CHAPTER SIX MONGODB

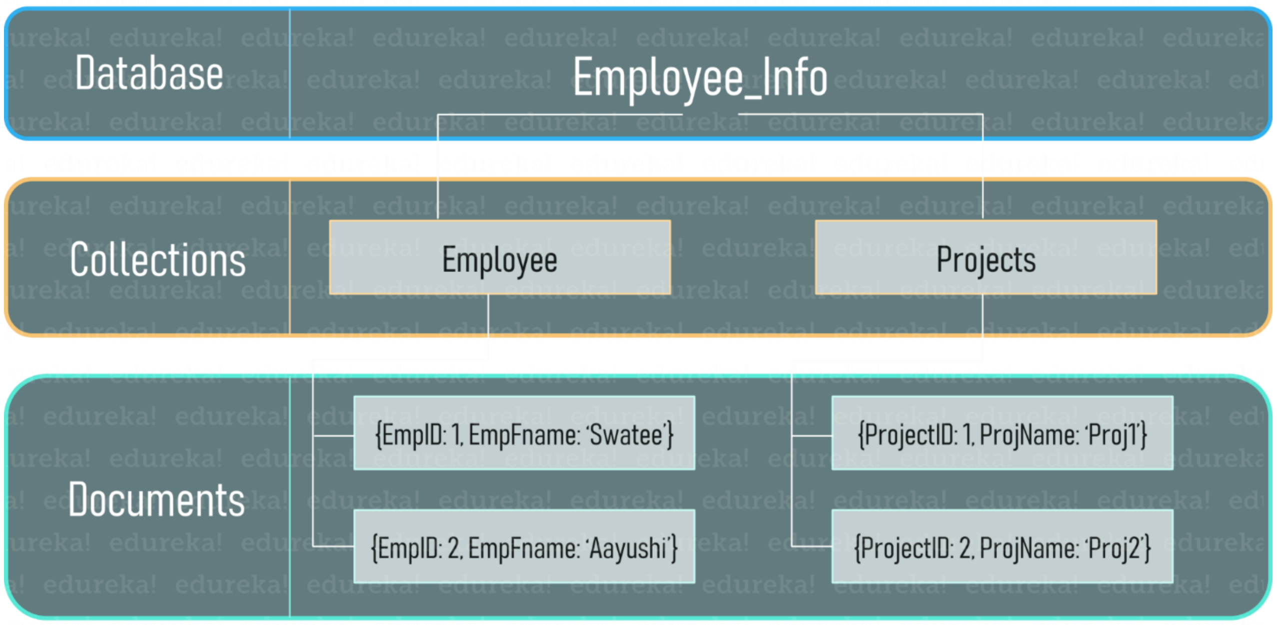
6.1 Introduction

Node.js is the most popular JavaScript framework when it comes to high-speed application development. **Node.js Professionals** often opt for a NoSQL database which can keep up with Node.js speed all while maintaining the performance of the application. MongoDB is a perfect fit for this kind of requirement as it has a very fast development cycle and performs quite efficiently.

6.2 What is NoSQL Database?

NoSQL, or most commonly known as Not only SQL database, provides a mechanism for storage and retrieval of unstructured data. This type of database can handle a humongous amount of data and has a dynamic schema. So, a NoSQL database has no specific query language, no or a very few relationships, but has data stored in the format of collections and documents.

So, a database can have a **'n'** number of collections and each collection can have 'm' number of documents. Consider the example below.


Database
Employee_Info
Collections
Employee
Projects
Documents
{EmpID: 1, EmpFname: 'Swatee'}
{EmpID: 2, EmpFname: 'Aayushi'}
{ProjectID: 1, ProjName: 'Proj1'}
{ProjectID: 2, ProjName: 'Proj2'}

As you can see from the above image, there is an Employee Database which has 2 collections i.e. the Employee and Projects Collection. Now, each of these collections has Documents, which are basically the data values. So, you can assume the *collections to be your tables and the Documents to be your fields in the tables*.

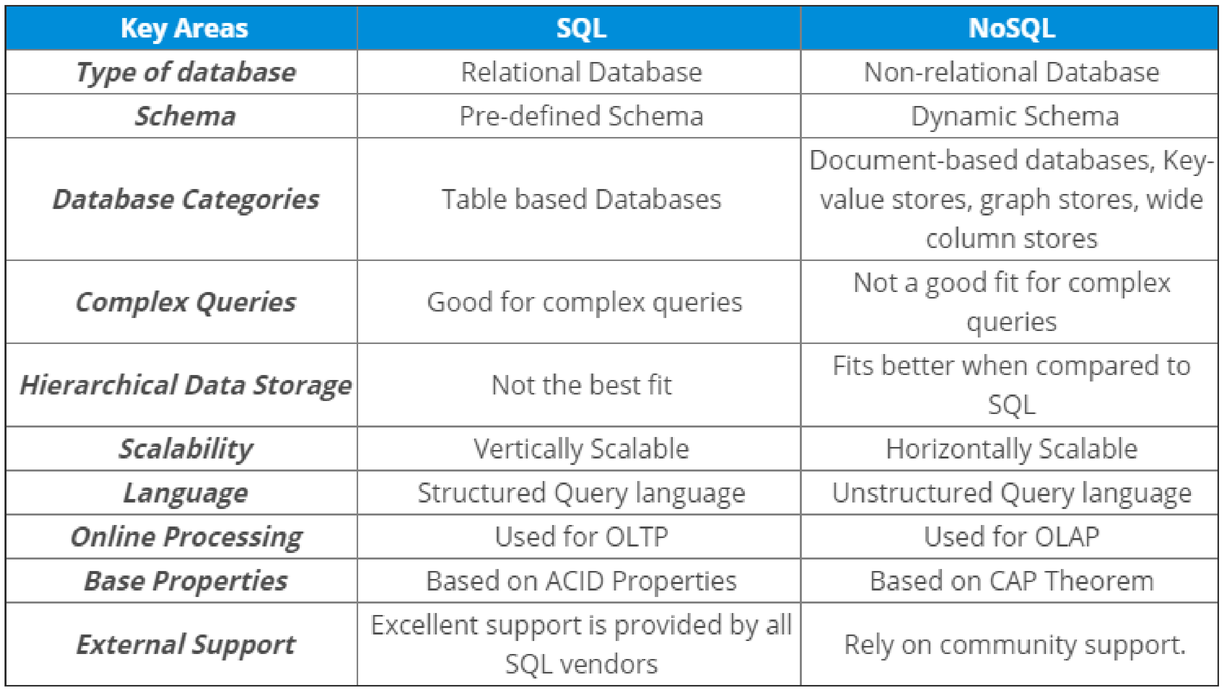
There is a list of NoSQL databases which are used quite heavily in the industry, some of which I have listed below:

1. MongoDB
2. Hbase
3. Cassandra
4. Amazon SimpleDB
5. Hypertable

6.3 SQL vs NoSQL

So, in this face off, I will be comparing both these databases based on the following grounds:

1. Type of Database
2. Schema
3. Database Categories
4. Complex Queries
5. Hierarchical Data Storage
6. Scalability
7. Language
8. Online Processing
9. Base Properties
10. External Support



Key Areas	SQL	NoSQL
Type of database	Relational Database	Non-relational Database
Schema	Pre-defined Schema	Dynamic Schema
Database Categories	Table based Databases	Document-based databases, Key-value stores, graph stores, wide column stores
Complex Queries	Good for complex queries	Not a good fit for complex queries
Hierarchical Data Storage	Not the best fit	Fits better when compared to SQL
Scalability	Vertically Scalable	Horizontally Scalable
Language	Structured Query language	Unstructured Query language
Online Processing	Used for OLTP	Used for OLAP
Base Properties	Based on ACID Properties	Based on CAP Theorem
External Support	Excellent support is provided by all SQL vendors	Rely on community support.

Table 1: Differences between SQL and NoSQL – SQL vs NoSQL

6.4 Introduction to MongoDB

MongoDB is an open source non-relational database that stores the data in the form of collections and documents. This kind of databases preserves most of the functionalities while offering horizontal scalability. This eases the work of a developer by providing persistence to the data and enhancing agility.

MongoDB stores the JSON documents in the form of collections having dynamic schemas. It stores all the related information together which enhances the speed of query processing. This way, it also helps in bringing down the gap between the key-value stores and relational databases.

Below I have listed down a few of the most intriguing features of MongoDB:

- **Indexing:** It makes use of indexes that helps in improving the search performance.
- **Replication:** MongoDB distributes the data across different machines.
- **Ad-hoc Queries:** It supports ad-hoc queries by indexing the BSON documents & using a unique query language.
- **Schemaless:** It enhances the flexibility of the data and needs no script to modify or update data.
- **Sharding:** It makes use of sharding which eases the deployment of very large data sets and provides high throughput operations.

6.5 Node.js MongoDB Demo

Here I will be creating a CRUD application for Course Management with the help of Node.js and Express.js and use MongoDB to store the data. In this application, I will be taking course details like name, id, duration, and fee as inputs. For that, I will be creating a few view files which will act as an interface. Then in order to handle the data, I will be needing a controller as well which will help in manipulating the data. Finally, I will be needing a few model files to store the data. So basically, I will be following an MVC pattern for this application development.

open the command prompt and navigate to your project directory. Now you need to set up the project configurations for that, type in the below command and provide the necessary details:

```
npm init
```

Now, you need to install the required packages. So, in this project, I am using the below packages:

- **express.js:** It is a web framework.
- **express-handlebars:** It is a template engine and helps in creating client-side applications.
- **mongoose:** Helps in communicating with MongoDB.
- **body-parser:** Helps in converting the POST data into the request body.
- **nodemon:** Helps in automatically restarting the server whenever the code changes.

In order to install these packages, type in the following command:

```
npm i --s express express-handlebars mongoose body-parser
```

Since I want to install nodemon such that it can access any file in the directory, I will be installing it with the global command:

```
npm i -g nodemon
```

Create the file to establish connectivity between Node.js and MongoDB. For that, first, you need to create a folder inside the project directory and name it ‘model’. Inside this folder, create a javascript file with the name ‘**mongodb.js**‘ and type in the below code:

mongodb.js

```
const mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/mongo-demo', {
   useNewUrlParser: true,
   useUnifiedTopology: true
})
.then(() => console.log('connected to mongodb successfully....'))
.catch(err =>console.log('failed to connect to mongodb',err));

//Connecting Node and MongoDB
require('./course.model');
```

Now, you need to define the schema of your course database. For that, create a new JS file within the model folder and name it ‘**course.model.js**‘. So, I am using four fields in my course object I am using four fields which are name, id, duration, and fee. To create this file, type in the below-given code.

course.model.js

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

//Attributes of the Course object
var courseSchema = new mongoose.Schema({
courseName: {
type: String,
required: 'This field is required!'
},
courseId: {
type: String
},
author: {
type: String
},
courseDuration: {
type: String
},
courseFee: {
type: String
},
discount: {
type: Number
}
});

mongoose.model('Course', courseSchema);
```

Now, you need to create the root file called ‘**script.js**‘. This file is the entry point of this application and will contain all the connection paths in it. You need to be really careful while providing the paths in this file as it might result in an error or application failure. Along with this, it is also responsible for invoking the server and establish the connection. In order to create this file, type in the below code:

script.js

```
require('./model/mongodb')
const courseController = require('./controllers/courseController');

//Import the necessary packages
const express = require('express');
var app = express();
const bodyparser = require('body-parser');

app.use(bodyparser.urlencoded({extended: true}));
app.use(bodyparser.json());

//Create a welcome message and direct them to the main page
app.get('/', (req, res) => {
   res.send('Welcome to our app');
});

//Set the Controller path which will be responding the user actions
app.use('/api/courses', courseController);

//Establish the server connection
//PORT ENVIRONMENT VARIABLE
```

```
const port = process.env.PORT || 8080;
app.listen(port, () => console.log(`Listening on port ${port}..`));
```

Next, in order to handle the user requests, you need to create the router file. For that first, create a folder and name it ‘controller’ and within this folder create a file with the name ‘**courseController.js**‘. In this file, we will be dealing with the CRUD operations related to the employee. Below is the code for creating this file:

courseController.js

```
//Import the dependencies
const express = require('express');
const mongoose = require('mongoose');
//Creating a Router
var router = express.Router();
//Create Course model CLASS
const Course = mongoose.model('Course');

//Router Controller for CREATE request
router.post('/', (req,res) => {
insertIntoMongoDB(req, res);
});
 //Router Controller for UPDATE request
router.put('/', (req,res) => {
    updateIntoMongoDB(req, res);
});

//Creating function to insert data into MongoDB
function insertIntoMongoDB(req,res) {
var course = new Course();
course.courseName = req.body.courseName;
course.courseId = req.body.courseId;
course.courseDuration = req.body.courseDuration;
course.courseFee = req.body.courseFee;
course.author = req.body.author;
course.discount = req.body.discount;
course.save()
    .then((courseSaved) => res.send(courseSaved).status(201))
    .catch(err => res.send(err).status(400));
}
```

```
//Creating a function to update data in MongoDB
function updateIntoMongoDB(req, res) {
Course.findOneAndUpdate({ _id: req.body._id }, req.body, { new: true })
    .then(course => res.send(course))
    .then(err => res.send(err).status(400));
}
//Router to retrieve the complete list of available courses
router.get('/', (req,res) => {
Course.find()
    .then(courses => res.send(courses))
    .catch(err => res.send(err).status(404));
});
//Comparision operators
// eq (equal)
//ne (not equal)
//gt (greater than)
//gte (greater than or equal to)
//lt (less than)
//lte (Less than or equal to)
//in
//nin (not in)
router.get('/comparision', (req,res) => {
    Course
    //.find({discount:{$gt: 100}})
    .find({discount:{$in: [100,150,200]}})
    .sort({courseName:1})
        .then(courses => res.send(courses))
        .catch(err => res.send(err).status(404));
    });
    //or
    //and
    router.get('/logical', (req,res) => {
        Course
        .find()
        //.or([{author:'Stanley'},{discount:{$in:[100,150]}}])
        .and([{discount:{$gte:100}},{author:'Stanley'}])
        .sort({courseName:1})
            .then(courses => res.send(courses))
            .catch(err => res.send(err).status(404));
        });

        router.get('/expressions', (req,res) => {
            Course
            //starts with
            .find({ author: /^sta/i})
            //ends with
            //.find({author:/Sta$/i})
            //contains
           //.find({author:/.*st.*/i})
            //.or([{author:'Stanley'},{discount:{$in:[100,150]}}])
            .and([{discount:{$gte:100}},{author:'Stanley'}])
            .sort({courseName:1})
                .then(courses => res.send(courses))
                .catch(err => res.send(err).status(404));
            });
router.get('/advanced/:author', (req,res) => {
    Course.find({author: req.params.author})
    .select({author:1,courseId:1})
        .then(courses => res.send(courses))
        .catch(err => res.send(err).status(404));
    });

router.get('/count/:author', (req,res) => {
    Course
    .find({author: req.params.author})
            .then(count => res.send(count))
            .catch(err => res.send(err));

    });
//Router to update a course using it's ID
router.get('/:id', (req, res) => {
Course.findById(req.params.id)
    .then(course => res.send(course))
    .catch(err => res.send(err).status(404));
});

//Router Controller for DELETE request
router.delete('/:id', (req, res) => {
Course.findByIdAndRemove(req.params.id)
    .then(course => res.send(course))
    .catch(err => res.send(err).status(404));
});
//count documents
function dispayCount(author){
Course.find({author:author}).countDocuments()
    .then(count =>console.log({total:count }))
    .catch(err => console.error(err));
}
//dispayCount('Stanley')
module.exports = router;
```

Comparison Query Operators

NOTE
For details on specific operator, including syntax and examples, click on the specific operator to go to its reference page.

For comparison of different BSON type values, see the specified BSON comparison order.

Name	**Description**
$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$in	Matches any of the values specified in an array.
$lt	Matches values that are less than a specified value.

Name	Description
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$nin	Matches none of the values specified in an array.

Logical Query Operators

NOTE
For details on specific operator, including syntax and examples, click on the specific operator to go to its reference page.

Name	Description
$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.

Name	**Description**
$not	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
$nor	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.
$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.

CHAPTER SEVEN: Authentication and authorization

7.1 What Is Authentication?

Authentication is the act of validating that users are who they claim to be. Passwords are the most common authentication factor—if a user enters the correct password, the system assumes the identity is valid and grants access.

Other technologies such as One-Time Pins, authentication apps, and even biometrics can also be used to authenticate identity. In some instances, systems require the successful verification of more than one factor before granting access. This multi-factor authentication (MFA) requirement is often deployed to increase security beyond what passwords alone can provide.

7.2 What Is Authorization?

Authorization in system security is the process of giving the user permission to access a specific resource or function. This term is often used interchangeably with access control or client privilege. Giving someone permission to download a particular file on a server or providing individual users with administrative access to an application are good examples. In secure environments, authorization must always follow authentication—users should first prove that their identities are genuine before an organization's administrators grant them access to the requested resources.

7.3 Authentication vs. Authorization

Let's use an analogy to outline the differences. If you need to enter a house and the door is locked, you need a set of keys to open it. Unlocking the door with the correct key and gaining access to the house verifies you have the right to enter. This process is authentication. The lock on the door only grants access to someone with the correct key, in much the same way that a system only grants access to users that have the correct credentials.

However, once you have entered the house, you may not have the owner's permission to access certain areas or appliances. Imagine that your neighbor has asked you to feed her pets while she is away. In this example, you have the authorization to access the kitchen and open the cupboard storing the pet food. However, you can't go into your neighbor's bedroom as she did not explicitly permit you to do so. Even though you had the right to enter the house (authentication), your neighbor only allowed you access to certain areas (authorization).

7.4 Hashing Passwords with Node.js and Bcrypt

The bcrypt library on NPM makes it really easy to hash and compare passwords in Node. If you're coming from a PHP background, these are roughly equivalent to password_hash() and password_verify().

Installing

To use the library, simply install with NPM:

```
npm install --save bcrypt
```

Then include it in your module:

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

Creating and Verifying a Password Hash

Bcrypt supports both sync and async methods. The asynchronous approach is recommended because hashing is CPU intensive, and the synchronous version will block the event loop and prevent your app from handling other requests until it finishes.

Thus, while the sync version is more convenient, it's best to stick with async if you're concerned about performance.

Asynchronous Version

Hashing a password is as simple as this. The second argument is the number of rounds to use when generating a salt.

```
bcrypt.hash('myPassword', 10, function(err, hash) {
  // Store hash in database
});
```

To verify the password later on:

```
bcrypt.compare('somePassword', hash, function(err, res) {
  if(res) {
   // Passwords match
  } else {
   // Passwords don't match
  }
});
```

Synchronous Version

If you prefer a synchronous approach, you can do this instead:

```
let hash = bcrypt.hashSync('myPassword', 10);
// Store hash in database
```

To verify the password later on:

```
if(bcrypt.compareSync('somePassword', hash)) {
 // Passwords match
} else {
 // Passwords don't match
}
```

7.5 Implementing JWT based authentication in Node.js

Authentication allows your application to know that the person who is sending a request to your application is actually who they say they are. The JSON web token (JWT) is one method for allowing authentication, without actually storing any information about the user on the system itself.

In this lecture, we will demonstrate how JWT based authentication works, and how to build a sample application in Node.js to implement it.

The JWT format

Let’s say we have a user called `user1`, and they try to log into an application or website. Once successful they would receive a token that looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InVzZXIxIiwiZXhwIjoxNTQ3OTc0MDgyfQ.2Ye5_w1z3zpD4dSGdRp3s98ZipCNQqmsHRB9vioOx54
```

This is a JWT, and consists of three parts (separated by `.`):

1. The first part is the header (`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`). The header specifies information like the algorithm used to generate the signature (the third part). This part is pretty standard and is the same for any JWT using the same algorithm.
2. The second part is the payload (`eyJ1c2VybmFtZSI6InVzZXIxIiwiZXhwIjoxNTQ3OTc0MDgyfQ`), which contains application specific information (in our case, this is the username), along with information about the expiry and validity of the token.
3. The third part is the signature (`2Ye5_w1z3zpD4dSGdRp3s98ZipCNQqmsHRB9vioOx54`). It is generated by combining and hashing the first two parts along with a secret key.

Now the interesting thing is that the header and payload are not encrypted. They are just base64 encoded. This means that anyone can view their contents by decoding them.

Which will show its contents as:

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

Similarly, the contents of the payload are:

```
{"username":"user1","exp":1547974082}
```

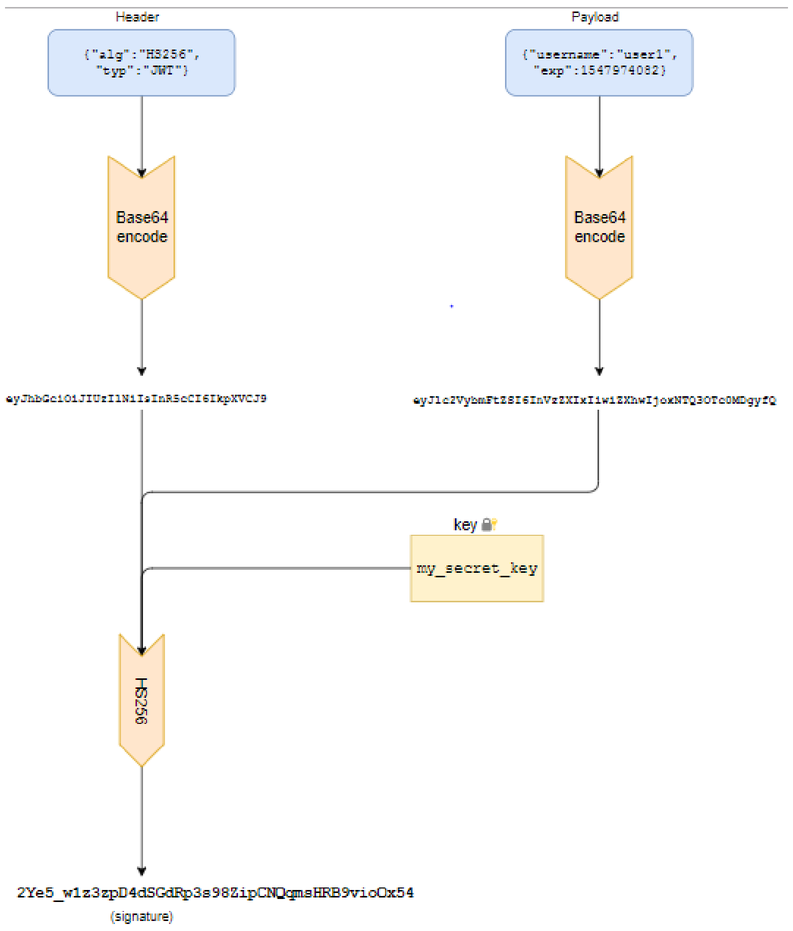
How the JWT signature works

So if the header and signature of a JWT can be read and written to by anyone,
what *actually* makes a JWT secure? The answer lies in how the last part (the signature) is generated.

Let’s pretend you’re an application that wants to issue a JWT to a user (for example, `user1`) that has successfully signed in.

Making the header and payload are pretty straightforward: The header is more or less fixed, and the payload JSON object is formed by setting the user ID and the expiry time in unix milliseconds.

The application issuing the token will also have a key, which is a secret value, and known only to the application itself. The base64 representations of the header and payload are then combined with the secret key and then passed through a hashing algorithm (in this case its `HS256`, as mentioned in the header)

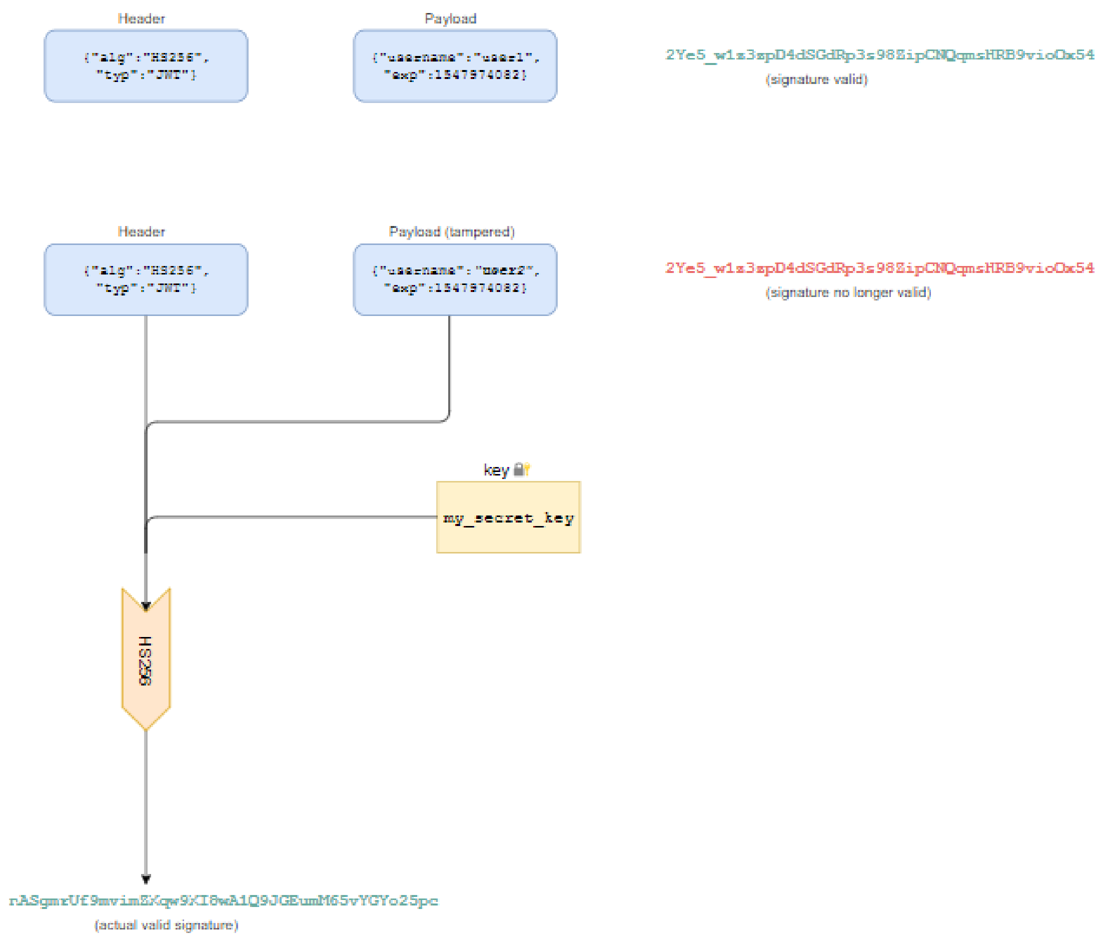


The details of how the algorithm is implemented is out of scope for this lecture, but the important thing to note is that it is one way, which means that we cannot reverse the algorithm and obtain the components that went into making the signature… so our secret key remains secret.

Verifying a JWT

In order to verify an incoming JWT, a signature is once again generated using the header and payload from the incoming JWT, and the secret key. If the signature matches the one on the JWT, then the JWT is considered valid.

Now let’s pretend that you’re a hacker trying to issue a fake token. You can easily generate the header and payload, but without knowing the key, there is no way to generate a valid signature. If you try to tamper with the existing payload of a valid JWT, the signatures will no longer match.



In this way, the JWT acts as a way to authorize users in a secure manner, without actually storing any information (besides the key) on the application server.

Implementation in Node.js

Now that we've seen how JWT based authentication works, let's implement it using Node.

Creating the HTTP server

Let's start by initializing the HTTP server with the required routes in the index.js file. We've used express as the server framework:

```
const express = require('express')

const bodyParser = require('body-parser')
const cookieParser = require('cookie-parser')


const { signIn, welcome, refresh } = require('./handlers')


const app = express()
app.use(bodyParser.json())
app.use(cookieParser())


app.post('/signin', signIn)
app.get('/welcome', welcome)
app.post('/refresh', refresh)


app.listen(8000)
```

We can now define the signIn and welcome routes.

Handling user sign in

The /signin route will take the users credentials and log them in. For simplification, we're storing the users information as an in-memory map in our code:

```
const users = {
  user1: 'password1',
  user2: 'password2'
}
```

So for now, there are only two valid users in our application: user1, and user2. Next, we can write the signIn HTTP handler in a new file handlers.js. For this example we are using the jsonwebtoken library to help us create and verify JWT tokens.

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


const jwtKey = 'my_secret_key'

const jwtExpirySeconds = 300


const users = {

  user1: 'password1',

  user2: 'password2'

}


const signIn = (req, res) => {

  // Get credentials from JSON body

  const { username, password } = req.body

  if (!username || !password || users[username] !== password) {

    // return 401 error is username or password doesn't exist, or if password does

    // not match the password in our records

    return res.status(401).end()

  }


  // Create a new token with the username in the payload

  // and which expires 300 seconds after issue
  const token = jwt.sign({ username }, jwtKey, {
    algorithm: 'HS256',
    expiresIn: jwtExpirySeconds
  })
  console.log('token:', token)

  // set the cookie as the token string, with a similar max age as the token
  // here, the max age is in milliseconds, so we multiply by 1000
  res.cookie('token', token, { maxAge: jwtExpirySeconds * 1000 })
  res.end()
}
```

If a user logs in with the correct credentials, this handler will then set a cookie on the client side with the JWT value. Once a cookie is set on a client, it is sent along with every request henceforth. Now we can write our welcome handler to handle user specific information.

Handling post authentication routes

Now that all logged in clients have session information stored on their end as cookies, we can use it to:

- Authenticate subsequent user requests
- Get information about the user making the request

Let’s write our welcome handler in handlers.js to do just that:

```
const welcome = (req, res) => {
  // We can obtain the session token from the requests cookies, which come with every request
  const token = req.cookies.token
```

```
// if the cookie is not set, return an unauthorized error
if (!token) {
  return res.status(401).end()
}

var payload
try {
  // Parse the JWT string and store the result in `payload`.
  // Note that we are passing the key in this method as well. This method will throw an error
  // if the token is invalid (if it has expired according to the expiry time we set on sign in),
  // or if the signature does not match
  payload = jwt.verify(token, jwtKey)
} catch (e) {
  if (e instanceof jwt.JsonWebTokenError) {
    // if the error thrown is because the JWT is unauthorized, return a 401 error
    return res.status(401).end()
  }
  // otherwise, return a bad request error
  return res.status(400).end()
}

// Finally, return the welcome message to the user, along with their
// username given in the token
```

```
  res.send(`Welcome ${payload.username}!`)
}
```

Renewing your token

In this example, we have set a short expiry time of five minutes. We should not expect the user to login every five minutes if their token expires. To solve this, we will create another /refresh route that takes the previous token (which is still valid), and returns a new token with a renewed expiry time.

To minimize misuse of a JWT, the expiry time is usually kept in the order of a few minutes. Typically the client application would refresh the token in the background.

```
const refresh = (req, res) => {
  // (BEGIN) The code uptil this point is the same as the first part of the `welcome` route
  const token = req.cookies.token


  if (!token) {
    return res.status(401).end()
  }


  var payload
  try {
    payload = jwt.verify(token, jwtKey)
  } catch (e) {
    if (e instanceof jwt.JsonWebTokenError) {
      return res.status(401).end()
    }
    return res.status(400).end()
  }
  // (END) The code uptil this point is the same as the first part of the `welcome` route

  // We ensure that a new token is not issued until enough time has elapsed
  // In this case, a new token will only be issued if the old token is within
  // 30 seconds of expiry. Otherwise, return a bad request status
  const nowUnixSeconds = Math.round(Number(new Date()) / 1000)
  if (payload.exp - nowUnixSeconds > 30) {
    return res.status(400).end()
  }

  // Now, create a new token for the current user, with a renewed expiration time
  const newToken = jwt.sign({ username: payload.username }, jwtKey, {
    algorithm: 'HS256',
    expiresIn: jwtExpirySeconds
  })

  // Set the new token as the users `token` cookie
  res.cookie('token', newToken, { maxAge: jwtExpirySeconds * 1000 })
  res.end()
}
```

We’ll need to export the handlers at the end of the file:

```
module.exports = {
  signIn,

  welcome,

  refresh

}
```