**Chapter 7**

**Basic Authentication and Authorization with JSON Web Tokens**

**Authentication and Authorization**

When hosting web services, there would be certain web api methods that you want to restrict access to. In such cases, you might require the user to provide some user credentials to authenticate himself/herself.

Authentication is the process of determining whether someone is who he/she declares himself/herself to be. Authentication technology provides access control for systems by checking to see if a user's credentials(usually some login id and password) match the credentials in a database of authorized users.

However, the web's application protocols, HTTP and HTTPS, are stateless, meaning that strict authentication would require users to re-authenticate each time they access a protected resource. Rather than burden end users with that process for each interaction over the web, web systems can rely on token-based authentication, in which authentication is performed once at the start. The authenticating system issues a signed authentication token to the end-user application, and then that token is appended to every request from the client when a protected resource is required. This unique token verifies your identity and can authenticate who you are, and authorize various resources you have access to.

In this chapter, we will be looking at JSON Web Tokens (JWT) as a means of helping you perform authentication and authorization.

Wikipedia Definition: JSON Web Token is a JSON-based open standard (RFC 7519) for creating access tokens that assert some number of claims. For example, a server could generate a token that has the claim "logged in as admin" and provide that to a client. The client could then use that token to prove that it is logged in as admin. The tokens are signed by one party's private key (usually the server's), so that both parties (the other already being, by some suitable and trustworthy means, in possession of the corresponding public key) are able to verify that the token is legitimate. The tokens are designed to be compact, URL-safe and usable especially in a web-browser single-sign-on (SSO) context. JWT claims can be typically used to pass identity of authenticated users between an identity provider and a service provider, or any other type of claims as required by business processes.

The JWT is composed of a Header, a Payload, and a signature that proves the integrity of the message to the receiving server. Content encoded inside a JWT is digitally signed, either using a secret utilizing the HMAC algorithm, or by leveraging the Public Key Infrastructure (PKI) model with a private/public RSA configuration.

## JWT Structure - Wikipedia

|  |  |  |
| --- | --- | --- |
| **Header** | {  **"alg"** : "HS256",  **"typ"** : "JWT"  } | Identifies which algorithm is used to generate the signature  HS256 indicates that this token is signed using HMAC-SHA256.  Typical cryptographic algorithms used are [HMAC](https://en.wikipedia.org/wiki/HMAC) with [SHA-256](https://en.wikipedia.org/wiki/SHA-256) (HS256) and [RSA signature](https://en.wikipedia.org/wiki/Digital_signature) with SHA-256 (RS256). JWA (JSON Web Algorithms) [RFC 7518](https://tools.ietf.org/html/rfc7518) introduces many more for both authentication and encryption.[[9]](https://en.wikipedia.org/wiki/JSON_Web_Token#cite_note-9) |
| **Payload** | {  **"loggedInAs"** : "admin",  **"iat"** : 1422779638  } | Contains a set of claims. The JWT specification defines seven Registered Claim Names which are the [standard fields](https://en.wikipedia.org/wiki/JSON_Web_Token#Standard_fields) commonly included in tokens[[1]](https://en.wikipedia.org/wiki/JSON_Web_Token#cite_note-Bradley-1). Custom claims are usually also included, depending on the purpose of the token.  This example has the standard Issued At Claim (iat) and a custom claim (loggedInAs). |
| **Signature** | HMAC-SHA256(  base64urlEncoding(header) + '.' +  base64urlEncoding(payload),  secret  ) | Securely validates the token. The signature is calculated by encoding the header and payload using [Base64url Encoding](https://en.wikipedia.org/wiki/Base64#URL_applications) and concatenating the two together with a period separator. That string is then run through the cryptographic algorithm specified in the header, in this case HMAC-SHA256. The *Base64url Encoding* is similar to [base64](https://en.wikipedia.org/wiki/Base64), but uses different non-alphanumeric characters and omits padding. |

The three parts are encoded separately using [Base64url Encoding](https://en.wikipedia.org/wiki/Base64#URL_applications), and concatenated using periods to produce the JWT:

**const** token = base64urlEncoding(header) + '.' + base64urlEncoding(payload) + '.' + base64urlEncoding(signature)

The above data and the secret of "secretkey" creates the token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dnZWRJbkFzIjoiYWRtaW4iLCJpYXQiOjE0MjI3Nzk2Mzh9.gzSraSYS8EXBxLN\_oWnFSRgCzcmJmMjLiuyu5CSpyHI

Our focus for this chapter would be the creation of such a mechanism to use JWT as a means to authorize usage of protected resources after authentication is successfully done.

Let’s consider a typical scenario which we will attempt to implement. The user will send a request with the required credentials such as email and password to the server. The server checks to see if the credentials are valid. If they are, the server creates a JWT token using the desired payload and a secret key. Then the server sends it back to the client. The client, in turn, saves the token to use it in every other request the user will send. The practice of adding a token to the request headers is a way of authorizing the user to access resources.

In Node JS, there is a JWT library that we can use to create JWT tokens and perform JWT verification.

Let’s continue on with the MYFIRSTWS project we did previously and install the jsonwebtoken library in our project directory:

|  |
| --- |
| npm install jsonwebtoken --save |

Then we define a new function in user.js (inside userDB object) to handle the login checking based on the email and password submitted by the user. This function will return 1 representing success and 0 representing failure upon successful query of the database:

|  |
| --- |
| loginUser: function (email,password, callback) {    var conn = db.getConnection();  conn.connect(function (err) {  if (err) {  console.log(err);  return callback(err,null);  }  else {  console.log("Connected!");    var sql = 'select \* from user where email=? and password=?';    conn.query(sql, [email,password], function (err, result) {  conn.end();    if (err) {  console.log(err);  return callback(err,null);    } else {    var msg=”{\”result\”:\””+result.length+”\”}”;  return callback(null, msg);    }  });    }    }); |

We can then define a new route in app.js to handle the login:

|  |
| --- |
| app.post('/api/login',function(req,res){  var email=req.body.email;  var password=req.body.password;  user.loginUser(email,password,function(err,result){  if(!err){  res.send("{\"result\":\""+result +"\"}");  }else{  res.status(500);  res.send(err.statusCode);  }  });  }); |

You can do your own testing to ascertain that the basic login function works.

With the login checking function done, the next step is for us to configure and create the json web token which we will return upon successful login. To do this, we would first define a config js file to hold the secret key.

In the main project folder, create a new config.js file which we will define a secret key for use in our json web token generation:

|  |
| --- |
| var secret='s12xyz00'; //your own secret key  module.exports.key=secret; |

Once this is done, let’s proceed back to user.js to import config.js, jsonwebtoken and also modify the loginUser function. The jsonwebtoken library contains a sign function which we can use to generate a json web token to return to the caller of the webservice upon successful verification of the login credentials. We will also encode the user’s role and user id in the jwt. If validation fails, we will return a null value for the token.

|  |
| --- |
| var config=require('../config.js');  var jwt=require('jsonwebtoken');  ….  loginUser: function (email,password, callback) {    var conn = db.getConnection();  conn.connect(function (err) {  if (err) {  console.log(err);  return callback(err,null);  }  else {    console.log("Connected!");    var sql = 'select \* from user where email=? and password=?';    conn.query(sql, [email,password], function (err, result) {  conn.end();    if (err) {  console.log(err);  return callback(err,null);    } else {    //console.log(config.key);  var token="";  if(result.length==1){  token=jwt.sign({id:result[0].userid,role:result[0].role},config.key,{  expiresIn:86400//expires in 24 hrs  });    }    return callback(null,token);    }  });    }    });  }  … |

Graphical user interface, text, application, email

Description automatically generated

**Verifying the token**

Now that we have created the token, we would also need to authorize a verified user through the token when a “protected” webservice route/resource is being called.

Whenever the user wants to access a protected route or resource, the user 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> |

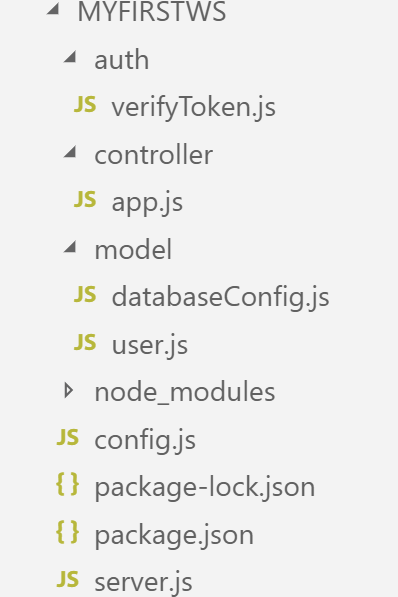
Let us assume the insert user Post method web api route ('/api/user') can only be called by authenticated users with a valid token. Let us create a new folder **auth** in the main project folder. Inside the auth folder, we will create a new verifyToken.js file which will have a middleware function verifyToken that checks for the presence of the token and verify its validity. Source code of verifyToken.js is listed below:

|  |
| --- |
| var jwt=require('jsonwebtoken');  var config=require('../config');  function verifyToken(req,res,next){  console.log(req.headers);  var token=req.headers['authorization']; //retrieve authorization header’s content  console.log(token);  if(!token || !token.includes('Bearer')){ //process the token  res.status(403);  return res.send({auth:'false',message:'Not authorized!'});  }else{  token=token.split('Bearer ')[1]; //obtain the token’s value  console.log(token);  jwt.verify(token,config.key,function(err,decoded){//verify token  if (err){  res.status(403);  return res.end({auth:false,message:'Not authorized!'});  }else{  req.userid=decoded.userid; //decode the userid and store in req for use  req.role=decoded.role; //decode the role and store in req for use  next();  }  });  }  }  module.exports=verifyToken; |

Finally to apply the verifyToken middleware in our application, we modify app.js accordingly.

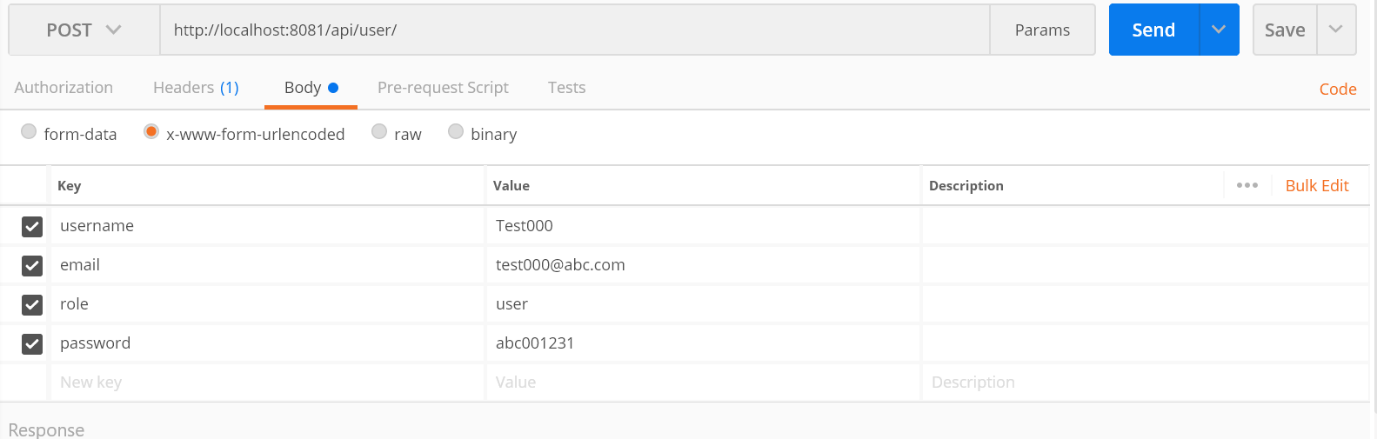
|  |
| --- |
| var verifyToken=require('../auth/verifyToken.js');  …  app.post('/api/user',verifyToken,function(req,res){  ……  }); |

Below shows the folder and files in the MYFIRSTWS Project after you complete the above enhancement.

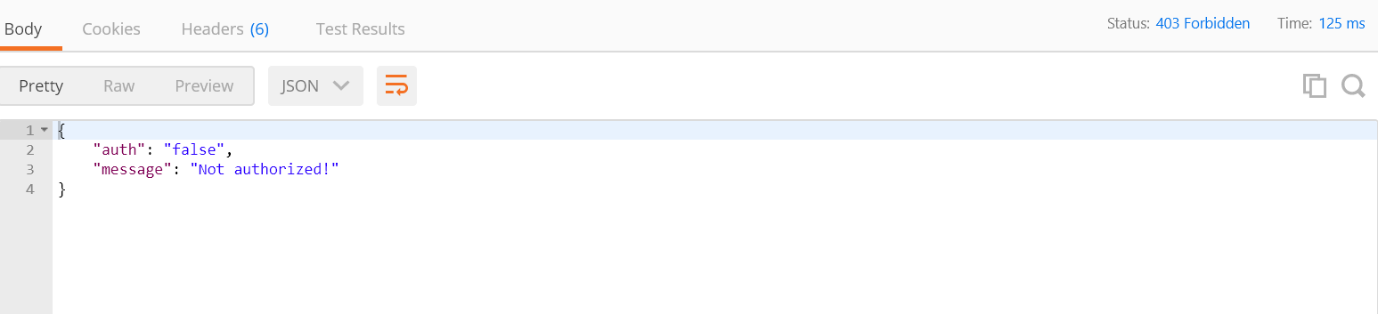


**Testing**

1. Use POSTMAN and target <http://localhost:8081/api/user/> with POST method and the below Message body parameters.

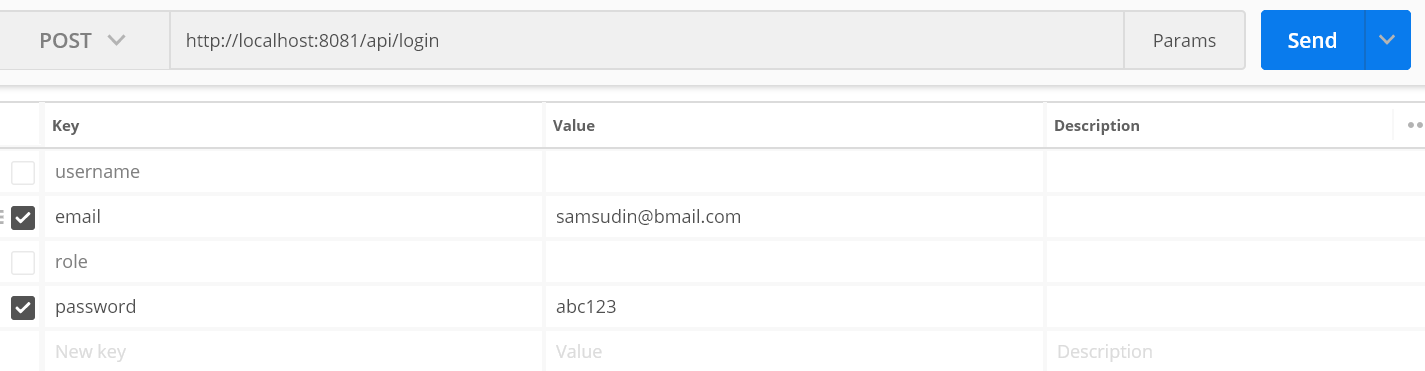


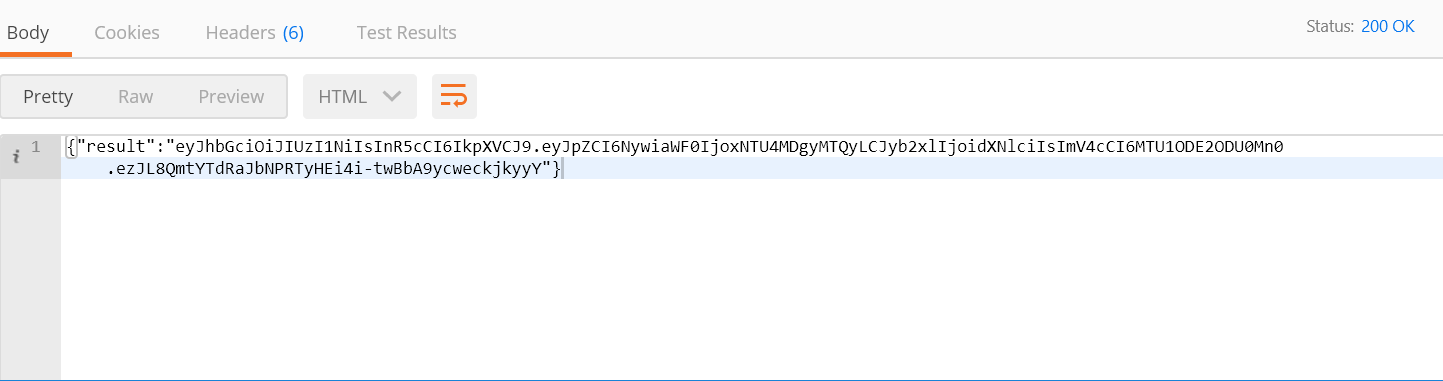
Once sending the request, we would notice that a message indcating not authorized is returned with status 403.



2)

Now we try to obtain a valid JWT by logging in with a user account. You can use your own email and password based on the actual data in your database.

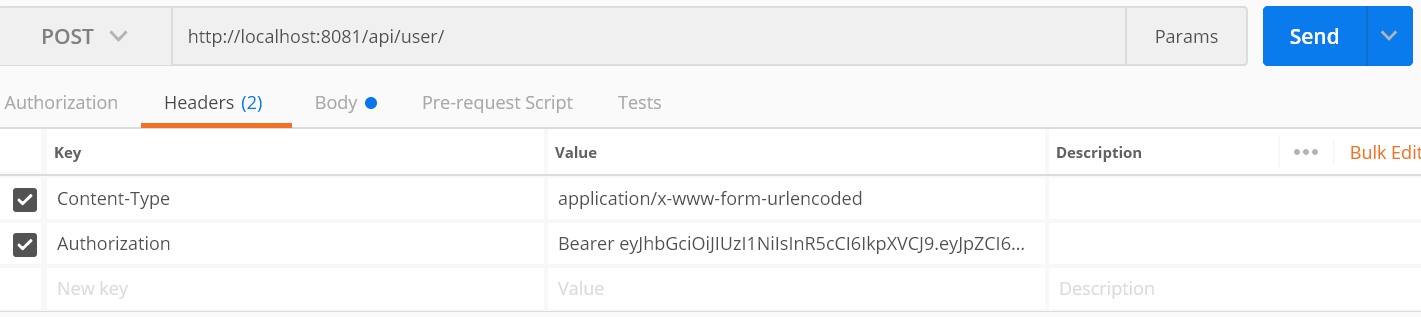


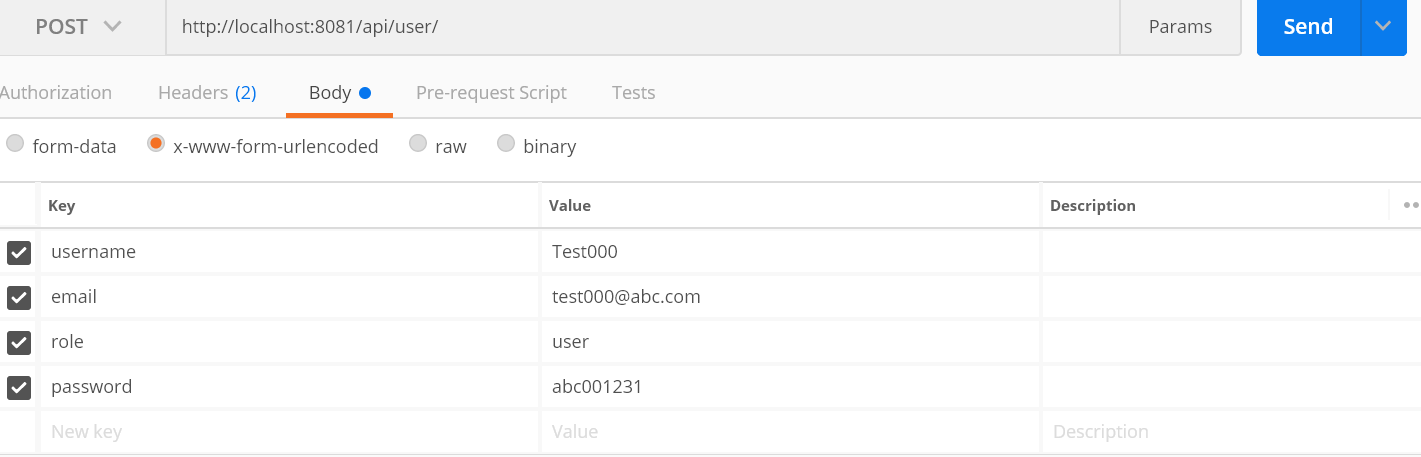


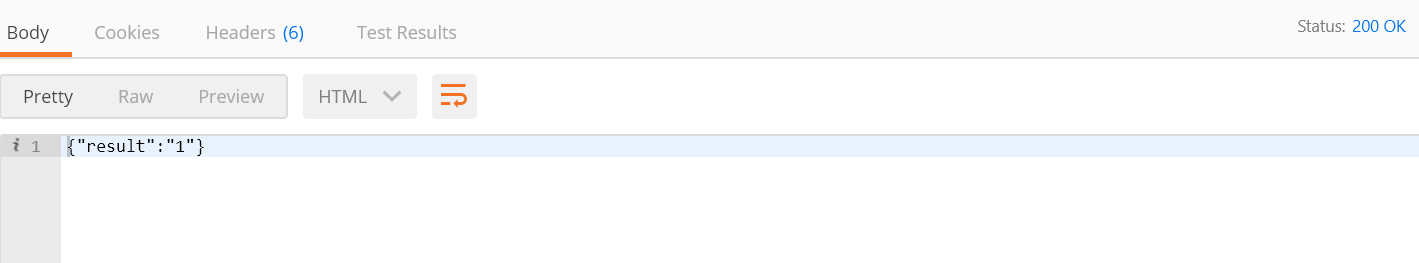
In this sample run, the following result key is returned: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NywiaWF0IjoxNTU4MDgyMTQyLCJyb2xlIjoidXNlciIsImV4cCI6MTU1ODE2ODU0Mn0.ezJL8QmtYTdRaJbNPRTyHEi4i-twBbA9ycweckjkyyY

For web APIs that require special access permission, we can use this jwt for verification until it expires.

Next we attempt to access the POST method for http://localhost:8081/api/user/







Record successfully inserted and authenticated.