable	Initial Value	Current Value
accessToken	(empty)	(empty)
refreshToken	(empty)	(empty)
userId	(empty)	(empty)

60 Click "Save"
61 Select "Local Development" from the environment dropdown

API Testing Requests

Test 1: User Registration

Request Setup:

62 Create a new request in the collection
63 Name it "Register User"
64 Set method to POST
65 Set URL to {{baseUrl}}/v1/api/auth/register
66 Set Headers:
   - Content-Type: application/json

67 Set Body (raw JSON):

```
{
  "email": "testuser@example.com",
  "password": "TestPassword123!",
  "name": "Test User"
}
```

Pre-request Script:

Add this script to generate unique email for each test run:

```
// Generate unique email for each test
const timestamp = new Date().getTime();
const email = `testuser${timestamp}@example.com`;
```

```
pm.environment.set("testEmail", email);
```

Update Body to use variable:

```
{
  "email": "{{testEmail}}",
  "password": "TestPassword123!",
  "name": "Test User"
}
```

Tests Script:

Add this script to validate response and extract tokens:

```
// Check response status
pm.test("Registration successful", function () {
    pm.response.to.have.status(201);
});

// Check response structure
pm.test("Response has correct structure", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.status).to.equal("success");
    pm.expect(jsonData.data.user).to.exist;
    pm.expect(jsonData.data.tokens).to.exist;
});

// Extract and save tokens
pm.test("Tokens extracted and saved", function () {
    var jsonData = pm.response.json();
    pm.environment.set("accessToken", jsonData.data.tokens.accessToken);
    pm.environment.set("refreshToken", jsonData.data.tokens.refreshToken);
    pm.environment.set("userId", jsonData.data.user.id);
});
```

Execute Request:

68 Click "Send" button
69 View the response in the Response panel
70 Verify status code is 201
71 Check that tokens are saved in environment variables

Test 2: User Login

Request Setup:

72 Create a new request: "Login User"
73 Set method to POST
74 Set URL to {{baseUrl}}/v1/api/auth/login
75 Set Headers:
   - Content-Type: application/json

76 Set Body (raw JSON):

```
{
```

```
  "email": "{{testEmail}}",
  "password": "TestPassword123!"
```

```
}
```

Tests Script:

```
pm.test("Login successful", function () {
```

```
    pm.response.to.have.status(200);
});

pm.test("Response has tokens", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.data.tokens.accessToken).to.exist;
    pm.expect(jsonData.data.tokens.refreshToken).to.exist;
});

pm.test("Save new tokens", function () {
    var jsonData = pm.response.json();
    pm.environment.set("accessToken", jsonData.data.tokens.accessToken);
    pm.environment.set("refreshToken", jsonData.data.tokens.refreshToken);
```

```
});
```

Test 3: Get Current User

Request Setup:

77 Create a new request: "Get Current User"
78 Set method to GET
79 Set URL to {{baseUrl}}/v1/api/users/me
80 Set Headers:
   - Authorization: Bearer {{accessToken}}

Tests Script:

```
pm.test("Get user successful", function () {
```

```
    pm.response.to.have.status(200);
});

pm.test("Response contains user data", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.data.user.id).to.exist;
    pm.expect(jsonData.data.user.email).to.exist;
});

pm.test("User email matches login email", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.data.user.email).to.equal(pm.environment.get("testEmail"));
```

```
});
```

Test 4: Refresh Token

Request Setup:

81 Create a new request: "Refresh Token"
82 Set method to POST
83 Set URL to {{baseUrl}}/v1/api/auth/refresh
84 Set Headers:
   - Content-Type: application/json
85 Set Body (raw JSON):

```
{
  "refreshToken": "{{refreshToken}}"
}
```

Tests Script:

```
pm.test("Token refresh successful", function () {
    pm.response.to.have.status(200);
});

pm.test("New tokens generated", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.data.tokens.accessToken).to.exist;
    pm.expect(jsonData.data.tokens.refreshToken).to.exist;
});

pm.test("Save new tokens", function () {
    var jsonData = pm.response.json();
    pm.environment.set("accessToken", jsonData.data.tokens.accessToken);
    pm.environment.set("refreshToken", jsonData.data.tokens.refreshToken);
});
```

Test 5: Logout

Request Setup:

86 Create a new request: "Logout"
87 Set method to POST
88 Set URL to {{baseUrl}}/v1/api/auth/logout
89 Set Headers:
   - Authorization: Bearer {{accessToken}}

Tests Script:

```
pm.test("Logout successful", function () {
    pm.response.to.have.status(200);
});

pm.test("Logout message received", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.message).to.equal("Logout successful");
});
```

Test 6: Health Check

Request Setup:

90 Create a new request: "Health Check"
91 Set method to GET
92 Set URL to {{baseUrl}}/health

Tests Script:

```
pm.test("Health check successful", function () {
    pm.response.to.have.status(200);
});

pm.test("Service is healthy", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.status).to.equal("ok");
});
```

Running Test Collections

Manual Testing:

93 Execute requests individually by clicking "Send"
94 Review responses and test results
95 Check environment variables are updated correctly

Automated Testing:

96 Click the collection name
97 Click "Run" (or use the Runner icon)
98 Select the collection and environment
99 Click "Run Auth Service API"
100 View test results and execution summary

Test Results:

The collection runner displays:

- Number of tests passed/failed
- Response times
- Request/response details
- Console output for debugging

Postman Collection Export

To share the collection with team members:

101 Right-click the collection
102 Select "Export"
103 Choose JSON format
104 Save the file
105 Share with team members who can import it

Security Considerations

Password Security

Password Hashing - Passwords are hashed using bcryptjs with 12 salt rounds, making them resistant to brute force attacks. Hashed passwords cannot be reversed to obtain the original password.

Password Validation - Passwords must meet complexity requirements (8+ characters, uppercase, lowercase, number, special character) to prevent weak passwords.

Never Store Plain Text - The service never stores, logs, or transmits plain text passwords.

JWT Security

Token Signing - Tokens are signed using HMAC SHA-256 with a secret key. The signature prevents token tampering.

Token Expiration - Access tokens expire after 15 minutes, limiting the window of vulnerability if a token is compromised.

Refresh Token Rotation - Refresh tokens should be rotated on each use in production environments.

Secure Storage - Tokens should be stored securely on the client:

- Web: HttpOnly, Secure cookies or secure localStorage
- Mobile: Secure storage (Keychain on iOS, Keystore on Android)

Database Security

Connection Pooling - Database connections are pooled and reused, reducing connection overhead.

Parameterized Queries - All database queries use parameterized statements to prevent SQL injection.

User Isolation - Database user has limited privileges, restricted to the auth_service database.

API Security

CORS - Cross-Origin Resource Sharing is enabled to allow requests from authorized domains.

Helmet - Security headers are set using Helmet.js to protect against common web vulnerabilities.

Rate Limiting - API endpoints are rate-limited to prevent brute force attacks and abuse.

Input Validation - All user input is validated and sanitized before processing.

Production Recommendations

106. **Change JWT_SECRET** - Use a strong, randomly generated secret (minimum 32 characters)
107. **Enable HTTPS** - Deploy behind a reverse proxy with SSL/TLS
108. **Use Environment Variables** - Never commit secrets to version control
109. **Implement Logging** - Log authentication events for security auditing

110 **Monitor Rate Limits** - Adjust rate limiting based on traffic patterns
111 **Regular Updates** - Keep dependencies updated for security patches
112 **Database Backups** - Implement regular database backups
113 **Access Control** - Restrict database access to authorized services only

Troubleshooting

Common Issues and Solutions

Issue: "Cannot connect to database"

Symptoms: Service fails to start with database connection error

Solutions:

114 Verify MySQL is running: `sudo systemctl status mysql` (Linux) or `brew services list` (macOS)
115 Check database credentials in .env file
116 Verify database user exists: `mysql -u auth_user -p`
117 Check database host and port are correct
118 Ensure database user has privileges: `GRANT ALL PRIVILEGES ON auth_service.* TO 'auth_user'@'localhost';`

Issue: "Port 3000 already in use"

Symptoms: Service fails to start with "EADDRINUSE" error

Solutions:

119 Change PORT in .env file to an available port (e.g., 3001)
120 Kill process using port 3000: `lsof -i :3000` then `kill -9 <PID>`
121 On Windows: `netstat -ano | findstr :3000` then `taskkill /PID <PID> /F`

Issue: "Invalid or expired token"

Symptoms: Authenticated requests return 401 error

Solutions:

122 Verify access token is included in Authorization header
123 Check token format: `Bearer <token>` (with space)
124 Verify token hasn't expired (default 15 minutes)
125 Use refresh endpoint to get new token
126 Check JWT_SECRET matches between requests

Issue: "User already exists"

Symptoms: Registration fails with duplicate user error

Solutions:

127. Use different email address for registration
128. Clear database and restart: DROP DATABASE auth_service;
129. Verify email uniqueness constraint in database

Issue: "Password must contain..."

Symptoms: Registration fails with password validation error

Solutions:

130. Ensure password meets all requirements:
    - Minimum 8 characters
    - At least one uppercase letter
    - At least one lowercase letter
    - At least one number
    - At least one special character (!@#$%^&*)
131. Example valid password: SecurePass123!

Issue: "Rate limit exceeded"

Symptoms: Requests return 429 Too Many Requests

Solutions:

132. Wait for rate limit window to reset (default 15 minutes)
133. Adjust RATE_LIMIT_MAX_REQUESTS in .env for development
134. Check for automated scripts making excessive requests

Debug Mode

Enable debug logging by setting NODE_ENV to development:

```
NODE_ENV=development npm start
```

This enables console logging of:

- Registration and login attempts
- Database queries
- Token generation and verification

- Middleware execution

Database Inspection

View database contents for debugging:

```
mysql -u auth_user -p auth_service
```

```
# View all users
SELECT id, email, name, account_status, created_at FROM users;

# View specific user
SELECT * FROM users WHERE email = 'user@example.com';

# Count total users
SELECT COUNT(*) FROM users;

# Delete test user (if needed)
```

```
DELETE FROM users WHERE email = 'test@example.com';
```

References

135 Node.js Official Documentation - https://nodejs.org/en/docs/
136 Express.js Documentation - https://expressjs.com/
137 MySQL Documentation - https://dev.mysql.com/doc/
138 JWT (JSON Web Tokens) - https://jwt.io/
139 bcryptjs Documentation - https://github.com/dcodeIO/bcrypt.js
140 Helmet.js Security - https://helmetjs.github.io/
141 CORS Middleware - https://github.com/expressjs/cors
142 Express Rate Limit - https://github.com/nfriedly/express-rate-limit
143 Postman Documentation - https://learning.postman.com/
144 OWASP Authentication Cheat Sheet - https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html