05 FastAPI - Auth

Introduction to JWT Authentication

Authentication is a critical component of web applications, ensuring that only authorized users can access protected resources. JSON Web Tokens (JWT) offer a modern, stateless approach to authentication that works particularly well with FastAPI's asynchronous architecture.

What is JWT?

A JSON Web Token (JWT) is a compact, URL-safe means of representing claims between two parties. It consists of three parts:

1. **Header**: Contains metadata about the token, like the algorithm used for signing
2. **Payload**: Contains the actual data (claims) being transmitted
3. **Signature**: Ensures the token hasn't been tampered with

These three parts are encoded and concatenated with dots: `header.payload.signature`

Setting Up Our Project

Let's start by setting up our environment and installing the necessary dependencies:

```
# Install required packages
# pip install fastapi uvicorn python-jose[cryptography] passlib[bcrypt] python-multipart

# Import necessary libraries
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from datetime import datetime, timedelta
from pydantic import BaseModel
from typing import Optional

# Initialize FastAPI app
app = FastAPI(title="JWT Authentication Demo")
```

Creating Our User Models

Let's define Pydantic models to represent our users:

```
# Define user models
class User(BaseModel):
    username: str
    email: Optional[str] = None
    full_name: Optional[str] = None
    disabled: Optional[bool] = None

class UserInDB(User):
    hashed_password: str
```

```
# For demo purposes, we'll use a simple in-memory user database
# In a real application, you would use a proper database
fake_users_db = {
    "johndoe": {
        "username": "johndoe",
        "full_name": "John Doe",
        "email": "johndoe@example.com",
        "hashed_password":
"$2b$12$EixZaYVK1fsbw1ZfbX3OXePaWxn96p36WQoeG6Lruj3vjPGga31lW",
        "disabled": False,
    }
}
```

Password Handling and Security

We need to securely handle passwords, which means we'll hash them rather than storing them in plain text:

```
# Set up password hashing functionality
pwd_context = CryptContext(schemes=["bcrypt"])

# OAuth2 configuration with password flow
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

# Verify password
def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

# Generate password hash
def get_password_hash(password):
    return pwd_context.hash(password)
```

Setting Up JWT Configuration

Now, let's define our JWT settings:

```
# JWT configuration
SECRET_KEY = "YOUR_SECRET_KEY_HERE"  # In production, use a proper secret key
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30  # Token expiration time

# Token model
class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    username: Optional[str] = None
```

User Authentication Functions

Let's create functions to authenticate users and create JWT tokens:

```python
# Get user from database
def get_user(db, username: str):
    if username in db:
        user_dict = db[username]
        return UserInDB(**user_dict)
    return None

# Authenticate user
def authenticate_user(fake_db, username: str, password: str):
    user = get_user(fake_db, username)
    if not user:
        return False
    if not verify_password(password, user.hashed_password):
        return False
    return user

# Create access token
def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    to_encode = data.copy()

    # Set expiration time
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)

    to_encode.update({"exp": expire})

    # Create JWT token
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

User Dependency for Protected Routes

Now, let's create a dependency that will verify tokens and extract the current user:

```python
# Get current user from token
async def get_current_user(token: str = Depends(oauth2_scheme)):
    # Define credential exception
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        # Decode JWT token
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")

        if username is None:
            raise credentials_exception
```

```
        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception

    # Get user from database
    user = get_user(fake_users_db, username=token_data.username)

    if user is None:
        raise credentials_exception

    return user

# Get active user (an additional check for account status)
async def get_current_active_user(current_user: User = Depends(get_current_user)):
    if current_user.disabled:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user
```

Implementing API Endpoints

Now, let's implement our API endpoints, including the token endpoint for login and a protected endpoint:

```
# Token endpoint for user login
@app.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm =
Depends()):
    # Authenticate the user
    user = authenticate_user(fake_users_db, form_data.username,
form_data.password)

    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # Create access token with expiration time
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username}, expires_delta=access_token_expires
    )

    # Return token
    return {"access_token": access_token, "token_type": "bearer"}

# Protected route - requires valid token
@app.get("/users/me/", response_model=User)
async def read_users_me(current_user: User = Depends(get_current_active_user)):
    return current_user
```

```
# Public route - no authentication required
@app.get("/")
def read_root():
    return {"message": "Welcome to the JWT Authentication API"}
```

Complete Application

Here's what our complete FastAPI application looks like when all components are brought together:

```
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from datetime import datetime, timedelta
from pydantic import BaseModel
from typing import Optional

# Initialize FastAPI app
app = FastAPI(title="JWT Authentication Demo")

# Define user models
class User(BaseModel):
    username: str
    email: Optional[str] = None
    full_name: Optional[str] = None
    disabled: Optional[bool] = None

class UserInDB(User):
    hashed_password: str

# For demo purposes, we'll use a simple in-memory user database
# In a real application, you would use a proper database
fake_users_db = {
    "johndoe": {
        "username": "johndoe",
        "full_name": "John Doe",
        "email": "johndoe@example.com",
        "hashed_password":
"$2b$12$EixZaYVK1fsbw1ZfbX3OXePaWxn96p36WQoeG6Lruj3vjPGga31lW",  # "secret"
        "disabled": False,
    }
}

# Set up password hashing functionality
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# OAuth2 configuration with password flow
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

# JWT configuration
SECRET_KEY = "09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"  #
Use your own secret key in production
ALGORITHM = "HS256"
```

```python
ACCESS_TOKEN_EXPIRE_MINUTES = 30  # Token expiration time

# Token model
class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    username: Optional[str] = None

# Verify password
def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

# Generate password hash
def get_password_hash(password):
    return pwd_context.hash(password)

# Get user from database
def get_user(db, username: str):
    if username in db:
        user_dict = db[username]
        return UserInDB(**user_dict)
    return None

# Authenticate user
def authenticate_user(fake_db, username: str, password: str):
    user = get_user(fake_db, username)
    if not user:
        return False
    if not verify_password(password, user.hashed_password):
        return False
    return user

# Create access token
def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    to_encode = data.copy()

    # Set expiration time
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)

    to_encode.update({"exp": expire})

    # Create JWT token
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

# Get current user from token
async def get_current_user(token: str = Depends(oauth2_scheme)):
```

```
    # Define credential exception
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        # Decode JWT token
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")

        if username is None:
            raise credentials_exception

        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception

    # Get user from database
    user = get_user(fake_users_db, username=token_data.username)

    if user is None:
        raise credentials_exception

    return user

# Get active user (an additional check for account status)
async def get_current_active_user(current_user: User = Depends(get_current_user)):
    if current_user.disabled:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user

# Token endpoint for user login
@app.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm =
Depends()):
    # Authenticate the user
    user = authenticate_user(fake_users_db, form_data.username,
form_data.password)

    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # Create access token with expiration time
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username}, expires_delta=access_token_expires
```

```
    )

    # Return token
    return {"access_token": access_token, "token_type": "bearer"}

# Protected route - requires valid token
@app.get("/users/me/", response_model=User)
async def read_users_me(current_user: User = Depends(get_current_active_user)):
    return current_user

# Another protected route that returns additional user information
@app.get("/users/me/items/")
async def read_own_items(current_user: User = Depends(get_current_active_user)):
    return [{"item_id": "Foo", "owner": current_user.username}]

# Public route - no authentication required
@app.get("/")
def read_root():
    return {"message": "Welcome to the JWT Authentication API"}

# Route to add a new user (simplified for demo)
@app.post("/users/", response_model=User)
async def create_user(username: str, password: str, email: str = None, full_name:
str = None):
    if username in fake_users_db:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Username already registered"
        )

    hashed_password = get_password_hash(password)
    fake_users_db[username] = {
        "username": username,
        "email": email,
        "full_name": full_name,
        "hashed_password": hashed_password,
        "disabled": False
    }

    return {
        "username": username,
        "email": email,
        "full_name": full_name,
        "disabled": False
    }

# Run the FastAPI app with: uvicorn main:app --reload
```

Step-by-Step Explanation of the Authentication Flow

Let's break down the JWT authentication flow in our FastAPI application:

1. User Registration

Though simplified in our example, a typical application would have a registration endpoint where users provide credentials, which are then securely stored with the password hashed.

2. User Login (Token Generation)

1. The user sends their username and password to the `/token` endpoint
2. The system verifies the credentials against the stored data
3. If valid, the system creates a JWT containing the user's identity ("sub" claim)
4. The token includes an expiration time for security
5. The system returns the token to the user

3. Accessing Protected Resources

1. For subsequent requests to protected endpoints, the user includes the JWT in the Authorization header (`Bearer <token>`)
2. The `get_current_user` dependency:
   - Extracts the token from the request
   - Verifies the token's signature using the secret key
   - Checks that the token hasn't expired
   - Retrieves the user identity from the token
   - Fetches the complete user data
3. If all checks pass, the endpoint function receives the user data and processes the request
4. If any check fails, the user receives a 401 Unauthorized error

Best Practices for JWT Authentication

1. **Keep your SECRET_KEY secure**: Never commit it to version control. Use environment variables in production.
2. **Set appropriate token expiration**: Short-lived tokens (15-30 minutes) provide better security.
3. **Use HTTPS**: Always use HTTPS in production to encrypt tokens in transit.
4. **Consider using refresh tokens**: For long-lived sessions, implement a refresh token system.
5. **Store tokens securely**: On the client side, store tokens in secure, HTTP-only cookies or local storage with proper security measures.
6. **Implement token revocation**: In critical systems, maintain a blacklist of revoked tokens.

Testing Your JWT Authentication

You can test your API using FastAPI's built-in Swagger UI, which is available at `/docs` when you run your application:

1. Start your application with:

```
uvicorn main:app --reload
```

2. Open your browser and navigate to `http://127.0.0.1:8000/docs`
3. Click on the `/token` endpoint, then "Try it out"

4. Enter the test credentials (username: "johndoe", password: "secret")
5. You'll receive a token that you can use to access protected endpoints
6. Click on one of the protected endpoints, like `/users/me/`
7. Click "Authorize" at the top of the page and enter your token
8. Now you can try out the protected endpoint

Conclusion

JWT authentication provides a secure, stateless way to handle user authentication in FastAPI applications. By following this guide, you've learned how to:

- Set up JWT authentication in FastAPI
- Create and verify JWTs
- Protect routes using dependencies
- Implement user authentication flows

This implementation provides a solid foundation that you can build upon for your specific application needs. Remember that security is critical, so always follow best practices and keep your dependencies updated.

Is there any specific part of the JWT authentication flow you'd like me to explain in more detail?