Authentication and Authorization

Overview

Authentication and authorization are two fundamental concepts in security that ensure only authorized users can access protected resources. In this lecture, we will explore the basics of authentication and authorization, their importance, and how to implement them in FastAPI.

Authentication

Authentication is the process of verifying the identity of a user or entity. It ensures that the user is who they claim to be. There are several types of authentication, including:

- *Username-Password Authentication*: This is the most common type of authentication. Users provide a username and password to access a system.
- *Token-Based Authentication*: Users receive a token after authentication, which is used for subsequent requests.
- *OAuth*: A standardized authorization framework that allows users to grant third-party applications limited access to their resources.

Authorization

Authorization is the process of determining what actions an authenticated user can perform. It ensures that users can only access resources they are allowed to access. There are several types of authorization, including:

- *Role-Based Access Control (RBAC)*: Users are assigned roles, and access is determined based on these roles.
- *Attribute-Based Access Control (ABAC)*: Access is determined based on user attributes, such as department or job function.

Importance of Authentication and Authorization

Authentication and authorization are crucial for protecting sensitive data and ensuring the security of a system. Without proper authentication and authorization, unauthorized users may be able to access sensitive data or perform malicious actions.

FastAPI Implementation

FastAPI provides several tools and libraries to implement authentication and authorization. Here is a simple example of authentication and authorization using FastAPI and JWT tokens:

```
from fastapi import FastAPI, HTTPException, Depends
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import jwt
from pydantic import BaseModel

app = FastAPI()

SECRET_KEY = "secret-key"

# Define a token URL
token_url = "/token"
```

```python
# Define a user model
class User(BaseModel):
    username: str
    password: str

# Define a token model
class Token(BaseModel):
    access_token: str

# Create a dictionary to store users
users = {
    "user1": "password1",
    "user2": "password2"
}

# Create a function to authenticate users
def authenticate_user(username: str, password: str):
    if username in users and users[username] == password:
        return True
    return False

# Create a function to generate JWT tokens
def generate_token(username: str):
    payload = {"sub": username}
    token = jwt.encode(payload, SECRET_KEY, algorithm="HS256")
    return token

# Create a route to obtain a token
@app.post(token_url, response_model=Token)
async def login(user: User):
    if authenticate_user(user.username, user.password):
        token = generate_token(user.username)
        return {"access_token": token}
    raise HTTPException(status_code=401, detail="Invalid username or password")

# Create a function to verify JWT tokens
def verify_token(token: str):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
        return payload["sub"]
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token has expired")
    except jwt.InvalidTokenError:
        raise HTTPException(status_code=401, detail="Invalid token")

# Create a dependency to verify tokens
async def get_current_user(token: HTTPAuthorizationCredentials = Depends(HTTPBearer())):
    username = verify_token(token.credentials)
    return username

# Create a protected route
@app.get("/protected")
async def protected_route(username: str = Depends(get_current_user)):
    return {"message": f"Hello, {username}!"}
```

This example demonstrates:

- ***Authentication***: Users can obtain a JWT token by providing a valid username and password.
- ***Authorization***: The `get_current_user` dependency verifies the JWT token and retrieves the username, which is then used to authorize access to protected routes.

Role-Based Access Control (RBAC) Example

To implement RBAC, you can modify the `User` model to include roles and create a dependency to verify roles:

Role-Based Access Control (RBAC) Example

```
from fastapi import FastAPI, HTTPException, Depends
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import jwt
from pydantic import BaseModel

app = FastAPI()

# Define a secret key for JWT
SECRET_KEY = "secret-key"

# Define a token URL
token_url = "/token"

# Define a user model with roles
class User(BaseModel):
    username: str
    password: str
    role: str

# Define a token model
class Token(BaseModel):
    access_token: str

# Create a dictionary to store users with roles
users = {
    "user1": {"password": "password1", "role": "admin"},
    "user2": {"password": "password2", "role": "user"}
}

# Create a function to authenticate users
def authenticate_user(username: str, password: str):
    if username in users and users[username]["password"] == password:
        return True
    return False

# Create a function to generate JWT tokens
def generate_token(username: str):
    payload = {"sub": username}
    token = jwt.encode(payload, SECRET_KEY, algorithm="HS256")
    return token

# Create a route to obtain a token
@app.post(token_url, response_model=Token)
async def login(user: User):
```

```
    if authenticate_user(user.username, user.password):
        token = generate_token(user.username)
        return {"access_token": token}
    raise HTTPException(status_code=401, detail="Invalid username or password")

# Create a function to verify JWT tokens
def verify_token(token: str):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
        return payload["sub"]
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token has expired")
    except jwt.InvalidTokenError:
        raise HTTPException(status_code=401, detail="Invalid token")

# Create a dependency to verify tokens
async def get_current_user(token: HTTPAuthorizationCredentials = Depends(HTTPBearer())):
    username = verify_token(token.credentials)
    return username

# Create a dependency to verify roles
async def get_current_active_admin(username: str = Depends(get_current_user)):
    if users[username]["role"] != "admin":
        raise HTTPException(status_code=403, detail="Forbidden")
    return username

# Create a protected route for admins
@app.get("/admin-only")
async def admin_only_route(username: str = Depends(get_current_active_admin)):
    return {"message": f"Hello, admin {username}!"}

# Create a protected route for all users
@app.get("/user-only")
async def user_only_route(username: str = Depends(get_current_user)):
    return {"message": f"Hello, {username}!"}
```

This example demonstrates:

- *Role-Based Access Control (RBAC)*: Users are assigned roles, and access is determined based on these roles.
- *Protected Routes*: Routes are protected by dependencies that verify the user's role.

Best Practices for Authentication and Authorization

1. *Use Secure Password Storage*: Store passwords securely using a strong hashing algorithm like bcrypt or Argon2.
2. *Use HTTPS*: Use HTTPS to encrypt data in transit and prevent eavesdropping.
3. *Validate User Input*: Validate user input to prevent SQL injection and cross-site scripting (XSS) attacks.
4. *Implement Rate Limiting*: Implement rate limiting to prevent brute-force attacks.
5. *Use Secure Tokens*: Use secure tokens like JWT to authenticate users and authorize access to protected routes.