**Prompt to generate pseudo code from technical document:**

You are a software architect tasked with creating detailed pseudo code for an admin authentication module in a banking application.

\*\*Context:\*\* This is a FastAPI-based banking application with admin user management functionality.

\*\*Task:\*\* Generate comprehensive pseudo code for the admin authentication module based on the following specifications:

\*\*Module Requirements:\*\*

- File: `Backend/app/routes/admin.py`

- Framework: FastAPI with async/await support

- Database: SQLAlchemy with async sessions

- Authentication: JWT tokens with bcrypt password hashing

\*\*Functional Specifications:\*\*

1. \*\*Admin Registration Endpoint:\*\*

- Route: POST /admin/register

- Purpose: Create new admin accounts

- Input validation: username (3-100 chars), password (8-128 chars), email (valid email format)

- Business rules: username and email must be unique

- Security: password hashing with bcrypt

- Response: admin\_id, username, email (HTTP 201)

- Error handling: HTTP 409 for duplicate username/email

2. \*\*Admin Login Endpoint:\*\*

- Route: POST /admin/login

- Purpose: Authenticate admin users

- Input: identifier (username or email), password

- Authentication flow: try username first, then email lookup

- Security: password verification against stored hash

- Response: JWT access token with user details (HTTP 200)

- Error handling: HTTP 401 for invalid credentials

\*\*Database Schema:\*\*

- Table: admin

- Fields: admin\_id (UUID, PK), username (String, unique), email (String, unique), password\_hash (String), created\_at (DateTime)

\*\*Security Requirements:\*\*

- Password hashing using bcrypt

- JWT token generation with configurable expiration

- Input sanitization (trimming whitespace, email normalization)

- Generic error messages for security

- Unique constraints at database level

\*\*Dependencies:\*\*

- FastAPI components (APIRouter, Depends, HTTPException, status)

- SQLAlchemy async session

- Password utilities (hash\_password, verify\_password)

- JWT utilities (create\_access\_token)

- Admin CRUD operations

- Pydantic schemas for validation

\*\*Generate pseudo code that includes:\*\*

1. \*\*Module Structure:\*\*

- Import statements and dependencies

- Router configuration with prefix "/admin" and tags ["admin"]

2. \*\*Registration Endpoint Logic:\*\*

- Input sanitization (username trimming, email lowercase)

- Duplicate checking (username and email uniqueness)

- Password hashing process

- Database record creation

- Response generation

- Error handling for conflicts

3. \*\*Login Endpoint Logic:\*\*

- Identifier processing (username/email detection)

- Database lookup strategy (username first, then email)

- Password verification process

- JWT token generation with claims

- Response formatting

- Authentication failure handling

4. \*\*Error Handling Strategy:\*\*

- HTTP status code mappings

- Security-conscious error messages

- Exception handling patterns

5. \*\*Security Implementation:\*\*

- Input validation and sanitization

- Password security measures

- Token generation details

- Data protection considerations

\*\*Format the pseudo code as:\*\*

PSEUDO CODE FOR ADMIN AUTHENTICATION MODULEPURPOSE: [Brief description]

IMPORTS:[Required dependencies]

ROUTER SETUP:[Router configuration]

ENDPOINT 1: POST /admin/register[Detailed implementation steps]

ENDPOINT 2: POST /admin/login[Detailed implementation steps]

ERROR HANDLING:[Exception handling strategy]

FEATURES:[Security implementation details]

\*\*Requirements:\*\*

- Use clear, step-by-step pseudo code syntax

- Include all technical implementation details

- Specify data validation and sanitization steps

- Detail security measures and error handling

- Make it comprehensive enough for direct implementation

- Follow FastAPI and async programming patterns

- Include proper type hints and response models

Generate the complete pseudo code following this structure and requirements.

**Generated Pseudo Code:**

PSEUDO CODE FOR ADMIN ROUTES MODULE

MODULE: admin\_routes

PURPOSE: FastAPI router for admin authentication endpoints

IMPORTS:

- FastAPI components: APIRouter, Depends, HTTPException, status

- Database: AsyncSession from sqlalchemy.ext.asyncio

- Local modules: get\_db, admin schemas, password utilities, JWT utilities, admin CRUD operations

ROUTER SETUP:

- Create APIRouter with prefix "/admin" and tags ["admin"]

ENDPOINT 1: POST /admin/register

INPUT: AdminRegisterIn payload, database session

OUTPUT: AdminOut response, HTTP 201 status

LOGIC:

1. Extract and sanitize username (strip whitespace)

2. Extract and sanitize email (lowercase and strip whitespace)

3. Check if username already exists in database

- If exists: raise HTTP 409 "Username already exists"

4. Check if email already exists in database

- If exists: raise HTTP 409 "Email already exists"

5. Hash the provided password

6. Create new admin record in database with username, email, and hashed password

7. Return AdminOut with admin\_id (as string), username, and email

ENDPOINT 2: POST /admin/login

INPUT: AdminLoginIn payload, database session

OUTPUT: TokenOut response

LOGIC:

1. Extract and sanitize identifier (username or email)

2. Try to find admin by username first

3. If not found, try to find admin by email (lowercase)

4. If admin not found: raise HTTP 401 "Invalid credentials"

5. Verify provided password against stored password hash

- If invalid: raise HTTP 401 "Invalid credentials"

6. Create JWT access token with admin\_id as subject and username/email as extra claims

7. Return TokenOut with access\_token, userId, userName, and userEmailid

ERROR HANDLING:

- Username conflicts: HTTP 409 with descriptive message

- Email conflicts: HTTP 409 with descriptive message

- Invalid credentials: HTTP 401 with generic message for security

- All database operations are async and use dependency injection

SECURITY FEATURES:

- Password hashing using bcrypt

- JWT token generation with configurable expiration

- Input sanitization (trimming whitespace, email normalization)

- Generic error messages for authentication failures

- Unique constraints on username and email at database level

**Prompt to generate code from pseudo code:**

Create a FastAPI admin authentication module with the following exact specifications:

\*\*File Structure:\*\*

- File: `Backend/app/routes/admin.py`

- Router prefix: "/admin" with tags ["admin"]

\*\*Dependencies Required:\*\*

- FastAPI: APIRouter, Depends, HTTPException, status

- SQLAlchemy: AsyncSession from sqlalchemy.ext.asyncio

- Local imports: get\_db, admin schemas, password utilities, JWT utilities, admin CRUD operations

\*\*Schema Requirements:\*\*

- AdminRegisterIn: userName (str, 3-100 chars), password (str, 8-128 chars), userEmail (EmailStr)

- AdminOut: admin\_id (str), userName (str), userEmail (EmailStr)

- AdminLoginIn: identifier (str, accepts username or email), password (str, 8-128 chars)

- TokenOut: access\_token (str), token\_type (str, default "bearer"), userId (str), userName (str), userEmailid (EmailStr)

\*\*Database Model (Admin):\*\*

- admin\_id: UUID primary key

- username: String(100), unique, not null

- username: String(255), unique, not null

- password\_hash: String(255), not null

- created\_at: DateTime with timezone

\*\*CRUD Functions Needed:\*\*

- get\_admin\_by\_username(db, username) -> Admin | None

- get\_admin\_by\_email(db, email) -> Admin | None

- create\_admin(db, username, email, password\_hash) -> Admin

\*\*Password Security:\*\*

- hash\_password(plain: str) -> str (using bcrypt)

- verify\_password(plain: str, hashed: str) -> bool

\*\*JWT Functions:\*\*

- create\_access\_token(subject: str, extra: Dict[str, Any] | None = None) -> str

\*\*Endpoint 1: POST /admin/register\*\*

- Response model: AdminOut, status 201

- Input validation: strip username, lowercase and strip email

- Duplicate checks: username and email uniqueness

- Error handling: HTTP 409 for conflicts

- Return: admin\_id as string, username, email

\*\*Endpoint 2: POST /admin/login\*\*

- Response model: TokenOut

- Authentication: try username first, then email (lowercase)

- Password verification: compare with stored hash

- Error handling: HTTP 401 for invalid credentials

- JWT creation: subject=admin\_id, extra={username, email}

- Return: access\_token, userId, userName, userEmailid

\*\*Code Style Requirements:\*\*

- Use async/await for all database operations

- Dependency injection for database session

- Proper error handling with specific HTTP status codes

- Input sanitization (trimming, case normalization)

- Generic error messages for security (no specific failure reasons)

- Type hints for all function parameters and return types

- Follow FastAPI best practices for route definitions

\*\*Security Considerations:\*\*

- Password hashing with bcrypt

- JWT tokens with configurable expiration

- Input validation and sanitization

- Unique constraints at database level

- Generic authentication error messages

Generate the complete admin.py file with exact imports, router setup, and both endpoints following these specifications precisely.

**Generated Code:**

*from* fastapi *import* APIRouter, Depends, HTTPException, status

*from* sqlalchemy.ext.asyncio *import* AsyncSession

*from* app.db *import* get\_db

*from* app.schemas.admin *import* AdminRegisterIn, AdminOut, AdminLoginIn, TokenOut

*from* app.security.password *import* hash\_password, verify\_password

*from* app.security.jwt *import* create\_access\_token

*from* app.crud.admin *import* get\_admin\_by\_username, get\_admin\_by\_email, create\_admin

router = APIRouter(*prefix*="/admin", *tags*=["admin"])

@router.post("/register", *response\_model*=AdminOut, *status\_code*=status.HTTP\_201\_CREATED)

async def register\_admin(*payload*: AdminRegisterIn, *db*: AsyncSession = Depends(get\_db)):

    username = payload.userName.strip()

    email = payload.userEmail.lower().strip()

*if* *await* get\_admin\_by\_username(db, username):

*raise* HTTPException(*status\_code*=409, *detail*="Username already exists")

*if* *await* get\_admin\_by\_email(db, email):

*raise* HTTPException(*status\_code*=409, *detail*="Email already exists")

    admin = *await* create\_admin(db, *username*=username, *email*=email, *password\_hash*=hash\_password(payload.password))

*return* AdminOut(*admin\_id*=str(admin.admin\_id), *userName*=admin.username, *userEmail*=admin.email)

@router.post("/login", *response\_model*=TokenOut)

async def login\_admin(*payload*: AdminLoginIn, *db*: AsyncSession = Depends(get\_db)):

    identifier = payload.identifier.strip()

*# Try username first, then email*

    admin = *await* get\_admin\_by\_username(db, identifier)

*if* not admin:

        admin = *await* get\_admin\_by\_email(db, identifier.lower())

*if* not admin:

*raise* HTTPException(*status\_code*=status.HTTP\_401\_UNAUTHORIZED, *detail*="Invalid credentials")

*if* not verify\_password(payload.password, admin.password\_hash):

*raise* HTTPException(*status\_code*=status.HTTP\_401\_UNAUTHORIZED, *detail*="Invalid credentials")

*# Subject is admin\_id (UUID string)*

    token = create\_access\_token(*subject*=str(admin.admin\_id), *extra*={"username": admin.username, "email": admin.email})

*return* TokenOut(

*access\_token*=token,

*userId*=str(admin.admin\_id),

*userName*=admin.username,

*userEmailid*=admin.email,

    )