Securing APIs

Understanding Security

Introduction to the next set of APIs

APIs to be created:

- Rate a product.
- Add products to a cart.
- Retrieve products from the cart.
- Remove a product from the cart.

Need for User Authentication

- User authentication is required for user-specific actions.
- Actions such as adding items to the cart or placing orders require user identification.
- Without authentication, it becomes difficult for the system to determine user-specific actions.

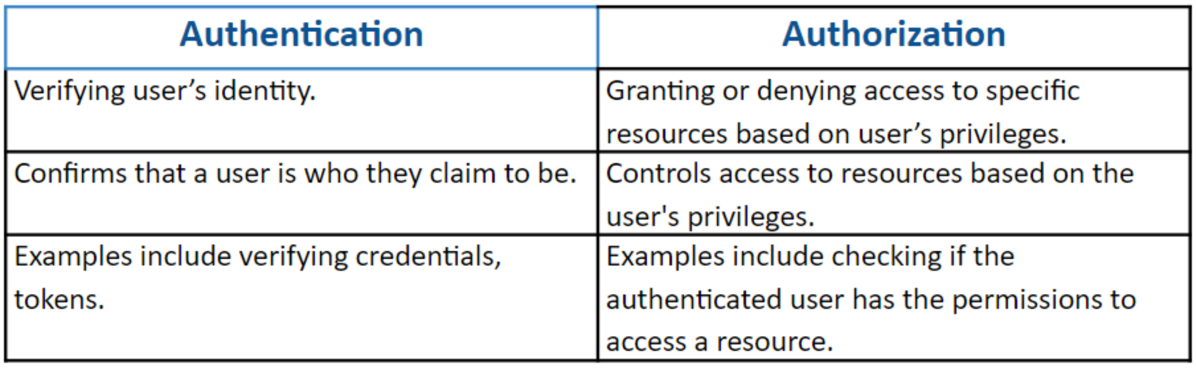
Reasons for Securing the Application

- Controlled access: Only logged-in users should access certain features.
- Data privacy: User data should not be accessible to everyone.
- Examples: Misuse of private data on social media platforms.

How to Secure the Application

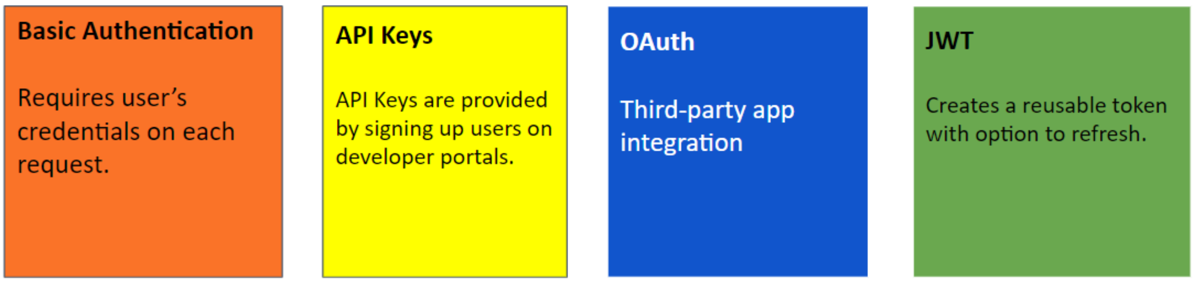
- Authentication and authorization are essential.
- Authentication verifies the identity of a user.
- Authorization determines what a user can access based on their privileges.

Authentication and Authorization



Authentication	Authorization
Verifying user’s identity.	Granting or denying access to specific resources based on user’s privileges.
Confirms that a user is who they claim to be.	Controls access to resources based on the user's privileges.
Examples include verifying credentials, tokens.	Examples include checking if the authenticated user has the permissions to access a resource.

Types of Authentication



Basic Authentication

Requires user’s credentials on each request.

API Keys

API Keys are provided by signing up users on developer portals.

OAuth

Third-party app integration

JWT

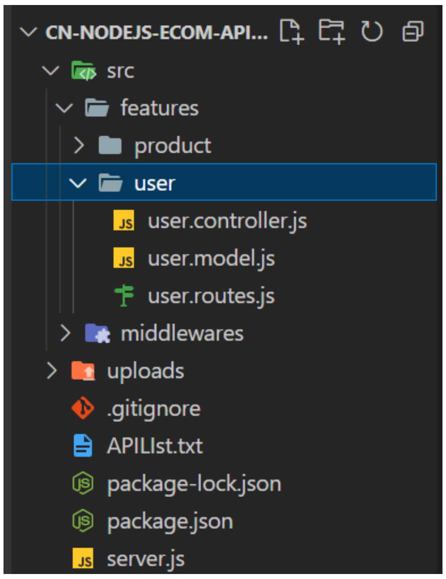
Creates a reusable token with option to refresh.

- **Basic authentication**: Users provide credentials with each request.
- **API keys:** Unique keys for user identification (used by third-party APIs).
- **OAuth (Open Authentication):** Allows authentication using third-party applications (e.g., Google or Facebook).
- **JWT (JSON Web Token):** Popular authentication method using tokens (more details in future videos).

User APIs

Securing REST API Application

- The first step towards securing the application is by creating two APIs for user registration and login.
- Add two APIs to the user controller: one for sign up and another for sign in.
- The sign-up API will accept user details such as email, name, password, and user type (customer or seller).
- The sign-in API will require the user to provide their email and password.
- Create a new folder named "user" inside the existing "features" directory.
- Within the "user" folder, three files are created: user.model.js, user.controller.js, and user.routes.js.



- The **user.model.js** file defines a User class with a constructor that takes parameters for name, email, password, and user type.
- A default user is created with the name "Seller User," email "seller@ecom.com," password "password1," and user type "seller."

```
export default class UserModel {
  constructor(name, email, password, type, id) {
    this.name = name;
    this.email = email;
    this.password = password;
    this.type = type;
    this.id = id;
  }

  static SignUp(name, email, password, type) {
    const newUser = new UserModel(
      name,
      email,
      password,
      type
    );
    newUser.id = users.length + 1;
    users.push(newUser);
    return newUser;
  }

  static SignIn(email, password) {
    const user = users.find(
      (u) =>
        u.email == email && u.password == password
    );
    return user;
  }
}

var users = [
  {
    id: 1,
    name: 'Seller User',
    email: 'seller@ecom.com',
    password: 'Password1',
    type: 'seller',
  },
];
```

- The **user.controller.js file** is created with a controller class that includes two functions: signUp and signIn, which both take request and response parameters.

```
import UserModel from './user.model.js';

export default class UserController {
  signUp(req, res) {
    const {
      name,
      email,
      password,
      type,
    } = req.body;
    const user = UserModel.SignUp(
      name,
      email,
      password,
      type
    );
    res.status(201).send(user);
  }
  signIn(req, res) {
    const result = UserModel.SignIn(
      req.body.email,
      req.body.password
    );
    if (!result) {
      return res
        .status(400)
        .send('Incorrect Credentials');
    } else {
      return res.send('Login Successful');
    }
  }
}
```

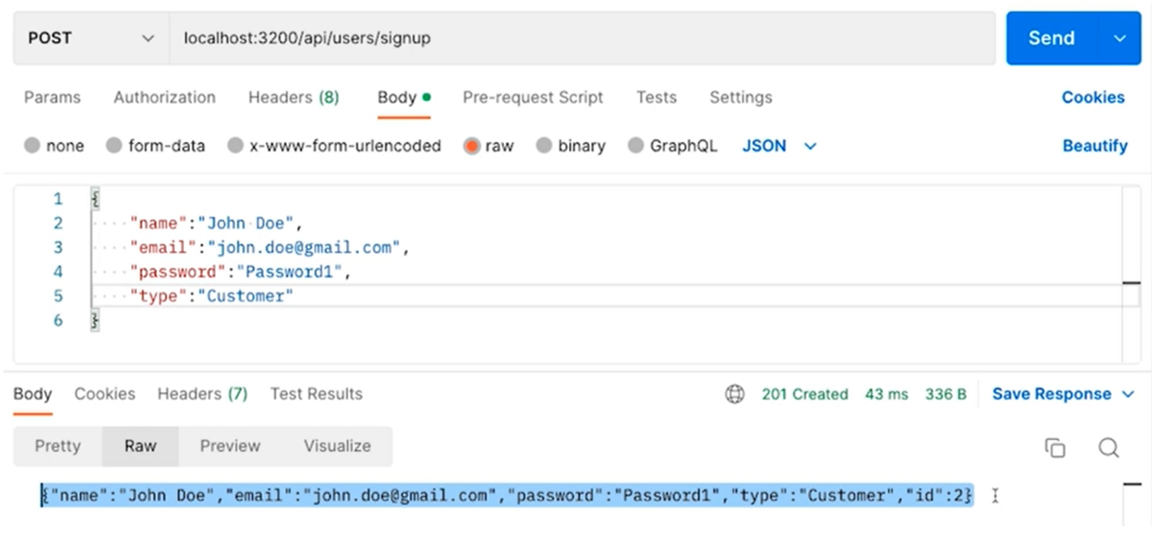
- In the user.routes.js file, create routes for user sign-up and sign-in.
- The routes for sign-up and sign-in are both POST requests.

```
// Manage routes/paths to ProductController

// 1. Import express.
import express from 'express';
import UserController from './user.controller.js';

// 2. Initialize Express router.
const userRouter = express.Router();

const userController = new UserController();

// All the paths to controller methods.

userRouter.post('/signup', userController.signUp);
userRouter.post('/signin', userController.signIn);

export default userRouter;
```

Testing User APIs

We have to test the user sign-up and user sign-in APIs before proceeding to secure other routes.

- Open Postman to perform the testing both sign-up and sign-in requests.
- The URL for the sign-up request is specified as "localhost:3000/api/users/signup."
- Set up the request body in Postman for the sign-up API. It should contain the user's name, email, password, and type.
- Send the sign-up request. The response indicates that a new user has been created with an ID of 2.



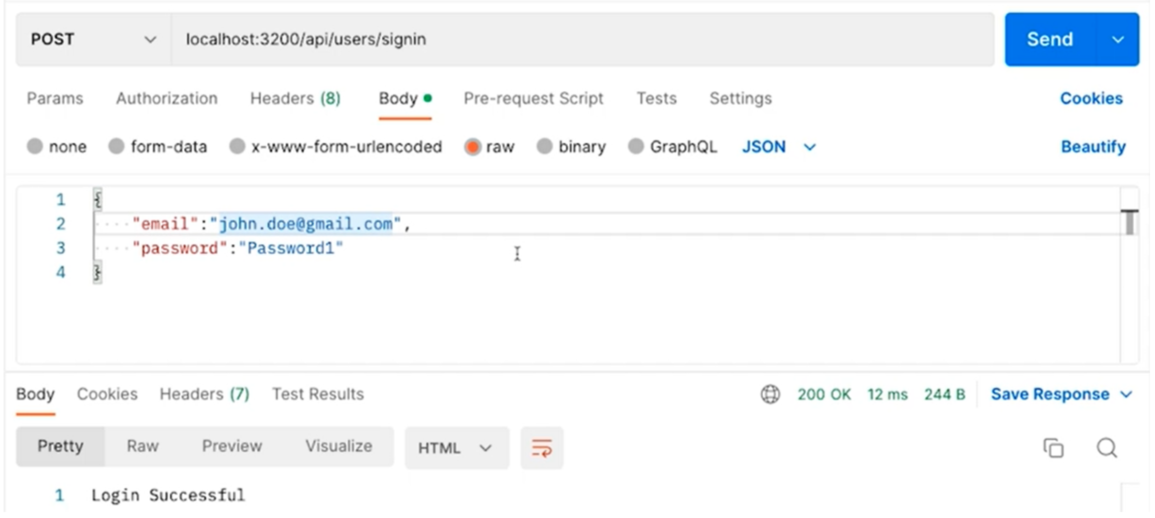
localhost:3200/api/users/signup

```
{
    "name":"John Doe",
    "email":"john.doe@gmail.com",
    "password":"Password1",
    "type":"Customer"
}
```

201 Created 43 ms 336 B

```
{"name":"John Doe","email":"john.doe@gmail.com","password":"Password1","type":"Customer","id":2}
```

- Test the sign-in API. The request body for sign-in only requires the user's email and password.
- Provide a valid email and password, and send the request. The response confirms a successful login with the message "Login successful."



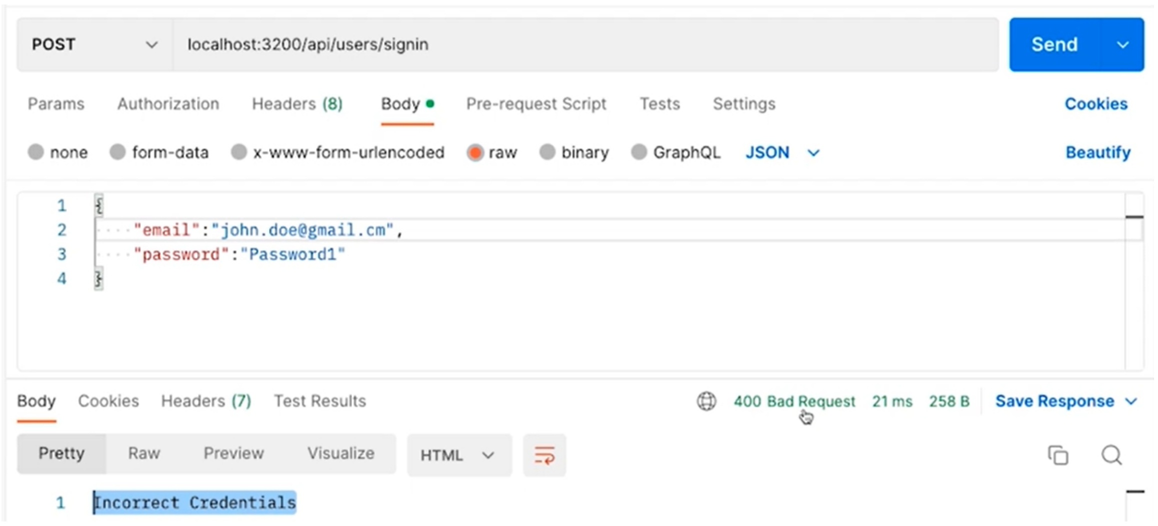
localhost:3200/api/users/signin

```
{
    "email":"john.doe@gmail.com",
    "password":"Password1"
}
```

200 OK 12 ms 244 B

```
Login Successful
```

- If incorrect credentials are provided, the response will indicate a bad request with the message "Incorrect credentials."



localhost:3200/api/users/signin

```
{
    "email":"john.doe@gmail.cm",
    "password":"Password1"
}
```

400 Bad Request 21 ms 258 B

```
Incorrect Credentials
```

- It is noted that although the sign-up and sign-in APIs are functioning, the application is not yet secure. Access to other APIs is possible without authentication.
- Simply exposing the login API does not make the application secure. Each request needs to be validated to ensure the user is logged in.

Basic Authentication

1. Objective: Implement basic authentication to secure the APIs in the application.
2. Basic authentication mechanism:
   - Credentials (username and password) provided by the client will be checked on each request.
   - Middleware named "basicAuth" will be used to simplify the authentication process and ensure data validation against attacks like SQL injections.
   - The "express-basic-auth" package will be installed to set up basic authentication.
3. Middleware Implementation:
   - Create a middleware named "basicAuth" in the "middlewares" folder.
   - The middleware will compare the received email and password with the user data stored in the user model.
   - If the credentials do not match, an error will be returned.
   - If the credentials are correct, the middleware will proceed to the next middleware.
   - Secure string comparison will be performed using the "safeCompare" function to protect against timing attacks.
   - Code Implementation:

```
import bAuth from 'express-basic-auth';
import UserModel from '../features/user/user.model.js';
```

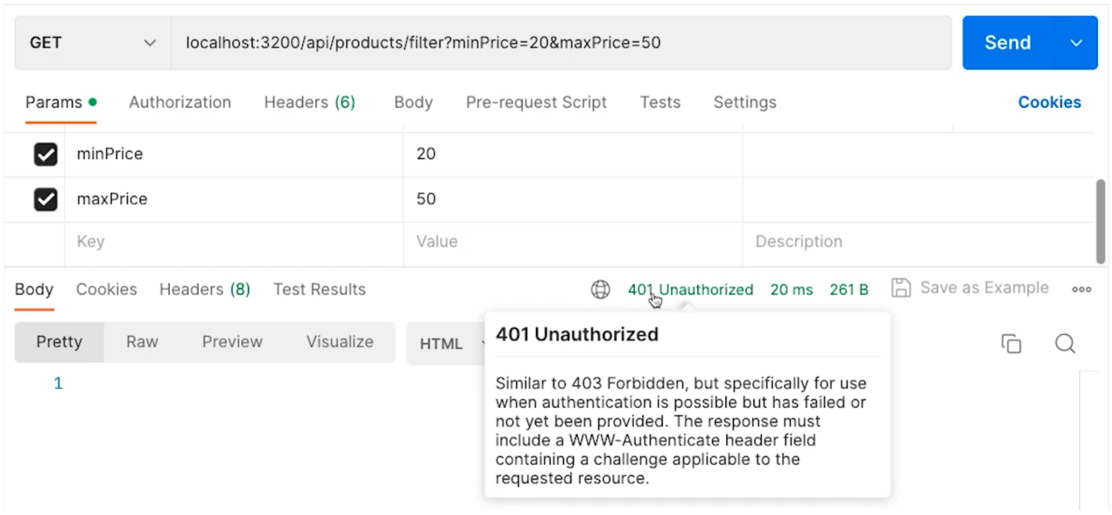
```
const basicAuthorizer = (username, password) => {
  // 1. Get users
  const users = UserModel.getAll();
  // 2. Compare email
  const user = users.find((u) =>
    bAuth.safeCompare(username, u.email)
  );
  if (user) {
    // 3. Compare password and return
    return bAuth.safeCompare(
      password,
      user.password
    );
  } else {
    // 4. Return error message
    return res.status(401).send('Unauthorized');
  }
};

const authorizer = bAuth ({
  authorizer: basicAuthorizer,
  challenge: true,
});
export default authorizer;
```

5. Usage:
   - Import the "authorizer" middleware.
   - Apply the "authorizer" middleware to the relevant API routes that require authentication (e.g., products APIs) to enforce authentication for accessing those routes.

```
server.use(
  '/api/products',
  authorizer,
  productRouter
);
```

Testing Basic Authentication

- Start the server using the command "node server".
- Use Postman for API testing. The "API products" request is selected, which has been secured by the authorizers.
- Accessing the API without providing any credentials results in a 401 unauthorized response, indicating the lack of authorization.

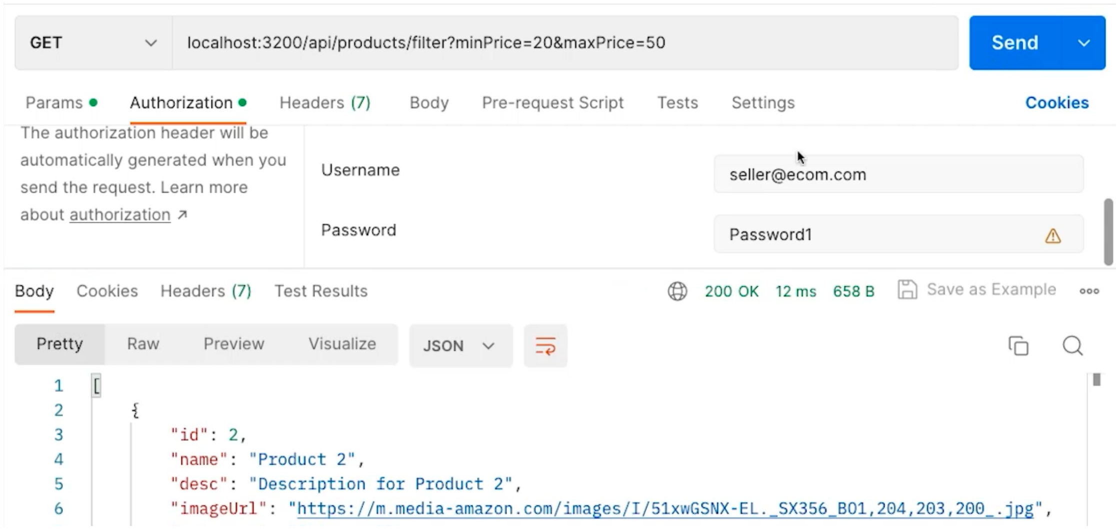


	Key	Value	Description
✓	minPrice	20	
✓	maxPrice	50	

401 Unauthorized

Similar to 403 Forbidden, but specifically for use when authentication is possible but has failed or not yet been provided. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource.

- Basic authentication credentials (username and password) are added to the Postman request under the Authorization tab.
- The "Basic Auth" option is selected, and the username and password are entered.
- Upon sending the request with the correct credentials, the data (all the products based on the filter) is successfully retrieved.



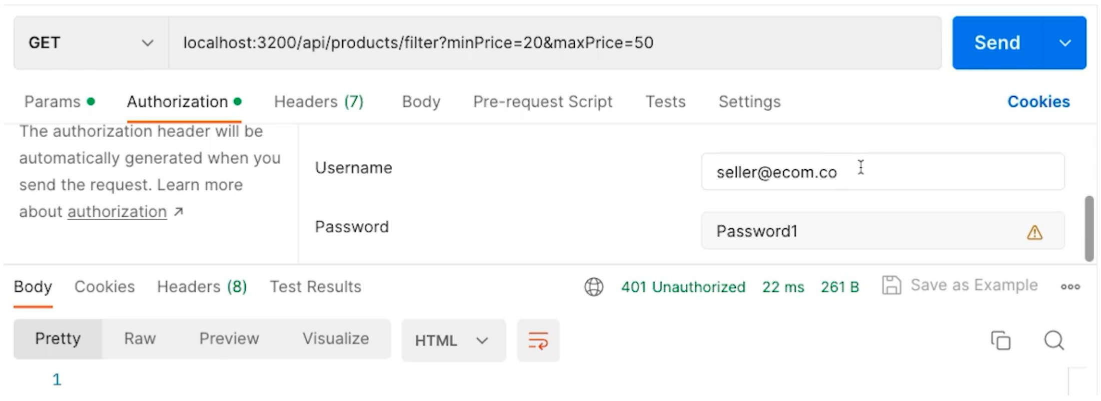
The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Username: seller@ecom.com

Password: Password1

```
[
    {
        "id": 2,
        "name": "Product 2",
        "desc": "Description for Product 2",
        "imageUrl": "https://m.media-amazon.com/images/I/51xwGSNX-EL._SX356_BO1,204,203,200_.jpg",
```

- Request with incorrect credentials returns a 401 unauthorized response.


GET localhost:3200/api/products/filter?minPrice=20&maxPrice=50

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Username: seller@ecom.co

Password: Password1

401 Unauthorized 22 ms 261 B

- Basic authentication requires providing username and password with each request to the server.
- The server verifies the provided credentials for every incoming request, granting access if the credentials are correct and returning an error response if they are incorrect.

Understanding JWT

Problems with the implementation of basic authentication

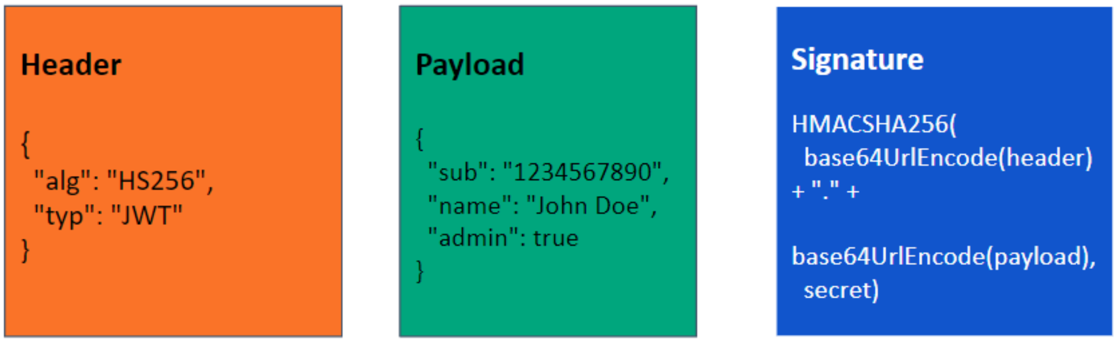
- **No Encryption:** The first problem is the lack of encryption in basic authentication. While tools like Postman use basic encoding, it can be easily decoded, so a strong encryption technique is needed to securely share credentials across applications.
- **Need Strong Password:** The second problem is that weak passwords can be easily cracked using brute force attacks. Brute force attacks involve trying different combinations of characters to guess the password, and passwords like "password1" are particularly vulnerable.
- **Client Needs to Store Credentials:** Another problem is that the client application needs to store the username and password, typically in the browser. This can be risky if the browser is accessed by multiple people because anyone can look into the browser's cookies or stored data to find the credentials.

JSON Web Token (JWT) for securing the application

- JWT is a token-based authentication method that creates a token instead of requiring credentials on every request. The token is scalable, easy to implement, and stateless.
- **Structure of JWT:** The token consists of three parts separated by dots: the header (algorithm and token type), the payload (user information and permissions), and the signature (encrypted information).

JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```



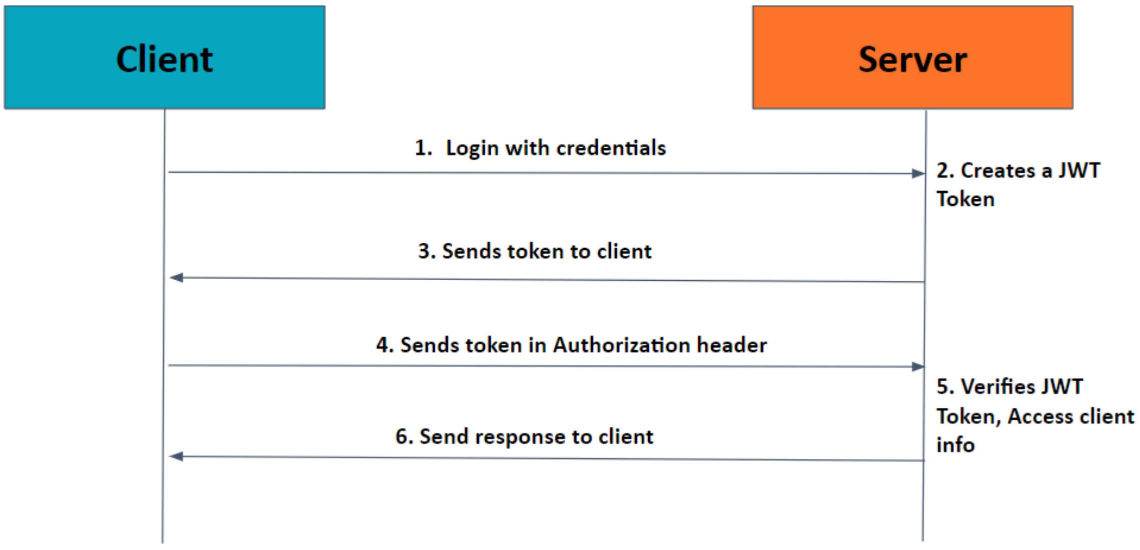
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

```
HMACSHA256(
  base64UrlEncode(header)
+ "." +

base64UrlEncode(payload),
  secret)
```

- Sensitive data should not be stored in the payload but mentions that user permissions are a suitable example of data to include.
- JWT is recommended because it provides a stateless behavior of authentication, allowing the server to verify the token without needing to store client information.
- Visit the [jwt.io](jwt.io) website to explore the details and examples of JWT tokens.

- **Process of using JWT :** It involves the client logging in with their credentials through the login API, the server verifying the credentials and generating a JWT token, and then sending the token back to the client. The client stores the token, typically in the browser, and includes it in the authorization header when making requests to secure APIs.


Client
Server
1. Login with credentials
2. Creates a JWT Token
3. Sends token to client
4. Sends token in Authorization header
5. Verifies JWT Token, Access client info
6. Send response to client

The server verifies the token and provides a response if the verification is successful. If the token has been modified or expired, an error with the status code 401 is returned.

JWT Authentication - 1

Implementing JWT authentication in our Express REST API application.

- Install JSON Web Token package, available on npmjs.com, as the tool to create and manage JWT tokens for Express.
- The first step is to install the JSON WebToken package using the command `npm install jsonwebtoken`
- Import the **'jsonwebtoken'** package as **'jwt'** in the code.
- Use **jwt.sign** method as mentioned as the function to create the token.
- In the sign-in API, after calling the model's signIn method, instead of sending a "login successful" message, create the token and send it to the client.
- The sign function of JWT is used to sign the token with the provided algorithm, private key, and payload.

Changes in **user.controller.js** file:

```
import UserModel from './user.model.js';
import jwt from 'jsonwebtoken';

export default class UserController {
  signUp(req, res) {
    const {
      name,
      email,
      password,
      type,
    } = req.body;
    const user = UserModel.signUp(
      name,
      email,
      password,
      type
    );
    res.status(201).send(user);
  }

  signIn(req, res) {
    const result = UserModel.signIn(
      req.body.email,
      req.body.password
    );
    if (!result) {
      return res
        .status(400)
        .send('Incorrect Credentials');
    } else {
      // 1. Create token.
      const token = jwt.sign(
        {
          userID: result.id,
          email: result.email,
        },
        'AIb6d35fvJM4O9pXqXQNla2jBCH9kuLz',
        {
          expiresIn: '1h',
        }
      );

      // 2. Send token.
      return res.status(200).send(token);
    }
  }
}
```

- The payload should not contain sensitive data like passwords but can include information such as user ID and authorization permissions.
- The user ID is stored in the payload using the value result.id.
- The private key, which is used for signing and verifying the token, should be a strong and secure key. Online key generators are recommended for generating such keys.
- Set the "**expiresIn**" option to one hour, indicating that the token will be invalid after that time.
- After creating the token, return it to the client with a status code of 200 (OK).
- The modified login API now generates a token and sends it back to the client.

- The token serves as the client's validation key and can be used to access secure routes.

JWT Authentication - 2

- Create a new middleware called "jwt.middleware.js"
- Define the middleware function with the parameters: request, response, and next.
- **Reading the token:**
  - Retrieve the token from the client.
  - Check if the token exists; if not, return an error message (unauthorized access) with a status code of 401.
- **Checking token validity:**
  - Import the JSON Web Token (JWT) library.
  - Use the JWT library's "verify" function to check if the token is valid.
  - Pass the token and the signing key used during token creation to the "verify" function.
  - Wrap the verification process in a try-catch block to handle errors.
  - If any error occurs during verification, return an unauthorized access error (status code 401).
  - If the verification is successful, retrieve the payload from the verified token.
  - Print the payload for testing purposes.
  - If the token is present, not empty, and valid, call the next middleware in the pipeline.
  - Export the JWT Auth middleware to be used in securing specific routes.
  - Apply the JWT middleware to secure routes.
  - Replace the existing authorizer with the JWT Auth middleware.
  - **jwt.middleware.js** file

```
import jwt from 'jsonwebtoken';

const jwtAuth = (req, res, next) => {
  // 1. Read the token.
  const token = req.headers['authorization'];

  console.log(token);
  // 2. if no token, return the error.
  if (!token) {
    return res.status(401).send('Unauthorized');
  }
  // 3. check if token is valid.
  try {
    const payload = jwt.verify(
      token,
      'AIb6d35fvJM4O9pXqXQNla2jBCH9kuLz'
    );
    console.log(payload);
  } catch (err) {
    // 4. return error.
    console.log(err);
    return res.status(401).send('Unauthorized');
  }
  // 5. call next middleware
  next();
};

export default jwtAuth;
```

- **Testing the API**:
  - Use Postman to test the API without any authentication, resulting in an "unauthorized" response.
  - Generate a token and include it in the authorization header.
  - Send the request the response should be successful.
  - Verify the payload received from the token.

Summarising it

Let's summarise what we have learned in this module:

- Created APIs for rating a product, adding products to a cart, retrieving products from the cart, and removing a product from the cart.
- Learned about reasons for securing the application including controlled access and data privacy.
- Created user API for user registration and login, including sign-up and sign-in functionalities.
- Tested user sign-up and sign-in APIs using Postman.

- Implemented Basic authentication using middleware to check credentials on each request.
- Learned about JWT (JSON Web Token) as a token-based authentication method for securing the application.
- Implemented JWT authentication in the Express REST API application using the jsonwebtoken package.

Some Additional Resources:

- **Production Best Practices: Security**
- **How To Add Basic Authentication To An Express Web App**
- **AUTHENTICATION AND AUTHORIZATION IN EXPRESS.JS**