NodeJs

Class	Intro to NodeJS
Created	@Mar 2, 2021 3:04 PM

1. Base commands on terminal

- Running scripts

```
>>>node fileName.js
```

- Running a fie

2. Node module system

- The Node module system is what allows us to import or export features between scripts
- require()

```
//require imports a module
const fs = require('fs') //import file system core module

const authModule = require('./auth.js'); //import a module from own file
```

- module.exports()

```
//module.exports() specifies what is exported
const varA = 'a';
function fx() {

//single export
module.exports = varA;

//attaching as property
module.exports.varA = varA; //say this was in script.js
//in other file:
const moduleImport = require('./script.js');
moduleImport.varA; //'a'

//exporting an object
module.exports = {
  varA: varA,
  fx: fx,
}
//in other file:
```

```
const moduleImport = require('./script.js');
moduleImport.varA; //'a'
```

- using ESM
    1. In package.json, add "type": "module" to use the import syntax
- Loading third party modules with npm
    - Initializing npm
        1. in parent folder, not in specific js file, run npm init
        2. This creates a configuration file, package.json, to manage dependencies, similar to pubspec.yaml in Dart
    - Installing local packages
        1. What are local packages? Local packages are packages that are installed in your machine locally; They appear in your package.json configurations;
        2. run npm install packageName
        3. This creates a node_modules folder(for first package installed) where all installed modules are in
            - github uploads or project downloads may exclude node_modules, in which case to install the dependencies according to what is specified in package.json,
            - run npm install
            - npm install checks the package.json files for dependency information and installs all of it to create a new node_modules folder or update it
        4. package-lock.json will also be created
    - Installing modules globally
        1. Installing modules globally is to install them on our entire "runtime" - The module will be available in all our NodeJs applications
        2. run npn install packageName -g

3. File system and CL Args

- CL args
    - CL args are what you pass into the command line as additional information

```
>>> node app.js extra1 extra2
```

- CL args are available through the process object(which is the equivalent of document object; window === global)

```
console.log(process.argv); //Returns an array of args provided
//[ 'path to nodeJs executable', 'path to app.js file', 'extra1', 'extra2',...]
//first two elements will always be present,
//while the third(and subsequent elements are optional

console.log(process.argv[2]); //extra1
```

- Using Yargs, a package to manage process.argv, would simplify things a lot

1. Without yargs

```
//in CL
>>> node app.js add --title="myTitle"

//in app.js
const command = process.argv[2]; //'add'
const title = process.argv[3];//'--title="myTitle'

//How to we process the string easily?
```

2. With yargs

```
//in CLI
>>>node app.js
//in app.js
console.log(yargs.arv);
//{ _: [],'$0': 'app.js' }

>>>node app.js add --title="myTitle" extra1 extra2
console.log(yargs.argv);
//{_: ['add','extra1','extra2'], title: 'myTitle', '$0': 'app.js'}
```

   - yargs.argv returns an object, with a minimum of 2 properties
   - _ returns a list of 'unformatted' arguments
   - $0 returns the fileName
   - Any other property of the object was from a formatted argument

3. Creating commands with yargs

```
//create add command
yargs.command({
  command: 'add',
  describe: 'add a new note',
  handler: () =>{
    console.log('added note')
  }}).argv
//When add is passed as a parameter, it is recognized as a command
```

```
//Handler is then executed

//creating multiple commands
yargs.command({
  command: 'add',
  describe: 'add a new note',
  handler: () =>{
    console.log('added note')
}).command({
  command: 'read',
  describe: 'reads an existing note',
  handler: () => {
    console.log('read note')
  },
}).argv
```

4. To receive additional information(options) in commands created,

```
//Suppose adding a note requires a --title="" and --body=""
yargs.command({
  command: 'add',
  describe: 'adds a new note',
  builder: {
    title: {
      //all objects in yarn can have a describe property
      describe: "Note title",
    }
  },
  //handler can receive argv, which will contain the options as a property,
  //recall {_: ['add','extra1','extra2'], title: 'myTitle', '$0': 'app.js'}
  handler: (argv) =>{
    console.log('added note',argv)
  },
})

//to make title required, set demandOptions: true
//to make the value of title to be a string, set type:'string'
yargs.command({
  command: 'add',
  describe: 'adds a new note',
  builder: {
    title: {
      describe: "Note title",
      demandOption: true, //*****required*****
      type: 'string', //*****string*****
    }
  },
  handler: (argv) =>{
    console.log('added note',argv);
  },
})
```

5.

-
-

4. JSON

- JSON.stringify()

```
const book = {
  title: 'Ego is the enenmy',
  author: 'Ryan Holiday',
}
//Filesystem in node only accepts strings
//To store objects, we need a way to convert objects into string representation
const bookJsonString = JSON.stringify(book);
console.log(bookJsonString);
//{"title":"Ego is the enenmy","author":"Ryan Holiday"}
```

  - note: single quotes are converted to double
  - propertyName is now in double quotes
  - bookJSON is of <string> type, bookJSON.author is undefined

- JSON.parse()

```
//JSON.parse() takes in string and converts it into an object
const parsedData = JSON.parse(bookJsonString);
console.log(parsedData); //{ title: 'Ego is the enenmy', author: 'Ryan Holiday' }
console.log(parsedData.title); //Ego is the enenmy
```

5. Debugging in Node

- The debugging tools in node work with chrome's v8 engine

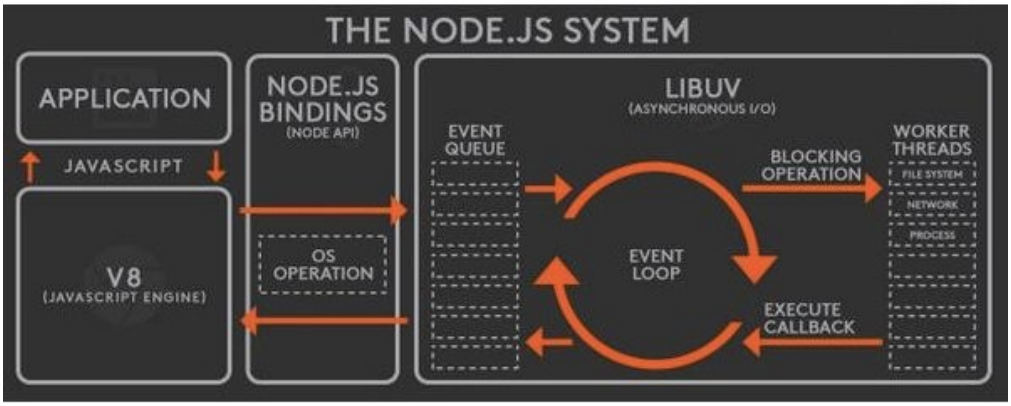
```
const addNote = (title, body) => {
  const data = loadNotes();
  debugger; //*****Adds Node's debugging
  const duplicateNotes = data.filter((note)=>note.title === title);
  if (duplicateNotes.length >0) {
    console.log(chalk.red.inverse('title taken'));
    return;
  }
  data.push({
    title: title,
    body: body,
  });
  updateNotes(data);
  console.log(chalk.green.inverse(`${title} added!`));
};
```

  - Node debugger do anything by default, until we run node inspect fileName.js
  - In chrome, we can go to chrome://inspect

- Click on inspect on one of the targets
- Under filesystem tab, click the + icon to add your project file
- Open app.js
- Click the blue icon on the top right to run script
- The debugger keyword will pause execution automatically, and we will have access to all values available at that lexical location
- Going back to the terminal, press ctrl+c twice to exit from node debugger

6. NodeJS runtime

- Diagram



7. encodeURIComponent

- Encodes a text string as a valid component of a Uniform Resource Identifier

8. Callback patterns

- Callbacks should always accept either an error or a response

```
getLatLng('Boston', (error, response) => {
  if (error) {
    console.log("Error: ", error);
    return;
  }
  getWeatherData(response,  (error, response) => {
    if (error) {
      console.log("Error: ", error);
```

```
      return;
    }
    console.log(response);
  })
});
```

- Callbacks should be received as arguments
  - Here the anonymous function that contains if clause and getWeatherData is the callback being passed as argument
- Callback chaining: Passing the output of one callback as input to the other
  - Here getLatLng's returned response is passed into getWeatherData
-

-

9. ES6

- Object property shorthand

```
const name= 'Alex'
const age = 24;
const user = {
  name, //shorthand for name: name,
  age,
  location: 'singapore',
}
```

- Object destructuring

```
const user1 = {
  name: 'Alex'
  age: 26,
  location: 'singapore',
}

const {name,location: currentLocation,spouse = false, rating,} = user1;
//shorthand for
//user1 = user.name
//currentLocation = user1.location
//spouse = user.spouse ? user1.spouse : false;
//rating = user1.rating
console.log(name); //'Alex'
console.log(currentLocation); //'singapore'
console.log(rating); //undefined

const fx = (request, {name,age,location}) => {
}
fx('register', user1) //automatically destructures
```

- Note that destructuring something that is undefined will throw an error, therefore when destructuring arguments, use this instead

```
const fx = (request, {name,age,location} = {}) => {
}
```

  - This uses the default parameter feature to give a default value of an empty object IF no value was passed in
  - This works because destructing off an empty object would simply return undefined for each variable and no error is thrown
-
-

10. HTTP requests in Node

- Without library

```
const http = require('http');
const url = 'someUrl';
const request = http.request(url, response=> {
  let data = '';
  response.on('data', (chunk) =>{
    data+=chunk.toString();
  });
  response.on('end',()=>{
    const result = json.parse(data);
    console.log(result);
  });
  return result;
});
//error handling
request.on((error)=> {});
request.end(); //runs json.parse and console.log part and signifies end of request
```

  - Node's own http package is very low level and it is easer to use libraries
- With request libray

```
const request = require("postman-request");
const url = 'someUrl';
//request two arguments, first is an object with information such as url,
//whether to parse response to json automatically, etc
//second is a callback
request({ url, json: true }, (error, response) => {
    //error handling
    if (error) {
      console.log("Unable to connect to location services", "");
      return;
    }
    if (response.body.features.length === 0) {
```

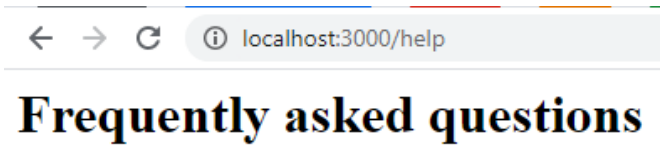
```
      console.log("Bad search terms, try another");
      return;
    }
    return response; //or do something with it immediately
  });
```

11. ExpressJS

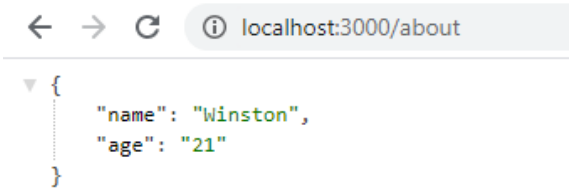
- Creating an express server

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

//creates express process
const app = express();

//.get executes a callback when a user reaches a certain endpoint
//.get('/app', ()=>{}) triggers when ____.com/app is reached
app.get('', (req,res) => {
  res.send('Home');
})

app.get('/help', (req,res) => {
  //automatically processes string as HTML and renders
  res.send('<h1> Frequently asked questions </h1>');
})

app.get('/about', (req,res) => {
  //processes objects as json, and parses it into a string before sending to request
  res.send({
    name: 'Winston',
    age: '21',
  });
})

//.listen can only be called once, and it starts the express process to continously listen for connections
app.listen(3000, ()=> {
  console.log('Express has started on port 3000')
});
```

- Rendered pages
  - /help



- /about



- Only objects are seen as JSON, sending an array would result in no change to the array

```
app.get('/about', (req,res) => {
  res.send(
    [{
    name: 'Winston',
    age: '21',
      },
    {
      name: 'Ray',
      age: 20,
    }]
  );
})
```

- Takeaways:
- Express creates a process that listens to connections
- Depending on which endpoint(s) are reached, it sends some data(local/server)
- Instead of string data, we can send entire html/javascript files
- We can create multiple express processes which would run multiple 'serversSending static assets

- index.html is special - it is immediately sent
- Express provides two variables, __dirname and __fileName, that return the directory of the current folder and current file
- Using the path core module, we create a path to our public folder where we store files
  - the path module introduces methods like path.join() etc
  - path.join

    ```
    path.join(D:\Development\NodeJs\web-server\src, '../public/index.html');
    //D:\Development\NodeJs\web-server\public\index.html
    ```

  -
- Calling app.use() after creating express

  ```
  app.use(express.static(D:/Development/NodeJs/web-server/public));
  //serves static content from specified path
  ```

  - What app.use() does: Mounts the specified middleware function(s) at the specified path
  - Without a path, app.use mounts to every request to the app
  - Middleware functions refer to functions that have access/receive request and response objects and the next middleware function in the request-response cycle
  - If the middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function
  - In this case, express.static() is executed on every request, and will allow you to load in static files that are in the public folder
- CSS files
  - Create a css folder in public
  - In the html file, add to <head>
  - <link rel= 'stylesheet' href="/css/styles.css">
  - note href attributeValue no longer should have a relative path, but an absolute path

    ```
    //relative path
    <link rel= 'stylesheet' href="./css/styles.css">

    //absolute path
    <link rel= 'stylesheet' href="/css/styles.css">
    ```

  -

-
- Serving dynamic pages with Templating using HandleBars
  1. Allows rendering of dynamic documents
  2. Allows easy creation of reusable code
  3. To set up:

```
const app = express();
app.set('view engine', 'hbs'); //arg1: setting, arg2: value
```

     - Here we configure our engine to use hbs, such that .render() uses the hbs engine
     - .render() renders a view using specified view engine
  4. Place views(hbs templates) into a view folder in the root of the project
  5. Set up an endpoint to that view using app.get()

```
app.get('',(req,res)=> {
  res.render('index', {
    option1: val1,
    option2: val2,
    //...
  })
)
```

     - There is no need for a path, since Express expects the views in root/views
     - There is also no need to specify the type/extension of file
     - The second argument is options - an object that provides optional values that will be accessible in the .hbs file
  6. To customize the Views directory, you need to declare it somehow

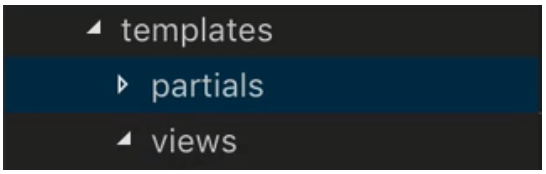
```
const viewsPath = path.join(__dirname, '../newPath')
//recall dirname is current folder

//now we need to declare to Express where our views folder is
app.set('views', viewsPath)
```

  7. To access the options to display dynamic content

```
//in app.js
app.get('', (req,res)=> {
  res.render('index', {
    title: 'Weather App',
    author: 'Winston'
  });
})
```

```
//in index.hbs
<h1>{{title}}</h1> //Title = Weather App
```

- Partials with Handlebar
  1. Partials = a template(that is therefore reusable) that is part of a larger page - such as headers, footers, navigation bar etc
  2. Partials require importing from the hbs module(unlike all the previously mentioned features which were built in)
  3. The convention is to split your partials from views - we do this by creating a views AND partials folder within a template



  4. The setup for partials is slightly different from views - it is not built in but uses the hbs import

```
const path = require('path');
const express = require('express');
const hbs = require('hbs');
const app = express();

//define paths for config
const publicDirectoryPath = path.join(__dirname, '../public/');
const viewsPath = path.join(__dirname, '../templates/views');
const partialsPath = path.join(__dirname, '../templates/partials');

//Set up configuration for hbs engine and views path
app.set('view engine', 'hbs');
app.set('views',viewsPath);
hbs.registerPartials(partialsPath);
```

     - Note: partials are loaded when hbs.registerPartials are called - if scripts are not defined yet, an error will be thrown
     - If using Nodemon, configure to restart when partials are modified

```
>>> nodemon src/app.js -e js,hbs
```

  5. Adding partials to html file

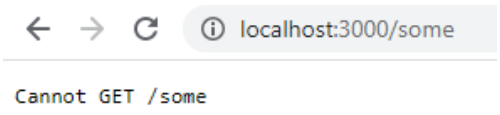
```
<body>
  {{>header}}
  <h1>Frequently asked questions</h1>
  <h3>{{helpMessage}}</h3>
</body>
```

6. To use options in partials, use same syntax(no imports etc)

```
<h1>{{title}}</h1>
```

7.

- Error Handling
  - Creating a 404 page
    1. When a user tries to reach an endpoint that doesnt exist, Express shows



    2. To prevent that, we add a "catch" at the bottom of all .get() calls that will run and render some content

```
app.get('*', (req,res)=>{
  res.send('404 PAGE NOT FOUND');
})
//* means everything is a match
```

       - How express works is when there is a request to an endpoint, it goes from top to below to search for matches
       - * is the last route which means the route the request is looking for does not exist\
  - Queries in Express
    - Recall that .get()'s callback accepts req/res
    - So far we know res is the response our server provides to the request - to send or render
    - The other argument, req is how we access requests made to the server

```
app.get('/products', (req,res)=> {
  console.log(req.query)
  console.log(req.query.search);
  res.send({
```

```
    products: [],
  })
})

//http://localhost:3000/products?search=games&rating=5
//{search: 'games, rating: '5'}
//games
```

- queries start after the "?" and are separated by "&"
- queries can be accessed on the server by req.query
- individual queries can be acessed by req.query.queryName

- To make a query compulsory, we need to write conditional code

```
app.get('/products', (req,res)=> {
  if (!req.query.search) {
    return res.send({
      error: 'You must provide a search term',
    });
  }
  console.log(req.query)
  console.log(req.query.search);
  res.send({
    products: [],
  })
})
```

-

- Other commands
  - On response object
    1. response.status(statusCode);
       - Changes the response status to code passed in
       - All other commands should be executed before response.send()
       - It is possible to chain commands on response object

         ```
         catch(error=>response.status(400).send(error));
         ```

       -
    2.
  -
- Route parameters
  - Express provides route parameters so that endpoints can be reached with parameters

- The most common use case for this is fetching data

```
app.get("/users/:iid", (request,response)=> {
  console.log(req.params) //returns all the route parameters in object
  // { id: '19024u10foas123gkn" }
});
```

-
-

12. Deploying to heroku

- Ensure heroku cli is installed
  - If doing fresh install, restart vscode after heroku cli is installed
- Set up SSH keys with heroku

```
$ heroku keys:add
```

- Similar to git, in root folder, run heroku create

```
$ heroku create projectName
```

  - This creates a server on heroku and returns the url to that server(along with a link to where your git repository is)
- Provide instructions of what to do with our code in package.json
  1. Specify which file to run in package.json.scripts

```
{
  "name": "web-server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    //"test": "echo \"Error: no test specified\" && exit 1"
    "start ":"node src/serverApp.js" //*****
  },
  //....
  }
}
```

  - This basically is setting up the command npm run start to run the specified files (npm run accepts script to run, in this case start is the script)
  - Can be tested in your terminal simply run npm run start

2. Change the port we are listening to

```
const app = express();
const port = process.env.PORT || 3000; //*****
//3000 is fallback port in case we start locally

//...

//At end of file,
app.listen(port, ()=> {
  console.log(`Node has started at ${port}`);
});
```

3. aa

- Configure devDependencies
  1. There are some dependencies such as nodemon that is only required during development and not production
  2. In order to prevent redundant installs on production servers, one has to make sure that the dependencies for development are listed in devDependencies in package.json

```
{
  //....
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js -e js,hbs,css"
  },
  //....
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

  3.

- Push to heroku remote

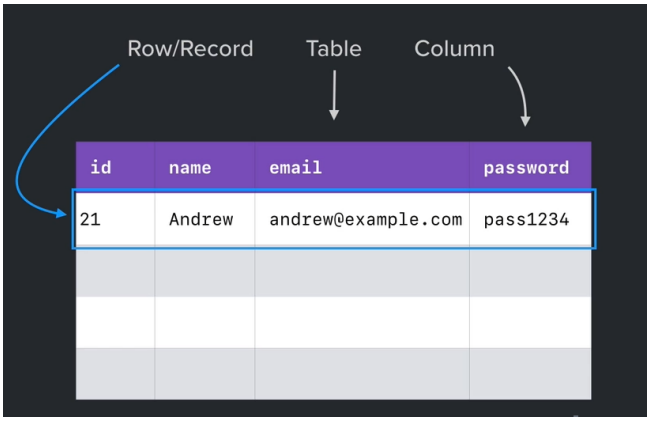
```
$ git remote //shows all configured remotes
>>>heroku
>>>origin
$ git push heroku master
```

-

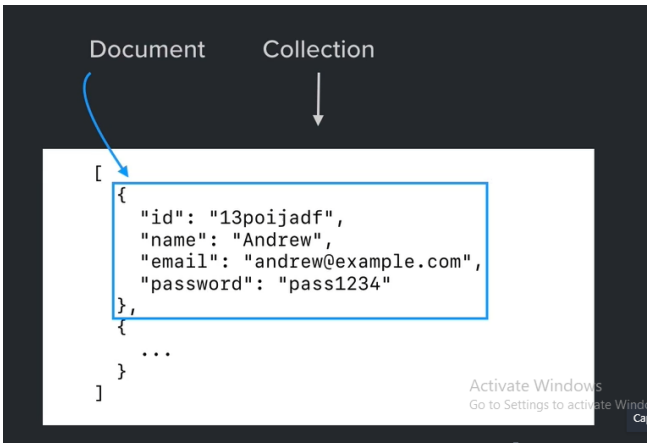
13. MongDB

- MongoDB is a NoSQL database
  1. SQL stands for structured query language

- Data is stored in tables - rows=record and columns



2. NoSQL stands for Not Only structured query language
    - Data is stored in collections(f) and documents and fields- similar to firestore



    - A collection is a list of entries
    - A document contains fields: Map<FieldName, FieldValue>

- Setup
    - Install community server zip
    - Extract in desired location and rename folder to MongoDB
    - Create a folder of any desired directory and name to store data
    - Run /d/..../mongod.exe —dbpath path/to/datafolder
    - Server should now be up and listening
    - Install database GUI viewer: robo 3t

    - Create a mongodb connection with preferred configurations
- Test CRUD operations
    - Create(insert) documents(.insert,.insertMany)

```
const mongodb = require('mongodb');
//access to functions for CRUD operations
const MongoClient  = mongodb.MongoClient;

const connectionUrl = 'mongodb://127.0.0.1:27017';
const databaseName = 'task-node-app';
```

        - Connect to the client and attempt to create a collection and document

```
MongoClient.connect(connectionUrl, {useUnifiedTopology: true,},(error, client)=> {
  if(error) {
    return console.log('Unable to connect to database');
  }
  //creates or accesses database
  const db = client.db(databaseName);
  //creates or accesses collection and inserts a document
  db.collection('users').insertOne({
    name: 'Winston',
    //unlike firestore, insert expects an object not a Map
    age: 27,
  },(error, result)=> {
      if (error) {
        return console.log('Unable to add user');
      }
      //result.ops returns an array of documents that were inserted
      console.log(result.ops);
    }
  );
 })

//[ { name: 'Winston', age: 27, _id: 6045b504a7dce62d48d9ed53 } ]
```

            - Note: to insert multiple documents, use insertMany() which accepts an array of objects instead of a single object
            - Note2: insert and insertMany returns a promise if no callback function was passed in
            - In Update operations, we show how to work with a promi
    - Read(.findOne,.find)

```
const {MongoClient, ObjectId} = require('mongodb');

const connectionUrl = 'mongodb://127.0.0.1:27017';
const databaseName = 'task-node-app';


MongoClient.connect(connectionUrl, {useUnifiedTopology: true,},(error, client)=> {
```

```
  if(error) {
    return console.log('Unable to connect to database');
  }
  //creates or accesses database
  const db = client.db(databaseName);
  db.collection('users').findOne(
    {
      name: 'jen',
      _id: new ObjectId("6045b727bbba7e47e4c206e2")
    }, (error, result)=> {
    if (error) {
      return console.log('Unable to fetch');
    }
    console.log('Fetched data');
    console.log(result);
  })
  //Object

  db.collection('users').find({age: 27,}).toArray((error, result)=> {
    console.log(result);
  });
  //3 Array<Objects>
  db.collection('users').find({age: 27,}).count((error, count)=> {
    console.log(count);
  });
  //3
})
```

- .findOne
    1. .findOne returns a document if it exists or null if it does not
    2. The first argument is an object of query fields
    3. for .findOne, if multiple documents satisfy the query, the first one found is returned
    4. finding documents based on ID is possible, but must be converted to hexadecimal from its string representation as seen above
- .find
    1. .find returns a Cursor: a cursor is a pointer to the data in the database
    2. .find does not accept a callback function
    3. Instead, we can use methods on the returned cursor, which opens up more use case possbilities
    4. To get an array of results, use .toArray() which receives a callback function
    5. .count() returns the number of results that satisfies the condition
    6. The advantage of Cursors are therefore in being able to specify the kind of query, and in so, we reduce load times and efficiency
- Update

```
MongoClient.connect(connectionUrl, {useUnifiedTopology: true,},(error, client)=> {
  if(error) {
    return console.log('Unable to connect to database');
  }
  const db = client.db(databaseName);
  db.collection('tasks').updateOne(
    //filterQuery object is first argument
    {
    completed: false,
    },
    //update object is second argument
    {
      //keys are update operators
      //values are objects containing the fields to update
      $set: {
        completed: true,
      }
    }).then(console.log).catch(console.log);
 })
```

    - updateMany works the exact same way as updateOne except it applies to many
  - Delete
- Collections and documents
  - Notice that documents have auto generated IDs that are called ObjectId
  - Viewing a document in robo 3t

  - In SQL databases, the IDs follow a simple incrementing pattern(1,2,3,4)\
  - In noSQL databases, the IDs are known as GUI - globally unique identifiers; this removes the requirement of having to determine what the next ID will be, allowing scaling in a distributed system more efficient
    1. Having multiple databases is not an issue as documents have globally unique ids
    2. Another advantage is being able to generate the ID of a document before inserting into a database using ObjectID from mongodb's package
    3. ObjectID is a 12 byte value, where the first 4bytes represent the second after unix epoch(time of creation), the next 5 bytes represent a random value, and the last 3 bytes represent a counter(starts with a random value)
    4. The way ObjectID was designed means having a complex GUI without needing a central authority that determines the next
    5. In the above photo, we see _id is stored as ObjectId('hexadecimalString')

6. ObjectId() takes in a hexadecimal string and converts it to hexadecimal for it to be stored on the computer, reason being: Hexadecimals(12byte) take up half the memory of hexadecimal strings(24byte)
7. To manually the _id property value, you must create a new object with

```
const {MongoClient, ObjectID} = require('mongodb');

const id = new ObjectID();

//later on, when creating the object to pass into  .insert,
_id: id,
//_id only accepts <ObjectID>
```

- MongoDB and Promises
  - For HTTP requests or any CRUD operations on MongoDB, such as in Mongoose where we call .save()or find(), if the operation was completed even if the query failed to return anything or threw an error, the Promise is considered fulfilled
- Hosting MongoDB
  - Previously we tested our MongoDB operations with Mongoose on a locally created MongoDB using mongod.exe
  - Many services can host mongoDB for you to access, but we will try Atlas
    - Register and create a free cluster
    - Connect to the cluster which will prompt a setup
    - First, allow all IPs to connect as we will be deploying using Heroku
    - Next, create a database user
    - Set connection method to MongoDB Compass
    - Download and install Compass, copying the string URI value from the created DB user when creating a connection
- After hosting mongodb, we deploy our application on heroku
  - Ensure that git is already initialized/local repository exists
  - heroku create task-node-app
  - Set up heroku environment variables for deployed server

```
heroku config
//shows all env var
heroku config:set key=value key2=value2
//adds to env var
heroku config
//key=value
//key2=value2
```

```
heroku config:unset key=value key2=value2
heroku config
//unset removes env var
```

- After that, push to heroku's git

```
git push heroku main
```

- This will push code from main branch to heroku's own created git repository that it references from

- Query and projection operators
  - https://docs.mongodb.com/manual/reference/operator/query/
- Indexes
  ▼ Context:
    - consider that you have a deck of 52 cards, four suits - Ace through King. If I shuffle the deck into random order and ask you to pick out the 8 of hearts, you would have to individually flip through each card until you find it. This means that on average you would have to go through half the deck, which is 26 cards!
    - Now, consider that we divided the cards into four piles by suit, each pile shuffled randomly. Now if I asked you to pick out the 8 of hearts, you would select the pile with the hearts suit, which would take on average two to find, and then flip through the 13 cards. Approximately, it would take seven flips to find, thus nine totals. This is seventeen flips (26-9) faster than just scanning the whole deck. Right?
    - Let's take it one step further and split the individual piles into two groups (one Ace to 6, the other 7 to King). In this case the average search would cut to 6.
    - This is exactly the power of indexing.
  - What is an index?
    - Indexes are special data structures that store a small portion a collection's data in an easy to traverse form
    - An index is therefore technically just a copy of selected data from a collection
    - The creation of indexes can be considered to be a trade of between space and time - creating indexes adds overhead in the form of additional writes and storage space, but to improve lookout speeds
  - How do indexes work?
    - Indexes target the documents individual field(although it can be further configured to instead store a set of fields)

- When an index is used to fetch documents, it will traverse the index until it finds the matching field, which points to a document in that collection
- By defualt, the default index is created on the document._id field i.e. the index stores _id values which points to the actual document - querying by _id therefore traverses this index, and very quickly can fetch the actual document

- Order of index data
  - MongoDB indexes store the fields/set of fields in order of their values, which supports efficient equality or range-based query operations

  - The above illustrates the ordering of data in an ascending order, specified by (score:1)- such that a query for example, the score of 20, would be efficient as it can directly search the lower half etc
  - Creating indexes is covered below

- Creating indexes
  - Single Field index
    - Document
      - Document example

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

      - Index creation

```
db.records.createIndex( { score: 1 } )
```

        - The value of the field in the index specification describes the kind of index for that field
        - For example, a value of 1 specifies an index that orders items in ascending order
        - A value of -1 specifies an index that orders items in descending order
    - Embedded document
        - Document example

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

        - Index creation

```
db.records.createIndex( { "location.state": 1 } )
```

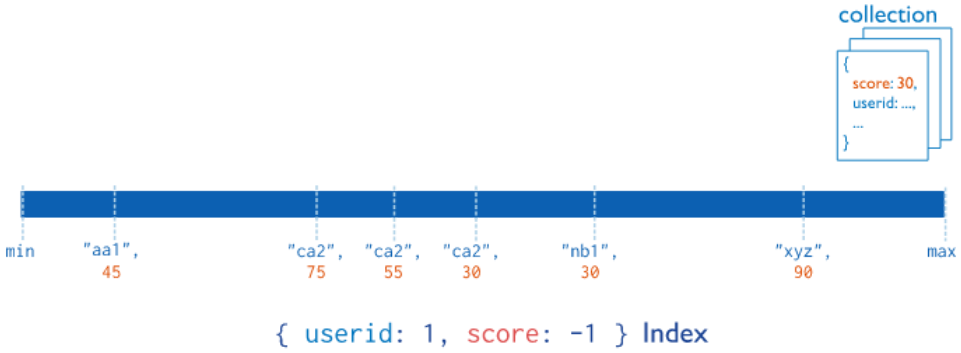
- Compound indexes
    - Creating a compound index
        - Document example

```
{
 "_id": ObjectId(...),
 "item": "Banana",
 "category": ["food", "produce", "grocery"],
 "location": "4th Street Store",
 "stock": 4,
 "type": "cases"
}
```

        - Index creation

```
db.products.createIndex( { "item": 1, "stock": 1 } )
```

        - The order of the fields listed in a compound index is important
        - The index will contain references to documents sorted first by the values of the item field and, within each value of the item field, sorted by values of the stock field
    - Sort order
        - Consider the above created compound index



- { userid: 1, score: -1 } :the index sorts first by userid and then, within each userid value, sorts by score
- In other words, only if the userid of documents are equal, do they consider sorting by score, in descending order

- Returning sorted results
  - Another reason why a index should be sorted, is in its ability to return sorted data

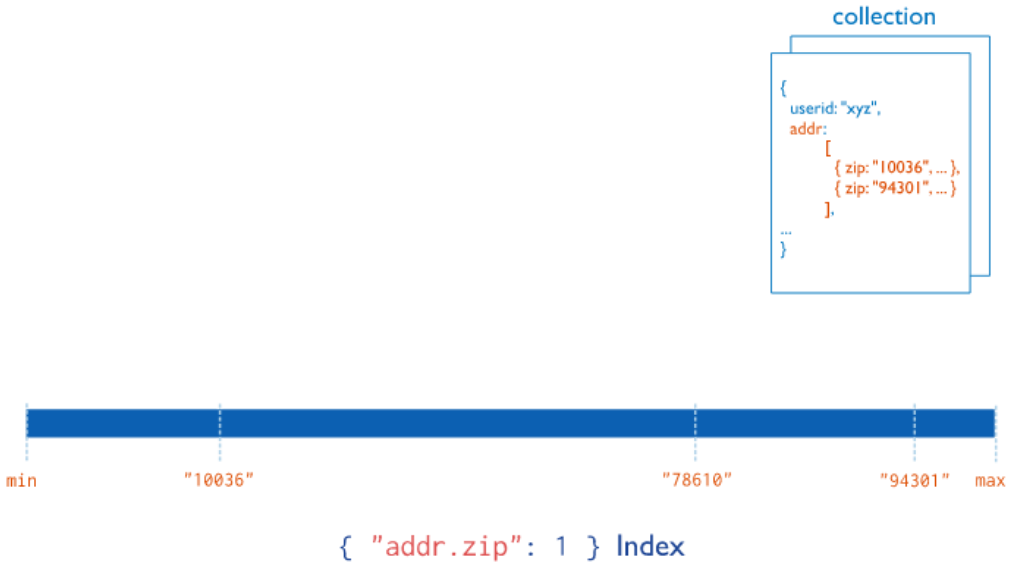
```
db.events.find().sort( { username: 1, date: -1 } )
//db.events.find().sort( { username: -1, date: 1 } )
```

    - This means to sort() the results in the following order: first by ascending usernamee then by date, similar to the way an index sorts its own data
    - The second sort also works, just in reverse order
    - However the sort below does not work/is not supported

```
db.events.find().sort( { username: 1, date: 1 } )
```

      - Intuitively, after sorting by ascending username, the dates are sorted in descending order due to the way the index was sorted

- Multikey index
  - When an index is created based on a field that contains an array, MongoDB automatically creates a multikey index - separate index entries for each element in that array



    - These multikey indexes allow queries to select documents based on the matching element
    - If there is a chance that elements of the array can have exacty same values, then make the field unique(the prerequisite for this is: MongoDB cannot create a unique index on the specified index field(s) if the collection already contains data that would violate the unique constraint for the index
    - For ranged results(using.find()), see index bounds: https://docs.mongodb.com/manual/core/multikey-index-bounds/
  - Geospatial index: when you know there will be queries based on location
  - Text index: when you know queries wil be on fields that contain only strings
  -
- Weight of indexes
  - For each indexed field in the document, MongoDB multiplies the number of matches by the weight and sums the results
  - Using this sum, MongoDB then calculates the score for the document
  - It is therefore possible to configure what results should be prioritised in a query by adjusting the weight of that indexed field
  - https://docs.mongodb.com/manual/tutorial/control-results-of-text-search/
-

14. Rest APIs with Mongoose

- Mongoose is an ODM: an object document mapper - allows you to map your objects in nodejs
- Example code

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

//provide the connnectionUrl followed by database name straight away
mongoose.connect('mongodb://127.0.0.1:27017/task-mongoose-app', {
  useUnifiedTopology: true,
  useCreateIndex: true, //ensures when mongoose works with mongodb,indexes are created for quick access
});

//Creating a model
const User = mongoose.model('User', { //creates a model named user
  //creates a name field that we can configure properties on
  //these properties act as validation to screen data
  name: {
    type: String, //has tons of other types, refer to docs
  },
  age: {
    type: Number,
  }
})

//Creating an instance of defined User model
const user1 = new User({
  name: 'Weng',
  age: 27,
})

//Operations on model with database
user1.save().then(console.log).catch(console.log);
//{ _id: 604775eb174d530acc0ee5ef, name: 'Weng', age: 27, __v: 0 }
//__v is created by mongoose, represents the version number
```

  - Mongoose connects to the MongoDB differently, the database name is included in the url we pass in
  - Mongoose allows us to create models(think classes) where validation on fields can be configured
  - In the above, we ensure that the name field must be of String type and if not, then an error will be thrown
  - Interacting with the Database is automatically handled by Mongoose, which allows for the syntax of calling methods on instances of models
  - Mongoose also automatically creates a collection for each model, with the name argument passed in - converts it to lowercase, and adds an 's' for plural form
- Data validation and sanitization
  - Validation is where data conforms to rules while sanitization is where data is altered before calling .save( )

- Built in validation is mentioned in the docs, but for more flexibility it is possible to create custom validation

```
const User = mongoose.model('User', {
  name: {
    type: String,
    required: true,
  },
  age: {
    type: Number,
    validate(value) {
      if (value<0) {
        throw new Error();
      }
    }
  }
})
```

  - If an error is thrown from calling validate, ValidationError is thrown by Mongoose
  - Therefore customizing ones validation is through setting conditions
- It is however easier to use packages that already have implemented validators and sanitization
- One such package is the "npm i validator"

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

const User = mongoose.model('User', {
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    validate(value) {
      return validator.isEmail(value) ? new Error : true;
    }
  }
})
```

-

- Creating a schema

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

const userSchema = mongoose.Schema(
  {
  name: {
    type: String,
```

```
    required: true,
  },
  email: {
    type: String,
    required: true,
    validate(value) {
      return validator.isEmail(value) ? new Error : true;
    }
  }
}
)

const User = mongoose.model('User', userSchema)
```

- Creating a schema allows us to take advantage of additional features of Mongoose such as using middleware, which are specified on the schema itself

- Using middleware
  1. Without middleware functions: new request → run route handler (express maps the request to its corresponding route handler)
  2. With middleware: new request → middleware function execution → route handler
  3. Middleware functions are simply callback functions that have access to response and request objects and are able to specify the sequence of exeuction(what executes before or after it) andwhen(by calling next()
  4. We use middleware functions on servers to perform certain actions before passing it on to the route handler - most commonly, there is hashing passwords, verifying tokens, logging of data
  5. Usually, middlewares are specific to certain routes only, as specified by app.use()

     ```
     app.use(
       (request,response,next)=>
     )
     ```

  6. pre middleware functions are executed before specified operation happens, in a one after another fashion

     ```
     userSchema.pre('save', async function(next) {
       bcrypt.hash(password);
     })
     ```

     - In document middleware functions, this refers to the document. In query middleware, this refers to the query
     - Middleware functions have access to next, a function that we use to execute the next middleware

- Middleware functions should not be arrow functions as the value of this becomes lexical dependent
- It is important to note that there are operations that will overwrite middleware functione executions
- For example, the Query Middleware .findOneAndUpdate and update() will not execut the pre and post save() hooks

7. post middleware functions are executed after specified operation happens
8. Setting up middleware for individual routes

```
const middlewareFn = require('../middleware/someMiddleware');
const router = new express.Router();

router.get('/users', middlewareFn,async (req, res) => {
    try {
        const users = await User.find({})
        res.send(users)
    } catch (e) {
        res.status(500).send()
    }
})
```

- In the above, middlewareFn is executed first, followed by the arrow function(which also is a middleware function)

9.

- Creating relationships between models
  - In a project, we decide that Users can create tasks and that task should have a owner field
  - Mongoose allows us to set a property in the field called ref
  - ref creates a relationship between that field and another model

```
const Task = mongoose.model('Task', {
  description: {
    type: String,
    required: true,
    trim: true,
  },
  completed: {
    type: Boolean,
    default: false,
  },
  owner: {
    type: mongoose.Schema.Types.ObjectId,
    required: true,
    ref: 'User',
  }
})
```

- This does nothing yet on its own; However, when we do want to use that relationship to draw some data,

```
const task = await Task.findById('someID');
await task.populate('owner').execPopulate();
console.log(task.owner); //returns entire user document with associated _id
```

  - .populate().execPopulate() allows us to populate a field with data from a relationship, where it automatically will return the entire user document related to the task
- Another way to setup a relationship between two documents is through the Scheme.virtual property
  1. .virtual() allows you to set up a virtual attribute(as if added onto a scheme), but it is simply a way for mongoose to know the relationship between two documents

```
userSchema.virtual('tasks', {
  ref: 'Task',
  localField: "_id",
  foreignField: "owner",
})
const user = await User.findById('someId');
await user.populate('tasks').execPopulate();
console.log(user.tasks);
//finds all returns all task returned by user with associated _id
```

     - ForeignField is the name of the field at other end of the relationship
     - The localField is the name of the field that the other end uses, which is _id
     - Why do we use virtual?
  2.
-

15. REST API

- REST - API: Representing state transfer - application program interface
  1. Representing: Client works with representations of the data stored in the DB
  2. State transfer: State is transferred from the server to the client, the server is stateless - each request from the client contains everything the client needs for(operations, data, authentication)

3. Requests are made via HTTP requests to the server > Server fulfills the request and rends a JSON response with a status code > Client renders things with the data
4. HTTP requests

Request

```
POST /tasks HTTP/1.1
Accept: application/json
Connection: Keep-Alive
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI...

{"description": "Order new drill bits"}
```

   - Three main pieces to a request
   - Line 1: Request line: contains request/path protocol/version
   - Line2-5: Request headers: key-value pairs that are meta data attached to the request
     - You can have as many request headers as required
   - Last line: Request body: A json object
5. HTTP response

Response

```
HTTP/1.1 201 Created
Date: Sun, 28 Jul 2019 15:37:37 GMT
Server: Express
Content-Type: application/json

{"_id": "5c13ec6400d614654ed7e5b5", "description": "Order new drill bits",
"completed": false}
```

   - Line 1: status line: protoco/version statusCode textRepresentationOfStatusCode
   - Line2-5: response headers: metadata
   - Last Line: response body: json object
6.

- What it does: Allows clients to access and manipulate resources using a set of predefined operations
  1. It is important to define and expose necessary operations clients can do with the api such as (CRUD)

2. Create = POST/collectionName,
3. Read = GET/collectionName(mulitple readings) or GET/collectionName/idOfTask
    - Reading usually involves two operations: single readings vs multiple readings
4. Update = PATCH/collectionName/idOfTask
5. Delete = DELETE/collectionName/idOfTask

- Setting up our REST api
    - Create an express server

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

const app = express();
const port = process.env.PORT || 3000;

//Automatically parses incoming json into objects
app.use(express.json());

app.post('/users',(request,response)=>{
  response.send('Testing');
});

app.listen(port, ()=> {
  console.log('Server is up on port ', port);
})
```

        - Recall app.use() is used for mounting middleware functions to requests, where if no path was provided, then the function is executed on every request
        - app.use(express.json()) is used to convert all request bodies from json into an object auotmaticallyt
    - Define how it handles requests
- CRUD operations
    - Create = post

```
app.post('/users',(request,response)=>{
  const user = new User(request.body);
  user.save()
  .then(()=>response.status(201).send(user))
  .catch(error=>response.status(400).send(error));
});
```

        - Recall that request.body is the JSON object sent to the server
    - Read = get

```
app.get('/users/:id', (request,response)=> {
  const _id = request.params.id;
  User.findById(_id)
  .then(result=> result ? response.send(result): response.status(404).send("No such user"))
  .catch(error=>response.status(500).send());
})
```

- The route passed in is one that has a route parameter, as denoted my the colon ":id"
- Route parameters can be accessed by request.params which returns an object containing key=param name, value=param value passed in
- localhost:3000/users/604847e34be48a1e38c00b9c means request.params.id === 604847e34be48a1e38c00b9c
- Note for fetch operations on MongoDB, if the query search fails to return a result, the operation is still considered to have completed/promise is fulfilled
- For fetch operations by id, Mongoose also automatically converts the hexadecimal string into its hexadecimal representation automatically

- Update

```
app.patch('/users/:id', (request,response) => {
  //to define what updates are allowed,
  const updates = Object.keys(request.body);
  //Creates an array of keys(updates) request attempts to update
  const allowedUpdates = ['name', 'age', 'email', 'password'];
  const isValid = updates.every((update)=>allowedUpdates.includes(update));
  //isValid is a boolean, true if all items in updates satisfy the condition
  if (isValid) {
    User.findByIdAndUpdate(
      request.params.id, request.body,
      {new: true, runValidators: true},
    )
    .then(result=> result
      ? response.status(200).send(result)
      : response.status(404).send())
    .catch(error=>response.status(400).send(error));
  }
  response.status(400).send('Invalid update request');
})
```

- By default, if the client tries to request for an update on a field that does not exist, such as pets or hobbies, Mongoose ignores the patch request and simply returns the query result
- However, we can handle how to reject invalid requests on our own

- Delete

```
app.delete('/users/:id', (request,response)=> {
  User.findByIdAndDelete(request.params.id)
```

```
  .then(result=> result ? response.send(result) : response.status(404).send())
  .catch(error=>response.status(500).send());
})
```

- Again, recall that queries will fulfill the promises even if there was no result, so one has to check if result exists
- This also means the response.status(500).send() will only exist if you passed in an invalid id, where invalid means it is not a hexadecimal string

- Separating route files
  1. It is often a good idea to separate route handlers into separate route files, where each file contains the route handlers for a unique collection
  2. This involves the following steps: Creating a new router, calling methods on the router such as get/post/delete, then attaching the created router to the app using app.use()

```
const router  = new express.Router();
router.get('/test', (req,res)=> {
  res.send('This is a new router')
})
app.use(router);
```

- What are Router objects?
- A Router object is an isolated instance of middleware and routes. It is a "mini application" capable of performing middleware and routing operations
- This means that we are able to use app.use() on it AND HTTP method routes on it, such as .get, .post etc

  3. Recall, app.use() accepts a path and a middleware function. If no path is specified, then the middleware function is executed on all requests
  4. In a routers folder, create a router file for each collection

```
const express = require('express');
const User = require('../models/user');
const router = new express.Router();

router.post('/users',(request,response)=>{
  const user = new User(request.body);
  user.save()
  .then(()=>response.status(201).send(user))
  .catch(error=>response.status(400).send());
});
router.get('/users', (request,response)=> {
  User.find({})
  .then(result=>response.status(200).send(result))
  .catch(error=>response.status(500).send());
})
router.get('/users/:id', (request,response)=> {
  User.findById(request.params.id)
```

```
  .then(result=> result ? response.send(result): response.status(404).send("No such user"))
  .catch(error=>response.status(500).send());
})
router.patch('/users/:id', (request,response) => {
  const updates = Object.keys(request.body);
  const allowedUpdates = ['name', 'age', 'email', 'password'];
  const isValid = updates.every((update)=>allowedUpdates.includes(update));
  if (isValid) {
    User.findByIdAndUpdate(request.params.id, request.body, {new: true, runValidators: true})
  .then(result=> result ? response.status(200).send(result) : response.status(404).send())
  .catch(error=>response.status(400).send());
  return;
  }
  response.status(400).send('Invalid update request');
})
router.delete('/users/:id', (request,response)=> {
  User.findByIdAndDelete(request.params.id)
  .then(result=> result ? response.send(result) : response.status(404).send())
  .catch(error=>response.status(500).send());
})
```

- Migrate your route handlers over, and instead of attaching to app using app.get() or app.post,
- Attach them to the router using router.post, router.get
- Export the router,
- Then in the index.js attach the router to the app using app.use(router);

5.

- Authentication and security

1. Secure passwords
    - Storing passwords is never a good idea as a compromised database would mean having no secondary protection for passwords
    - Instead, passwords should be stored as hashes ⇒ where we pass in the password into a hash function that generates a hash
    - One commonly used hashing alogrithm is bcyrpt from npm

```
bcrypt.hash(String<Password>, Int<Rounds>)
//Rounds refer to how many times the hashing algorithm is executed

bcrypt.compare(String<Passsword>, String<HashedPassword>)
//hashes the Password and compares it to the hashed one to return a bool
```

        - Generally, you only need to use .hash and .compare from this library
    - Hashing vs encrypting: With encryption, one is able to reverse outputs to obtain inputs; Hashing on the other hand, is a one way process, it is not possible to convert a hashed string back into the original string

- Hashing is done before saving a User/his password, therefore, we make use of Schema.pre(), to use a middleware that executes before save()

```
userSchema.pre('save', async function(next) {
  const user = this;
  if (user.isModified('password')) {
    user.password = await bcrypt.hash(user.password, 8);
  }
  next();
})
```

  - This means, before save() executes on a document, check if the password field was modified and if so, hash it before storing
  - This also means that in your route handlers, ensure that .save() is called. This means using .update() or findByIdAndUpdate() should not be used
  - Fields are considered to be modified when first created or when updated
  - .isModified is provided by Mongoose
-

2. User Authentication
   - User Authentication is done through post requests
   - We therefore configure our router(to create a route handler) and schema(define the reusable login function to use in route handlers) to handle such a request

     First, creating the reusable function on the schema

```
//To define a static loginfunction,
userSchema.statics.login = async(email,password) => {
  const user  = User.findOne({email}) //same as {email: email}
  if (!user) {
    throw new Error('Unable to login');
  }
  const isMatch = await bcyrpt.compare(password, user.password);
  if (!isMatch) {
    throw new Error('Incorrent login credentials');
  }
  return user;
}
```

     Then, creating a route handler in the router

```
router.post('/users/login', async(request,response)=> {
  try {
    const user = await User.login(request.body.email, request.body.password);
    response.send(user);
  } catch(e) {
    response.status(400).send();
```

```
  }
})
```

- Note, one must ensure that email addresses are unique, by ensure that the email field in userSchema has unique: true

```
const userSchema = mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
      trim: true,
    },
    email: {
      type: String,
      unique: true,
      required: true,
      trim: true,
      lowercase: true,
      validate(value) {
        if (!validator.isEmail(value)) {
          throw new Error('Invalid email');
        }
      }
    },
    password: {
      type: String,
      required: true,
      trim: true,
      minlength: 7,
      validate(value) {
        if(value.toLowerCase().includes('password')) {
          throw new Error('Password contains \'password\'');
        }
      }
    },
    age: {
      type: Number,
      default: 0,
      validate(value) {
        if(value<0) {
          throw new Error('Invalid age');
        }
      }
    }
  }
);
```

-

3.

- JWT - Json Web Tokens
  1. All express routes should be categorized into two categories
  2. Public - such as Sign In or Login

3. Private - one has to be authenticated to access the route
4. The way this is done is : When a user logins or registers, return a token. When a private route is accessed, require a token
5. JWT is the common standard used to manage this and it supports many operations, most commonly, authentication
6. Creating authentication tokens with JWT

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

const token = jwt.sign(
  {_id: user.id},
  "thisisasecretsignature",
  {expiresIn:'1 second',
);
console.log(token);
//eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
//.eyJfaWQiOiJwbGFjZWhvbGRlcklkIiwiaWF0IjoxNjE1NDU0NzY3fQ
//.NXO02l4W_P9Y8CoB3v7xEXaiK2HtRksDBnOERpyVYN4
```

   - .sign() returns a token, and accepts 3 arguments: the first is what data is embedded in the created token, the second is a unique secret signature that shows the token has not be tampered with. The third is a options object which can be used to configure
   - As shown above, the JWT is split into 3 parts by periods
   - The first is the header, a base-64 encoded json string - contains meta information on what type of token this is and the algorithm used to generate it etc
   - The second is the payload/body, a base-64 encoded json string - contains the data we provided as first argument to .sign + some meta data like timeCreated etc
   - The last is the signature - second argument we used in .sign and is used to verify the token
7. Validating tokens
   - jwt.verify(token, signature) will either return data(the payload/body) of the token or thrown an error

     ```
     const token = jwt.sign({_id: "placeholderId"}, "thisisasecretsignature");
     console.log(jwt.verify(token,"thisisasecretsignature"));
     //{ _id: 'placeholderId', iat: 1615455231 }
     ```

   -
8. Signing and Loggin in - Returning tokens
   1. Ignoring all the above, the first step is returning a token after authentication
   2. Tokens are unique to individual users themselves

3. Therefore the generation of tokens does not need the entire User collection, but only the user logging in
4. This means that unlike authentication, the function we use to return a token for a user should not be defined in the schema using Schema.statics - the function will be a method and not a static method
5. Methods are defined using Schema.methods
    1. The difference between Schema.statics and Schema.methods is in Schema.statics this refers to the Model, while in Schema.methods, this refers to the instance
    2. As we will be using the value of this binding regularly, arrow functions are avoided

```
//First, add a tokens field to userSchema

const userSchema = mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
      trim: true,
    },
    email: {
      type: String,
      unique: true,
      required: true,
      trim: true,
      lowercase: true,
      validate(value) {
        if (!validator.isEmail(value)) {
          throw new Error('Invalid email');
        }
      }
    },
    password: {
      type: String,
      required: true,
      trim: true,
      minlength: 7,
      validate(value) {
        if(value.toLowerCase().includes('password')) {
          throw new Error('Password contains \'password\'');
        }
      }
    },
    age: {
      type: Number,
      default: 0,
      validate(value) {
        if(value<0) {
          throw new Error('Invalid age');
        }
      }
    },
    tokens: [{
      token: {
        type: String,
        required: true,
      }
```

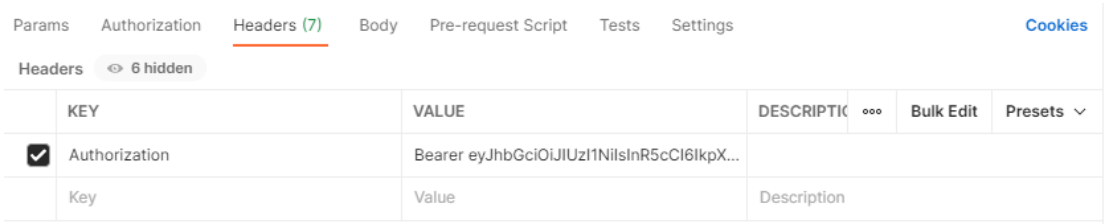
```
    }]
  }
);
```

- Above, we see that tokens contain an array of objects where each object contains a token key-value pair
- Objects are seen as a collection, therefore, the User collection has a document>document has a tokens field> tokens field has an array of collections>each collection contains a document with only 1 field, token(_id is another field auto generated for each document)
- tokens must be an array, since it is possible for a user to be logged onto multiple devices

```
userSchema.methods.generateAuthToken = async function() {
  const user = this;
  const token = jwt.sign({_id: user._id.toString()}, 'privateKey');
  user.tokens = user.tokens.concat({token});
  await user.save();
  return token;
}
```

6. Recall that only signup and login does not require tokens to be verified, while every other private operation should require tokens

9. Adding tokens to requests
   - Tokens are added to request.headers



   - The value is a String, where we must cut off the 'Bearer ' portion

```
const token = request.header('Authorization').replace('Bearer ', '');
//.replace replaces the first string with the second
//effectively, replacing 'Bearer ' with nothing, which returns the token itself
```

10. Verifying tokens for private requests
    - Involves using middleware for all private request
    - We create a middleware functions folder, and name a file 'Auth' which will verify tokens for all private requests

11.

- Postman
  - If using postman to send request, authentication can be set a lot faster
  - First, create an environment with environment variables(url, authToken)
  - Next, to the main collection, add authentication to use the authToken environment variable
  - Then to create and login requests, make sure authToken environment variable is set up to be the returned value
    - For all requests, set up authentication to inherit from parent(main collection who uses the env var)
- Sorting, Pagination and Filtering
  - The obvious advantage is improvement to queries when we fetch less data
  - Filtering allows us to be more specific about what data we want to fetch
    - For this reason, we use populate as it accepts a match field

```
await request.user.populate({
            path: 'tasks',
            match: {
                completed: true,
            }
        }).execPopulate()
```

    - The above will populate user.tasks with tasks whose completed value is true
    - To receive queries, such as ?complete=false, use request.query.queryName

```
router.get('/tasks', auth, async (request, response) => {
    try {
        const match = {};
        if (request.query.complete) {
            match.completed = request.query.completed === 'true'
        }
        await request.user.populate({
            path: 'tasks',
            match, //match: match
        }).execPopulate();
        response.send(request.user.tasks);
    } catch (e) {
        response.status(500).send()
    }
})
```

    - Note, queries are always in string form

    - Therefore, when handling boolean queries, compare the query to a boolean.toString()

- Pagination
    - Pagination is creating pages of data on request, so that you do not fetch all data at once
    - A google search for example splits all its results into pages, which are only loaded when the user choose to
    - Pagination requires two options to be configured: limit and skip
    - These options are configured on the populate argument as well

```
router.get('/tasks', auth, async (request, response) => {
    try {
        const match = {};
        if (request.query.complete) {
            match.completed = request.query.completed === 'true'
        }
        await request.user.populate({
            path: 'tasks',
            match, //match: match
            options: {
              limit: parseInt(request.query.limit),
              skip: parseInt(request.query.skip),
            }
        }).execPopulate();
        response.send(request.user.tasks);
    } catch (e) {
        response.status(500).send()
    }
})
```

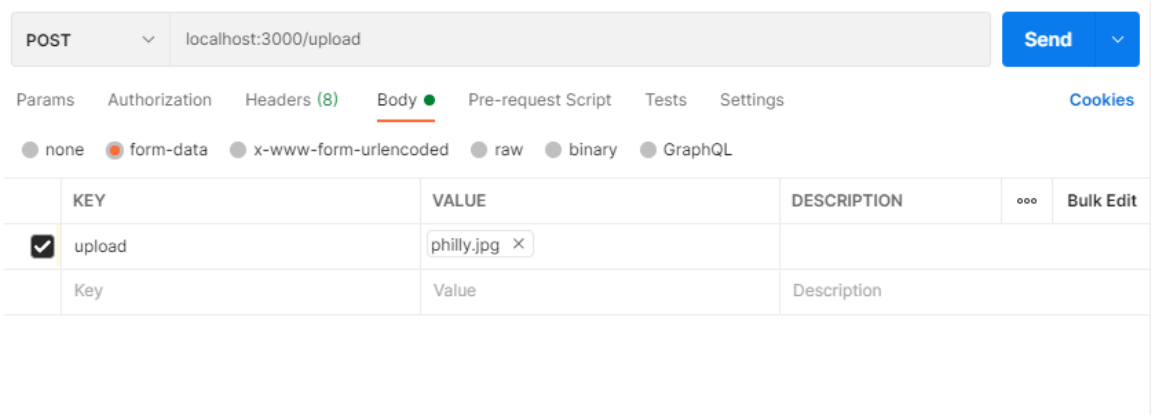
        - Again, all query values are strings, so parseInt() must be used

- Sorting
    - sorting uses the sortBy query, which involves two pieces of information - fieldName and sortOrder
    - For example /tasks?sortBy=createdAt_asc
    - sorting is configured on the sort field of the populate argument

```
router.get('/tasks', auth, async (request, response) => {
    try {
        const match = {};
        const sort = {};
        if (request.query.completed) {
            match.completed = request.query.completed === 'true'
        }
        if (request.query.sortBy) {
            const parts = request.query.sortBy.split(":");
            sort[parts[0]] = parts[1]==='asc'? 1: -1;
        }
```

```
        await request.user.populate({
            path: 'tasks',
            match,
            options: {
                limit: parseInt(request.query.limit),
                skip: parseInt(request.query.skip),
                sort,
            }
        }).execPopulate();
        response.send(request.user.tasks);
    } catch (e) {
        response.status(500).send()
    }
})
```

- File uploads
  - File uploads are not suported by default on Express servers
  - We use a tool called multer, which exposes a middleware function that we use, upload.single(), which takes in a string value of the fileName
  - File uploads are post requests, where the data we pass in is in the form of form-data
    - The key of the form-data should match that which is passed into upload.single()




```
const multer = require('multer');
//returns a multer instance that provides methods for generating middleware
//that will process file uploads
const upload = multer({
  dest: 'images',
})
app.post('/upload', upload.single('upload'), (request,response)=> {
  response.send();
})
```

  - Here, when /upload receives a post request, upload.single() will look for a value with key=upload and store it somewhere on the local file system(because dest: is configured)

- dest should not be set in production, instead, one should store the file on the server the app is deployed on
- if dest is not configured, multer will attach the file to the request object for the routehandler to access it
- In production, we can directly store the image in binary on the User instance. To do so, we need to declare another field for the User model

```
avatar: {
      type: Buffer,
    }
```

- After that we story the binary data, which is available on request.file.buffer to it

```
router.post('/upload/me/avatar',
  upload.single('avatar'),
  async (request,response)=> {
    request.user.avatar = request.file.buffer;
    awaitrequest.user.save();
    response.send();
    },
    (error,request,response, next)=>response.status(400).send(error.message)
);
```

-

- Validation for file upload
  - When users upload files, we need to check two key information: file size and file type
  - This is done by configuring multer

```
const upload = multer({
  dest: 'images',
  limits: {
    fileSize: 1000000,
  },
  fileFilter(request,file,callback) {
    //Callback accepts two arguments,
    //first one is an error, the second one is a boolean value
    callback(new Error('Unsupported file type')) //an error is thrown
    callback(undefined,true) //an upload is expected
    //Therefore we conditionally call the callback
  }
})
```

    - fileFilter is provided by multer and is automatically executed when uploads.single() is called
    - the second argument, file, contains file information that is defined on the docs - fieldname, originalname, encoding etc

- originalname is what we use to determine filetype as it is the name of the file on the computer

```
const upload = multer({
  dest: 'images',
  limits: {
    fileSize: 1000000,
  },
  fileFilter(request,file,callback) {
    //Callback accepts two arguments, first one is an error,
    //the second one is a boolean value
    if (!file.originalname.endsWith('.pdf')) {
      return callback(new Error('Unsupported file type'))
      //an error is thrown
    }
    callback(undefined,true) //an upload is expected
  }
})
```

- Error handling
  - When validation fails, we want to be able to send our own response object
  - This can be done with the middleware chaining, where the pattern is after the error, a middleware that accepts the error object will handle it

```
router.post('/upload/me/avatar',
  upload.single('avatar'),
  (request,response)=> {
    response.send();
  },
  (error,request,response, next)=>response.status(400)
    .send(error.message);
)
```

- Fetching binary data and exporting as jpg type

```
router.get('/users/:id/avatar', async (request,response)=> {
    try {
        const user = await User.findById(request.params.id);
        if (!user || !user.avatar) {
            throw new Error();
        }
        response.set('Content-Type', 'image/jpg');
        //default Content-Type was set to 'application/json' for us by express
        //now we say that the return type is jpg
        response.send(user.avatar);
    } catch (e) {
        response.status(404).send();
    }
})
```

- Final product

```
const upload = multer({
    limits: {
        fileSize: 1000000,
    },
    fileFilter(request, file, callback) {
        if(!file.originalname.match(/\.(jpg|jpeg|png)$/)) {
            return callback(new Error('Unsupported file type(s)'));
        }
        callback(undefined,true);
    }
})

router.post('/users/me/avatar',
  auth,upload.single('avatar'),
  async (request,response)=> {
    request.user.avatar = request.file.buffer;
    await request.user.save();
    response.send();
  },
  (error,request,response, next)=>response.status(400).send(error.message)
);

router.delete('/users/me/avatar', auth, async (request,response)=> {
    if (!request.user.avatar) {
        return response.status(400).send({message: 'Avatar does not exist'});
    }
    request.user.avatar = undefined;
    await request.user.save();
    response.send({message: 'Avatar deleted!'});
})
router.get('/users/:id/avatar', async (request,response)=> {
    try {
        const user = await User.findById(request.params.id);
        if (!user || !user.avatar) {
            throw new Error();
        }
        response.set('Content-Type', 'image/jpg');
        //default Content-Type was set to 'application/json' for us by express
        //now we say that the return type is jpg
        response.send(user.avatar);
    } catch (e) {
        response.status(404).send();
    }
})
```

- Setting auto-cropping and image formatting using Sharp

```
router.post('/users/me/avatar',
  auth,upload.single('avatar'),
  async (request,response)=> {
    const buffer = await sharp(request.file.buffer)
      .resize({width:250, height:250}).png().toBuffer();
    request.user.avatar = buffer;
    await request.user.save();
    response.send();
  },
  (error,request,response, next)=>response.status(400).send(error.message)\
);
```

-
-

- Email notifications with SendGrid
  - Set up SendGrid by integrating their email API key, creating a sender identitify, verifying the email of sender identity, and completed the email API integration
  - In a separate folder and file,

```
const sgMail = require('@sendgrid/mail');
const sendGridAPIKey = "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";
sgMail.setApiKey(sendGridAPIKey);

const sendWelcomeEmail = (email, name)=> {
  sgMail.send({
    to: email,
    from: 'winston.lim.cher.hong@gmail.com',
    subject:'Thank you for joining the app',
    text: `Welcome to the app, ${name}. Let me know your valuable feedback`,
  })
}
const sendExitEmail = (email, name)=> {
  sgMail.send({
    to: email,
    from: 'winston.lim.cher.hong@gmail.com',
    subject:'We are sorry you\'re leaving',
    text: `Let us know how we can improve and better suit your needs`,
  })
}


module.exports = {
  sendWelcomeEmail,
  sendExitEmail,
}
```

    - While email operations are asynchronous, there is no need to await response
- Environment variables
  - Environment variables are variables that exist only in certain environments - locally created env var will not appear on production when deployed(for example, heroku provides its own env var which will expose process.env.PORT, for deployed express server to listen with)
  - Environment variables are also more secure - they are not exposed to the public and allow you to store API keys, passwords etc
  - To configure environment variables, in a separate config folder, create a dev.env file
  - .env files accept one key-value pairs per line

```
PORT=3000
SENDGRID_API_KEY=SG.HIAWGAWIGNAIGNSPOVSPOVKk
```

    - Note, do not pass in string values with quotes
- After configuring environment variables, we need a way to register the .env file
    - One such package is the env-cmd package which is cross platform compatible
    - Use —save-dev flag as this is only required locally to register our local env var
    - It is convenient to run env-cmd whenever we start the server. However, in our package.json script, 'start' is usually reserved as it is used by servers when deployed
    - We therefore attach this on 'dev'

```
"scripts": {
    "start": "node src/index.js",
    "dev": "env-cmd -f ./config/dev.env nodemon src/index.js"
  },
```

    - Again, environment variables will be accessible via process.env.variableName
-

16. Deployment to Production(wrt task-node-app)

- Create a local database to work with - in this case we use mongoDB and run a locally using mongod.exe
- Use a ODM - in this case Mongoose
- Use ODM to connect to database
- Create an express server, and run the ODM script that connects to local server

▼ Define User model for authentication

- Define schema - name,email,password,tokens, ensure options set to timestamps: true for schema
- Define properties for schema - type, required, trim, unique, minlength, default
    - unique is for emails, where no email can register twice
    - validate is a function that will throw a Validation error if an error is thrown inside it
    - Use validators that are already out there such as npm validator which provides methods such as <bool> isEmail()
- Define Schema static methods
    - Static methods are available on the Model itself, and have access to the entire User collection, which usually is only required by 1 method - Logging in
        - For loggin in, ensure that passwords are encrypted, we used bcrypt which provides a .compare(password, user.password)

        - Registration is already provided with .save()
            - For registration, passwords must be encrypted, so we use a middleware that runs before .save()
            - Middleware uses .isModified('password') from mongoose to check if the instance password value changed, and if so calls bcrypt.hash()
    - Define Schema methods
        - Methods are available on all instances and only has access to the instance itself
        - The most common methods are .generateAuthToken, which only needs access to a instance itself, and .toJSON, which is automatically executed by express for all response.body objects
        - .generateAuthToken uses jwt, calls jwt.sign() and stores the token inside the instance property tokens
        - .toJSON, is overwritten by us to return our custom object of choice
    - Define Schema middleware functions
        - Usually .pre will be for 'save' and 'delete'
        - For 'save' as mentioned above, passwords will be check and hashed
        - For 'remove', a check is done to remove all data of the user that exists in other collections before the user is removed from the collection

▼ Define User router

- C(post) = Create user, Login user, Logout user, Logout all
- R(get) = Read Profile
- U(patch) = Update user
- D = Delete user, Delete avatar
- For created routes, categorize them into two - private and public
- For all private ones, the user must be authenticated i.e. have a token
    - The token is passed in through headers, and accessed through request.header('Authorization')
- This is managed with an authentication middleware which will draw out the token, call jwt.verify which compares the signatures, then checks if user exists on our DB and if that user has the same token in its tokens property

```
const user = await User.findOne({_id: decoded._id, 'tokens.token': token});
//'tokens.token' string property syntax
//is for searching through the tokens array
//comparing each token object to a value
```

        - For optimized performance, the middleware also attaches the fetched user and token object to request - request.user = user; request.token = token;
        - The token is for logout purposes
    - All private routes will then execute auth before its route handler
    - For patch requests on Users, there must be a check to filter what can be updated

```
const updates = Object.keys(request.body);
    const allowedUpdates = ['name', 'email', 'password', 'age'];
    const isValidOperation = updates.every((update) => allowedUpdates.includes(update));

    if (!isValidOperation) {
        return response.status(400).send({ error: 'Invalid updates!' });
    }
    try {
        updates.forEach((update) => request.user[update] = request.body[update])
        await request.user.save()
        response.send(request.user);
    } catch(e) {
      //.....
    }
```

- use the User router on express
- Define other models routers and use them
- Decide environment variables - such as DB connectionUrl, private API keys
- When ready to deploy, follow the same order
- First host the database
- Next deploy the server
    - Register environment variables

17. Testing

- Why tests
    - Resuable automatic tests that keeps running
    - Fewer unexpected errors, test cases should be aware of all errors
    - Collaboration is easier - changes can be tested
    - Profiling is easy - do they changes affect performance
    - The idea of tests - to run tests on every small progress so no breaking changes are introduced
- Using Jest

- Testing is done in local development therefore they are devDependencies

```
//npm i jest --save-dev
"scripts": {
    "start": "node src/index.js",
    "dev": "env-cmd -f ./config/dev.env nodemon src/index.js",
    "test": "jest --watch"
  },
```

  - —watch will run tests after the test case file is saved
- Create a root subfolder called tests where we store test cases
- test cases are identified by .test
- Jest creates global functions -such as test() available to us
- Testing outputs

```
test('Value of a sum',()=> {
  const total = 10+3;
  expect(total).toBe(13);
})
```

  1. expect() is another global function provided by JestJs, and it can check many things, such as .resolves(), toBeDefined() etc
  2. .toBe() is the most common method used
  3. expect automatically prints diagnostic messages such as expected value vs output etc
- Testing asynchronous code
  - The callback function passed into test() accepts a done argument
  - done is passed in, Jest knows that asynchronous code is running, and will not evaluate the test until done() is called

```
const addPositiveIntegers = (a,b) => {
  return new Promise((resolve,reject)=>{
    setTimeout(()=> {
      if (a<0||b<0) {
        return reject('Numbers must be non-negative');
      }
      resolve(a+b);
    }
  });
}

test('Adds two positive integers',(done)=> {
  add(2,3).then((sum)=>{
    expect(sum).toBe(5);
    done();
  });
});
```

```
test('Adds two positive integers',async()=>{
  const sum = await add(2,3);
  expect(sum).toBe(5);
  done();
});
```

- Jest lifecycle methods(Setup and Teardown)
  - Lifecycle methods are code that are run just before or after a test is executed, similar to .pre() and post() in Mongoose
  - beforeEach() accepts a handler that executesbefore each test
  - afterEach()
- Testing with express
  - Create a test.env file and provide environment variables
    - This should be the same as the dev.env file except it uses a different mongoDB
  - Register the env for testing

```
"scripts": {
    "start": "node src/index.js",
    "dev": "env-cmd -f ./config/dev.env nodemon src/index.js",
    "test": "env-cmd -f ./config/test.env jest --watch"
  },
```

  - Configure properties for jest in package.json according to documentation

```
"jest": {
    "testEnvironment": "node"
  },
```

  - Next, we use a npm library called supertest which makes testing especially easy on express servers - supertest allows for easy syntax for testing of http requests
    - supertest also does not require the actual server to be up and running - you do not need to call app.listen(port,//...)
    - npm i supertest —save-dev
  - Refactor files such that express server is created in a separate file from where the application listens to a port;

```
const express = require('express');
require('./db/mongoose');
const userRouter = require('./routers/user');
const taskRouter = require('./routers/task');

const app = express();
```

```
app.use(express.json());
app.use(userRouter);
app.use(taskRouter);

module.exports = app;
```

```
const app = require('./app'); //runs above script to load express server
const port = process.env.PORT;

app.listen(port, ()=> {
  console.log('Server is up on port ', port);
})
```

- This is because supertest is able to directly interact with the express app instead of through http requests

```
const request = require('supertest');
const app = require('../src/app');

test('Create a user', async()=> {
  await request(app).post('/users').send({
    name: 'Winston',
    email: 'winston.lim.cher.hong@gmail.com',
    password: 'password'
  }).expect(201);
})
```

  - expect() asserts that the response.status should be 201
  - The problem with testing with created users is that future tests will all fail, since the email can only be registered once
  - Therefore, we need to wipe the database everytime a test is done
  - This is done using jest lifecycle method beforeEach

- Create more deep assertions and tests for all authentication utilities

```
test('Should create a user', async()=> {
  const response = await request(app)
  .post('/users').send({
    name: 'Winston',
    email: 'winston.lim.cher.hong@gmail.com',
    password: 'testing'})
  .expect(201);

  //Assert that the database was updated
  const user = await User.findById(response.body.user._id);
  expect(user).not.toBeNull();

  //Assert that database was updated with correct user
  expect(response.body).toMatchObject({
    user: {
      name: 'Winston',
```

```
      email: 'winston.lim.cher.hong@gmail.com',
    },
    token: user.tokens[0].token,
  })
})
```

- Mocking
  - The process of replacing functions in NodeJS with mock functions
  - For example, for SendGrid.send() is executed on every use created and it shouldn't
  - To create mocks, first create a folder **__mocks__** which is what JestJs identifies where to search
  - Inside that folder, we mock a package
    - E.g. for sendgrid,

```
const sgMail = require('@sendgrid/mail');
sgMail.send({
    to: email,
    from: 'winston.lim.cher.hong@gmail.com',
    subject:'Thank you for joining the app',
    text: `Welcome to the app, ${name}. Let me know your valuable feedback`,
  })
```

    - We see that the package is scoped
    - The equivalent of that is to create a '@sendgrid' folder, which contains a 'mail.js' file
    - Inside mail.js, must provide the functions that were used, in this case only .send

```
module.exports = {
  setApiKey() {

  },
  send () {

  }
}
```

    - Jest automatically uses mock functions that were mocked inside the __mock__ folder
  -

- Fixtures
  - Fixtures are things that allow you to set up the environment you are goint to run in
  - For tests, we store assets we need to run tests, such as files that are going to be uploaded

- For file uploading, the code looks like that

```
test('Should upload avatar image', async ()=> {
  await request(app)
  .post('/users/me/avatar')
  .set('Authorization', `Bearer ${testUser.tokens[0].token}`)
  .attach('avatar','tests/fixtures/profile-pic.jpg')
  .expect(200);

  //Assert that avatar binary data was stored
  const user = await User.findById(testUserId);
  expect(user.avatar).toEqual(expect.any(Buffer));
})
```

  - As we know, files are passed in through form-data. In supertest, this is with the attach method
    - The form-key is passed in as first argument
    - The path to the file to be attached is passed in as second argument
- In fixtures, we also set up db.js, which will create a testUser for us to work with in all testing suites.
- toEqual() is used when comparing objects
  - toBe() is similar to ===
  - toEqual() runs its own algorithm to compare properties of two objects

```
expect({}).toBe({});
//failed
expect({}).toEqual({});
//passed
```

- expect.any() simply returns a generic object with a type
  - When coupled with toEqual(), we can check if user.avatar is of type = Buffer

```
expect(user.avatar).toEqual(expect.any(Buffer));
```

18. Web sockets

- Similar to HTTP, web sockets is a protocol that allows us to communicate with the server
- Web sockets allow for full-duplex communication: bi-direction communications between clients and servers
  - There is a persistent connection between the two for as long as it needs to
- This is unlike http where the communication is one directional

- Socket.io
  1. Setup
     - After installing socket.io dependency
     - Express server must be refactored such that we have access to a http server(which express does behind the scenes) for loading our web socket

```
const express = require('express');
const http = require('http');
const path = require('path');
const socketio = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = socketio(server);
const port = process.env.PORT || 3000;

const publicDirectoryPath = path.join(__dirname, '../public/');

app.use(express.static(publicDirectoryPath));

io.on('connection', ()=> {
  console.log('New web socket connection');
})

server.listen(port, ()=> {
  console.log(`Server is up on port ${port}`);
})
```

     - io.on() registers a event handler to a event - in this case 'connection'
     - Socket.io works only if the front end(client) can communicate with the server
     - Socket.io therefore provides its own client library through a script
     - Therefore, in index.html, we make the given script available to our client-side.js script

```
<script src= '/socket.io/socket.io.js'></script>
<script src= '/scripts/chat.js'></script>
```

       - Loading in the provided socket.io script allows our chat.js script to have access to functions to communicate with the http server
       - For example, calling io() will create a connection with the server
  2. Events and socket
     - Socketio works around events - it receives and responds with events
     - When a new connection event occurs for , a socket is sent to the server
     - A socket is an object that contains some information about the new connection, and a unique socket is returned from each connection event

```
io.on('connection', (socket)=> {
  console.log('new websocket connection');
});
```

- socket comes with methods we can use to communicate with the connected client

```
io.on('connection', (socket)=> {
  console.log('new websocket connection');
  socket.emit('countUpdate');
});
```

  - socket.emit() basically pushes/responds with an event
  - 'countUpdate' is a custom created event
  - When the server emits an event, the client must receive it

- On the front end,

```
const socket = io();
```

  - Recall that io() sends a connection 'request' event to the server, and will receive a returned socket object that it can use to further communicate with the server
  - Communicate = receive and send events
  - To the previously emitted event 'countUpdate', the client handles the event in a similar fashion

```
const socket = io();
socket.on('countUpdate', ()=> {
  //do something
});
```

3. Pushing data from server to client

```
//Server
let count =0;
io.on('connection', (socket)=> {
  socket.emit('countUpdate', count);
});

//Frontend
const socket = io();
socket.on('countUpdate', (count)=> {
  console.log(count);
});
```

- To pass data through socket.emit(), simply pass the data in as second(&3rd,4th,5h, as many) argument, and receive it on event handler

4. Pulling data from server

```
//Recall pulling is when client reqeusts for data
document.querySelector('#increment').addEventListener('click', ()=> {
  console.log('clicked')
  socket.emit('increment');
})

//Server
let count =0;
io.on('connection', (socket)=> {
  socket.emit('countUpdate', count);

  socket.on('increment', ()=> {
    count+=1;
    socket.emit('countUpdate', count);
  })
});
```

- When the increment button is clicked, the server receives a 'increment' event and responds with 'countUpdate'

5. socket vs io
    - Observing example 4, we see that io and socket are very similar - they have the .on() method and as we will soon learn, the .emit() method too
    - A socket is created for every unique connection, e.g. a new client ==new tab/window
    - io on the other hand, exists as the same object for all connections - and is useful when we want to interact with all active connections at once
    - For example, if a user increments the count by 1, we want that to apply to all connections - this is not possible in example 4, where the countUpdate is only reflected in that connection, because it is called on the socket
    - To do this, we must emit the event to all sockets

```
let count =0;
io.on('connection', (socket)=> {
  socket.emit('countUpdate', count);
  socket.on('increment', ()=>{
    count+=1;
    io.emit('countUpdate', count);
  });
```

6. socket.broadcast
    - In the event you want to emit an event to all connections except the current one, use socket.broadcast.emit()

7. Connection events
    - io.on('connection, ()⇒{}) is for when a connection is created
    - socket.on('disconnect') is when another socket disconnects, for example when another user leaves the chat

```
io.on('connection', (socket)=> {
  socket.emit('message', 'Welcome');
  //sends to all other connected sockets
  socket.broadcast.emit('Another user has joined the chat');
  socket.on('disconnect', ()=> {
    io.emit('message', 'A user has left the chat');
  }
});
```

        - When a socket disconects, it will automatically call emit('disconnect') - handled by socket.io

8. Event acknowledegements
    - Acknowledgements are called by an event receiver when it successfully receives a event to inform the sender that it has received and handled the event
    - Acknowledgements are handled with the sender(.emit) registering a callback function to execute when the receiver(.on) acknowledges

```
//client
const socket = io();
socket.on('serverMessage', (message,acknowledge)=> {
  console.log(message);
  acknowledge('Delivered');
});

//server
io.on('connection', (socket)=> {
  //acknowledgementMessage == delivered
  socket.emit('serverMessage', messageData, (acknowledgementMessage)=> {
    console.log(acknowledgementMessage);
  });
});
```

        - .emit() will accept a acknowledgement callback function as its **last** argument, which executes when the event handler calls acknowledge();
        - .on()'s callback function, similarly accepts an acknowledgement callback function as its last argument, and when called, triggers the acknowledgement callback
            - note that as with all callbacks, both (message,acknowledge) can be renamed to anything

```
const socket = io();
socket.on('serverMessage', (response, callback)=> {
  console.log(response);
  callback();
});
```

    - The above code works exactly the same
- The functionality acknowledgements provide is mainly a way for the server to respond to any event
- Most commonly, acknowledgements are used to perform some sort of validation to requests, as an error handler(only called when there is an error)

```
//server
socket.on('send-message', (message,error)=> {
    const filter = new Filter();
    if (filter.isProfane(message)) {
      return error('Message contains profanity')
    }
    io.emit('server-message', message);
  })

//client
const socket = io();
socket.emit('send-message', messageData, (errorMessage)=> {
    console.log(errorMessage);
  });
```

- Other use cases could be when you have to wait for the server to respond before executing something
    - When a form is submitted and you disable buttons on the form, you should only enable them after the server receives and sends back a response
-

9. Socket rooms
    - It is common that sometimes, you want to send responses to specific sockets only; how do we configure which sockets to send to?
    - So far, socket.emit, io.emit and socket.broadcast.emit are the three ways we communicate with sockets
    - socket.join() is a method available only on the server, and it introduces two more ways to interact with sockets

```
socket.on('join', ({username, room})=> {
    socket.join(room);
    socket.emit('server-message', generateResponse('Welcome'));
    socket.broadcast.to(room)
```

```
    .emit('server-message', generateResponse(`${username} has joined the chat`));
  })
```

- One way is shown above, socket.broadcast.to() means to everyone but current socket in the given room
- Another way is io.to(room).emit() which means to everyone in the room

- in the io.connection, a unique id is provided for each socket, accessible by socket.id
- This allows for all events to have access to that unique id, and explains why typically all crud operations for users of a room, involves the connection id

10. SocketIDs
    - For each connection, a unique id exists
11.

•

19. ChatApp

1. Here we learn socketio and some front end

▼ First, initialize the project with npm/dependencies, and create the express server

- Create a new project and initialize node
- Create two folders - src and public
    1. in src - index.main
    2. in public - create 3 folders assets, styles, scripts
    3. in public, create an index.html file
- In index.main, create an express server, and serve the public directory and render index.html

```
const express = require('express');
const path = require('path');
const app = express();
const port = process.env.PORT || 3000;
const publicDirectoryPath = path.join(__dirname, '../public/');
app.use(express.static(publicDirectoryPath));

app.listen(port, ()=> {
  console.log(`Server is up on port ${port}`);
})
```

- process.env allows you to access environmental variables

- Creating local env var works by creating a config(remember to .gitignore) folder inside root directory, and creating a .env file
- The file must be served up - we used env-cmd, which allows us to load .env files with scripts

```
"scripts": {
    "start": "node src/index.js",
    "dev": "env-cmd -f ./config/dev.env nodemon src/index.js",
    "test": "env-cmd -f ./config/test.env jest --watch --runInBand"
  },
```

- __dirname is the path of the current directory(folder)
- app.use() receives a middleware function. If no path is specified, then the middleware is executed on all requests
- express.static() is a predefined express middleware that helps serve static content

▼ Next, install socket.io and refactor code

```
const express = require('express');
const http = require('http');
const path = require('path');
const socketio = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = socketio(server);
const port = process.env.PORT || 3000;

const publicDirectoryPath = path.join(__dirname, '../public/');

app.use(express.static(publicDirectoryPath));

server.listen(port, ()=> {
  console.log(`Server is up on port ${port}`);
})
```

- socketio is initialized with a http server, therefore requiring us to create a reference to the http server created by express()
- normally the server is created behind the scenes and not required, but only for socket.io, we need it

▼ Install other javascript libararies- mustache, moment and qs using npm works

- How to use mustache to render templates
  - Create a div where the template should be rendered

```
<div id='messages'></div>
```

- Create script where the template will be loaded from

```
<script id='message-tempalte' type='text/html'>
  <div>
    <p>{{message}}</p>
  </div>
</script>
```

  - {{}} is a syntax from mustache that allows you to insert dynamic values into a template
  - the dynamc values must be passed in when Mustache.render() is called to compile the script
- In client script, create references to the above two created objects

```
//Elements
const $messages = document.querySelector('#messages');

//Templates
const messageTemplate = document.querySelector('$message-template').innerHTML;
```

- Compile the template with data we want to render

```
const hmtl = Mustache.render(messageTempalte, {
  message: 'This is a test'
});
```

- Inserting the complied template into the div

```
$messages.insertAdjacentHTML('beforeend', html)
```

- Advanced: passing in arrays into Mustache.render(), for a

```
<ul class='users'>
    {{#users}}
      <li>{{username}}</li>
    {{/users}}
  </ul>
```

  - users is an array, #users and /users signifies the start and end of the array
  - within that is how each element in the array should be rendered, along with what property of the element
  - users.keys() = username, room, id
  - In this case we only want their usernames in a list

-
- Using moments to format time
  - Javascript's build in Date object returns a timestamp, a large number representing the number of seconds that have elapsed since 1st Jan 1970
  - This is universally accepted the Unix epoch, and is the format that works in most programming languages
  - Unfortunately, it is not very readable, and therefore libraries such as moments help to format these timestamps into human readable formats
  - moment(someTimeStamp).format('h:mm a'),
  - To learn how to pass tokens to format for custom time representations, head over to momentjs documentation
- Using Qs to format query strings
  - All query strings can be accessed with location.search, a global variable provided by the browser
  - Again, this will be client side javascript as it involves the browser
  - QS
-

▼ Create events for socket.io on the client and server

- Start with the server and think of all possible requests
  - For the chat app, we must listen to sent messages, sent locations, disconnections, and when working with rooms, joins
  - If working with rooms, then things to take note are: For CRUD, C is done on 'join', D is done on 'disconnect', for C, you must register a room property, for validation
  - Update is irrelevant for this, but R is used for 'private events', where you fetch the user according to socket.id, such that the event is only handled when it matches the user.room
- Create CRUD operations for relevant data
  - Each CRUD operation should clean, validate and handle errors for inputs
- Build client side html/ javascript and create event handlers
  - In general, client inputs are converted into event emittors - buttons, inputs etc
  - server responses are converted into rendered content - insertAdjacentHTML, innerHTML
  - Specific to rooms, usually one event listener would be for roomData changes - which is whenever a user 'join's or 'disconect's

- You need at least two screens - 1 for joining a room, 1 being the room itself