

Full Stack Web Development

Authentication & Authorization

Job Connector Program

Introduction to Authentication & Authorization

User **Authentication & Authorization** is one of the important part of any web application.

In simple words, **Authentication** is the process of verifying who a user is (**who you are**), and Authorization is the process of verifying what they have access to (**what you are allowed to do**).



What is Authentication?

Authentication is a process to verify that someone or something is who they say they are. Technology systems typically use some form of authentication to secure access to an application or its data. For example, when you need to access an online site or service, you usually have to enter your username and password.

Behind the scenes, it compares the username and password you entered with a record it has on its database. If the information you submitted matches, the system assumes you are a valid user and grants you an access. System authentication in this example presumes that only you would know the correct username and password.

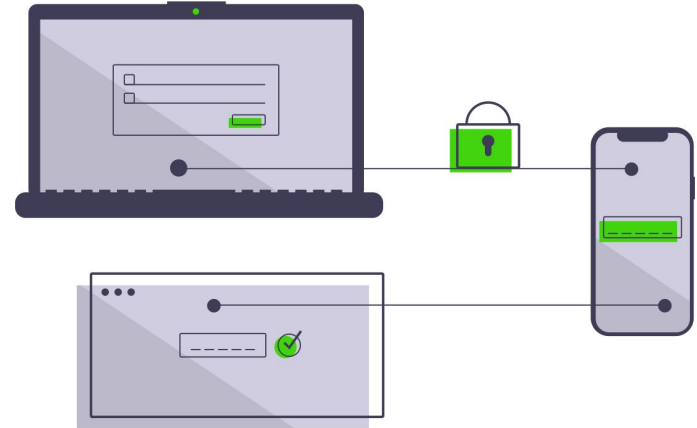
Why We need Authentication?

The purpose of authentication is to verify that someone or something is who or what they claim to be. Typically, authentication protects items of value, and in the information age, it protects systems and data.



Common Types of Authentication

Systems can use several mechanisms to authenticate a user. Typically, to verify your identity, authentication processes use something you know, something you have or something you are.



Common Types of Authentication

- **Passwords and security questions** are two authentication factors that fall under the something-you-know category. As only you would know your password or the answer to a particular set of security questions, systems use this assumption to grant you access.
- Another common type of authentication factor uses something you have. **Physical devices such as USB security tokens and mobile phones** fall under this category. For example, when you access a system, and it sends you a One Time Pin (OTP) via SMS or an app, it can verify your identity because it is your device.
- The last type of authentication factor uses something you are. **Biometric authentication mechanisms** fall under this category. Since individual physical characteristics such as fingerprints are unique, verifying individuals by using these factors is a secure authentication mechanism.

What is Authorization?

Authorization is the security process that determines a user or service's level of access. In technology, we use authorization to give users or services permission to access some data or perform a particular action.

For example, in a coffee shop business, Rahul and Lucia have different roles. As Rahul is a barista, he may only place and view orders. Lucia, on the other hand, in her role as manager, may also have access to the daily sales totals. Since Rahul and Lucia have different jobs in the coffee shop, the system would use their verified identity to provide each user with individual permissions. It is vital to note the difference here between authentication and authorization. **Authentication verifies the user** (Lucia) before allowing them access, and **authorization determines what they can do once the system has granted them access** (view sales information).

- Authorization systems exist in many forms in a typical technology environment. For example, Access Control Lists (ACLs) determine which users or services can access a particular digital environment. They accomplish this access control by enforcing allow or deny rules based on the user's authorization level. For instance, on any system, there are usually general users and super users or administrators. If a standard user wants to make changes that affect its security, an ACL may deny access. On the other hand, administrators have the authorization to make security changes, so the ACL will allow them to do so.
- In any enterprise environment, you typically have data with different levels of sensitivity. For example, you may have public data that you find on the company's website, internal data that is only accessible to employees, and confidential data that only a handful of individuals can access. In this example, authorization determines which users can access the various information types.

The Difference Between Authentication and Authorization

authentication and authorization may sound alike, but each plays a different role in securing systems and data. Unfortunately, people often use both terms interchangeably as they both refer to system access. **However, they are distinct processes. Simply put, one verifies the identity of a user or service before granting them access, while the other determines what they can do once they have access.**



The Difference Between Authentication and Authorization

The best way to illustrate the differences between the two terms is with a simple example:

Let's say you decide to go and visit a friend's home. On arrival, you knock on the door, and your friend opens it. She recognizes you (**authentication**) and greets you. As your friend has authenticated you, she is now comfortable letting you into her home. However, based on your relationship, there are certain things you can do and others you cannot (**authorization**). For example, you may enter the kitchen area, but you cannot go into her private office. In other words, you have the authorization to enter the kitchen, but access to her private office is prohibited.

After understanding the concept of authentication and authorization, lets implement that into your projects.

In this part, we assume you already know about:

- NodeJs with ExpressJs
- SQL with MySQL as database
- ORM with sequelize
- Postman to perform api testing

Authentication & Authorization in NodeJs

First initialize your projects and setup your db connection on config/config.json adjust your username, password, database, and host based on your db connection

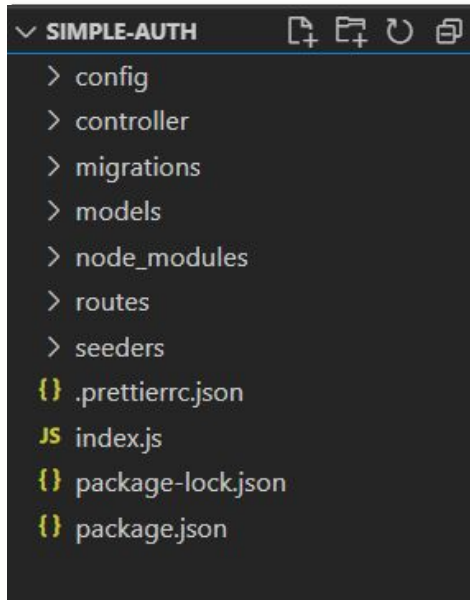


```
npm init --y
npm install express sequelize mysql2 body-parser
sequelize-cli init
```



```
{
  "development": {
    "username": "root",
    "password": null,
    "database": "database_development",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
}
```

Authentication & Authorization in NodeJs



After initialize your projects and setup your db connection, we should prepare some folder and index.js as picture besides:

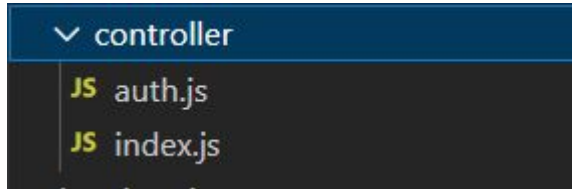
- controller
- routes
- index.js

Authentication & Authorization in NodeJs

In folder controller, create this several files:

- auth.js
- Index.js

We would working on it with this files later.



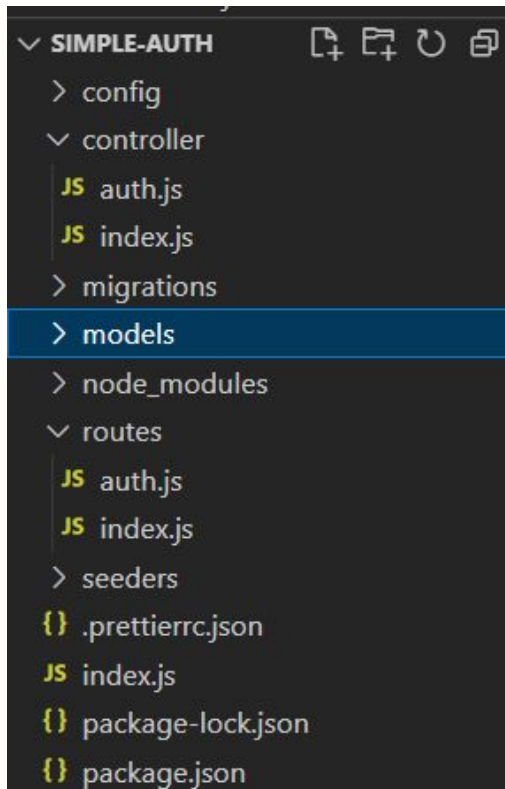


In folder routes, create this several files:

- auth.js
- role.js
- Index.js

We would working on it with this files later.

Authentication & Authorization in NodeJs



Your projects would be look like this. Next, we will setup our tables that would be used in this projects.

Authentication & Authorization in NodeJs

```
user.js

const { DataTypes } = require('sequelize');

const User = (sequelize) => {
  return sequelize.define('user', {
    username: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    email: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    password: {
      type: DataTypes.STRING,
    },
    isAdmin: {
      type: DataTypes.BOOLEAN,
      defaultValue: false,
    },
  });
};

module.exports = User;
```

Go to models directory and then create new files:

- user.js

Write down this code. In this case, we would create user table into our db through sequelize.

Authentication & Authorization in NodeJs

```
index.js

const express = require('express');

const app = express();
app.use(express.json());

//sync db purposes
const db = require('./models');
db.sequelize.sync({ alter: true });

// Import Routes
const { authRoutes } = require('./routes');

// Route Middlewares
app.use('/auth', authRoutes);

const port = 3000;
app.listen(port, function () {
  console.log('Server running on localhost:' + port);
});
```

After defining our models, let's work on index.js in the root of projects

Create our server apps through this code.

In this case, we would generate our tables through sync approach from sequelize.

We would work on routes named as auth.

Authentication in NodeJs



```
const authRoutes = require('./auth');  
  
module.exports = {  
  authRoutes,  
};
```

Go to routes/index.js and write down this code.

Every route that we create would be handled in this file.

Authentication in NodeJs

```
auth.js

const { authControllers } = require('../controller');

const router = require('express').Router();

router.post('/', authControllers.register);
router.get('/', authControllers.login);

module.exports = router;
```

Go to routes/auth.js and write down this code.

We will define each router would be use for auth. In this case there are login and register as an example.

Authentication in NodeJs



```
const authControllers = require('./auth');  
  
module.exports = {  
  authControllers,  
};
```

Go to controller/index.js and write down this code.

Every controller that we create would be handled in this file.

Authentication in NodeJs

```
auth.js

const db = require('../models');
const User = db.user;

const AuthController = {
  register: async (req, res) => {
    try {
      const { username, email, password } = req.body;

      const isEmailExist = await User.findOne({ where: { email } });
      if (isEmailExist) {
        return res.status(409).json({
          message: 'email has been used',
        });
      }
      await User.create({ username, email, password });
      return res.status(200).json({
        message: 'register success',
      });
    } catch (err) {
      console.log(err);
      return res.status(err.statusCode || 500).json({
        message: err.message,
      });
    }
  },
};

module.exports = AuthController;
```

Go to controller/auth.js and write down this code.

In this code you will create register action. Where you would check email availability. If its available then create register process would be successful.

Try to create user with post method:

<http://localhost:3000/auth>

Put username, email, and password through body.

Authentication in NodeJs

```
auth.js

login: async (req, res) => {
  try {
    const { email, password } = req.body;

    const checkLogin = await User.findOne({ where: { email, password } });
    if (!checkLogin) {
      return res.status(404).json({
        message: 'email or password is incorrect',
      });
    }
    return res.status(200).json({
      message: 'login success',
    });
  } catch (err) {
    console.log(err);
    return res.status(err.statusCode || 500).json({
      message: err.message,
    });
  }
},
```

Still in the same file, create another method for login after the register.

Based on authentication rules, only the one who login that knows email and password for him/her self. If its not, authentication process will send an error.

In this example, login process was using email and password as a terms of login. If email and password are same as on database, it would show login success message.

Try to login through get method:

<http://localhost:3000/auth>

Put email and password through body

Authentication in NodeJs

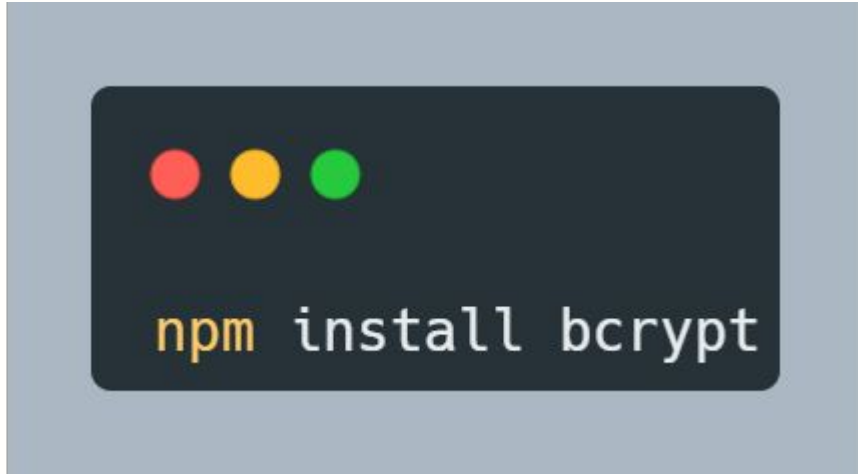


```
{  
  "message": "login success"  
}
```



```
{  
  "message": "email or password is incorrect"  
}
```


Authentication in NodeJs with Encrypt Data



Lets check your users table on the db. You can see an user password since it was written clearly. Lets secure his password using bcrypt.

This package would be useful to encrypt and decrypt any information through your projects.

Read more for documentation:

<https://www.npmjs.com/package/bcrypt>

Authentication in NodeJs with Encrypt Data

Lets check your users table on the db. You can see an user password since it was written clearly. Lets secure his password using bcrypt.

This package would be useful to encrypt and decrypt any information through your projects.

```
//import bcrypt
const bcrypt = require('bcrypt');

//create random secure key
const salt = await bcrypt.genSalt(10);
//bcrypt.hash last parameter could be replaced with any string
const hashPassword = await bcrypt.hash(req.body.password, salt);
```

```
npm install bcrypt
```

Authentication in NodeJs with Encrypt Data

```
auth.js

register: async (req, res) => {
  try {
    const { username, email, password } = req.body;

    const isEmailExist = await User.findOne({ where: { email } });
    if (isEmailExist) {
      return res.status(409).json({
        message: 'email has been used',
      });
    }
    const salt = await bcrypt.genSalt(10);
    const hashPassword = await bcrypt.hash(req.body.password, salt);
    await User.create({ username, email, password: hashPassword });
    return res.status(200).json({
      message: 'register success',
    });
  } catch (err) {
    console.log(err);
    return res.status(err.statusCode || 500).json({
      message: err.message,
    });
  }
},
```

Lets modify our controller/auth.js and implement bcrypt while encrypting user password in register process.

Dont forget to import bcrypt packages at the top of your code. After that try to register with another account. Check out the password in users table it wont plain ever again.

```
const bcrypt = require('bcrypt');
```

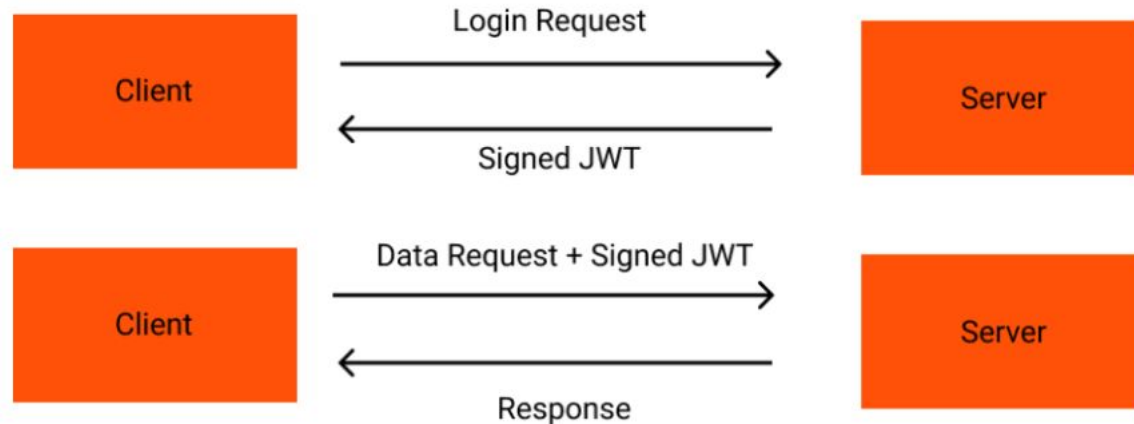
Authentication in NodeJs with Encrypt Data

```
login: async (req, res) => {
  try {
    const { email, password } = req.body;
    const isEmailExist = await User.findOne({ where: { email }, raw: true });
    if (!isEmailExist) {
      return res.status(401).json({
        message: 'email not found',
      });
    }
    const isValid = await bcrypt.compare(password, isEmailExist.password);
    if (!isValid) {
      return res.status(401).json({
        message: 'email or password incorrect',
      });
    }
    return res.status(200).json({
      message: 'login success',
    });
  } catch (err) {
    console.log(err);
    return res.status(err.statusCode || 500).json({
      message: err.message,
    });
  }
},
```

Next, we need to modify login. In this code you will compare between plain password with the encrypted password in users table using `bcrypt.compare` method. Try to login with the new register user and check out the response.

You just secure your password data.

JSON Web Token (JWT) are a good way of securely transmitting information between parties because they can be signed (Information Exchange). Even though we can use JWT with any type of communication method, today JWT is very popular for handling authentication and authorization via HTTP.



In this case, we would like to create user access level in our projects. Which mean, only permitted user role could access some of information in our server. Lets say we separate our user role into two:

- Basic-user
- Admin-user.

Basic user is not allowed to accessing list of all users from database, while admin user is granted to access all users information.

Authorization with JWT

```
const { DataTypes } = require('sequelize');

const User = (sequelize) => {
  return sequelize.define('user', {
    username: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    email: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    password: {
      type: DataTypes.STRING,
    },
    isAdmin: {
      type: DataTypes.BOOLEAN,
      defaultValue: false,
    },
  });
};

module.exports = User;
```

Lets take a look at models/user.js

There is isAdmin field that we define before, we would used it to define false as a basic user, while true as admin user.

Try to create new user and set as admin user.

Authorization with JWT



```
npm install jsonwebtoken
```

Before we re going to next step, lets install JSON Web Token package into our projects.

Find out more about the documentation:

<https://www.npmjs.com/package/jsonwebtoken>

Authorization with JWT

```
login: async (req, res) => {
  try {
    const { email, password } = req.body;
    const isEmailExist = await User.findOne({ where: { email }, raw: true });
    if (!isEmailExist) {
      return res.status(401).json({
        message: 'email not found',
      });
    }
    const isValid = await bcrypt.compare(password, isEmailExist.password);
    if (!isValid) {
      return res.status(401).json({
        message: 'email or password incorrect',
      });
    }
    let payload = { id: isEmailExist.id, isAdmin: isEmailExist.isAdmin };
    const token = jwt.sign(payload, 'coding-its-easy', { expiresIn: '1h' });
    return res.status(200).json({ token, message: 'login success' });
  } catch (err) {
    return res.status(err.statusCode || 500).json({
      message: err.message,
    });
  }
},
```

Go to controller/auth.js and focus on login process.

Import JSON Web Token package into this file. After that use method `jwt.sign` to generate token.

First parameter would be a data to convert, the second one is secret key to handle your token, and the last params is optional, but in this case we could set expires token time.

Authorization with JWT

```
const jwt = require('jsonwebtoken');
```

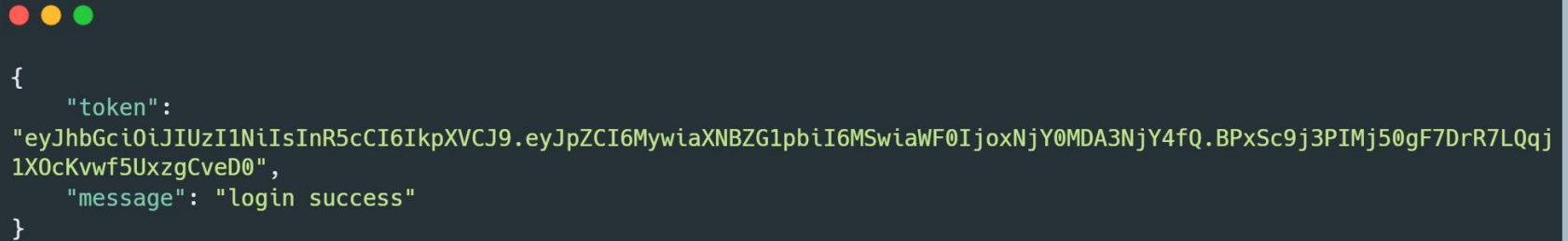
```
const token = jwt.sign(
  payload,
  'coding-its-easy',
  { expiresIn: '1h' }
);
```

Don't forget to import JSON Web Token packages into controller/auth.js

In jwt.sign method we put expires time. This would be useful to set session time for user access. For example mobile banking apps implement session login time. Token was created when user login and has been set expires time. When token expired, user login session would be no longer available and need to create a new one and suddenly you were log out from the apps.

Authorization with JWT

Try to login, if its success would show response like image shown below.



```
{
  "token":
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MywiaXNBZG1pb2I6MSwiaWF0IjoxNjY0MDA3NjY4fQ.BPxSc9j3PIMj50gF7DrR7LQqj1X0cKvwf5UxzgCveD0",
  "message": "login success"
}
```

Authorization with JWT

```
middleware/auth.js

const verifyToken = (req, res, next) => {
  let token = req.headers.authorization;
  if (!token)
    return res.status(401).send('Access Denied / Unauthorized request');

  try {
    token = token.split(' ')[1]; // Remove Bearer from string

    if (token === 'null' || !token)
      return res.status(401).send('Unauthorized request');

    let verifiedUser = jwt.verify(token, 'coding-its-easy');
    if (!verifiedUser) return res.status(401).send('Unauthorized request');
    req.user = verifiedUser;
    next();
  } catch (error) {
    res.status(400).send('Invalid Token');
  }
};
```

Create folder named as middleware and create file named as auth.js

We would check request authorization by token session since user has been login.

Jwt.verify method would check validity of your token session.

Authorization with JWT

```
middleware/auth.js

const checkRole = async (req, res, next) => {
  if (req.user.isAdmin) {
    return next();
  }
  return res.status(401).send('Unauthorized!');
};

module.exports = { verifyToken, checkRole };
```

```
middleware/auth.js

const jwt = require('jsonwebtoken');
```

After checking user session time, we need to check user role. This means only admin users could access some information. If request made by admin, then it would give a valid response.

Don't forget to import JSON Web Token packages!

Authorization with JWT

```
routes/auth.js

const { authControllers } = require('../controller');
const { verifyToken, checkRole } = require('../middleware/auth');
const router = require('express').Router();

router.post('/', authControllers.register);
router.get('/', authControllers.login);
router.get('/users', verifyToken, checkRole, authControllers.findAllUser);

module.exports = router;
```

Create new routes in order to retrieve all of users data from server. Put middleware that we just created before into this route. It would prevent basic-user or non sessions user from accessing this routes.

Authorization with JWT

```
findAllUser: async (req, res) => {  
  try {  
    const users = await User.findAll({ raw: true });  
    return res.status(200).json({  
      result: users,  
    });  
  } catch (error) {  
    return res.status(error.statusCode || 500);  
  }  
},
```

Create method named as `findAllUser` in order to get information all users from the database as a final action. This action would only execute if user is admin user and token session is still valid.

Test your project on postman using this flow:

1. Register new user and add isAdmin in the body and set true as a value
2. Login with the new user that you just created before.
3. Copy the token strings from response.
4. Go to <http://localhost:3000/auth/users> and go to tab authorization and select Bearer Token as a type and paste the token from response into token field, check image on the next slide as example.
5. Check out the response, and try with non admin user to check the difference

http://localhost:3000/auth/users

Save



GET

http://localhost:3000/auth/users

Send

Params

Authorization

Headers (11)

Body

Pre-request Script

Tests

Settings

Cookies

Type

Bearer Token

Token

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...

The authorization header will be automatically generated when you send the request. Learn more about [authorization](#)

Body

Cookies

Headers (7)

Test Results



Status: 400 Bad Request

Time: 5 ms

Size: 249 B

Save Response

Pretty

Raw

Preview

Visualize

HTML



1 Invalid Token

Thank You!

