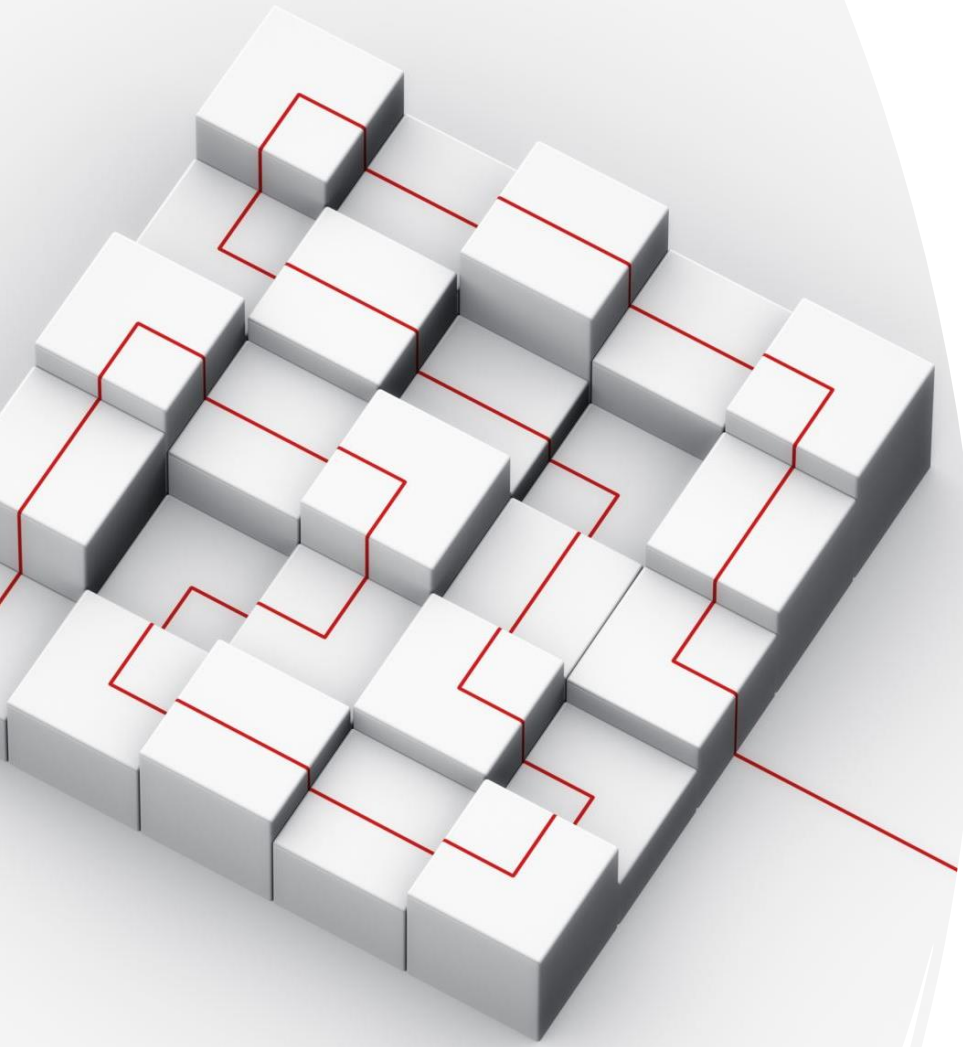
INT 161

Basic Backend Development

RESTful API

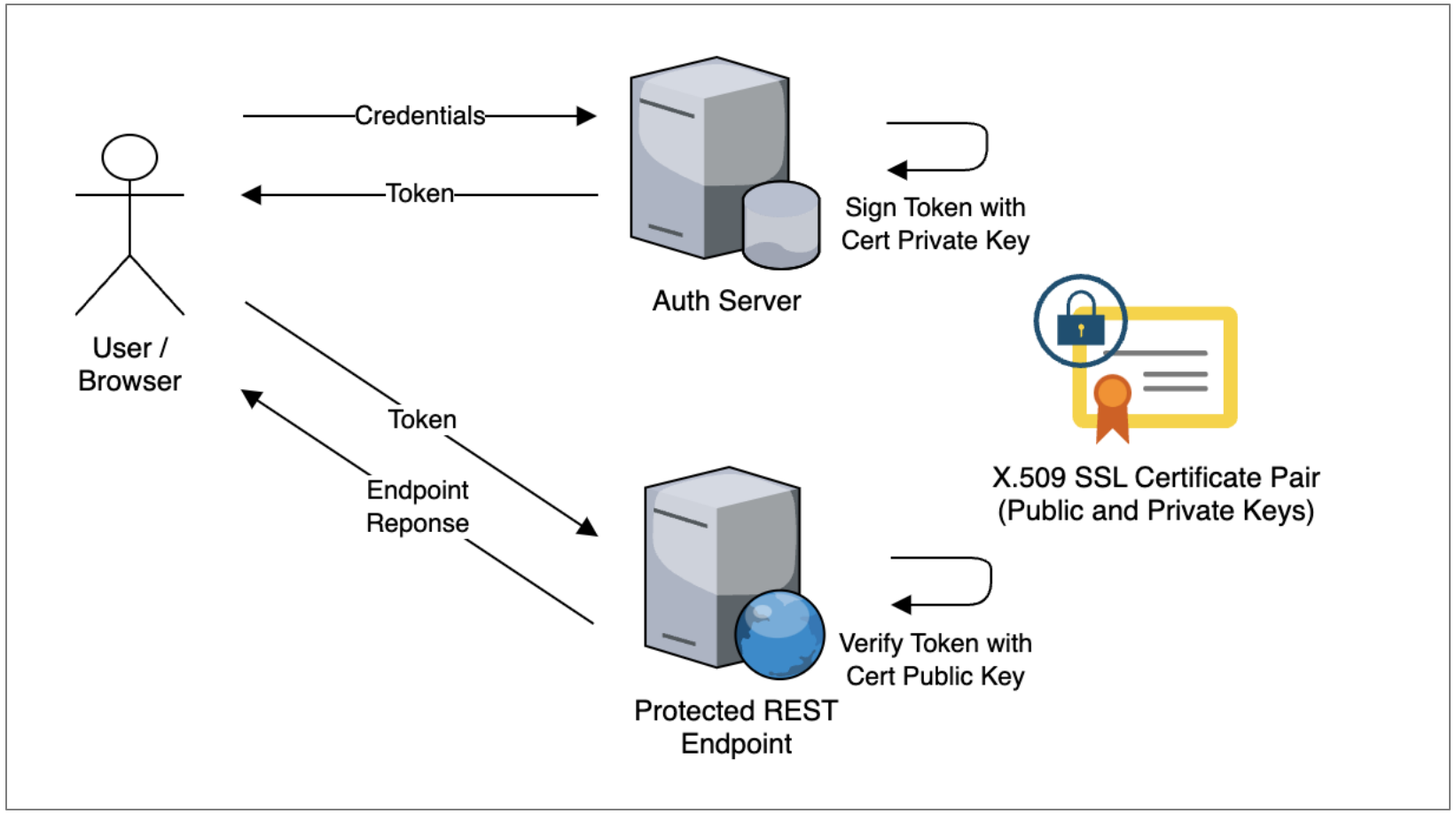
JWT-based Authentication



Unit Objectives

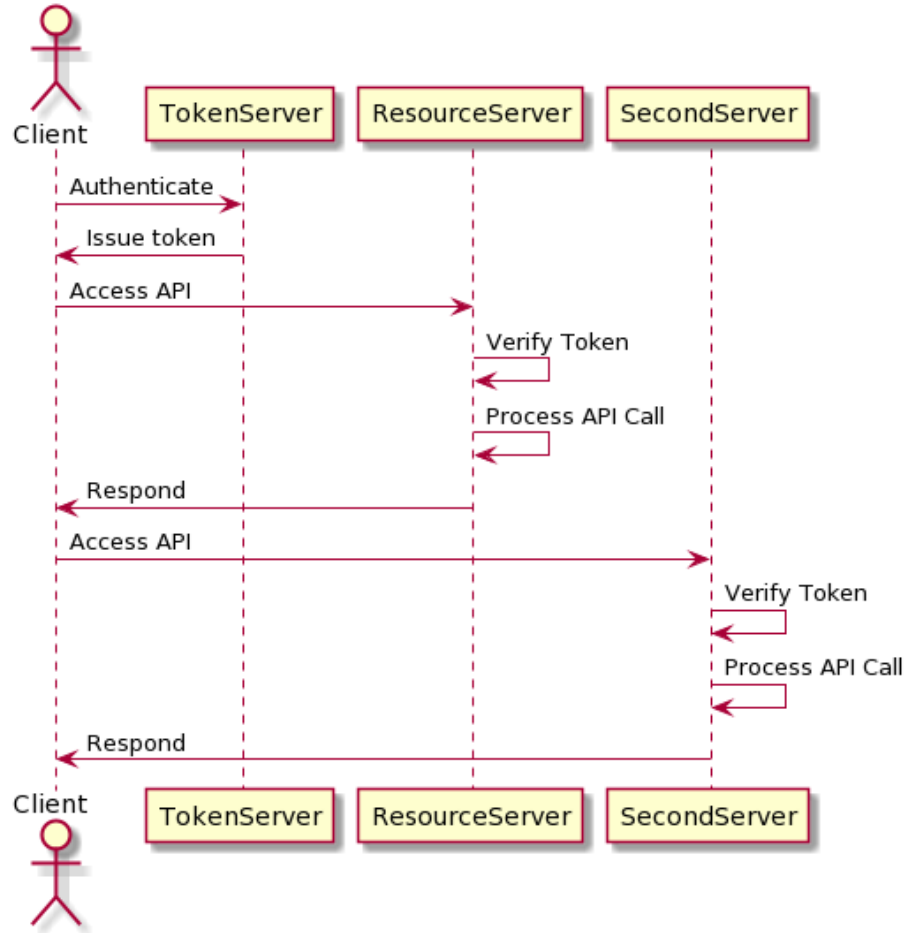
- After completing this unit, you should be able to:
  - Understand JWT Fundamentals
  - Apply in Node.js + Express

JWT-based Authentication



Session based vs Token based

- Scalability/Micro Service
  - A session often lives on a server or a cluster of servers.
- Some data that needs to be remembered across various parts of a website.



Register User flow

- User submits registration form (Send request to API with email, password, name
- Server validates & creates user record (active = 0)
- Create verification token (JWT or UUID) and sends verification email with link: https://your-app.com/verify-email?token=<verification_token>
- User clicks verification link, request to endpoint /api/auth/verify?token=...
- Server verifies the token then update active = 1
- Server responds success

Create table user in sakila db

```
create table sakila.user
(
    id          smallint auto_increment primary key,
    username    varchar(40)   not null,
    email       varchar(100)  not null,
    role        enum ('ADMIN', 'MANAGER', 'USER') default 'USER'        not null,
    created_on timestamp                          default CURRENT_TIMESTAMP not null,
    updated_on timestamp                          default CURRENT_TIMESTAMP not null,
    password    char(98)  not null,
    active      tinyint(1)                        default 0               not null,
    constraint user_email_uindex unique (email)
);
```

Argon2: Password Hashing

- In modern information technology systems, secure storage and transmission of personal and sensitive data are recognized as important tasks.
- These requirements are achieved through secure and robust encryption methods.
- Argon2 is an advanced cryptographic algorithm that emerged as the winner in the Password Hashing Competition (PHC), offering a concrete and secure measure.
- Argon2 also provides a secure mechanism against side-channel attacks and cracking attacks using parallel processing (e.g., GPU).

Argon2's Cryptographic Password Hashing Parameters

- The input parameters used in the Argon2 algorithm can be summarized as follows:
  - **Salt**: A randomly generated value. The salt is added to each user's password to enhance the encryption process. This ensures that users with the same password do not have the same hash value. (**128** or 64 bits)
  - **Hash Length**: This parameter depends on the intended usage. The Argon2 algorithm authors claim that a value of **128 bits** should be sufficient for most applications.
  - **Memory:** The memory used by the algorithm. To make hash cracking more expensive for an attacker, you want to make this value as high as possible.
  - **Parallelism**: A parameter that determines the number of threads or tasks processed concurrently. A higher parallelism degree allows for more simultaneous processing, resulting in faster encryption. (**Twice the amount of available CPU cores**)
  - **Iterations**: The number of iterations over the memory. The execution time correlates linearly with this parameter. It allows you to increase the computational cost required to calculate one hash.

Using argon2 with Express.js

> npm install argon2

./tests/test-argon2.js

```
import argon2 from 'argon2';
async function hashUserPassword(plainPassword) {
    try {
        const hash = await argon2.hash(plainPassword, {
            type: argon2.argon2id, // Recommended type for general use
            memoryCost: 102400,    // Adjust memory usage in KB
            timeCost: 3,           // Adjust the number of iterations
            parallelism: 1         // Adjust based on your CPU cores
        });
        return hash;
    } catch (error) {
        console.error("Password hashing failed:", error);
        throw error;
    }
}
const hashPassword = await hashUserPassword("1234abCD!");
console.log('valid password: ',await argon2.verify(hashPassword,"1234abCD!"));
console.log(hashPassword, hashPassword.length);
```

> node tests/test-argon2.js

What is JSON Web Token?

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- This information can be verified and trusted because it is digitally signed.
- JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.
- Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens.
- Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties.
- When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

Signing vs Encryption

- Signing ensures integrity - verifies authenticity and data consistency.
- Encryption ensures confidentiality - keeps data hidden.
- Common algorithms:
  - HMAC (HS256)
  - RSA
  - ECDSA

What is the JSON Web Token structure?

- In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:
  - Header
    - The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
  - Payload
    - Which contains the claims. Claims are statements about an entity (typically, the user) and additional data.
  - Signature
    - The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.
  - Putting all together

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

JSON Web Token structure

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJh
QGdtYWkuY29tIiwiaWF0IjoxNzExM
zQzNzM1LCJleHAiOjE3MTEzNTA5M
zV9.nrKL5p1b2Q_lrs1d2dPvJoIzW5
4-YihoEdCoZhBDbcc
```

Online Decoder: https://jwt.io

Decoded

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

The Payload

- Within the payload, as we can see, there are a number of keys with values. These keys are called “claims”, and the JWT specification has seven of these specified as “registered” claims:
  - iss Issuer
  - sub Subject
  - aud Audience
  - exp Expiration
  - nbf Not Before
  - iat Issued At
  - jti JWT ID
- When building a JWT, we can put in any custom claims we wish.
- The list above simply represents the claims that are reserved both in the key that’s used, and the expected type.

How do JSON Web Tokens work?

- In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required.
- Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

Token Lifecycle

- Issue: Token created upon successful login.
- Validate: Each request includes token for verification.
- Renew: Refresh token issues new access token.
- Revoke: Invalidate token upon logout or compromise.

Using JWT with Express.js

```
> npm install jsonwebtoken
```

```
> npm install dotenv
```

./tests/test-jwt.js

```
import jwt from 'jsonwebtoken';
import 'dotenv/config';

console.log('Secret key: ',process.env.JWT_SECRET);
const userInfo = { id: 1001, username: "somchai", email:"a@x.com",role:"ADMIN" }

const accessToken = jwt.sign(userInfo, process.env.JWT_SECRET, { expiresIn: process.env.JWT_EXPIRE});
console.log('JWT:', accessToken);

jwt.verify(accessToken, process.env.JWT_SECRET, (err, claims) => {
   if (err) console.log(403, 'Forbidden');
   console.log(claims)
});
```

```
> node tests/test-jwt.js
```

.env

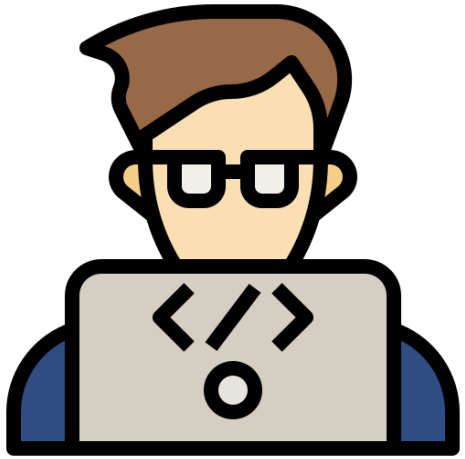
```
DATABASE_URL="mysql://root:143900@localhost:3306/sakila"
JWT_SECRET="aab856017ee097f73e5b63265a4dfdc270330697cedc550aa170789a4168f12b"
JWT_EXPIRE='30m’

SMTP_HOST="smtp.gmail.com"
SMTP_PORT=587
SMTP_USER=”*****@gmail.com"
SMTP_PASS=”**** nqll hwnd ****"
APP_URL="http://localhost:3000"
```

Setting	Value
SMTP Server	`smtp.gmail.com`
Port	`465` (SSL) or `587` (TLS)
Username	Your full Gmail address (e.g., `yourname@gmail.com`)
Password	Your Gmail password OR, preferably, an App Password
Authentication	Required
Secure Connection	SSL or TLS (depending on port)

Important notes:

- **App Passwords:** If you use 2-Step Verification, you will need to generate an App Password in your Google account settings to use with your email client.
- **Security:** Using an App Password is more secure than allowing "less secure app access".
- **Port differences:**
  - Use port `465` with SSL.
  - Use port `587` with TLS (recommended for most modern applications).

Practices



Add Model User/UserRole: `prisma.schema`

```
model User {
  id       Int      @id @default(autoincrement()) @db.SmallInt
  username String   @db.VarChar(40)
  email    String   @unique(map: "user_email_uindex") @db.VarChar(100)
  password String   @db.VarChar(100)
  role     UserRole @default(USER)
  active   Boolean  @default(false)
}

enum UserRole {
  ADMIN
  MANAGER
  USER
}
```

User Repository(1/2) : `user-repository.js`

```
import {PrismaClient} from "@prisma/client";
const prisma = new PrismaClient();

export const findAll = async () => {
    return await prisma.user.findMany({
        select: {
            id: true,
            username: true,
            email: true,
            role: true,
            active: true,
        }
    });
}
export const create = async (user) => {
    return await prisma.user.create({
        data: user,
    });
}
```

```
export const findById = async (id) => {
    return await prisma.user.findUnique({
        where: {id: id},
        select: {
            id: true,
            username: true,
            email: true,
            role: true,
            active: true,
        }
    });
}
```

User Repository(2/2) : `user-repository.js`

```
export const findByEmail= async (email) => {
    return await prisma.user.findFirst({
        where: {email: email},
        select: {
            id: true,
            username: true,
            email: true,
            role: true,
            active: true,
            password: true,
        }
    });
}
```

```
export const update = async (id, data) => {
    return await prisma.user.update({
        where: {id: id},
        data : data,
    });
}
```

Modify Error Response: `/errors/error-response.js`

```
module.exports = {
  notFoundError: function (resourceId, resourceName) {
    return this.notFoundError(`${resourceName} not found for ID ${resourceId}`);
  },
  duplicateItem: function (itemName, resourceName) {
    return this.conflictError(`${resourceName} already exists for ${itemName}`);
  },
  notFoundError: function (message) {
    const  err = new Error(message);
    err.code = 'NOT_FOUND';
    err.status = 404;
    return err;
  },
  conflictError: function (message) {
    const  err = new Error(message);
    err.code = 'CONFLICT';
    err.status = 409;
    return err;
  },
  badRequestError: function (message) {
    const  err = new Error(message);
    err.code = 'BAD_REQUEST';
    err.status = 400;
    return err;
  },
  unauthorizedError: function (message) {
    const  err = new Error(message);
    err.code = 'UNAUTHORIZED';
    err.status = 401;
    return err;
  }
}
```

Create Jwt utility (1/2) : /utils/jwt.js

```
import jwt from 'jsonwebtoken';
import 'dotenv/config';

const JWT_SECRET = process.env.JWT_SECRET;
const JWT_EXPIRES = process.env.JWT_EXPIRE | "30m";

export const generateToken = (user, age = JWT_EXPIRES) => {
   user.iss = 'sakila_rental_video@bangkok.com';
   return jwt.sign(user, JWT_SECRET,{ expiresIn: age });
};

export const verifyToken = (token) => {
   return jwt.verify(token, JWT_SECRET);
};
```

Create email utility (1/2) : /utils/email.js

> npm install nodemailer

```
import nodemailer from "nodemailer";
import 'dotenv/config';

export const transporter = nodemailer.createTransport({
    host: process.env.SMTP_HOST || "smtp.gmail.com",
    port: process.env.SMTP_PORT || 587,
    secure: false,
    auth: {
        user: process.env.SMTP_USER, // e.g. your Gmail address
        pass: process.env.SMTP_PASS, // app password
    },
});
```

Create email utility (2/2) : /utils/email.js

```
export const sendVerificationEmail = async (email, token) => {
   const verifyUrl = `${process.env.APP_URL}/auth/verify-email`;
   const message = {
      from: `"Sakila Rental Video" <${process.env.SMTP_USER}>`,
      to: email,
      subject: "Verify your account",
      html: `
    <h2>Welcome to Sakila Rental Video!</h2>
    <p>Please click the 'Confirm' button below to verify your email address:</p>
    <form method="POST" action="${verifyUrl}" target="_blank">
     <input type="hidden" name="token" value="${token}">
     <button style="background-color: green;color: wheat" type="submit">Confirm</button>
    </form>
   `,
   };
   console.log(message);
   //await transporter.sendMail(message);
};
```

Auth Service (1/4) : auth-service.js

```
import errResp from "../errors/error-response.js";
import argon2 from "argon2";
import { generateToken } from "../utils/jwt.js";
import * as repo from "../repositories/user-repository.js";
import {sendVerificationEmail} from "../utils/email.js";

const argon2Param = {
    type: argon2.argon2id, // Recommended type for general use
    memoryCost: 102400,    // Adjust memory usage in KB
    timeCost: 3,           // Adjust the number of iterations
    parallelism: 2         // Adjust based on your CPU cores
}
```

Auth Service (2/4) : auth-service.js

```
export const registerUser = async (data) => {
    const {email, password} = data;
    const existing = await repo.findByEmail(email);
    if (existing) throw errResp.duplicateItem("User", email);

    data.password = await argon2.hash(password, argon2Param);
    const user = await repo.create(data);

    const token = generateToken({id: user.id, verifyKey: await argon2.hash(user.email, argon2Param)}, "7d");
    sendVerificationEmail(email, token)
        .then(() => console.log(`Verification email to ${email} has been sent`))
        .catch((err) => console.error("Email error:", err.message));
    return {id: user.id, email: user.email, role: user.role, username: user.username}
}
```

Auth Service (3/4) : auth-service.js

```
export const verifyEmail = async (userFromToken) => {
    const existing = await repo.findById(userFromToken.id);

    if (!existing) throw errResp.notFoundError("User does not exist");
    if(existing.active) throw errResp.conflictError(`User ${existing.email} is already active`);
    const valid = await argon2.verify(userFromToken.verifyKey, existing.email);
    if (!valid) throw errResp.unauthorizedError("Invalid verification data");
    existing.active = true;
    const user = await repo.update(existing.id, existing);
    user.password = undefined;
    return user;
}
```

Auth Service (4/4) : auth-service.js

```
export const login = async (data) => {
  const { email, password } = data;

  const user = await repo.findByEmail(email);

  if (!user) throw errResp.unauthorizedError("Invalid email or password");
  if (!user.active) throw errResp.unauthorizedError("User is not active");
  const valid = await argon2.verify(user.password, password);
  if (!valid) throw errResp.unauthorizedError("Invalid email or password");

  user.password = undefined;
  const token = generateToken(user);

  return { access_token: token };
}
```

Auth Controller: `auth-controller.js`

```
import * as authService from "../services/auth-service.js";
import errResp from "../errors/error-response.js";
import {verifyToken} from "../utils/jwt.js";

export const register = async (req, res) => {
    const result = await authService.registerUser(req.body);
    res.status(201).json(result);
};

export const verify = async (req, res) => {
    let { token } = req.body;
    if(! token) token = req.query.token;
    if(! token) throw errResp.badRequestError("Token is required");
    const userFromToken = verifyToken(token);
    if(userFromToken.verifyKey==undefined)
        throw errResp.unauthorizedError("Invalid verification data");
    const result = await authService.verifyEmail(userFromToken);
    res.status(200).json(result);
};
```

```
export const login = async (req, res) => {
    const user = req.body();
    const result = await authService.login(user);
    res.status(200).json(result);
};
```

Auth Router: `auth-route.js`

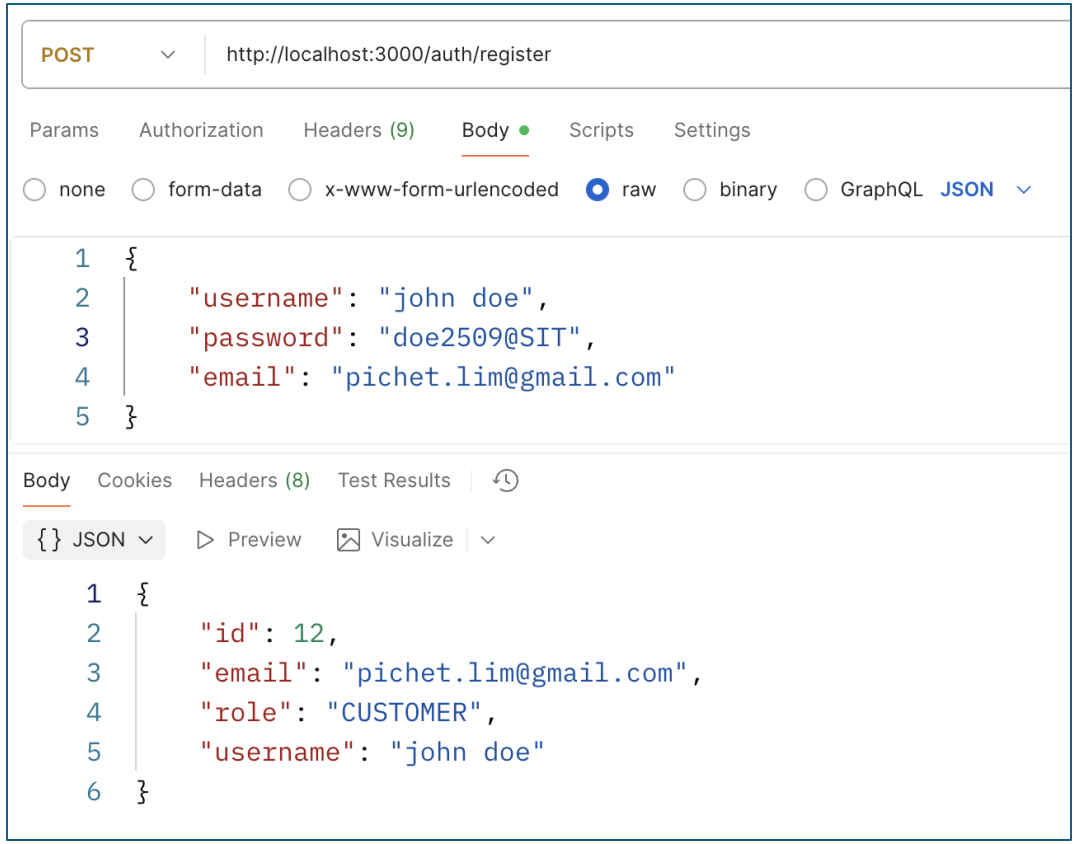
```
var express = require('express');
var router = express.Router();
const controller = require('../controllers/auth-controller');

router.post('/register', controller.register);
router.post('/verify-email', controller.verify);
router.post('/login', controller.login);

module.exports = router;
```

App.js

```
const authRouter = require('./routes/auth-route');
```

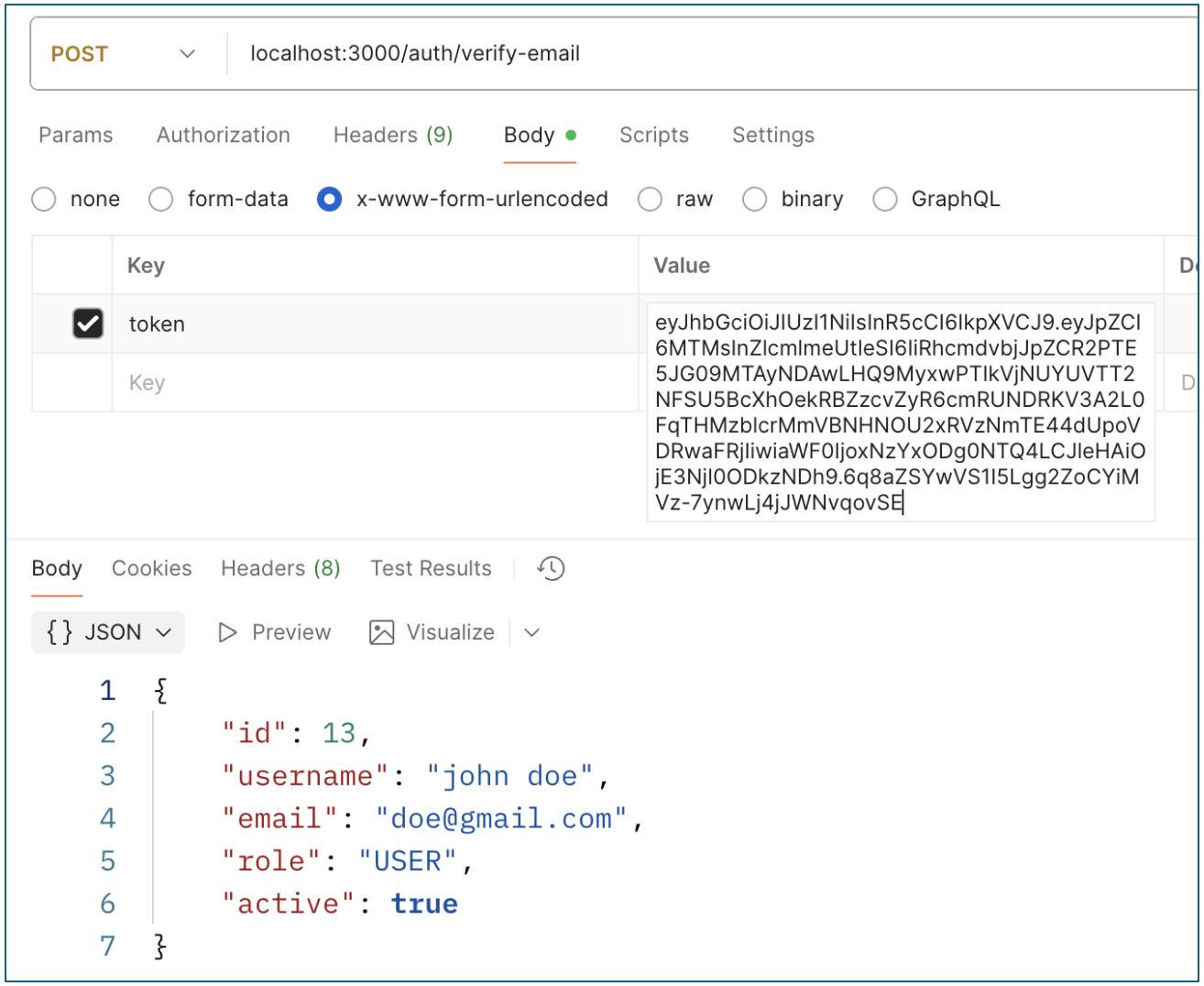
```
app.use('/auth', authRouter);
```

Register:

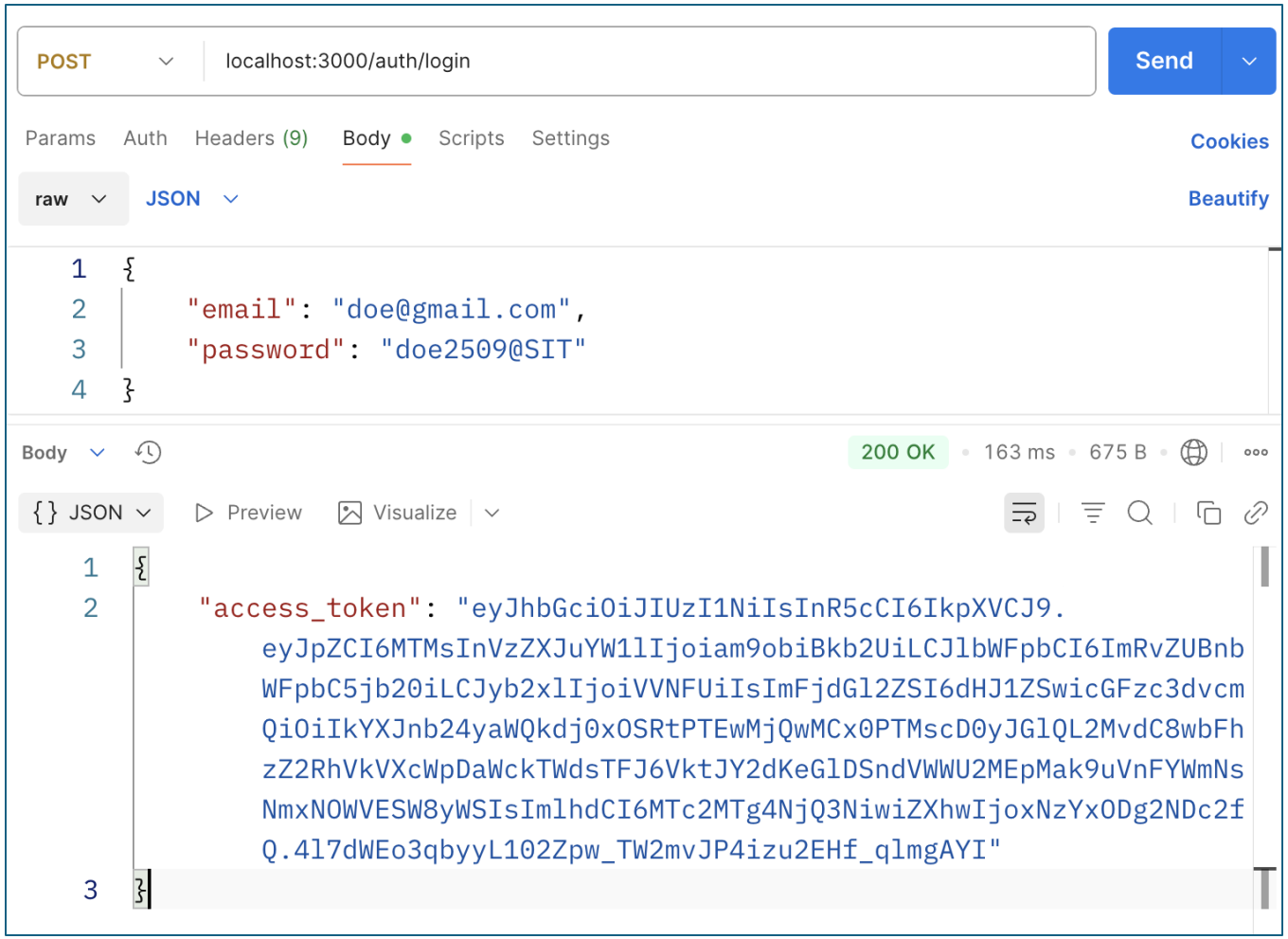


id	usern…	email	role	password	active
12	john doe	pichet.lim@gmail.com	USER	$argon2id$v=19$m=1…	0
13	john doe	doe@gmail.com	USER	$argon2id$v=19$m=1…	0

Verify:


POST
localhost:3000/auth/verify-email
Params
Authorization
Headers (9)
Body
Scripts
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
Key
Value
token
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTMsInZlcmlmeUtleSI6IiRhcmdvbjJpZCR2PTE5JG09MTAyNDAwLHQ9MyxwPTIkVjNUYUVTT2NFSU5BcXhOekRBZzcvZyR6cmRUNDRKV3A2L0FqTHMzblcrMmVBNHNOU2xRVzNmTE44dUpoVDRwaFRjIiwiaWF0IjoxNzYxODg0NTQ4LCJleHAiOjE3NjI0ODkzNDh9.6q8aZSYwVS1I5Lgg2ZoCYiMVz-7ynwLj4jJWNvqovSE
Body
Cookies
Headers (8)
Test Results
JSON
Preview
Visualize
{
"id": 13,
"username": "john doe",
"email": "doe@gmail.com",
"role": "USER",
"active": true
}

Login:


POST
localhost:3000/auth/login
Send
Params
Auth
Headers (9)
Body
Scripts
Settings
Cookies
raw
JSON
Beautify
{
"email": "doe@gmail.com",
"password": "doe2509@SIT"
}
Body
200 OK
163 ms
675 B
JSON
Preview
Visualize
{
"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpZCI6MTMsInVzZXJuYW1lIjoiam9obiBkb2UiLCJlbWFpbCI6ImRvZUBnb
WFpbC5jb20iLCJyb2xlIjoiVVNFUiIsImFjdGl2ZSI6dHJ1ZSwicGFzc3dvcm
QiOiIkYXJnb24yaWQkdj0xOSRtPTEwMjQwMCx0PTMscD0yJGlQL2MvdC8wbFh
zZ2RhVkVXcWpDaWckTWdsTFJ6VktJY2dKeGlDSndVWWU2MEpMak9uVnFYWmNs
NmxNOWVESW8yWSIsImlhdCI6MTc2MTg4NjQ3NiwiZXhwIjoxNzYxODg2NDc2f
Q.4l7dWEo3qbyyL102Zpw_TW2mvJP4izu2EHf_qlmgAYI"
}