Node.js Net Ninja Notes

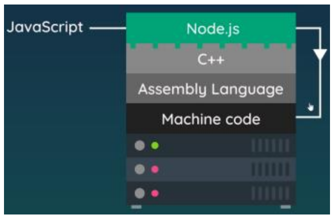
Last Updated: October 10, 2020

Source

- The Net Ninja Node.JS Crash Course (Link)

Tutorial 1 – Introduction and Setup

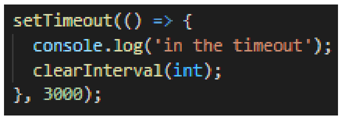
- Node.js allows us to run JavaScript directly on a server or on a computer, which was not allowed in the past.
    - The V8 engine is written in C++ by Google and allows JavaScript to compile to machine code inside of the browser at runtime.
    - JavaScript cannot be run outside of the browser environment.



- Node.js is written in C++ and wraps the V8 engine, so that the engine lives inside of Node.js.
    - Since Node.js is written in C++, the JavaScript can be compiled to machine code through the V8 engine outside of the browser environment.
- However, Node.js does more than just wrap the V8 engine as it can also:
    - Read and write files on a computer.
    - Connect to a database.
    - Act as a server for content.
- Obviously, since JavaScript is running outside of the browser, it loses access to the DOM, although that isn't needed anyways.
- Node.js is generally used to create web services that respond to user requests.
    - It's now an alternative to running Python, Ruby, or PHP on the back end.
- To create and run a JavaScript file using Node.js, simply install the latest version of Node.js in Windows or Linux and create a new file (test.js for example) and run it in the terminal by using node test.

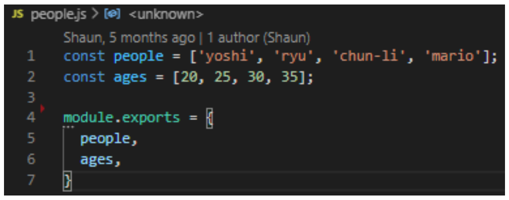
Tutorial 2 – Node.js Basics

- Node.js has a global object that contains premade functions, which can be viewed by using console.log(global).
    - We don't have to write global.setTimeout(...), but rather can just write setTimeout(...) since global is available to us out-of-the-box.



```
setTimeout(() => {
  console.log('in the timeout');
  clearInterval(int);
}, 3000);
```

- The __dirname and __filename commands are quite useful in Node.js, since we sometimes are working in different directories and need to formulate paths between them.
    - setTimeout(...) and setInterval(...) are both useful functions as well, since one can be used for timeouts while the other repeats at a set interval.
- Code can be broken up into modules, which allows us to import and export things to and from those files.
    - This allows code to be more modular, reusable, and maintainable.
    - To import a file or module, we can use require(...) which will find the file and run it.

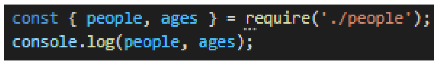


JS people.js > <unknown>

Shaun, 5 months ago | 1 author (Shaun)

```
const people = ['yoshi', 'ryu', 'chun-li', 'mario'];
const ages = [20, 25, 30, 35];

module.exports = {
  people,
  ages,
}
```

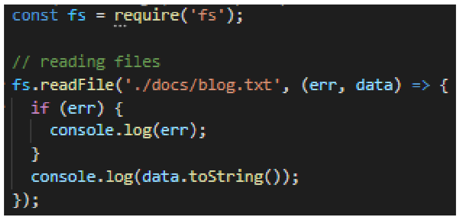
- If we want to export data from a module, we use the module.exports command.
    - We can then get those properties in another file by using require(...) along with destructuring (if multiple properties are exported).



```
const { people, ages } = require('./people');
console.log(people, ages);
```

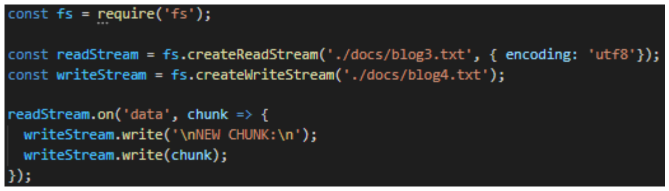
- Node.js also comes with its own modules, such as os, which gives us a lot of information about the operating system.
    - With the operating system module, we could get things such as the platform that the module is running on, or the home directory for the operating system.
- The Node.js file system core module can be used to do things like read files, write files, delete files, create directories, etc. and can be imported by using const fs = require('fs');

- If we use fs.readFile(...) for instance, which is an asynchronous function, we can pass in the path and file to read from and then can do something with the data from that file in an asynchronous fashion with the callback function.



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

// reading files
fs.readFile('./docs/blog.txt', (err, data) => {
  if (err) {
    console.log(err);
  }
  console.log(data.toString());
});
```

  - Note that when we use fs.writeFile(...), the text that we use as an input will overwrite that which was already in the file, if the file already existed.
  - fs.existsSync(...) can be used to check if a folder exists, and fs.mkdir(...) can be used to create a new directory.
  - fs.unlink(...) can be used to delete files.
- If we are working with much larger files, it can be more efficient to use streams instead to read and write from files.
  - With streams, we can start using the data before it's even been fully read by passing data along through a buffer.
  - This can be thought of in the same way as say a Netflix video is sent to you, where it is buffered and streamed one chunk at a time.
  - fs.createReadStream(...) can be used to create a read stream from a file, and readStream.on(...) can be used to read each buffered chunk of data as it is read from the file.

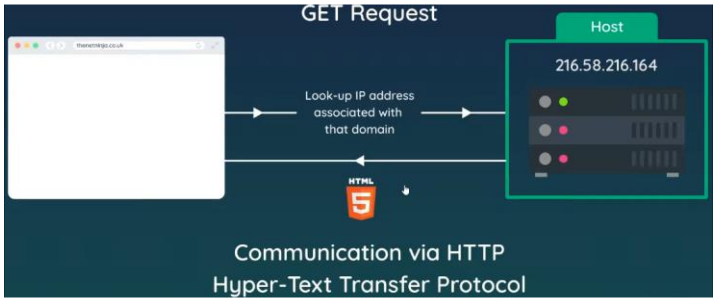


```
const fs = require('fs');

const readStream = fs.createReadStream('./docs/blog3.txt', { encoding: 'utf8'});
const writeStream = fs.createWriteStream('./docs/blog4.txt');

readStream.on('data', chunk => {
  writeStream.write('\nNEW CHUNK:\n');
  writeStream.write(chunk);
});
```

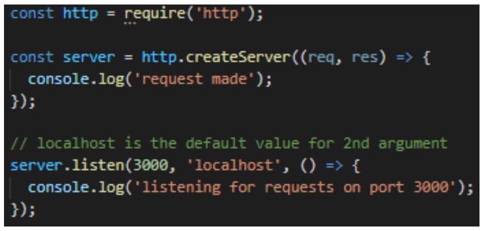
  - In the example, we are reading from one file in a utf8 encoded format and writing the buffered chunks to a different file.
  - Note that we could take advantage of pipes here to dramatically simplify the code by replacing the readStream.on(...) function with readStream.pipe(writeStream);
  - Although we did not talk about it here, a duplex stream is an option that allows us to read and write through it.

Tutorial 3 – Clients and Servers

- Each server, or host, will be associated with a unique IP address.
  - If we wanted to connect directly to our server, we could just type that IP address directly into the browser.
  - Domain names mask IP addresses, and browsers will look up the IP address associated with that domain before directing the browser to that IP address.
  - The server will then receive the request and respond, possibly with an HTML page.
- Whenever we type something into the browser and send off the request, that is called a GET request, which will return a resource.


GET Request
Host
216.58.216.164
Look-up IP address associated with that domain
HTML 5
Communication via HTTP
Hyper-Text Transfer Protocol

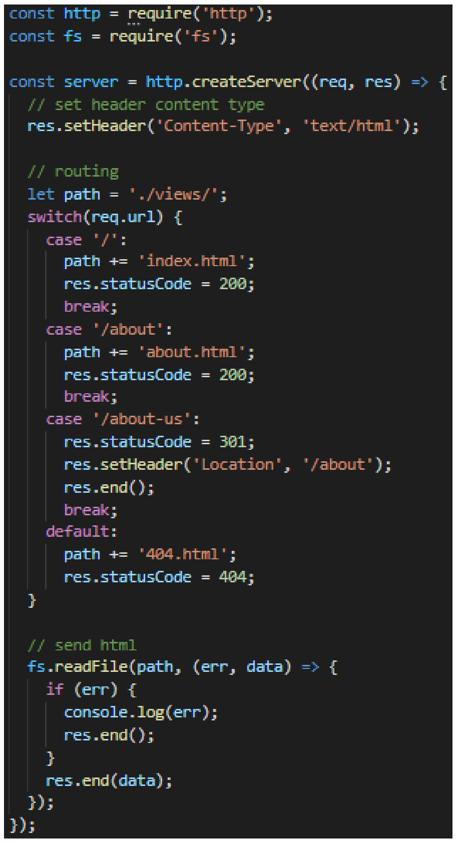
  - A POST request can be used to send data to a server for a web form, for example.
  - This communication between the server and the client occurs via Hyper-Text Transfer Protocol (HTTP), which is a set of instructions which dictates how that communication occurs.
- In Node.js, we must manually create a server which will live on the backend of our website.
  - We first import a core Node.js module called the HTTP module, and then use that module to create a server using http.createServer(...) which has a callback function that takes both a request object and a response object.



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

const server = http.createServer((req, res) => {
  console.log('request made');
});

// localhost is the default value for 2nd argument
server.listen(3000, 'localhost', () => {
  console.log('listening for requests on port 3000');
});
```

  - Note that the server can be stored in a variable and used later (e.g. for web sockets).
  - The request object contains information about the request, such as the URL that is being requested.
- In order to set the server to listen for requests, we have to use the server.listen(...) function, which takes in the port number to listen on along with the host name (e.g. localhost) and a function that runs with the method.
  - Note that localhost in this case is called a loopback IP address that is for your own machine, so the server for the website is your computer.

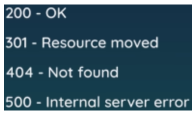
- Port numbers are like 'doors' into a computer that a software or service can communicate through.
  - Skype, Discord, Outlook, etc. are all assigned different port numbers on your computer through which they communicate so that information is kept separate from one another.
  - A common port number for localhost is 3000, although we should ensure that this port number is not being used by another program on your computer.
  - When we access our server, we type in the host name along with the port number (e.g. localhost:3000).
  - We then run the node file (e.g. node server) to start the server, which begins listening on the specified port.
- We should note that any log messages will be displayed on the server side and not in the browser.



```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
  // set header content type
  res.setHeader('Content-Type', 'text/html');

  // routing
  let path = './views/';
  switch(req.url) {
    case '/':
      path += 'index.html';
      res.statusCode = 200;
      break;
    case '/about':
      path += 'about.html';
      res.statusCode = 200;
      break;
    case '/about-us':
      res.statusCode = 301;
      res.setHeader('Location', '/about');
      res.end();
      break;
    default:
      path += '404.html';
      res.statusCode = 404;
  }

  // send html
  fs.readFile(path, (err, data) => {
    if (err) {
      console.log(err);
      res.end();
    }
    res.end(data);
  });
});
```

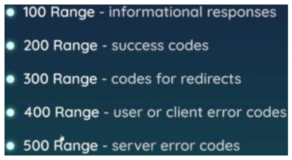
Tutorial 4 – Requests and Responses

- Note that whenever changes are made to the server code, the server must be restarted for the changes to take effect.
- Using req.url and req.method gives information about the URL that made the request, along with whether it is a GET, PUT, etc.
- When creating the response, we can provide back a response header by using res.setHeader(...), then res.write(...) to write to the response, and finally res.end() to send the response back to the client.
- Node.js can also be used to send back full HTML pages as the response to a user navigating pages.
  - Routing can be used since we can get the path for the page that the user has navigated to from the req.url method.
  - Of course, this would have limited use for a Single Page Application, such as that created by React or Angular.
- Status codes can also be provided along with the response to the browser, which will indicate to the browser how successful the response was.



200 - OK
301 - Resource moved
404 - Not found
500 - Internal server error

  - There are many different codes that exist, but most of them fall within certain ranges.



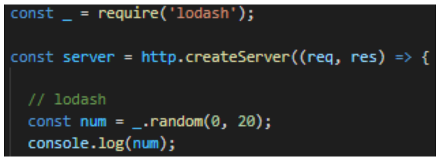
- 100 Range - informational responses
- 200 Range - success codes
- 300 Range - codes for redirects
- 400 Range - user or client error codes
- 500 Range - server error codes

- Redirects can be implemented by sending the proper status code of 301 along with a res.setHeader(...) with the redirect location defined (see screenshot).

Tutorial 5 – NPM

- There are thousands of 3rd party packages that can be used with Node.js, all of which can be installed using the Node Package Manager (NPM).
  - Nodemon is a package, for example, that allows the Node.js server to automatically refresh after changes have been made.
  - Using the command npm install -g nodemon will install the package globally (the -g part), which will make it available for all projects.
  - Now if we run the command nodemon server, the file will be run on a server that automatically updates with changes.
- If we are planning to use 3rd party packages in our project, we should run npm init to create a package.json file and a package-lock.json file.
  - Package-lock.json keeps track of dependency versions and shouldn't have to be updated, while package.json will keep track of all of the dependencies used for the project.
  - The major benefit of package.json is that you do not need to share your dependency folder – instead all dependencies can be installed from the package.json and package-lock.json files using npm install.
- Note that when installing a new dependency into our project, we shouldn't have to use npm i --save or anything since the dependency should automatically be added into the package.json file by newer versions of NPM.

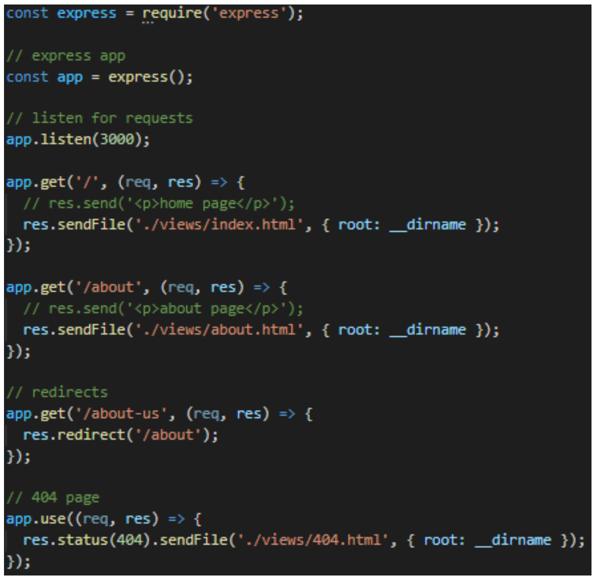
- A dependency called loadash is another useful one, since it contains a lot of utilities to make working with arrays, numbers, objects, strings, etc. much easier.
    - We need to use the require(…) method to load the dependency into our server, and it should be noted that Node.js will automatically know to look in the node_modules folder for the dependency.
    - This module allows you to use functions like .once(…) to use a function only one time, along with many others.



```
const _ = require('lodash');

const server = http.createServer((req, res) => {

  // lodash
  const num = _.random(0, 20);
  console.log(num);
```

Tutorial 6 – Express Apps

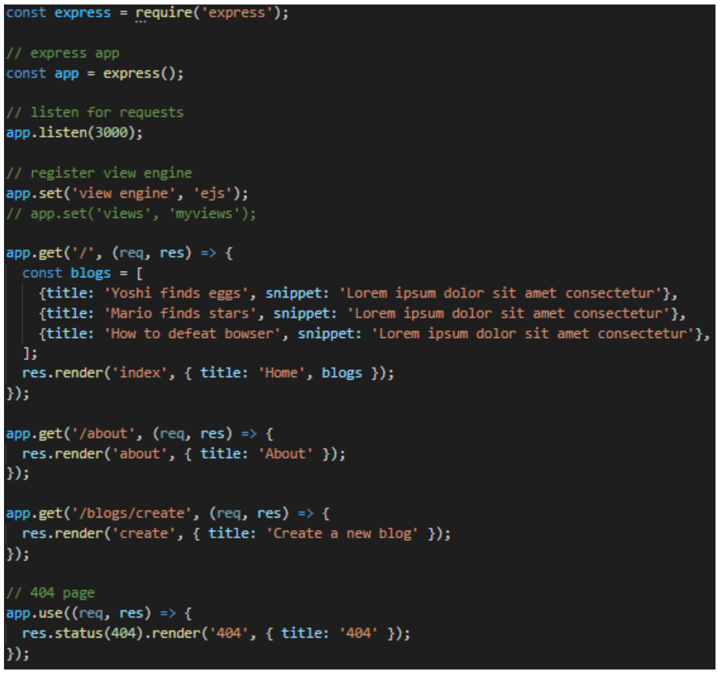
- Express allows us to write our server-side code in an elegant way, which can help us clean up the code.
    - Benefits include robust routing, high performance, high test coverage, HTTP helpers (redirection, caching, etc), support for template engines, content negotiation, and executables for generating applications quickly.
    - It can be installed by using npm install express.
- Once installed, we simply use require('express') and make express() equal to an app variable.
    - Then we can start using Express to help us with our routing.
    - Instead of having to use the res.setHeader(…), res.statusCode(…), res.readFile(…) and res.end() functions, we can just use res.sendFile(…) to automatically append the header info and send the page contents.
- It should be noted that we must provide the root path for the application for Express to use proper relative pathing.
    - Redirects are as simple as using res.redirect(…) inside of a route.
- For capturing invalid routes, we use app.use(…) along with the status code in res.status(…).
    - It should be noted that app.use(…) will only run if the code does not send a file further up in the code – that is, files that use Express run each time that a request is made to the server and only stop executing once either the end of the file is reached or a response is sent back to the client from res.sendFile(…), etc.
    - Therefore, positioning of an app.use(…) function is important in that it should be at the bottom of the file.
    - We also have to manually set the status code by using res.status(…) prior to sending the file.



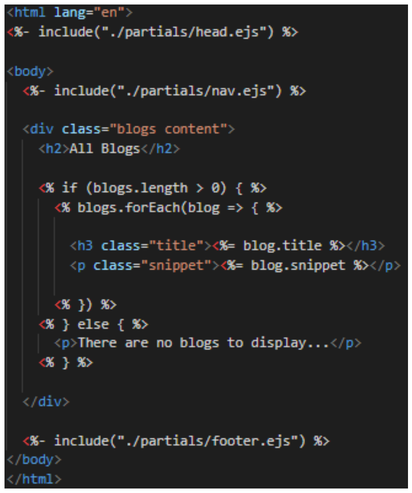
```
const express = require('express');

// express app
const app = express();

// listen for requests
app.listen(3000);

app.get('/', (req, res) => {
  // res.send('<p>home page</p>');
  res.sendFile('./views/index.html', { root: __dirname });
});

app.get('/about', (req, res) => {
  // res.send('<p>about page</p>');
  res.sendFile('./views/about.html', { root: __dirname });
});

// redirects
app.get('/about-us', (req, res) => {
  res.redirect('/about');
});

// 404 page
app.use((req, res) => {
  res.status(404).sendFile('./views/404.html', { root: __dirname });
});
```

Tutorial 7 – View Engines

- To handle dynamic page content such as blog posts, we can use a View Engine (or Template Engine) that is enabled by Express.
    - View engines allow us to inject dynamic data into them like dynamic variables, loops, and logic.
    - We're going to be using Embedded JavaScript Templating (EJS) as the view engine, but there are many options.
    - To install, simply run npm install ejs.
    - We set the view engine in the app by using app.set('view engine', 'ejs').

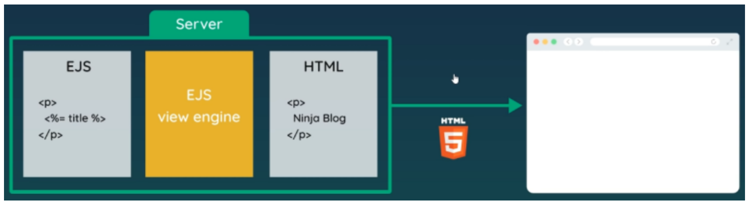


```
const express = require('express');

// express app
const app = express();

// listen for requests
app.listen(3000);

// register view engine
app.set('view engine', 'ejs');
// app.set('views', 'myviews');

app.get('/', (req, res) => {
  const blogs = [
    {title: 'Yoshi finds eggs', snippet: 'Lorem ipsum dolor sit amet consectetur'},
    {title: 'Mario finds stars', snippet: 'Lorem ipsum dolor sit amet consectetur'},
    {title: 'How to defeat bowser', snippet: 'Lorem ipsum dolor sit amet consectetur'},
  ];
  res.render('index', { title: 'Home', blogs });
});

app.get('/about', (req, res) => {
  res.render('about', { title: 'About' });
});

app.get('/blogs/create', (req, res) => {
  res.render('create', { title: 'Create a new blog' });
});

// 404 page
app.use((req, res) => {
  res.status(404).render('404', { title: '404' });
});
```



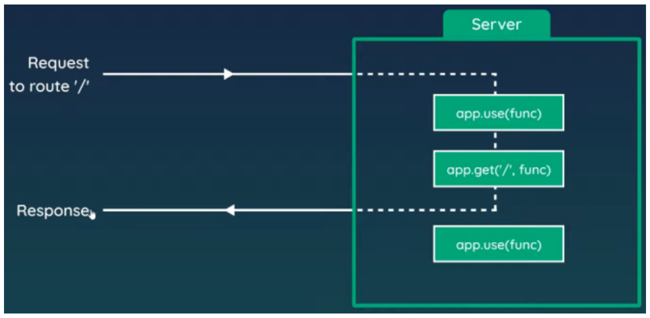
```
<html lang="en">
<%- include("./partials/head.ejs") %>

<body>
  <%- include("./partials/nav.ejs") %>

  <div class="blogs content">
    <h2>All Blogs</h2>

    <% if (blogs.length > 0) { %>
      <% blogs.forEach(blog => { %>

        <h3 class="title"><%= blog.title %></h3>
        <p class="snippet"><%= blog.snippet %></p>

      <% }) %>
    <% } else { %>
      <p>There are no blogs to display...</p>
    <% } %>

  </div>

  <%- include("./partials/footer.ejs") %>
</body>
</html>
```

- By default, EJS looks in the 'views' folder inside the Node.js file structure for EJS files.
  - Instead of HTML files, EJS looks for files with an .ejs extension, which are files that behave much like HTML that can take in the syntax of HTML but also with dynamic content (i.e. similar to jinja for our Databasify Python Flask project).
- Inside of the app.js Node file, we now use res.render(…) instead of res.sendFile(…) inside of our handler functions to render the HTML that is sent to the client.
- EJS tags are now used inside of the .ejs templates for defining the dynamic content, which is wrapped inside of a <% **dynamic content** %> tag.
  - We can now insert JavaScript in between the tags, which runs on the server as opposed to the front-end client.
  - When we are outputting a single value, we have to show an 'equals' sign before the value.
- We can pass data from a handler function into a template by using a second parameter in the render method and defining the data that we want to pass through to the template.
  - The data can then be displayed inside of a dynamic content tag in the template.
  - Note that arrays and other data structures can be passed into dynamic blocks as well, kind of like in Angular but much more verbose.

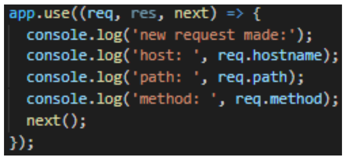


- As a quick summary, EJS templates are processed through the EJS view engine on the server to create static HTML files that are served to the client, which is called server-side rendering.
- EJS supports partials (or Partial Templates), which is basically an inheritance-type structure for EJS templates that allows for code reuse.
  - Partials are used by inserting an include tag <%- include(…) %> inside of the template, which looks for another EJS template at the location provided inside of the function.
  - Note that we use a – sign instead of an = sign inside the tag since = escapes special characters, which leaves us with a string value instead of raw HTML when the template is rendered (we don't want this since we want to serve HTML).
- We can also provide CSS along with our rendered pages by inserting the CSS inside of a top-level partial template (such as head.ejs for the header) within style tags – child partials will then have access to the CSS as well.
- As an aside you can import a font from Google Fonts into your application by going to fonts.google.com and copying the import statement for the font that you want.

Tutorial 8 – Middleware

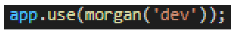


- Middleware is any code which run on the Node.js server between the server receiving a request and sending a response.
  - Note that app.get(…) handlers run for GET requests to a certain route only, while app.use(…) runs for any type of request, including routes and PUSH requests.
  - Middleware runs from top-to-bottom in our code until we either exit the code or return a response to the browser, making the order of the middleware very important.
- Some examples of middleware use include:
  - Logger middleware to log details of every request.
  - Authentication check middleware for protected routes.
  - Middleware to parse JSON data from requests.
  - Return 404 pages.



```
app.use((req, res, next) => {
  console.log('new request made:');
  console.log('host: ', req.hostname);
  console.log('path: ', req.path);
  console.log('method: ', req.method);
  next();
});
```

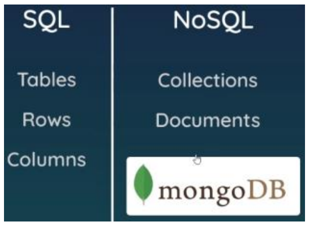
- Note that we have to include the next(); function call if we have app.use(...) middleware above our routes or else the routes will not be returned to the browser because the code will exit.
- A couple of popular middleware modules are morgan, which is an HTTP request logger, and helmet, which is security-focused by setting various HTTP headers for you.
  - There is middleware for sessions, cookies, validation, etc., so it's good to explore different middleware options.



```
app.use(morgan('dev'));
```

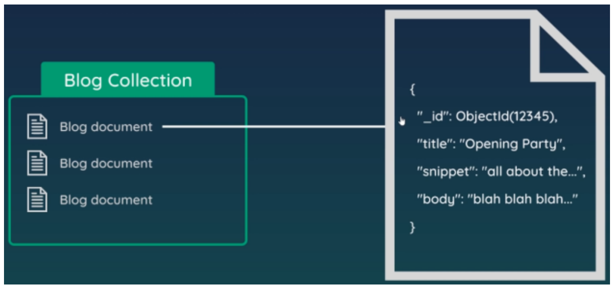
- The server automatically protects all files from users in the browser, so we can't just link to new files (e.g. CSS files) from inside of our templates and expect everything to work.
  - However, if we use some built-in Express middleware, such as express.static(...), we can specify a folder containing static files that we want to publicly make available to the browser – this allows us to expose and reference CSS files in our templates, for instance.

Tutorial 9 – MongoDB



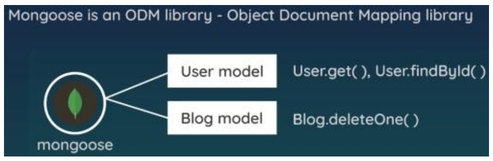
SQL	NoSQL
Tables	Collections
Rows	Documents
Columns	

- Databases are split up into SQL relational databases and NoSQL databases.
- NoSQL databases are split up into collections, which are like tables in SQL databases.
  - Each collection would be used to store different types of data or documents and would only store data or documents of that type.
- A document is like a record in a SQL database where it represents a single item of data.
  - These documents are stored in a format that looks very similar to JSON or JavaScript objects, where the format consists of a series of key-value pairs.
  - A unique ID is typically auto-generated as well.
- From our code, we could connect to a collection within a MongoDB database and save, read, update, or delete documents.
- When using a MongoDB database, we could either setup and connect to a local instance on our machine or connect to one setup in the cloud.
  - MongoDB Atlas is an excellent free cloud-based option.
  - You simply set up a MongoDB database cluster on the Atlas site, get the database URI and insert it into the Node.js application.

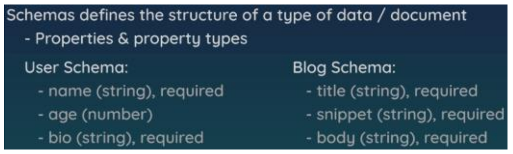


```
{
  "_id": ObjectId(12345),
  "title": "Opening Party",
  "snippet": "all about the...",
  "body": "blah blah blah..."
}
```

- Rather than using the native MongoDB API to make queries to the database, we could use an abstracted connection layer such as Mongoose to make the query work simpler.
  - Mongoose is an Object Document Mapping Library (ODM Library), which seems to be the NoSQL equivalent of an Object Relational Mapping Library (ORM Library).
  - Mongoose works by allowing us to create simple data models which can use database query methods to create, read, delete, and update database documents.
  - It queries the correct database collection for us based on the name of the model used, then it performs the required action and returns the response.


Mongoose is an ODM library - Object Document Mapping library
User model
User.get( ), User.findById( )
Blog model
Blog.deleteOne( )
mongoose

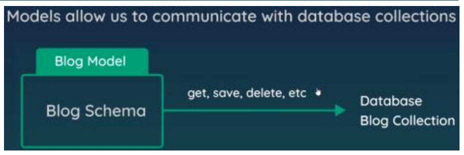
- When working with MongoDB and Mongoose, we need to create schemas and models.
  - Models will represent a data resource, and in this case, a document within a collection inside of a MongoDB database.
  - A schema defines the structure of a datatype or document stored in a collection – it defines the properties it should have, the type of properties, whether they are required, etc.
  - A model allows us to communicate with a database collection through either static or instance methods to read, save, delete, update, etc.



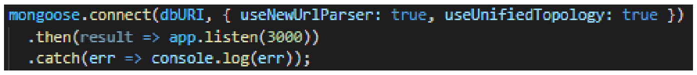
Schemas defines the structure of a type of data / document
- Properties & property types

User Schema:
- name (string), required
- age (number)
- bio (string), required

Blog Schema:
- title (string), required
- snippet (string), required
- body (string), required

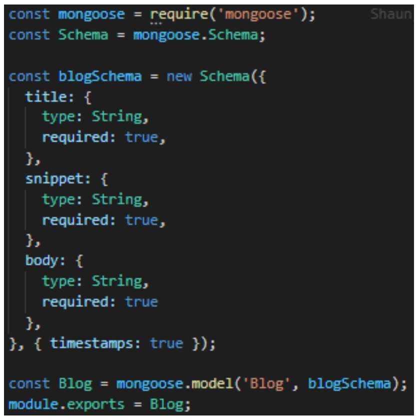


- Mongoose must first be installed using npm install mongoose, and then imported into Node.js through a require(...) function.
    - Then, a mongoose.connect(...) function can be used with the database URI and some additional options to connect with the MongoDB – the additional options being added to stop the deprecation warnings we would otherwise get.



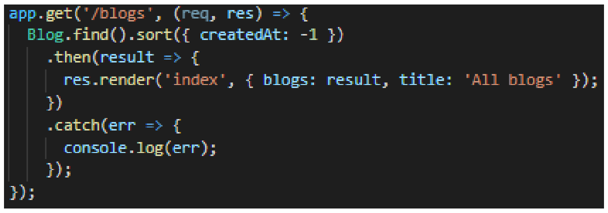
```
mongoose.connect(dbURI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(result => app.listen(3000))
  .catch(err => console.log(err));
```

    - This function is asynchronous and returns something like a promise, so we add a then(...) and catch(...) methods inside to handle the return.
    - Since we don't want the server listening for requests until a connection has been made, we move the app.listen(...) function inside of the asynchronous then(...) function.
- Models are typically created in a 'models' folder as JavaScript files.
    - Keep in mind that a schema is a thing that the model wraps around, while the model provides us with the interface through which we can communicate with the database collection for that document type.
    - Inside the model file, we create a schema object that includes all the properties that represent the data resource that we will be using.
    - There are also some properties that can be auto generated, like timestamps properties and IDs.



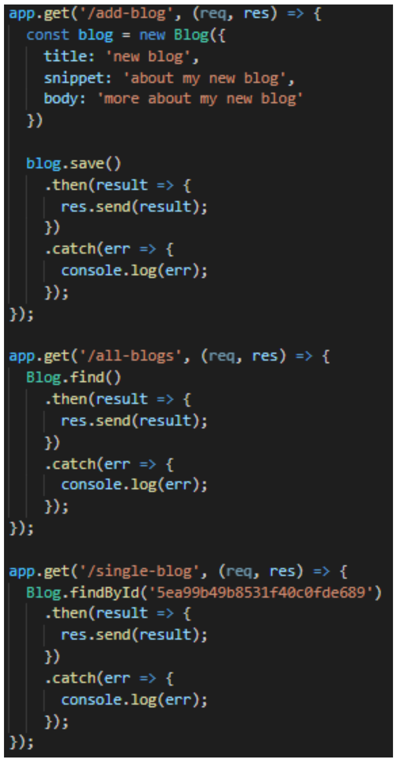
```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const blogSchema = new Schema({
  title: {
    type: String,
    required: true,
  },
  snippet: {
    type: String,
    required: true,
  },
  body: {
    type: String,
    required: true
  },
}, { timestamps: true });

const Blog = mongoose.model('Blog', blogSchema);
module.exports = Blog;
```

- Typically, the name of the model is provided with a capital letter and in singular format, and the model is returned by using mongoose.model(...).
    - The name provided to the model is important since it will be pluralized and used to attach to the correct collection within the database whenever we use the model.
    - The schema that we created is also added into the mongoose.model(...) function as an argument.
- Finally, we also must export the model so that it can be used elsewhere in the application.
- When we are ready to save data into the database, we can create a new instance of the model with the required data, and then use blog.save(), for instance, and handle the asynchronous promise with then(...) and catch(...).
    - Retrieving all the documents from a collection can be done by using the .find() asynchronous method.
    - Note that when we save a model, we work with an instance of the model, while when we retrieve the contents of a collection, we work with the model object itself.



```
app.get('/blogs', (req, res) => {
  Blog.find().sort({ createdAt: -1 })
    .then(result => {
      res.render('index', { blogs: result, title: 'All blogs' });
    })
    .catch(err => {
      console.log(err);
    });
});
```

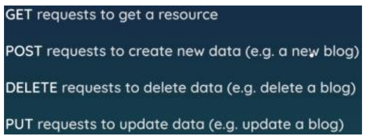
    - We can also retrieve single entries from the database by querying by ID.
- In practice, though, we want to get the data inside of one of our route handlers, and then pass an object containing an array of results to the template for display.



```
app.get('/add-blog', (req, res) => {
  const blog = new Blog({
    title: 'new blog',
    snippet: 'about my new blog',
    body: 'more about my new blog'
  })

  blog.save()
    .then(result => {
      res.send(result);
    })
    .catch(err => {
      console.log(err);
    });
});

app.get('/all-blogs', (req, res) => {
  Blog.find()
    .then(result => {
      res.send(result);
    })
    .catch(err => {
      console.log(err);
    });
});

app.get('/single-blog', (req, res) => {
  Blog.findById('5ea99b49b8531f40c0fde689')
    .then(result => {
      res.send(result);
    })
    .catch(err => {
      console.log(err);
    });
});
```

Tutorial 10 – Get, Post, and Delete Requests

- Besides the GET requests that we've seen so far, there are also POST, DELETE, and PUT requests.
    - We can reuse routes for different requests and create new routes to handle different types of requests.



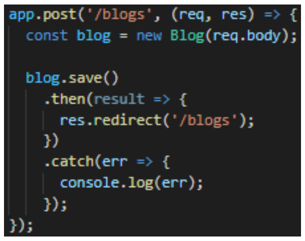
GET requests to get a resource

POST requests to create new data (e.g. a new blog)

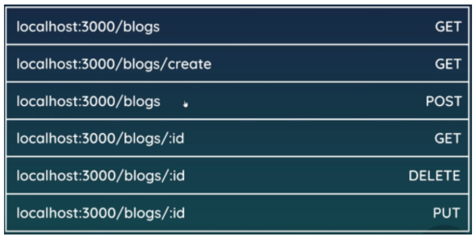
DELETE requests to delete data (e.g. delete a blog)

PUT requests to update data (e.g. update a blog)

- When performing a POST request, we can use some middleware such as express.urlencoded(…) to retrieve data attached to a form to send along with our request.

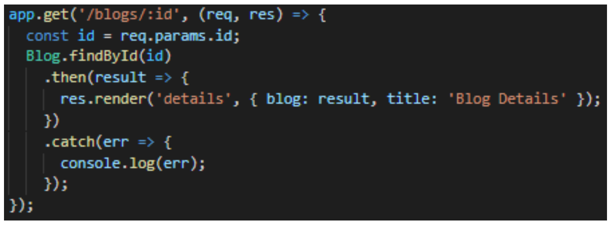


```
app.post('/blogs', (req, res) => {
  const blog = new Blog(req.body);

  blog.save()
    .then(result => {
      res.redirect('/blogs');
    })
    .catch(err => {
      console.log(err);
    });
});
```



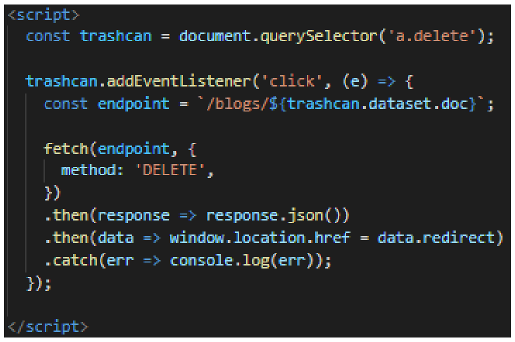
localhost:3000/blogs	GET
localhost:3000/blogs/create	GET
localhost:3000/blogs	POST
localhost:3000/blogs/:id	GET
localhost:3000/blogs/:id	DELETE
localhost:3000/blogs/:id	PUT

- Route parameters need to be used for getting a single document from the database or deleting a single document in a collection.
    - Route parameters are the part of the route that may change, which in the case of our example is the blog ID.
    - The route parameter for the ID, in this case, can be extracted from the request object itself and used to query the database for a single document.



```
app.get('/blogs/:id', (req, res) => {
  const id = req.params.id;
  Blog.findById(id)
    .then(result => {
      res.render('details', { blog: result, title: 'Blog Details' });
    })
    .catch(err => {
      console.log(err);
    });
});
```

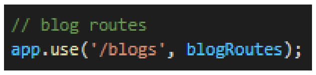
- The same idea can be applied to deleting documents from the database, where Blog.findByIdAndDelete(…) would be used instead of just Blog.findByID(…).
    - The major difference between the two is that for the DELETE request, a front-end JavaScript script must be written using the fetch(…) API to send the delete request to the server.
    - Another item that we must do is to send back a JSON object with the redirect route to the browser so that the front-end JavaScript code can force the redirect (i.e. the backend can't do this for the DELETE method).



```
<script>
  const trashcan = document.querySelector('a.delete');

  trashcan.addEventListener('click', (e) => {
    const endpoint = `/blogs/${trashcan.dataset.doc}`;

    fetch(endpoint, {
      method: 'DELETE',
    })
    .then(response => response.json())
    .then(data => window.location.href = data.redirect)
    .catch(err => console.log(err));
  });

</script>
```

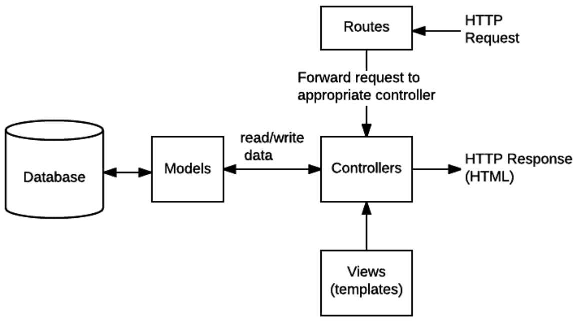
Tutorial 11 – Express Router & MVC

- The Express Router comes fully baked into Express and allows us to handle our routes more efficiently.
    - It allows us to split our routes into multiple files so that we can better manage them.
- To use the router, we can create new route files inside of a 'routes' folder in the project.
    - A router object is then created using express.Router(), with the router object replacing the app object that we previously were using.
    - The router code is then used inside of the app by exporting the router from the route file.
- Back in the main app file, we simply require(…) the route file and have app.use(…) included inside of the body of the file to insert the routes from the route file to that location in the file.



```
// blog routes
app.use('/blogs', blogRoutes);
```

    - Keep in mind that we can adjust the route scoping within the route files, since the app.use(…) provides scoping for the file already – this actually increases maintainability since routes in route files are relative to their app.use(…) paths.
- The app can also be broken out into a Model – View – Controller (MVC) design pattern.
    - This allows for better structuring of our files, making them more modular, reusable, and easier to read.
    - We have already been using the view (templates) and the models, so all that we need now is the controller.
- A 'controllers' folder is created within the project directory to hold our controller file.
    - The route handler functions are extracted from the app file and placed into standalone functions inside of the controller.
    - All of the controller functions must then be exported from the controller file so that they can be imported by the routes file and included inside the route handlers.
    - The result is a much cleaner routes file, and a far more modular and reusable application.



- The route file matches incoming requests and passes those requests to the correct controller function.
    - The controller communicates with the appropriate model to get data into the view.
    - The view then renders that data from its template, which then gets sent to the browser.