Smart IoT Bolt for Pipelines

AccountManager Microservice Documentation

Date: March 30, 2025

Version: 1.0

Table of Contents

---

1. Introduction

The AccountManager microservice is a crucial component of the Smart IoT Bolt for Pipelines system. It handles user authentication and authorization, ensuring that only authorized personnel can access the system's monitoring and control features. This document provides a comprehensive explanation of the AccountManager's structure, functions, and integration within the broader system.

The AccountManager follows a microservice architecture pattern, leveraging CherryPy for REST API implementation. It employs JWT (JSON Web Tokens) for secure authentication and provides role-based access control to differentiate between administrators and regular users.

---

2. Architecture Overview

The AccountManager microservice follows a structured architecture with clear separation of concerns:

Copy

```
MS_AccountManager/
├── __init__.py
├── main.py                  # CherryPy server & API routes
├── .env                     # Environment variables
├── requirements.txt         # Dependencies
├── controllers/
│   ├── __init__.py
│   └── auth_controller.py   # Authentication endpoints
├── models/
│   ├── __init__.py
│   └── user.py              # User data model
├── services/
│   ├── __init__.py
│   └── auth_service.py      # Authentication logic
└── utils/
    ├── __init__.py
    └── token_utils.py       # JWT token handling
```

- **Controllers**: Handle HTTP requests and responses
- **Models**: Define data structures and storage
- **Services**: Implement business logic
- **Utils**: Provide utility functions

The service follows a layered architecture:

1. **Presentation Layer** (Controllers): Handles HTTP requests/responses
2. **Service Layer** (Services): Implements business logic
3. **Data Layer** (Models): Manages data persistence
4. **Utility Layer** (Utils): Provides supporting functions

---

3. Module Descriptions

3.1 Main Module

`main.py`

This is the entry point of the microservice. It configures the CherryPy server, registers with the Resource Catalog, and mounts the controllers.

Key Functions:

`AccountManagerService.__init__`

- **Purpose**: Initializes the service and registers with the Resource Catalog
- **Parameters**: None
- **Returns**: None
- **Behavior**: Creates a new instance of the service and calls `register_with_catalog()`

`AccountManagerService.register_with_catalog`

- **Purpose**: Registers the microservice with the Resource Catalog
- **Parameters**: None
- **Returns**: None
- **Behavior**:
  1. Creates a service info dictionary with name, endpoint, description, and status
  2. Sends a POST request to the Resource Catalog
  3. Logs success or failure

`if __name__ == "__main__"`

- **Purpose**: Entry point when script is run directly
- **Behavior**:
  1. Configures CherryPy server with host and port from environment variables
  2. Sets up JSON handling tools
  3. Mounts the AuthController at the '/auth' path
  4. Creates an instance of AccountManagerService
  5. Starts the CherryPy engine

3.2 Controllers

`controllers/auth_controller.py`

This module handles the HTTP endpoints for authentication and user management.

Key Functions:

`AuthController.__init__`

- **Purpose**: Initializes the controller with an instance of AuthService
- **Parameters**: None
- **Returns**: None
- **Behavior**: Creates an instance of AuthService for handling authentication logic

`AuthController.login`

- **Purpose**: Endpoint for user login
- **HTTP Method**: POST
- **Parameters**: None (takes username and password from request body)
- **Returns**: JSON with token and user information
- **Behavior**:
  1. Extracts username and password from request body
  2. Validates input presence
  3. Calls auth_service.authenticate_user()
  4. Returns JWT token and user info or appropriate error

`AuthController.verify`

- **Purpose**: Endpoint to verify a token
- **HTTP Method**: GET
- **Parameters**: token (query parameter)
- **Returns**: JSON with user information if token is valid
- **Behavior**:
  1. Extracts token from query parameters
  2. Calls auth_service.verify_token()
  3. Returns user info or appropriate error

`AuthController.register`

- **Purpose**: Endpoint for user registration (admin only)
- **HTTP Method**: POST
- **Parameters**: None (takes user data from request body and admin token from headers)
- **Returns**: JSON with registration status and user information
- **Behavior**:
  1. Extracts user data from request body
  2. Extracts and verifies admin token from Authorization header
  3. Checks admin privileges
  4. Validates input
  5. Calls auth_service.register_user()
  6. Returns registration result or appropriate error

3.3 Models

`models/user.py`

This module defines the User model and provides methods for user management.

Key Functions:

`User.initialize_admin`

- **Purpose**: Initialize admin user from environment variables
- **Parameters**: None
- **Returns**: None
- **Behavior**:
  1. Gets admin credentials from environment variables
  2. Checks if admin user already exists
  3. Creates admin user with hashed password if not exists
  4. Logs success

`User.create`

- **Purpose**: Create a new user
- **Parameters**:
  - username (str): User's username
  - password (str): User's password
  - role (str, optional): User's role, defaults to 'user'
- **Returns**: Tuple (success: bool, result: dict or str)
- **Behavior**:
  1. Checks if username already exists
  2. Hashes the password using bcrypt
  3. Stores user in _users dictionary
  4. Returns success status and user info or error message

`User.verify`

- **Purpose**: Verify user credentials
- **Parameters**:
  - username (str): User's username
  - password (str): User's password
- **Returns**: Tuple (success: bool, result: dict or str)
- **Behavior**:

1. Checks if username exists
2. Verifies password against stored hash using bcrypt
3. Returns success status and user info (without password) or error message

`User.get`

- **Purpose**: Get user by username
- **Parameters**:
  - username (str): User's username
- **Returns**: User information (dict) or None
- **Behavior**:
  1. Retrieves user from _users dictionary
  2. Returns user info (without password) or None if not found

3.4 Services

`services/auth_service.py`

This module implements the business logic for authentication and user management.

Key Functions:

`AuthService.authenticate_user`

- **Purpose**: Authenticate a user and return a JWT token
- **Parameters**:
  - username (str): User's username
  - password (str): User's password
- **Returns**: Dict with token and user information
- **Behavior**:
  1. Calls User.verify() to check credentials
  2. Generates JWT token using user information
  3. Returns token and user info or raises exception

`AuthService.verify_token`

- **Purpose**: Verify a JWT token and return user information
- **Parameters**:
  - token (str): JWT token
- **Returns**: User information (dict) or None

- **Behavior**:
  1. Calls verify_jwt() to decode and verify token
  2. Extracts username from payload
  3. Retrieves user information using User.get()
  4. Returns user info or None if invalid

`AuthService.register_user`

- **Purpose**: Register a new user
- **Parameters**:
  - username (str): User's username
  - password (str): User's password
  - role (str, optional): User's role, defaults to 'user'
- **Returns**: Dict with message and user information
- **Behavior**:
  1. Calls User.create() to create new user
  2. Returns success message and user info or raises exception

3.5 Utilities

`utils/token_utils.py`

This module provides utility functions for JWT token generation and verification.

Key Functions:

`generate_token`

- **Purpose**: Generate a JWT token for the user
- **Parameters**:
  - user_data (dict): User information to encode in token
- **Returns**: JWT token (str)
- **Behavior**:
  1. Creates payload with username, role, expiration, and issue time
  2. Encodes payload with JWT_SECRET using HS256 algorithm
  3. Returns encoded token

`verify_token`

- **Purpose**: Verify a JWT token and return the payload if valid

- **Parameters**:
  - token (str): JWT token to verify
- **Returns**: Token payload (dict) or None
- **Behavior**:
  1. Decodes token using JWT_SECRET and HS256 algorithm
  2. Handles ExpiredSignatureError and InvalidTokenError
  3. Returns payload if valid or None if invalid

---

4. Authentication Flow

The AccountManager implements a standard JWT-based authentication flow:

1. **Login Process**:
   - User submits username and password to `/auth/login`
   - System verifies credentials against stored hash
   - If valid, system generates JWT token and returns to user
   - Token contains encoded user information and expiration time
2. **Authentication Process**:
   - User includes token in requests (query parameter or Authorization header)
   - System verifies token validity and expiration
   - If valid, access is granted based on user's role
3. **User Registration**:
   - Admin authenticates and receives token
   - Admin submits new user details to `/auth/register` with admin token
   - System verifies admin token and creates new user
   - New user can then authenticate using their credentials

```
┌──────────┐     ┌─────────────┐     ┌───────────────┐
│  Client  │     │ Auth Service│     │ User Database │
└────┬─────┘     └──────┬──────┘     └───────┬───────┘
     │                  │                    │
     │  login request   │                    │
     │─────────────────>│                    │
     │                  │ verify credentials │
     │                  │───────────────────>│
     │                  │ credentials valid  │
     │                  │<───────────────────│
     │                  │                    │
     │                  │─┐ generate         │
     │                  │ │ JWT token        │
     │                  │<┘                  │
     │ token response   │                    │
     │<─────────────────│                    │
     │                  │                    │
     │ protected request│                    │
     │ with token       │                    │
     │─────────────────>│                    │
     │                  │─┐ verify           │
     │                  │ │ token            │
     │                  │<┘                  │
     │ protected resource                    │
     │<─────────────────│                    │
     │                  │                    │
```

---

5. API Endpoints

The AccountManager exposes the following REST API endpoints:

Login Endpoint

- **URL**: `/auth/login`
- **Method**: POST
- **Body**:

  ```json
  {
    "username": "string",
    "password": "string"
  }
  ```

- **Success Response**:

  ```json
  {
    "token": "string",
    "user": {
      "username": "string",
      "role": "string"
    }
  }
  ```

- **Error Responses**:
  - 400: Username and password are required
  - 401: Invalid username or password
  - 500: Internal server error

Verify Token Endpoint

- **URL**: `/auth/verify`
- **Method**: GET
- **Query Parameters**: `token=string`
- **Success Response**:

  ```json
  {
    "username": "string",
    "role": "string"
  }
  ```

- **Error Responses**:
  - 400: Token is required
  - 401: Invalid or expired token
  - 500: Internal server error

Register User Endpoint

- **URL**: `/auth/register`
- **Method**: POST
- **Headers**: `Authorization: Bearer <admin_token>`
- **Body**:

json

```json
{
  "username": "string",
  "password": "string",
  "role": "string"  // Optional, defaults to "user"
}
```

- **Success Response**:

json

```json
{
  "message": "User registered successfully",
  "user": {
    "username": "string",
    "role": "string"
  }
}
```

- **Error Responses**:
  - 400: Username and password are required
  - 401: Authentication required
  - 403: Admin privileges required
  - 500: Internal server error

---

6. Security Considerations

The AccountManager implements several security measures:

1. **Password Security**:
   - Passwords are hashed using bcrypt with salt
   - Original passwords are never stored
2. **Token Security**:
   - JWTs are signed with a secret key
   - Tokens have an expiration time
   - Verification checks signature and expiration
3. **Role-Based Access Control**:
   - Users are assigned roles (admin, user)
   - Certain operations (like user registration) require admin privileges
4. **Best Practices**:

- No hardcoded credentials (loaded from environment variables)
- Proper error handling to avoid information leakage
- Input validation to prevent injection attacks

5. **Current Limitations**:
   - In-memory user storage (not persistent across restarts)
   - Basic authentication (to be replaced with Firebase Authentication)

---

7. Integration with Smart IoT Bolt System

The AccountManager plays a crucial role in the Smart IoT Bolt system by:

1. **Securing User Interfaces**:
   - Provides authentication for the Web Dashboard
   - Authenticates Telegram Bot users before allowing control actions
2. **Service Integration**:
   - Registers with the Resource/Service Catalog for discovery
   - Gets configuration from the catalog to avoid hardcoding
3. **Interface with Other Components**:
   - Web Dashboard uses AccountManager for login and session management
   - Telegram Bot verifies user identity before forwarding commands to Control Center
   - Control Center can verify token validity for sensitive operations

Integration flow:

Copy

```
┌───────────────┐     ┌─────────────────┐     ┌────────────────┐
│ Web Dashboard │     │ Account Manager │     │ Control Center │
└───────────────┘     └─────────────────┘     └────────────────┘
        │                      │                      │
        │ User Login Request   │                      │
        │─────────────────────>│                      │
        │ JWT Token            │                      │
        │<─────────────────────│                      │
        │                      │                      │
        │ Protected Operation  │                      │
        │ with JWT Token       │                      │
        │────────────────────────────────────────────>│
        │                      │                      │
        │                      │ Verify Token         │
        │                      │<─────────────────────│
        │                      │ Token Valid/Invalid  │
        │                      │─────────────────────>│
        │                      │                      │──┐
        │                      │                      │  │ Process
        │                      │                      │  │ Operation
        │                      │                      │<─┘
        │ Operation Result     │                      │
        │<────────────────────────────────────────────│
        │                      │                      │
```

---

8. Future Improvements

While the current implementation meets the core requirements, several enhancements are planned:

1. **Database Integration**:
   - Replace in-memory user storage with a persistent database
   - Consider MongoDB or PostgreSQL for user management
2. **Advanced Authentication**:
   - Implement Firebase Authentication as specified in requirements
   - Add support for OAuth 2.0 providers
3. **Enhanced Security**:
   - Implement rate limiting to prevent brute force attacks
   - Add two-factor authentication for critical operations
   - Implement token refresh mechanism
4. **Additional Features**:

- Password reset functionality
- User profile management
- Session management (logout, view active sessions)
- Audit logging for security events

5. **Performance Optimizations**:
    - Token caching to reduce verification overhead
    - Connection pooling for database interactions