nguyen minh quan

INTS  [Company address]

Khởi Tạo server

Table of Contents

[1. Access token vs ID Token 2](#_Toc144310490)

[a. ID token 2](#_Toc144310491)

[b. Access Token 3](#_Toc144310492)

[2. Express 5](#_Toc144310493)

[a. Đặt vấn đề 5](#_Toc144310494)

[b. Tạo server 5](#_Toc144310495)

[c. Truyền vào định dạng HTML 7](#_Toc144310496)

[d. Chia rount 8](#_Toc144310497)

[e. Sử dụng express 9](#_Toc144310498)

[3. Express from scratch 9](#_Toc144310499)

[a. Rounting 9](#_Toc144310500)

[b. Request POST config 11](#_Toc144310501)

[c. Xử lý thông tin của header 12](#_Toc144310502)

[d. Middleware 13](#_Toc144310503)

[e. Coookies 14](#_Toc144310504)

# Access token vs ID Token

## ID token

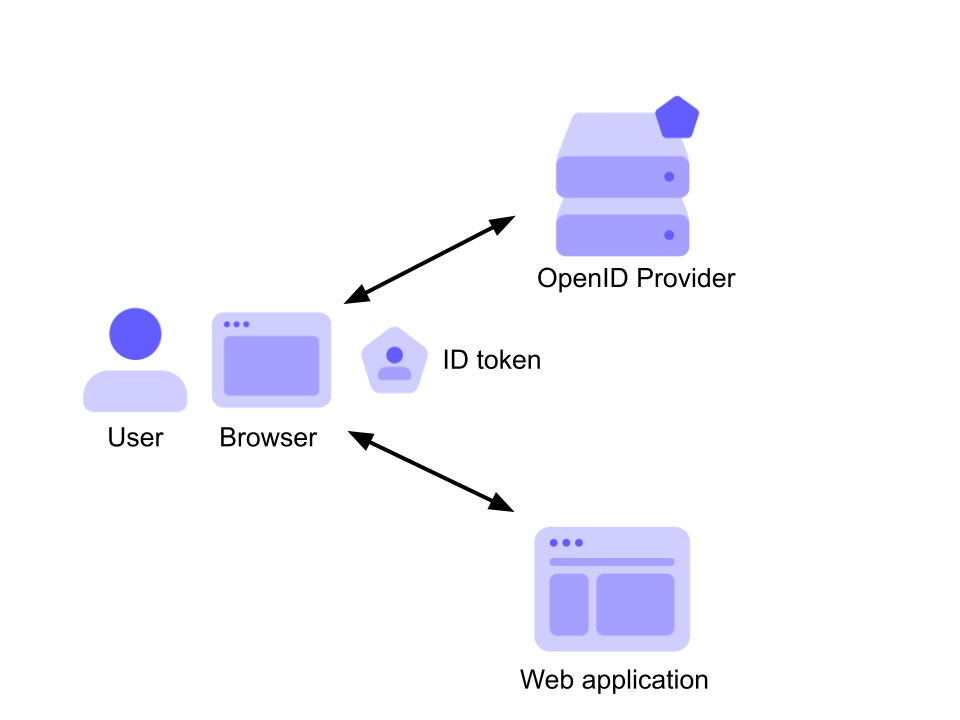


Figure ID token

ID token được dùng để khai báo về user và chính bản thân, có thể lấy được từ OpendID Provider, ID token được create ở phía client

[How to Explain Public-Key Cryptography and Digital Signatures to Non-Techies (auth0.com)](https://auth0.com/blog/how-to-explain-public-key-cryptography-digital-signatures-to-anyone/)

[ID Token and Access Token: What Is the Difference? (auth0.com)](https://auth0.com/blog/id-token-access-token-what-is-the-difference/)

## Access Token

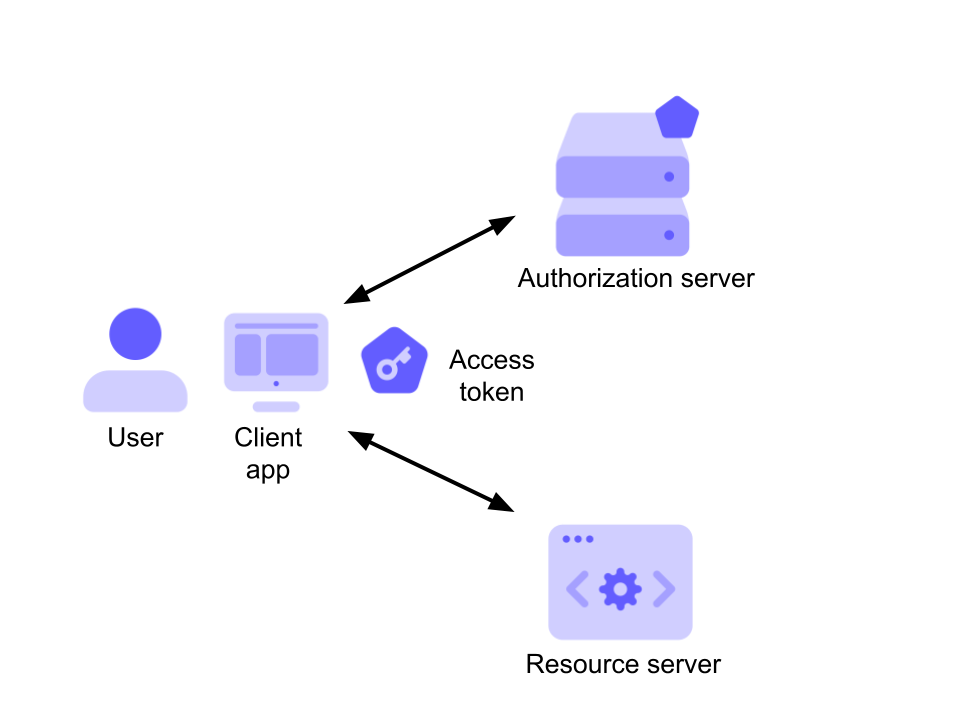


Figure Access Token

[ID Tokens VS Access Tokens: What's the Difference? - YouTube](https://www.youtube.com/watch?v=vVM1Tpu9QB4&ab_channel=OktaDev)



Figure Cheetsheet

# Express

## Đặt vấn đề

Tại sao cần express?

Express là 1 thư viện hỗ trợ việc giao tiếp get, post data, chia rounting với server, để hiểu được tiện ích của express, ta sẽ build 2 server with and without express

## Tạo server

const http = require('http')

// Nodejs đã hỗ trợ sẵn phương thức http

Tạo 1 biến server:

const server = http.createServer()

Check core của hàm thì ta thấy argument của method này là 1 function được tự động thêm vào sự kiện *request* .

/\*\*

     \* Returns a new instance of {@link Server}.

     \*

     \* The `requestListener` is a function which is automatically

     \* added to the `'request'` event.

     \* @since v0.1.13

     \*/

    function createServer<

        Request extends typeof IncomingMessage = typeof IncomingMessage,

        Response extends typeof ServerResponse = typeof ServerResponse,

    >(requestListener?: RequestListener<Request, Response>): Server<Request, Response>;

    function createServer<

        Request extends typeof IncomingMessage = typeof IncomingMessage,

        Response extends typeof ServerResponse = typeof ServerResponse,

    >(

        options: ServerOptions<Request, Response>,

        requestListener?: RequestListener<Request, Response>,

    ): Server<Request, Response>;

const server = http.createServer(function (req, res) {

})

Khi truyền 1 callback function vào createServer thì function này sẽ được chạy khi ta gửi 1 request lên server, và khi server trả về kết quả thì sẽ được lưu vào biến server.

server.listen(3000, function () {

})

Sau khi có instance server thì ta gọi method listen để lắng nghe sự kiện xảy ra trên server này

/\*\*

         \* Start a server listening for connections. A `net.Server` can be a TCP or

         \* an `IPC` server depending on what it listens to.

         \*

         \* Possible signatures:

         \*

         \* \* `server.listen(handle[, backlog][, callback])`

         \* \* `server.listen(options[, callback])`

         \* \* `server.listen(path[, backlog][, callback])` for `IPC` servers

         \* \* `server.listen([port[, host[, backlog]]][, callback])` for TCP servers

         \*

         \* This function is asynchronous. When the server starts listening, the `'listening'` event will be emitted. The last parameter `callback`will be added as a listener for the `'listening'`

Định nghĩa của method listen hỗ trợ cho TCP và cả IPC ( hệ máy tính công nghiệp hoặc để thiết kế hệ thống HMI)

Function này được định nghĩa là bất đồng bộ, khi server bắt đầu bắt sự kiện, sự kiện ‘listening’ sẽ được truyền đi, callback được coi là 1 listener.

Nói cách khác khi ta chạy lệnh node server.js thì function callback sẽ được chạy

const http = require('http')

const server = http.createServer(function (req, res) {

})

server.listen(3000, function () {

    console.log('server listening');

})

Khi server chạy thành công ta có thể thêm logic

const server = http.createServer(function (req, res) {

    res.write("hello")

    res.end("world")

})

## Truyền vào định dạng HTML

Tuy vậy server của chúng ta tạo ra không thể render ra định dạng HTML, in another way thì server không thể đọc được file là module file-system

Vì vậy ta gọi vào 1 module để giúp ta là được điều này

const fs = require('fs')

code để in ra nội dung HTML

const http = require("http");

const fs = require("fs");

const server = http.createServer(function (req, res) {

    res.writeHead(200, { "Content-Type": "text/html" });

    fs.readFile("./index.html", null, function (error, data) {

        if (error) {

            res.writeHead(404);

            res.write("File not found");

        } else {

            res.write(data);

        }

        res.end();

    });

});

server.listen(3000, function () {

    console.log("server listening");

});

## Chia rount

Khi muốn trả về nhiều rount, ta cần split code và làm 1 function để xử lý riêng

Tạo 1 file app.js

const url = require('url')

const fs = require('fs')

function renderHTML(path, res) {

    fs.readFile(path, null, function (error, data) {

        if (error) {

            res.writeHead(404);

            res.write("File not found");

        } else {

            res.write(data);

        }

        res.end();

    });

}

module.exports ={

    handleRequest: function (req, res) {

        res.writeHead(200,{'Content-Type': 'text/html'})

        const path = url.parse(req.url).pathname

        switch (path) {

            case '/':

                renderHTML('./index.html',res)

                break;

            case '/login':

                    renderHTML('./login.html',res)

                    break;

            default:

                res.writeHead(404);

                res.write('Route not defined')

                res.end()

        }

    }

}

Sau đó sửa lại server.js

const http = require("http");

const app = require('./app.js')

const server = http.createServer(app.handleRequest);

server.listen(3000, function () {

    console.log("server listening");

});

Thêm 1 file html và test

## Sử dụng express

Server ở trên working fine nhưng có khá nhiều boilerplate code, và khi ta muốn thao tác và truyền data vào file HTML thì mọi thứ sẽ còn phức tạp hơn, vì vậy ta dùng đến Express

Cài đặt Postman để test server

Struture of project inside js file

Sang phần sau ta sẽ build lại express để hiểu framework này hoạt động như nào under the hood

# Express from scratch

## Rounting

const express = require("express");

const app = express();

const PORT = 8000

const users = [

    { name: "Quan", age: "26" },

    { name: "Dung", age: "24" },

    { name: "Yen", age: "23" },

];

app.get('/', (req, res) => {

    res.send({

        msg: 'hello',

        user: {}

    })

})

app.get('/users', (req, res) => {

    res.status(200).send(users)

})

app.listen(PORT, function () {

})

Ta có thể quyết định data trả về về khi có request tới, đồng thời cũng quyết định được status trả về

    res.status(200).send(users)

Fun fact, nếu server error nhưng chúng ta vẫn trả về status 200 thì client vẫn nhận được request success

* Tạo 1 dynamic URL

// Dynamic URL

app.get('/users/:name', (req, res) => {

    const {name} = req.params;

    const user = users.find((user) => user.name === name )

    if(user) res.status(200).send({user})

    else res.status(404).send('Not found')

})

Giống như trong React có hỗ trợ build-in useParams thì express cũng hỗ trợ biến để lưu giá trị động “:name”

Trích xuất name từ req xong chúng ta lưu lại và find trong mảng user

app.get('/posts', (req, res) => {

    const {title} = req.query;

    const post = posts.find((post) => post.title === title )

    if(post) res.status(200).send({post})

    else res.status(404).send('Not found')

})

Sử dụng query thay vì params:

A screenshot of a computer

Description automatically generated

Hình req.query example

## Request POST config

Chuyển sang lệnh POST 1 cụm data lên server

app.post('/', (req, res) => {

    console.log("🚀 ~ file: rounting.js:19 ~ app.post ~ req:", req)

    res.send('Created')

})

Bằng cách log ra req ta có thể thấy được cấu trúc cơ bản của 1 req, những thông tin được client gửi lên cũng như các built-in

Ví dụ request mà Postman vừa gửi:

rawHeaders: [

'User-Agent',

'PostmanRuntime/7.32.3',

'Accept',

'\*/\*',

'Postman-Token',

'16241d76-fc9c-4157-badb-1004b08c973a',

'Host',

'localhost:8000',

'Accept-Encoding',

'gzip, deflate, br',

'Connection',

'keep-alive',

'Content-Length',

'0'

]

Tuy nhiên khi cố gửi lên 1 cục data bằng body thì kết quả trả về sẽ là:

🚀 ~ file: rounting.js:19 ~ app.post ~ req.body: undefined

app.post('/', (req, res) => {

    console.log("🚀 ~ file: rounting.js:19 ~ app.post ~ req.body:", req.body)

    res.status(200).send('Created')

})

Lý do vì chúng ta chưa setup middleware để xử lý dữ liệu gửi kèm

app.use(express.urlencoded({extended:false}))

🚀 ~ file: rounting.js:19 ~ app.post ~ req.body: [Object: null prototype] { name: 'Chien', age: '24' }

Tương tự, khi gửi dữ liệu lên dưới dạng 1 JSON thì ta cũng phải gọi hàm để support JSON middleware, về cơ bản thì trước khi req đi vào callback function ở post thì req cần đi qua 1 middleware để xác định dạng dữ liệu được gửi lên là gì

## Xử lý thông tin của header

Header gần giống như dữ liệu để sử dụng global, khi mà body chứ logic của business

app.post('/posts', (req, res) => {

    const {authorization} = req.headers

    if(authorization && authorization === '1234') {

        const post = req.body

        console.log("🚀 ~ file: rounting.js:29 ~ app.post ~ post:", post)

        posts.push(post)

        res.status(201).send(post)

    } else {

        res.status(404).send('Invalid request')

    }

})

Check authorization

## Middleware

Middleware bản chất là 1 function nhận vào 3 parameter là Request, Response và 1 function next() để chạy function middleware tiếp theo nếu có.

app.use((req, res, next) => {

    console.log(`${req.method} - ${req.url}`);

    next()

})

Với thiết lập này thì mỗi khi có request tới, app sẽ in req method và url ra màn hình trước rồi mới chạy logic bên trong

A black screen with white text

Description automatically generated

Hình Kết quả middleware trả về

Middleware come in handy đặc biệt khi ta muốn split logic ra để quản lý, ví dụ trong

app.post('/posts', (req, res) => {

    const {authorization} = req.headers

    if(authorization && authorization === '1234') {

        const post = req.body

        console.log("🚀 ~ file: rounting.js:29 ~ app.post ~ post:", post)

        posts.push(post)

        res.status(201).send(post)

    } else {

        res.status(404).send('Invalid request')

    }

})

Ta chỉ muốn app.post tập trung vào xử lý callback function logic, và tách phần code check logic ra riêng, ta có thể làm như sau:

function validateAuth(req, res, next) {

    const { authorization } = req.headers;

    if (authorization && authorization === "1234") {

        next();

    } else {

        res.status(404).send("Invalid request");

    }

}

app.post("/posts", validateAuth, (req, res) => {

    const post = req.body;

    posts.push(post);

    res.status(201).send(post);

});

Express hỗ trợ 2nd parameter để nhận vào và xử lý logic middleware trước, sau đó mới đi vào scope của callback function.

## Coookies

npm install cookie-parser

Gọi module

const cookieParser = require("cookie-parser");

app.use(cookieParser());

function validateCookie(req, res, next) {

    const { cookies } = req;

    if('session\_id' in cookies) {

        console.log('Session ID exsists');

        if (cookies.session\_id === '123456') {

            next();

        } else {

            res.status(404).send('Not authenreicated')

        }

    } else res.status(404).send('Not authenreicated')

}

app.get("/protected", validateCookie, (req, res) => {

    res.status(200).json({ msg: "protected success" });

});

Client gửi cookies lên và server sẽ check nếu khớp thì chạy logic trong callback

## Session

Việc lưu các sensitive infomation trên server để truy xuất sẽ an toàn hơn là lưu mọi thứ ở client

npm i express-session

interface SessionOptions {

        /\*\*

         \* This is the secret used to sign the session cookie. This can be either a string for a single secret, or an array of multiple secrets.

         \* If an array of secrets is provided, \*\*only the first element will be used to sign\*\* the session ID cookie,

         \*   while \*\*all the elements will be considered when verifying the signature\*\* in requests.

         \* The secret itself should be not easily parsed by a human and would best be a random set of characters

         \*

         \* Best practices may include:

         \* - The use of environment variables to store the secret, ensuring the secret itself does not exist in your repository.

         \* - Periodic updates of the secret, while ensuring the previous secret is in the array.

         \*

         \* Using a secret that cannot be guessed will reduce the ability to hijack a session to only guessing the session ID (as determined by the `genid` option).

         \*

         \* Changing the secret value will invalidate all existing sessions.

         \* In order to rotate the secret without invalidating sessions, provide an array of secrets,

         \*   with the new secret as first element of the array, and including previous secrets as the later elements.

         \*/

        secret: string | string[];

app.use(session({

    secret: ['some secret'],

    cookie: {maxAge: 3000},

    saveUninitialized: false

}))

Như đã thấy thì team Express họ recommend chúng ta sử dụng array để lưu secret như best practice vì sau này những secret này sẽ được update định kì

Khi send lên 1 request thì trong request sẽ có 1 property:

sessionID: 'pH7A7FHvllwe0jpzEtPFbMWTyIKy8bD1',

session: Session {

cookie: {

path: '/',

\_expires: 2023-08-31T03:22:53.967Z,

originalMaxAge: 3000,

httpOnly: true

}

},

Và sessionID sẽ thay đổi với mỗi request được gửi lên

sessionID: 'Fq-\_NNxNjl3ssKAQKo6AS3lSpYN1CLjK',

session: Session {

cookie: {

path: '/',

\_expires: 2023-08-31T03:26:32.392Z,

originalMaxAge: 3000,

httpOnly: true

}

}

Thêm middleware json và urlencoded để đọc data được gửi lên

app.use(express.urlencoded({

    extended:false

}))

app.use(express.json())

Tạo method post để kiểm tra

app.post("/login", (req, res) => {

    const { username, password } = req.body;

    console.log("🚀 ~ file: rounting.js:73 ~ app.post ~ username, password:", username, password)

    if (username && password) {

        if (req.session.authenticated) {

            res.json(req.session);

        } else {

            if (password === '1234') {

                req.session.authenticated = true;

                req.session.user = {

                    username,

                    password,

                };

                res.json(req.session);

            } else {

                res.status(404).send("Invalid user");

            }

        }

    } else res.status(404).send("Invalid user");

});

Kết quả trả về



Hình Kết quả trả về session của post

Session chứa các thông tin cơ bản như thời gian expire MaxAge là 3000, thời điểm expire, và các thuộc tính ta truyền vào là authenticated và user: {}

## Session store

Khởi tạo 1 instance của store và in ra console để xem struture, thêm store vào session

const store = new session.MemoryStore

A computer screen with green text

Description automatically generated

Hình This is how store look like

SessionID trong store sẽ match với ID của từng request

## Router và split code để moduler hơn

Về cơ bản thì code working fine nhưng ta sẽ muốn split logic ra thành nhiều module để dễ maintain và scale lên:

Đầu tiên tạo 1 folder router và import class Router.

Ta sẽ tạo tạm thời 2 route là user và post để lưu dữ liệu và export module ra để sử dụng