# Node JS Essentials

# Data Modelling

The first most important thing to remember is that we should structure data to match the ways the application queries and updates data. The second most important thing to remember is that we should never allow arrays to grow to infinity. There are 3 main types of relationships between datasets (collections).

* One to one
* One to [many]
* One to few: embed / child reference
* One to many: more read: embed / more write: parent reference
* One to ton: parent reference
* Many to many: two-way reference

## Referencing/Normalizing

The two related datasets should be kept separated, but the two datasets are connected through reference IDs. Then when a query happens for the main dataset, we [populate](#_Populating_(child_referencing)) the related dataset into the main dataset if necessary. Refer to Mongoose section for syntax.

### Child referencing

We use the children document IDs as references in an array in the parent document. This tightly couples parents and children, which is not always ideal.

### Parent referencing

We use the parent document ID as a reference in each child document.

### Two-way referencing

In each parent document, we keep ID references to children, and in each child document, we keep ID references to parents.

## Embedding/De-normalizing

The two related datasets are not kept separated, but one (child, related) is embedded into the other (parent, main). This way we will not be able to query for the embedded dataset separately. It is only accessible when querying for the main dataset.

## Making decisions

1. Type of relationship between datasets
2. Data access pattern of the related dataset (read/write)
3. Data closeness

# Node Modules

## 01 – HTTP

To create web servers capable of accepting requests and sending back responses. Core module, no NPM installation. Only require:

const http = require('http');

In order to create server we need 2 steps:

### Create server

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

receives a callback function with access to request and response objects. This will be called each time a new request hits server. In this implementation, we pass in the callback function into the createServer function. However, the callback function can be implemented based on certain events emitted by the HTTP module, like ‘request’:

server.on("request", (*req*, *res*) => {});

here is a list of events emitted by the HTTP module:

* Request
* Close
* List may be completed based on documentation…

Now we have methods on response object:

#### Res.end

This is used to send back a plain text response, including JSON. The ‘end’ method is actually used to determine the response body sent back to the client. The body usually contains the HTML of the website, or JSON data coming from the API, either of which should be determined in the ‘writeHead’ method beforehand.

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

  res.end("<string/json-file>");

});

**NOTE | each request can be responded only once with the ‘end’ method. Multiple listeners can be implemented with any action except sending the response, which actually ends the request-response cycle.**

#### Res.cookie

This method is used to send back a cookie as the response. The method accepts first a name for the cookie, then the data that will be sent with the cookie, which is usually a JWT, and finally an object of options for the cookie. This options object is usually defined in a separate variable and then inserted into the cookie method. Here is a couple of options that can be declared:

* Expiration limit (expires): usually determined as an environment variable in the configuration file
* Security (secure): if set to true, cookie will only be sent on a HTTPS encrypted connection. This would not work in development phase, because normally we are not using HTTPS at this stage. We should activate this option only in the production phase.
* HTTP-only cookie (httpOnly): if set to true, cookie cannot be accessed or modified in anyway by the browser. This can protect against [cross-site scripting attacks](#_Cross-site_scripting_attacks). This also makes it so that the browser will store the cookie in it, and send it back to the server along with each request.

res.cookie('<cookie-name>', <data>, {

    expires: <time-value>,

secure: true,

httpOnly: true,

  });

**NOTE | a name is the unique identifier for a cookie. If the client receives a new cookie with the same name, the old cookie will be replaced by the new one.**

#### Res.writeHead

The response object also has access to ‘writeHead’ method which is used to send back headers in the response.

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

res.writeHead(<status-code>, {header-object});

res.end("<text>");

});

Any headers can be sent into the header object. There are also standard headers like ‘Content-type’ which is used to make the browser expect a certain kind of response in the ‘end’ method.

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

res.writeHead(<status-code>, {

      "Content-type": "text/html",

   });

res.end("<html-code>");

});

**NOTE | headers should always be defined before sending the response.**

#### Res.write

Combined with readable streams of the FS module (createReadStream) and its ‘data’ event, will write data to the response object. Response object is implemented, by default, as a writable stream in FS module. this method accepts the data that should be written to the response object.

server.on("request", (*req*, *res*) => {

  const readable = fs.createReadStream("<file-path>");

  readable.on("data", (*chunk*) => {

    res.write(chunk); //NOT RECOMMENDED

  });

});

**NOTE | This method of the HTTP modules functions much slower than the createReadStream of the FS module. writing with ‘**[**pipe**](#_Readable_stream)**’ method of FS is recommended.**

#### Res.statusCode

Used to set the status code of the response. Here is a list of status codes with their meanings:

* 200: OK
* 201: Created
* 204: No content (deleted)
* 400: Bad request
* 401: Unauthorized
* 403: Forbidden
* 404: Not found
* 429: Too Many Requests
* 500: Internal server error
* May be completed according to documentation…

1. **Start server**: listen to incoming requests

server.listen(<port-number>, "<host-address>", () => {});

receives a callback function that will be called once the server starts listening. To run the script file (e.g. ‘index.js’) containing the above code:

node index.js

then as we enter ‘<host-address>:<port-number>’ in the browser, we receive the result of the callback function defined in the createServer method.

**NOTE | parameters of the command entered into the command line are accessible through the ’argv’ property of the ‘process’ variable. For example, with this command in the command line:**

node index.js --import

**the ‘process.argv’ will display an array containing related to each of the 3 parts of the command; node, index.js, and the import flag. It is possible to call functions in the script file based on the value of the flag.**

if (process.argv[2] === '--<value1>') {

  <function1>();

} else if (process.argv[2] === '--<value2>’) {

  <function2>();

}

## 02 – Express

A very popular Node JS framework, providing a higher level of abstraction over Node JS code. Requires NPM installation:

npm install express@4

It is a convention to implement all the code related to Express in a separate JavaScript file called ‘app.js’, and then include all the codes related to the server in a ‘server.js’ file.

Requiring the installed Express package:

const express = require('express');

Now ‘express’ is a function which upon calling, adds a bunch of methods to the variable into which it is stored. This variable can act as a router.

const app = express();

### Routing

To define routes in an Express application, follow RESTful APIs guidelines. Each routing implementation with its route handler is considered a middleware. Ideally, routers and their routes should be placed in a separate router file. Each resource should have its separate router (sub-application). Rout handlers in these routers would then refer to functions imported from separate Controller files.

#### Versioning routes

Usually implement API functionalities in a base route with a version. For example:

/api/v1/...

#### Acquiring URL parameters (req.params)

To define routes for HTTP requests from which parameters with their values can be derived:

/api/v1/tours/:<parameter>

For example:

<router>.get('/api/v1/tours/:<parameter>', (*req*, *res*) => {

const <variable> = req.params.<parameter>;

});

The value of the parameters defined in the route are now available in the ‘params’ property of the request object. The ‘params’ property is an object itself. Remember that this has nothing to do with the Params middleware.

**NOTE | The client is obligated to provide the value for the parameters expected in the URL. To determine optional parameters in the route we put a question mark after the parameter in the route.**

/api/v1/tours/:id?

**Client is now allowed to leave the parameter value empty in the URL, and the parameter will be returned as undefined, leaving no error.**

#### Acquiring query string

When the client queries for certain documents using a query string in the URL, the parameters of the query string will be accessible through ‘req.query’. For example, if there is a query string in the URL:

127.0.0.1:3000/api/v1/tours?duration=5&difficulty=easy

through ‘req.query’ we receive an object containing the query parameters:

{ duration: '5', difficulty: 'easy' }

This query object can be used as the filter object and passed into the find method in order to find certain documents matching the query.

const tours = await Tour.find(req.query);

**NOTE | it is very important to understand the difference between query parameters and URL parameters. With URL parameters, we predefine a specific route to accept a value for a specific parameter, like ID. With query parameters, we don’t define a specific route. The route that is usually used for queries is the route that is normally responded with all the documents of a collection.**

#### Nested routes & mergeParams

Nested routes are implemented in case one resource belongs to another. In this situation, there would be a special route in the main resource router that would lead the application toward the router of the related resource. So the router of the related resource is now used as a middleware in the main resource router. So in the router of the main resource:

<main-router>.use('/:<main-doc-id>/<related-resource-name>', <related-router>);

To provide access for the related router to acquire ID parameter from the URL, we activate the mergeParams option on the related router declaration in its file:

const router = express.Router({ mergeParams: true });

Now handler functions in the related resource controller file will have access to the ID parameter in the URL through ‘req.params’.

### Methods available on app (=express())

#### App.listen

To start a server. The method accepts a port number, and a callback function that will be called once the server starts listening.

app.listen(<port-number>, () => {});

#### App.<HTTP-method>

To define routes. This determines how the application will respond to a certain request (URL + HTTP method). Popular HTTP methods include ‘get’, ‘post’, ‘patch’, and ‘delete’. Each method accepts a route string, and a callback function, also called route handler, that has access to the request and response objects. It is called once the specified route with the specified method is hit. The difference with these request and response objects is that they have a lot more methods available on them.

app.get('<route>', (*req*, *res*) => {});

* **Res.status**: to determine the status code of the response.

app.get('<route>’, (*req*, *res*) => {

  res.status(200).json({

    status: 'success',

    data,

  });

});

* **Res.send**: to send back a simple string response.
* **Res.json**: to send back a JSON response. accepts a regular JS object containing, ‘status’, ‘message’, ‘data’ (envelop), and any custom fields. This object is then converted to JSON automatically. Using the json method automatically sets the ‘Content-type’ property of the header to ‘application/json’.
* **Req.params**: to access parameters defined in the URL. Returns an object containing URL parameters with their values in key/value pairs.
* **Req.header**: to access the request headers sent by the client.
* **Req.originalUrl**: holds the URL string without the host path.

**NOTE | route handlers should usually be defined as separate functions in a Controller file, from where they should be exported, and then imported to be used as route handlers when defining routes.**

#### App.all

The ‘all’ method stands for all types of HTTP request. This method is usually used to specify a route handler for any uncaught route. Therefore, it usually sends back a fail response. Such a route handler middleware should obviously be placed after all routers in the Express application. The ‘all’ method, similar to any HTTP method, accepts first the route, which will be any route here (\*), and then the callback function that has access to request and response objects, along with the next function, although this next function will have no use since this would most probably be the final middleware in the whole application’s stack.

app.all('\*', (*req*, *res*, *next*) => {

  res.status(404).json({

    status: 'fail',

    message: `Cannot find ${req.originalUrl} on this server.`,

  });

});

#### App.route.<HTTP-method>

To chain multiple HTTP methods on the same route. Route handlers should usually be defined as separate functions in a Controller JS file.

app.route('<route>').<HTTP-method1>(<route-handler1>, <route-handler2>).<HTTP-method2>(<route-handler1>, <route-handler3>);

**NOTE | don’t forget to call the next function in middleware that should lead to the execution of another middleware in the stack.**

#### App.use (global middleware)

To implement any custom middleware. Each middleware, between receiving the request and sending the response, has access to the request and response objects, along with a ‘next’ function which should necessarily be called in each middleware to terminate it and lead the request-response cycle toward the next middleware. The last middleware in a middleware stack is usually a route handler, where the next function is not called anymore, but the final response is sent to the client.

To create a custom middleware we call the ‘use’ method on the app. The ‘use’ method accepts first a route, and then the handler function. However, the route can simply be ignored, and only the handler function can be passed into the method. This will make the handler function to act as a global middleware in the sequence in which it was declared.

app.use('<route1>', <route-handler>);

app.use((*req*, *res*, *next*) => {})

This kind of middleware (global), based on its location in the code, can react to any request that hits the server. On the other hand, route handler middleware will only react to request hitting a certain route. Global middleware are usually declared before other middleware.

**NOTE | into the ‘use’ method, we only pass a function. We never call a function in this method. Even if we call a function here, that function would then have to return another function that would sit in the ‘use’ method waiting to be called in a future time. For instance, this is what the** [**helmet package**](#_Cross-site_scripting_attacks) **does.**

### Defining separate routers

const <router-name> = express.Router();

We then have methods available on the router:

#### Router.route.<HTTP-method>

to define routes and to determine the type of HTTP request and the route handler function.

<router>.route('<route1>/<route2>').<HTTP-method1>(<route-handler1>, <route-handler2>).<HTTP-method2>(<route-handler1>, <route-handler3>)

This separate router can be used as middleware by the express application (app).

app.use('<route1>', <router>);

This will redirect the application to the specified router for the specific route, and from there, the separate router will take care of requests. Separate routers would act as sub-applications of the main application or main router (app). Note that the root URL for sub-applications and the root URL for the main application are all defined with ‘/’ but they don’t point to the same path actually.

**CODE STRUCTURE-ROUTER:** In a separate router (sub-application) file we have in order:

1. Express required (along with other necessary requirements)
2. New router declared
3. Routes declared
4. Router exported

#### Router.use

Each and every router has the ‘use’ method available on them, which makes it possible to implement a middleware with that router. The ‘use’ method can accept first a route, and then a route handler which has access to the request and response objects, along with the next function. However, we can simply ignore the route, and only pass in a route handler function into the ‘use’ method. This will make it so that this route handler would act as a global middleware for that specific router, and in sequence, it will run before all routes and route handlers declared after it in the code.

router.use((*req*, *res*, *next*) => {});

### Aliasing

It involves defining a special route for a certain query that tends to be requested very often by the users. For example:

127.0.0.1:3000/api/v1/tours?limit=5&sort=-ratingAverage,price

Without aliasing, this will simply end up in the regular handler function responsible for filtering documents. But with aliasing, we won’t let this happen at first. Instead, we define a special route and route handler, which will act as a middleware before it reaches our regular handler functions.

router.route('/top-5-cheap').get(tourController. aliasTopTours, tourController.getAllTours);

This handler function will manipulate the query object to include the query string in the URL above, and then lead the request to the regular handler function responsible for filtering documents.

exports.aliasTopTours = (*req*, *res*, *next*) => {

  req.query.limit = '5';

  req.query.sort = '-ratingAverage,price';

  req.query.fields = 'name,price,ratingAverage,summary,difficulty';

  next();

};

With this implementation, whenever the client want this data, they should hit this URL:

127.0.0.1:3000/api/v1/tours/top-5-cheap

### Express middleware

#### Express body parser (express.json)

To make Express put the request body on the request object. Since this should be applied as a middle ware, it should be coded along with the ‘use’ method. This should always be implemented at the top of our Express application (app.js).

app.use(express.json());

as a result, the ‘body’ property will now be accessible on the request object when a ‘post’ request hits a specific route.

app.post('<route>', (*req*, *res*) => {

  console.log(req.body);

});

The amount of data that can be inserted in to the request body can also be limited by setting the ‘limit’ option for this middleware:

app.use(express.json({ limit: '10kb' }));

This is extremely useful to protect against [DOS attacks](#_Denial-of-Service_(DOS)_attacks).

#### Express URL-encoded middleware

This global middleware is used to parse URL-encoded data generated by an HTML form into the request body.

app.use(express.urlencoded({ extended: true, limit: '10kb' }));

The ‘urlencoded’ method receives an object of options, in which the ‘extended’ property should be set to true in case other options, like limit, are going to be defined.

**NOTE | the HTML form that is designed to encode user input data into the URL, should contain input elements with the ‘name’ attribute. The value of these attributes will be used as the keys of key/value pairs that would then be available in the request body after being parsed by this Express middleware.**

**Pug example:**

input#name.form\_\_input(type='text', name='name')

input#email.form\_\_input(type='email', name='email')

#### Express param middleware

This kind of middleware only applies to requests that have certain parameters in their URL. The middleware will have access, in addition to regular arguments of a middleware, to the value of the specified parameter in the URL.

router.param('id', (*req*, *res*, *next*, *val*) => {});

This middleware is usually declared as global middleware, before detailed routes, in sub-applications. The callback function can be declared in a separate Controller file, and then imported into the router. Since it is called by the middleware, it will have access to the middleware’s callback function arguments.

router.param('id', <handler-function>);

#### Express static middleware

It is used to server static files. It can be applied in the Express app file middleware stack, using the join method of Node’s [Path module](#_path.join).

app.set('views', path.join(\_\_dirname, './views'));

The static method accepts a file path that leads to the folder containing static files. That folder will then be considered as root if the client attempts to access its files. For example, if there is an ‘overview.html’ file in the public folder, the client should request it with this URL:

127.0.0.1:3000/overview.html

### Express error handling

In Express, we can define a middleware that will only get called if there is an error happening in the application. Express can identify an error handling middleware with the arguments of handler function. A regular middleware has access to request, response, and next arguments. An error handling middleware, first has access to the error object, and then the same 3 arguments as a regular middleware. An error handling middleware would be implemented as:

app.use((*err*, *req*, *res*, *next*) => {});

#### Global error handler middleware

A global error handling middleware is regularly only responsible for sending back an error response. This global error handling middleware should also account for the environment in which it is functioning. Error responses should look different when they are sent in the development environment compared to when they are sent in the production environment.

app.use((*err*, *req*, *res*, *next*) => {

  err.statusCode = err.statusCode || 500;

  err.status = err.status || 'error';

  if (process.env.NODE\_ENV === 'development') {

    sendErrorDev(err, res);

  } else if (process.env.NODE\_ENV === 'production') {

    sendErrorProd(err, res);

  }

});

Sending the error in each environment can be carried out with two separate functions.

const sendErrorDev = (*err*, *res*) => {

  res.status(err.statusCode).json({

    status: err.status,

    error: err,

    message: err.message,

    stack: err.stack,

  });

};

In development environment, we want to know everything about the error. But in production error, if it is an operational error, we send the error message to the client, and if it is a programming error, we send a generic error message.

const sendErrorProd = (*err*, *res*) => {

  if (err.isOperational) {

    res.status(err.statusCode).json({

      status: err.status,

      message: err.message,

    });

  } else {

    console.error('ERROR LOG!', err);

    res.status(500).json({

      status: 'Error',

      message: 'Something went very wrong!',

    });

  }

};

Now whenever the next function in any middleware is called with anything passed into it, Express will understand that an error has happened, and will run the global error handling middleware with what was passed into the next function, which would be the error object. For instance, the undefined route handler (app.all) is very likely to call the next function with an error object:

app.all('\*', (*req*, *res*, *next*) => {

  const err = new Error(`Can't find ${req.originalUrl} on this server!`);

  err.status = 'fail';

  err.statusCode = 404;

  next(err);

});

#### Extending Node Error class

It is a common practice to implement an error handling class that extends Node’s built-in error class. This custom class may receive the error message and the status code to create an error object.

class AppError extends Error {

  constructor(*message*, *statusCode*) {

    super(message);

    this.statusCode = statusCode;

    this.status = `${statusCode}`.startsWith('4') ? 'Fail' : 'Error';

    this.isOperational = true;

    Error.captureStackTrace(this, this.constructor);

  }

}

Now this global error handling class can be used in any point in the code. For example, again with the undefined route handler (app.all):

app.all('\*', (*req*, *res*, *next*) => {

  next(new AppError(`Can't find ${req.originalUrl} on this server.`, 404));

});

So the AppError class will create the error object, and passing this error object to the next function will lead the application toward the global error handling middleware defined beforehand.

#### Catching errors of asynchronous functions

We usually want to take away the error handling mechanism away from asynchronous functions handling business logics. These functions should do their regular tasks asynchronously, and if there is an error, it should be caught in one central catch method. For this purpose, we use the concept of **wrapper functions**. Here we wrap asynchronous functions into a function that returns another function to which a catch method is attached.

const catchAsync = (*fn*) => {

  return (*req*, *res*, *next*) => {

    fn(req, res, next).catch((*err*) => next(err));

  };

};

Then any asynchronous functionality can be wrapped into this function as:

const createTour = catchAsync(async (*req*, *res*, *next*) => {

const newTour = await Tour.create(req.body);

  res.status(201).json({

    status: 'success',

    data: {

      tour: newTour,

    },

  });

});

## 03 – FS (File System)

To read and write files. Core module, no NPM installation. Only require:

const fs = require("fs");

### Read files

#### Sync version

const textIn = fs.readFileSync("<file-path>", "<character-encoding>");

file path starts with ‘./’. Character encoding is usually ‘utf-8’.

#### Async version

fs.readFile("<file-path>", "<character-encoding>", (*err*, *data*) => {});

Async version receives a callback function with access to data and error. This will be called once the reading process is done.

**NOTE | it is not a good practice to perform a read process for a specific route, each time it is hit. It is strictly advised to perform the reading in top-level code synchronously, and then send back the data each time the route is hit.**

### Write files

#### Sync version

fs.writeFileSync("<output-file-path>", <string variable>);

file path starts with ‘./’. String to be written can be directly passed in or via a variable.

#### Async version

fs.writeFile("<file-path>", <string variable>, "<character-encoding>", (*err*) => {});

Async version receives a callback function with access only to error, not data. This will be called once the writing process is done. NOTE: character encoding may not be necessary here.

**NOTE | characters ‘./’ at the beginning of file paths in FS module refers to where the terminal is running. It is recommended to use ‘\_\_dirname’ which refers to the directory where the script that is currently being executed is located. Don’t use for requiring modules.**

**NOTE | When writing a route to a specific file, if we start with dot, it means that we should address the target file backwards. That is, starting from the file in which we are writing the route, and try to reach to the target file. On the other hand, if we start with ‘\_\_dirname’ we are actually using the absolute path that leads to the current file in which we are writing the route. From there, we can continue trying to reach the target file.**

### Streams

#### Readable stream

To create a readable stream on a request event:

server.on("request", (*req*, *res*) => {

  const readable = fs.createReadStream("<file-path>");

});

The readable stream has access to 3 events (data, end, error) and 2 functions:

* Events:
* Data: this event is emitted on the readable stream once a chunk of data is read and available to be consumed:

server.on("request", (*req*, *res*) => {

  const readable = fs.createReadStream("<file-path>");

  readable.on("data", (*chunk*) => {

    res.write(chunk); //NOT RECOMMENDED

  });

});

* End: this event is emitted when there is no more data to be read.

server.on("request", (*req*, *res*) => {

  const readable = fs.createReadStream("test-file.txt");

  readable.on("end", () => {

    res.end();

  });

});

* Error: this event is emitted when there is error reading the file.

server.on("request", (*req*, *res*) => {

  const readable = fs.createReadStream("testttt-file.txt");

  readable.on("error", (*err*) => {

    res.end("File not found!");

  });

});

* Functions:
* **Pipe**: writes each chunk of data to the response object (writable stream) as soon as read from file.

server.on("request", (*req*, *res*) => {

  const readable = fs.createReadStream("test-file.txt");

  readable.pipe(res); //RECOMMENDED

});

* **Read**

## 04 – URL

To analyze URL (URL parser) Core module, no NPM installation. Only require:

const url = require("url");

this will put the URL on the request object as a request hits the server.

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

  console.log(req.url);

});

The ‘url’ on request object is a string containing the route specified after the host and port number which is called query string. However, in case we want the query string to be parsed in to an object, we can use the ‘parse’ method on the URL module.

### Parsing URL

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

const { query, pathname } = url.parse(req.url, true);

}

the ‘parse’ methods accept the query string, and a Boolean value determining whether the query string should be parsed into an object. This object will contain numerous properties including ‘query’ and ‘pathname’.

Supposing the user has hit a URL like this:

<http://127.0.0.1:8000/product?id=123>

note that ‘req.url’ will be ‘/product?id=0’, and according to the code above:

* Pathname: ‘/product’ as string
* Query: { id: ‘123’ } as object

## 05 – Slugify

To make readable URLs out of names. Needs NPM installation.

npm install slugify

Then should be required:

const slugify = require('slugify');

the ‘slugify’ method can now convert a string into a URL-friendly one:

console.log(slugify(<string>/<string-variable>, { lower: true }));

The method accepts, after the string, an object of options. Setting the ‘lower’ option to true will produce a lower-case slug.

## 06 – Nodemon

To make Node JS applications automatically restart whenever a change happens in JS files. Needs NPM installation.

npm install nodemon --global

When installed globally, we can use it in the command line. So by running our entry file with Nodemon:

nodemon index.js

all the project files will be watched for any changes.

If installed locally as a save-dev:

npm install nodemon --save-dev

we cannot run it from the command line. Instead, we should define a script for it in the package JSON file.

"scripts": {

  "start": "nodemon index.js"

}

And then run in the command line:

npm run start

## 07 – Morgan (middleware)

Allows developers to see request information right in the console. Requires NPM installation.

npm install morgan

It also needs to be required:

const morgan = require('morgan');

to implement as middleware:

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

for each request that hits the server, information will be displayed as:

GET /api/v1/tours 200 6.705 ms – 8681

## 08 – Events

To use the built-in Node events and generate custom events. Core module, no NPM installation. Only require:

const EventEmitter = require("events");

This returns a class, from which an instance can be produced:

const myEmitter = new EventEmitter();

### Emit custom named event

To emit a custom event with code, and pass arguments into the emitter:

myEmitter.emit("<custom-event-name>", <argument>);

### Listen to custom named event

to set up listener for an event and receive arguments:

myEmitter.on("<custom-event-name>", (<argument>) => {});

Multiple listeners can be set up for one single event. Results of multiple listeners appear synchronously in the order of code declaration.

### Extending event emitter

To extend the original event emitter class of Node, in order to add functionalities to the class.

class Sales extends EventEmitter {

  constructor() {

    super();

  }

}

**NOTE** | Refer to OOP in JavaScript notes if necessary.

To create an instance of this new extended class:

const myEmitter = new Sales();

## 09 – Validator

An NPM library for validating data. It requires NPM installation:

npm install validator

and it also needs to be required:

const validator = require('validator');

it can be used in the schema to validate inputted data. For example:

name: {

      type: String,

     validate: [validator.isAlpha, 'Tour name should contain only letters.'],

    },

The ‘isAlpha’ validator checks if the inputted string contains only letters. If so, it returns true. Note that the isAlpha validator will also prevent spaces in the string.

The library also includes some sanitization functions that could be implemented into the mongoose schema in order to protect against [NoSQL query injection attacks](#_NoSQL_query_injection).

Visit the documentation: <https://github.com/validatorjs/validator.js/>

## 10 – Util

This module contains a promisifying method. No NPM installation required, but needs to be required into a file.

const util = require('util');

or

const { promisify } = require('util');

The promisify method returns a promise and should be awaited. in order to promisify a specific function:

await promisify(<function>)(<function-arguments>);

## 11 – Bcrypt

For information on how to use this package refer to [password encryption](#_Bcrypt) in Authentication section.

## 12 – Crypto

This module performs string encryption. No NPM installation required. Needs to be required into a file:

const crypto = require('crypto');

### Producing a random cryptographic string

To produce a random string we first use the ‘randomBytes’ function in which the length of the string should be determined (32), and then we chain the ‘toString’ method determining hexadecimal string.

const <variable1> = crypto.randomBytes(<number>).toString('hex');

### Hash a random string

Note that ‘Hashing’ refers to a type of encryption that will not allow the encrypted data to be decrypted. Comparisons should be done by converting the plaintext to ciphertext.

To encrypt a random string, usually created with the crypto module itself, we use the ‘createHash’ method in which the encryption standard, usually the ‘sha256’ or ‘sha512’, should be determined, and then the ‘update’ method into which the random string is passed, and finally a ‘digest’ method determining hexadecimal string.

const <variable2> = crypto.createHash('sha256').update(<variable1>).digest('hex');

### Symmetric encryption

Asymmetric encryption refers to a type of encryption that allows the encrypted data to be decrypted later. This method uses a key to encrypt data. The same key would then be used to decrypt data.

Three main components of a cryptographic system are:

* Plaintext
* Ciphertext
* Algorithm: ‘aes-256-cbc’ and ‘sha256’ seem to be the most popular ones.

#### Create cipher

In order to use symmetric encryption, we should first create a cipher function. This is done by calling the createCipheriv on the crypto module. The method accepts fist an algorithm, then a security key, and finally an initialization vector.

const cipher = crypto.createCipheriv(<algorithm>, <security-key>, <init-vector>);

#### Encrypt data

We would then use the cipher function, and call the update method on it. The update method receives first the data that should be encrypted, then the input encoding (mostly utf-8), and then the output encoding (mostly hex).

let encryptedData = cipher.update(data, 'utf-8', 'hex');

Note that the encryptedData variable is declared using ‘let’ because we still need to call another method on the cipher function in order to complete the encryption process, and that is the ‘final’ method, which again receives the output encoding.

encryptedData += cipher.final('hex');

### Symmetric decryption

To decrypt a symmetrically encrypted data, we should start by creating a decipher function.

#### Create decipher

To create a decipher function, we use the createDecipheriv method on the crypto module. The method receives exactly what the createCipheriv method received.

const decipher = crypto.createDecipheriv(<algorithm>, <security-key>, <init-vector>);

#### Decrypt data

We would then use the decipher function, and call the update method on it. The update method receives first the encrypted data that should be decrypted. Next, it receives the input encoding, which is the output encoding of the encryption stage (hex). Then it receives the output encoding which at this stage will be utf-8.

let decryptedData = decipher.update(encrypteddata, 'hex', 'utf-8');

again the variable is declared with ‘let’ since we still need to complete the decryption process by calling the final method on the decipher function. The final method will again receive the output encoding which will be utf-8, but strangely, we have to write it as utf8.

decryptedData += decipher.final('utf8')

## 13 – Nodemailer

Requires NPM installation:

npm install nodemailer

Needs to be required into a file:

const nodemailer = require('nodemailer');

In order implement an email sending functionality, we should declare a function with a couple of email options that will be passed into the actual email sending method.

const sendEmail = (*options*) => {};

The actual method that will be used to send an email is the ‘sendMail’ method that should be called on a transporter. This method receives an object of options about the email itself, and performs an asynchronous task. So the method should be awaited.

### Setting up a transporter

This is the service which takes care of actually sending emails. It can be Gmail or any other service that Nodemailer can work with. Some of the well-known services that can be used are SendGrid and Mailgun.

In order to define a transporter, we use the ‘createTransport’ method on the nodemailer object. This method receives an object including the ‘service’ property containing the name of the service that is going to be used, and the ‘auth’ property containing an object in which the ‘user’ and ‘password’ properties should be defined as they are stored in the configuration file.

const sendEmail = (*options*) => {

  const transporter = nodemailer.createTransport({

    service: 'Gmail',

    auth: {

      user: process.env.EMAIL\_USERNAME,

      password: process.env.EMAIL\_PASSWORD,

    },

  });

};

We can also use a development email service called Mailtrap. For this the object passed into the createTransport method would be:

const sendEmail = (*options*) => {

  const transporter = nodemailer.createTransport({

    host: process.env.EMAIL\_HOST,

    port: process.env.EMAIL\_PORT,

    auth: {

      user: process.env.EMAIL\_USERNAME,

      password: process.env.EMAIL\_PASSWORD,

    },

  });

};

### Setting up mail options

This should be regular JavaScript object containing the mail options. This object would then be passed into the ‘sendMail’ method called on the transporter. Here is a list of options that can be declared in this object:

* From: the email address from which the email is going to be sent. This should be defined as a string:

Omid Armat <omidarmat@gmail.com>

* To: the email address to which the email is going to be sent.
* Subject
* Text
* Html: used to convert the content of the ‘text’ property to HTML.

### Sending email

This sendEmail function can then be used across the whole application.

const sendEmail = async (*options*) => {

  // 1) create a transporter

  const transporter = nodemailer.createTransport({

    host: process.env.EMAIL\_HOST,

    port: process.env.EMAIL\_PORT,

    auth: {

      user: process.env.EMAIL\_USERNAME,

      password: process.env.EMAIL\_PASSWORD,

    },

  });

  const mailOptions = {

    from: 'Omid Armat <omidarmat@gmail.com>',

    to: options.email,

    subject: options.subject,

    text: options.message,

  };

  await transporter.sendMail(mailOptions);

};

## 14 – Path

Used to manipulate path names. No NPM installation, but needs to be required usually at the very beginning of the App file.

const path = require('path');

### path.join

The join method is used to join two paths together as a string, considering necessary slashes that should automatically be inserted. For instance it can be used for the [Express static middleware](#_Express_static_middleware), which is in direct connection with the [Pug template engine](#_Server-side_rendering_with) setup.

## 15 – Cookie parser

Used to parse cookies sent with requests. This will provide developers with access to cookies sent with requests through the ‘cookies’ property on the request object. Needs NPM installation.

npm install cookie-parser

Needs to be required into an App file:

const cookieParser = require('cookie-parser');

and the cookieParser should now be called in a global middleware in order to make the ‘cookies’ property available on the request object. This middleware is usually placed after the [Express body parser](#_Express_body_parser) middleware.

app.use(cookieParser());

As a result, for instance, if a user logs in and the server signs a token called ‘jwt’ and sends it to the client, this specific token would be sent back to the server with each request, and it would be accessible on:

req.cookie.jwt

# MongoDB

MongoDB is a NoSQL, document-based database popularly used with Node JS. Each database can contain multiple collections, and each collection can contain multiple documents.

## MongoDB Shell

The following commands can be used in Windows Power Shell to execute certain tasks.

### Creating a new database or switch to an existing one

As Mongo Shell opens, it usually opens the ‘test’ database by default. To create a new database or to switch to an already existing database:

use <database-name>

### Searching for all databases

Using this command, all databases created on MongoDB will be returned.

show dbs

### Creating a new collection and documents

Creating a collection can only be done with creating one or more documents at the same time. Create one single document using the insertOne method. This method accepts a regular JavaScript object, and it will convert it automatically to BSON.

db.<collection>.insertOne(

{

<key>: <value>,

...

}

)

Create multiple documents using the insertMany method. This method accepts multiple objects as documents.

### Searching for documents inside a collection

Using the ‘find’ method on the collection, can return with documents matching the query. If no query, all documents of that collection will be returned. Documents returned by this method are in JSON format.

db.<collection>.find()

### Searching for all collections in a database

Using this command, all collections created in a specific database will be returned.

show collections

### Quit MongoDB Shell

quit()

## MongoDB CRUD operations

CRUD operations include Creating, Reading, Updating, and Deleting documents in MongoDB.

### Create documents

* Db.collection.insertOne
* Db.collection.insertMany

### Reading documents

#### Finding documents matching a specific query

The find method can be used with no query, which will return all the documents in a certain collection. By passing a query object into the find method, we can search for documents that match the query.

db.<collection>.find({ <key>: <value> })

#### Finding documents matching a query range

By using MongoDB query operators, we can search for documents that matches a query range.

* $lte (less than or equal to)

db.<collection>.find({ <key>: {$lte: <number>} })

* $lt (less than)

db.<collection>.find({ <key>: {$lt: <number>} })

* $gte (greater than or equal to)
* $gt (greater than)
* Geospatial operators: used to find geospatial data matching a geospatial query. There are multiple geospatial operators, including:
* $geoWithin: accepts an object in which a geometry operator should be defined, representing the query area. In case we want to find geospatial data within a sphere with a defined center, we use ‘centerSphere’. This geometry operator receives an array that accepts as its first element an array of longitude and latitude, and then a radius value in Radians.

db.<collection>.find({ <key>: { $geoWithin: {$centerSphere: [[lng, lat], radius]} } });

**NOTE | Radian is the result of dividing a distance by earth’s radius. Earth’s radius – in miles: 3963.2 – in km: 6378.1**

**NOTE | to make geospatial queries more performant, it is recommended to implement** [**geospatial indexes**](#_Geospatial_index)**.**

#### And query

With ‘and’ query, query objects are inserted directly into the find method, while being separated with commas.

db.<collection>.find({ <key1>: {$lt: <number>}, <key2>: {$gte: <number>} })

#### Or query ($or)

With ‘or’ query, query objects are inserted in an array assigned to the ‘or’ operator. These objects are separated with commas.

db.<collection>.find({ $or: [ {<key1>: {$lt: <number>}}, {<key2>: {$gte: <number>}} ] })

### Updating documents

#### Partially update documents

Usually used to update parts of one or more documents.

* **Db.collection.updateOne**

The method accepts first the query with which it will find the document that is going to be updated. If the query matches multiple documents, only the first one will be updated. It will then accept an object containing the target key/value pair that is going to be updated in the found document using the $set operator. This $set operator will update an existing key/value pair, or it will create it if doesn’t exist.

db.<collection>.updateOne({ <key1>: <value1> }, { $set: {<key2>: <value2>} })

* **Db.collection.updateMany**

This method accepts first the query with which it will find documents that are going to be updated. It will then accept an object containing the target key/value pair that is going to be updated in the found documents using the $set operator. This $set operator will update an existing key/value pair, or it will create it if doesn’t exist.

db.<collection>.updateMany({ <key1>: {$gt: <number>}, <key2>: {$gte: <number>} }, { $set: {<key3>: <value>} })

#### Completely update (or replace) documents

Used to completely replace the content of a document.

* **Db.collection.replaceOne**
* **Db.collection.replaceMany**

### Deleting documents

The two methods accept a query object. If multiple documents match the query object of deleteOne, the first match will be deleted. If the query object of deleteMany is left empty, all documents in the collection will be deleted.

* **Db.collection.deleteOne**
* **Db.collection.deleteMany**

## MongoDB connection error

This error might happen because of wrong database credentials, or a connection interruption. This will cause an unhandled promise rejection that should get caught in a safety net implementation. This error will cause the whole application to fail. So what should usually be done along with handling this error, is to first close the server using the ‘close’ method on ‘server’, and then shut down the application using the ‘exit’ method on the ‘process’ object. The exit method accepts 0 as ‘success’, and 1 as uncaught exception.

process.on('unhandledRejection', (*err*) => {

  server.close(() => {

    process.exit(1);

  });

});

After the hosting platform or an implemented tool should restart the application.

# Mongoose

Mongoose is a software, also known as a driver, allowing Node applications to interact with MongoDB database. It is an Object Data Modelling (ODM) library that provides a higher level of abstraction over MongoDB. This makes a lot more functionality accessible to the developer, allowing for simpler and faster implementation of applications.

## MongoDB connection strings (hosted & local)

Connection is established using a connection string provided by Atlas, in case of hosted database. A typical connection string looks like and is usually stored in an environment variable:

DATABASE=mongodb+srv://<username>:<PASSWORD>@<cluster-name>.oijk0uw.mongodb.net/natours?retryWrites=true&w=majority

the password placeholder should usually be replaced by the value of another environment variable:

DATABASE\_PASSWORD=<password>

In case of local database, connection string would be:

DATABASE\_LOCAL=mongodb://localhost:27017/<database-name>

Number 27017 is derived from MongoDB server.

## Connecting MongoDB to Express application

Mongoose requires a NPM package installation.

npm install mongoose@5

it needs requiring in the server file:

const mongoose = require('mongoose');

all the configurations of MongoDB and Mongoose will be implemented in the ‘server.js’ file.

On the mongoose variable, the ‘connect’ method is accessible. The method accepts the connection string, along with an object of options.

mongoose.connect('<connection-string>', {

  useNewUrlParser: true,

  useCreateIndex: true,

  useFindAndModify: false,

});

Options used here deal with some deprecation warnings and should be followed as an instruction.

The connect method returns a promise, and can be consumed using a ‘then’ method, which will have access to the ‘connection’ object. Beware of errors returned from this method. They should probably be caught using a chained ‘catch’ method.

In case of hosted database, the connection string should contain the password. The complete connection string can be calculated using the environment variables defined before, and then stored into a variable.

## Schema

Used to model and describe the structure of data. A model is created out of a schema which will act as a database collection when programming.

### Creating a schema

const <schema-name> = new mongoose.Schema({<schema>}, {<options>});

the ‘Schema’ method accepts first the schema object, and then an object of options. In the schema object, fields should be defined by their ‘type’, and other necessary properties. ‘type’ determines the type of the value that can be passed into that specific field. Here is a list of data types:

* String or [String]: note that [String] won’t work if you are going to embed other related documents into a main document based on the elements of this array, through a document pre-hook. In this case you should use the Array type.
* Number or [Number]
* Boolean
* Date or [Date]
* Array (allows to receive an array of values of any type into a field)
* GeoJSON (special syntax introduced below)

const tourSchema = new mongoose.Schema({

  price: {

    type: Number,

    required: [true, '<error-message>'],

  },

});

Here is a list of different options that can be used for different types of values:

#### String

* Required (Boolean)
* Unique (Boolean)
* Default
* Enum ([<values>]): limits options for the value entered into a certain field.
* Trim (Boolean): removes any white space before and after the data
* Select (Boolean): determines whether or not a certain field can be exposed in the responses when data is queried, not when created.
* Set: declares a setter function which has access to the value inputted for the field.

<field-name>: {

      type: Number,

      set: (*val*) => {},

    },

#### Number

* Required (Boolean)
* Default
* Enum ([<values>])
* Select (Boolean)
* Set: declares a setter function which has access to the value inputted for the field.

#### GeoJSON

In order to specify a Geospatial data, Mongoose needs to receive a ‘type’ and a ‘coordinates’ field in an object which is not the schema type object. The type, should then contain an object which is the schema type object, containing a type field of String, and afterwards, the coordinates field should contain an array of numbers, that will represent longitude and latitude.

<field-name>: {

      type: {

        type: String,

        default: 'Point',

        enum: ['<value(s)>'],

      },

      coordinates: [Number],

      address: String,

      description: String,

    },

### Embedding in schema

In order to embed a related document into a main document, we should implement the related document’s model in an array into the main document’s model. This would introduce the related document as a sub-document, and not just a regular field of the main document.

<field-name>: [

      {},

    ],

### Referencing in schema

We only keep reference IDs and then when querying happens, we will perform a populate task if necessary.

#### Child referencing

For child referencing, we should define the related or child document’s model in an array in the parent document’s model, again introducing the related referenced document as a separate sub-document. In the referenced document model, the schema type object should contain the ‘type’ field set to mongoose schema object ID, and the ‘ref’ field set to the target Model that is actually being referenced.

<field-name>: [

      {

        type: mongoose.Schema.ObjectId,

        ref: '<target-Model>',

      },

    ],

**NOTE | It is not necessary to import the target Model into the model file where referencing is implemented.**

#### Populating (child referencing)

The populate process always happens in a query. But in order to make it happen in all find queries, it should be implemented as a find pre-hook. The populate method accepts an object that should contain the ‘path’ property in which the name string of the field which should be populated is inserted, and then optionally, a ‘select’ field where we can determine which fields should or should not be displayed in the response. Remember that if we need the populate process to happen for more than one field, multiple populate methods can be chained together.

<schema-name>.pre(/^find/, function (*next*) {

  this.populate({

    path: '<field-name>',

    select: '<field-name1 field-name2>',

  });

  next();

});

#### Parent referencing

For parent referencing, we simply define the schema type object for the parent data inside the child document’s model. The schema type object should contain the type field set to mongoose schema object ID, and the ‘ref’ field set to the parent document’s Model.

<field-name>: {

      type: mongoose.Schema.ObjectId,

      ref: '<field-name>',

    },

**NOTE | schema type object in parent referencing is not placed inside and array.**

#### Populating (parent referencing)

Virtual populating is also implemented in a find pre-hook since we want it to happen on all find queries. Again, multiple populate methods can be chained together in case more than one field needs to be populated.

<schema-name>.pre(/^find/, function (*next*) {

  this.populate({

    path: '<field-name1>',

    select: '<field-name2>',

  });

  next();

});

#### Virtual populating (parent referencing)

The problem with parent referencing is that we won’t be able to see the child documents when we query for the parent documents. The parent documents have no reference of their children document. To solve this problem, we use virtual populating. It is a bit similar to the way we define virtual properties. We implement a virtual property on the parent document’s schema.

The virtual method here accepts first the name of the field of this parent Model into which the children documents are going to get populated, and then an object in which we set the ‘ref’ property to the target Model of the documents that are going to get populated, then we set the ‘foreignField’ to the specific field in the child Model; the field that contains the parent document ID referring back to this parent document, and finally, we set the ‘localField’ to the field name of this parent document where ID is stored, which is regularly ‘\_id’.

tourSchema.virtual('<field-name>', {

  ref: '<target-Model>',

  foreignField: 'target-reference-field',

  localField: '\_id',

});

Then we can use the populate method on any find method.

<GET-function-name> = async (*req*, *res*, *next*) => {

  const <doc(s)> = await <Model>.findById(req.params.id).populate('<field-name>')

};

### Create a model out of schema

The model method which is available on the mongoose variable, accepts first the collection name which is the same is the variable into which the model is stored, and second, the schema out of which the model is created.

const <Collection-name> = mongoose.model('<collection-name>', <schema-name>);

It is a convention to start model names with a capital letter.

#### Model.countDocuments()

It returns the number of documents existing in a collection. This method returns a promise and should we awaited.

const numTours = await Tour.countDocuments();

### Virtual properties

Refers to fields that can be defined for documents without being actually recorded into the documents, with the aim of saving space. It is usually used for fields that their value can be calculated based on some other persisted fields. To create a virtual field, the ‘virtual’ method should be called on the schema. The method accepts the name of the virtual field. Usually a ‘get’ method should be chained to the ‘virtual’ method since virtual properties should be created when data is retrieved from the database. The ‘get’ method accepts a regular callback function, with the ‘this’ keyword referring to the current document.

<schema-name>.virtual('<field-name>').get(function () {});

In order for the virtual fields to appear in the response, the object of options passed into the schema should include the ‘virtual’ option set to true in both JSON and Object formats.

const <schema-name> = new mongoose.Schema(

  {<schema>},

  {

    toJSON: { virtuals: true },

    toObject: { virtuals: true },

  }

);

**NOTE | virtual properties cannot be used in data querying.**

### Data validation

#### Required

Can be used on all types of data. This obligates the client to input the data for a specific field. It receives an array, where the first element should usually be ‘true’ and the second element determines the message string that should be sent back as error when the client leaves the field empty.

required: [true, '<message>'],

#### Maxlength/minlength

Can be used on String type. Determines the number representing the max/min length of the input string as the first element of an array, and the error message string as the second element.

maxlength: [number, '<messsage>'],

#### Max/min

Can be used on Number and Date types. This determines the max/min input value. Receives an array where the first element specifies the amount of max/min, and the message string as the second element.

min: [number, '<message>'],

#### Enum

Restricts the value of a certain field to pre-defined values. Receives an object with ‘values’ and ‘message’ properties. The values field define the pre-defined values in an array. The message property receives the error message string.

enum: {

  values: [<value1>, <value2>, <value3>],

  message: '<message>'

      },

#### Custom validator

A custom validator is a simple function that should return either true or false. To implement a custom validator on a certain data field we insert the ‘validate’ property into the schema type object. The validate field receives an object in which the validator function is defined in addition to the message property. The validator function has access to the value inputted by the client.

<field-name>: {

      type: <type>,

      validate: {

        validator: function (*val*) {},

        message: '<message> {VALUE}',

      },

    },

**NOTE | Mongoose makes the message property have access to the inputted value through {VALUE} syntax.**

**NOTE | A validator will not execute if a document is being updated with methods like findOneAndUpdate. We can forcefully execute target validators in update methods by setting the ‘runValidators’ option to true. Even then, if the target validators needs the ‘this’ keyword (current document) in its code, it won’t be able to perform validation correctly, since when updating documents, the ‘this’ keyword is not available in validators. The ‘this’ keyword inside a validator function refers to the current document only if a new document is being ‘.create()’ed or a document is being ‘save()’ed. So if a document is being updated, and you want to run its validator, and that validator uses the ‘this’ keyword in its code, you have to call the ‘.save()’ method on the document to persist the update. The ‘.save()’ method will execute all validators and activate the ‘this’ keyword in the validators.**

### Indexes

Used to improve the read performance. MongoDB automatically create indexes on the ID field of each document. An index is an ordered list of the values of a certain field of all documents of a collection. We can manually create indexes on fields that are queried very often. We need to take into account the access patterns of the application in order to decide which fields should have indexes. If there is a collection with a high write-read ratio, then it would make no sense to create an index on any field in this collection.

The ‘index’ method should be called on the schema. This method first accepts an object where the target field name is defined as the key, and the value can be either 1 for ascending order and -1 for descending order, and then it accepts another object where options can be implemented.

#### Single-field index

<schema-name>.index({ <field-name>: 1 }, { <option> });

#### Geospatial index

This index is implemented for fields containing geospatial data and is used for [geospatial queries](#_Finding_documents_matching). Instead of 1 or -1, we should specify ‘2d’ for data on an imaginary 2d plane, or ‘2dsphere’ for data on the earth’s surface.

<schema-name>.index({ <field-name>: '2dsphere' });

#### Compound index

This is usually used when we know that two specific fields are often queried together.

<schema-name>.index({ <field-name1>: 1, <field-name2>: -1 });

**NOTE | compound indexes work for situations when we query for only one of its fields.**

**NOTE | for each field set to be unique in the schema, an index will be created automatically.**

**NOTE | to make indexes unique, the ‘unique’ property must be set to true in the object of options passed into the index method.**

<schema-name>.index({ <field-name1>: 1, <field-name2>: -1}, { unique: true });

**NOTE | implementing a compound index with the unique option might take a while to become functional programmatically.**

### Static methods

Static methods are declared on the ‘statics’ property of the schema, and then they should be called on the Model. In static methods, the ‘this’ keyword refers the Model. Here is how we declare a static method:

<schema-name>.statics.<function-name> = function (*<argument>*) {};

And here is how we call the static method.

<Model>.<function-name>(<argument>);

**NOTE | in case we need to call the static method inside a pre/post-hook we should call the method on ‘this.constructor’ since ‘this’ in a pre/post-hook refers to the current document, the constructor of which would certainly be the Model.**

<schema-name>.post('save', function () {

  this.constructor.<static-method>(<argument>);

});

### Instance methods

Instance methods are declared on the ‘methods’ property of the schema, and then they should be called on a document. In an instance method, the ‘this’ keyword refers to the current document. Here is how we declare an instance method:

<schema-name>.methods.<function-name> = function (*<argument>*) {};

And here is how we call the instance method:

<document>.<function-name>(<argument>);

## Mongoose CRUD operations

### Creating documents manually

The model created out of a schema will act as a class, from which an instance (document) can be created.

const <document-name> = new <Model>({});

into the model a document object should be passed. This document object should follow the structure and rules defined in the schema. This is not the regular way of creating document in real-world applications.

The created document has now access to a couple of methods.

#### Document.save

This will save the document to the specified collection (Model). The save method returns a promise and can be consumed using a ‘then’ method, which will have access to the document that was just persisted into the database. The saving process might also fail, and its errors can be caught with a chained ‘catch’ method.

<document-name>.save().then((*doc*) => {}) .catch((*err*) => {});

**NOTE | saving modified data to documents will always run the ‘required’ validators. To bypass schema validators, the save method can accept an object of options. For instance, we need this in the process of persisting the** [**password reset token**](#_Password_reset) **on the user document. (This method runs all validators???)**

await user.save({ validateBeforeSave: false });

#### Document.isModified

Returns Boolean based on checking whether or not the value of a certain field of the document, passed into the method, is changed.

if (<current-doc>.isModified('<field-name>')) {}

#### Document.isNew

Returns Boolean based on checking whether or not the document is just being created. This method should not be called right away.

if (<current-doc>.isNew) {}

### CRUD operations

It is important to remember that all database-related operations are done **asynchronously**. So all related methods should be awaited for their results as necessary.

#### Creating documents programmatically

In this way of creating documents, the ‘create’ method should be called on the Model. This method also accepts the document object that is going to be created, but now this object will come from the request body. This method also returns a promise and can be consumed.

<handler-function-name> = async (*req*, *res*) => {

  const <document> = await <Model>.create(req.body);

};

**NOTE | the ‘create’ method can either accept one document object, or an array of document objects.**

**NOTE | any additional fields inserted into the request body, which is not planned in the schema, would simply be ignored and not included in the created document.**

**NOTE | in handler functions, always remember to send a response to the request in order to finish the request-response cycle.**

<handler-function-name> = async (*req*, *res*) => {

  const <document> = await <Model>.create(req.body);

res.status(200).json({});

};

**NOTE | every one of CRUD handlers might fail to execute. Therefore a try/catch block may be necessary for all in order to handle errors. Consider using a central error handler function so that the try/catch block would be used only once, and not in each CRUD handler function.**

#### Reading documents

The ‘find’ method should be called on the Model, with a query object passed into it. All find methods return promises, and should be awaited.

* **Model.find**

Leaving the query object empty in the ‘find’ method will retrieve all documents existing in a the collection. Passing in a query object, will return all documents matching the query.

<handler-function-name> = async (*req*, *res*) => {

  const <documents> = await Tour.find();

};

* **Model.findOne**

This method finds the only document in the collection that matches the query object passed into it.

<handler-function-name> = async (*req*, *res*) => {

  const <document> = await Tour.findOne({<query>});

};

* **Model.findById**

This method accepts the ID value by which it will find the document in the collection. This method uses the ‘findOne’ method of MongoDB behind the scenes.

<handler-function-name> = async (*req*, *res*) => {

  const <documents> = await Tour.findById(<id>);

};

**NOTE | remember that when a request hits a route containing a parameter, the value of that parameter is accessible on the request object, in the ‘params’ property. So it is the route definition that makes this possible. Don’t take it as related to the Params middleware in no way.**

**NOTE | all find methods can be chained with a ‘select’ method to determine whether or not certain fields should be displayed in the results. In order to return fields that have been selected off from the schema, a + sign should be used in the ‘select’ method here before the field name.**

  const user = await User.findOne({ <field-name1>: <value> }).select('+<field-name2>');

#### Updating documents

The ‘findByIdAndUpdate’ method should be called on the Model. This method accepts first the ID value by which it will find the document that is going to be updated, then the target key/value that is going to be updated, and finally, an object of options.

exports.updateTour = async (*req*, *res*) => {

    const <doc> = await <Model>.findByIdAndUpdate(<id>, req.body, {

      new: true,

      runValidators: true,

});

}

In the options object, the ‘new’ property determines whether or not the updated document should be returned, and the ‘runValidators’ determine if the related validators defined in the schema should be executed before updating.

**NOTE | update methods don’t run validators by default. In order to validate the input data, we should forcefully run the target validators by setting the ‘runValidators’ option to true. Even then, if the target validator needs the ‘this’ keyword in its code, it won’t be able to perform validation correctly. Fix is to use the ‘.save()’ method on the document to persist updated fields.**

**NOTE | behind the scenes, ‘findByIdAndUpdate’ actually runs MongoDB ‘findOneAndUpdate’ function.**

#### Deleting documents

The ‘findByIdAndDelete’ method should be called on the Model. This method accepts only the ID value of the target document. The method returns a promise and should be awaited.

exports.deleteTour = async (*req*, *res*) => {

  try {

    await Tour.findByIdAndDelete(<id>);

};

**NOTE | behind the scenes, ‘findByIdAndDelete’ actually runs MongoDB ‘findOneAndDelete’ function.**

### Filtering documents

#### Simple filtering

Query parameters related to pagination, sorting, limiting, and fields should usually be excluded from the query string, since these are not the fields used inside each document object. These parameters are not used to find certain documents, but they determine how the found documents should be displayed in the response. The exclusion should be performed on a hard copy of the query object (req.query). These parameters should simply be deleted from the hard copy. To create the hard copy:

const queryObj = {...req.query};

fields that should be excluded might be stored in a separate variable:

const exculdedFields = ['page', 'sort', 'limit', 'fields'];

Each of the parameters should be removed from the hard copy:

exculdedFields.forEach((*field*) => delete queryObj[field]);

the ‘find’ method used on the Model should now be using the hard copy as the query object. If we want to execute the query with this simple implementation, we can await this find method right away.

const tours = await Tour.find(queryObj);

**NOTE | with the current implementation of filtering, query strings can only include ‘equals’, and not MongoDB operators gt, gte, lt, lte.**

**NOTE | the find method used on the Model, returns a query. Once this query is awaited, it will be executed, and it will return with found documents. When a query is executed, it is no longer available and nothing can be added to it, like sorting or pagination, or some advanced filtering. What should actually be done is to, first, store the query in a variable without awaiting it, which is also called building the query:**

const query = Tour.find(queryObj);

**We also might want to check for MongoDB operators in the query before building the query. After that, we usually chain all the query stages necessary, and finally await the query, or to execute the query, which will return with the documents that match the query, displayed in a way that also matches the query.**

const tours = await query;

**NOTE | when executing the query we can use the ‘explain’ method on the query, and it will return a huge amount of statistics about the query, including execution statistics.**

#### Advanced filtering

A standard way of writing more complex queries in the URL is to include the operator name in the brackets before equal signs.

127.0.0.1:3000/api/v1/tours?duration[gte]=5&difficulty=easy

Remembering how acquiring query string works, through the ‘req.params’ the query object is received as:

{duration: {gte: '5'}, difficulty: 'easy'}

So either one of operator names should be detected in the object and a ‘$’ sign should be inserted before it.

let queryStr = JSON.stringify(queryObj);

    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (*match*) => `$${match}`);

The replace method accepts first the string that should be detected, and then a callback function determining what the match should be replaced with. Now the query object in ‘req.query’ will look like:

{duration:{'$gte': '5'}, difficulty: 'easy'}

This stringified query object should be converted back to a JavaScript object so that it can be inserted into the find method in order to build the query. Note that we build it with ‘let’ so that later we can add methods to it.

let query = Tour.find(JSON.parse(queryStr));

#### Sorting documents

The URL requesting all documents sorted by, for example, the price field, would look like this:

127.0.0.1:3000/api/v1/tours?sort=price

**NOTE | once the sort functionality is implemented, this URL will make it so that the results will be sorted in an ascending order. To make it descending, a minus sign behind the field name should be inserted.**

127.0.0.1:3000/api/v1/tours?sort=-price

We should check if the query object (req.query) contains the ‘sort’ property. If so, we then chain the sort method to the query.

if (req.query.sort) {

      query = query.sort(req.query.sort);

    }

**NOTE | in case there are documents with the same price, we want them to be sorted based on a second field. The sort method of Mongoose can accept two field names separated with a space. However, since we cannot put spaces into the URL, we can separate them with a comma in the URL, and then replace the comma with a space to insert it into the sort method.**

if (req.query.sort) {

    const sortBy = req.query.sort.split(',').join(' ');

    query = query.sort(sortBy);

    }

#### Limiting fields

This will allow clients to choose which fields of the result documents they want to be displayed in the response. A URL requesting to limit fields would look like this:

127.0.0.1:3000/api/v1/tours?fields=name,duration,difficulty,price

We should now check the query object (req.query). If there is a ‘fields’ property, we then use the ‘select’ method on the query. Similar to ‘sort’, the ‘select’ method can receive multiple fields separated by comma. So the commas in the URL should be replaced with spaces before being passed into the select method.

if (req.query.fields) {

  const fields = req.query.fields.split(',').join(' ');

  query = query.select(fields);

    }

**NOTE | Mongoose creates some default fields in each document, that we don’t want to display in responses. We can implement this in an ‘else’ block, putting a minus sign before the field name that we want to be excluded from the results.**

if (req.query.fields) {

  const fields = req.query.fields.split(',').join(' ');

  query = query.select(fields);

    } else {

  query = query.select('-\_\_v');

    }

**NOTE | the minus sign can also be used in the URL. It will simply be included into the select method.**

#### Pagination

To implement pagination in Mongoose, we should use ‘skip’ and ‘limit’ methods on the query. These two methods together can define how many results should be displayed in each page, and which page should actually be displayed. A URL requesting a certain page with a certain limit of results on each page looks like:

127.0.0.1:3000/api/v1/tours?page=2&limit=10

So the value of ‘page’ will help us calculate the number that should be passed into the ‘skip’ value based on the number of results that should be displayed on each page, coming from the ‘limit’ field of the query object.

const page = req.query.page \* 1 || 1;

const limit = req.query.limit \* 1 || 100;

const skip = (page - 1) \* limit;

query = query.skip(skip).limit(limit);

## Aggregation pipeline

All documents of a certain collection go through a pipeline, where they are processed step by step in order to generate aggregated results. To build an aggregation pipeline, the ‘aggregate’ method should be used on the Model (collection). This method accepts an array consisting of different stages of the pipeline, each stage defined using an object. The result of performing calculations of the aggregation pipeline is then stored in an array, which can be saved into a variable.

const stats = await Tour.aggregate([]);

Similar to the ‘find’ method that returns a query, this method returns an aggregate object. Here is a list of different stages that can be used in a pipeline.

**NOTE | Stages can be repeated with different intentions.**

### Stages

#### Matching

This stage is very similar to querying data in MongoDB. To determine a stage as a matching stage, the **$match** operator should be used.

const stats = Tour.aggregate([

      {

      $match: { ratingAverage: { $gte: 4.5 }},

      },

    ]);

The $match operator receives an object, where the matching criteria is inserted just like querying.

The matching stage can be repeated as the final stage of a pipeline, probably to perform matching on the calculation results produced through the pipeline. This matching stage would have to perform matching process according to the new field names, and not the field names of the original documents, since they are actually not available any more. For example, the ‘\_id’ field name here was generated in the grouping stage of the pipeline.

{

  $match: { \_id: 'EASY' },

},

#### Grouping

The **$group** operator should be used. This operator receives an object, where the value of the ‘\_id’ property determines by which field existing in all documents, the matched documents will be grouped together. Setting this to null will simply group all matched documents in 1 big group, allowing for an overall calculations for all documents. New fields can be created here to include the result of different calculation operators:

{

     $group: {

       \_id: null,

},

Setting to a specific field name of the documents will group the matched documents in different groups that represent different values recorded for the specific field.

{

     $group: {

       \_id: '$difficulty',

},

Here is a list of different calculation operators:

* **$avg**

To calculate the average for a specific field of documents. This operator receives a specific field name that exists in each document with a $ sign in front of it.

<new-field-name>: { $avg: '$<document-field-name>' },

* **$min / $max**

To calculate the minimum/maximum value that has been recorded for a field among all matched documents. This operator receives a specific field name that exists in each document with a $ sign in front of it.

<new-field-name>: { $min: '$<document-field-name>' },

* **$sum**

Used to calculate different kinds of total values. This operator can receive either a field name with a $ sign in front of it, or a number, mostly 1, to add that specific number for each single document that is being processed into the pipeline.

<new-field-name>: { $sum: '$<document-field-name>' },

<new-field-name>: { $sum: 1 },

#### Sorting

This stage will sort the result of the calculations done through the pipeline. So field names of the original documents cannot be used here. Results can only be sorted according to the field names created by the pipeline. Into the field name by which the sorting will happen, we can pass either 1 or -1, for ascending or descending order respectively.

      {

        $sort: { avgPrice: 1 },

      },

#### Unwinding

This stage is used to create instances of a document based on an array of values that is stored in a field of that document. The **$unwind** operator should be used on a field name that contains an array of values in each and every document.

{

   $unwind: '$<field-name>',

},

#### Add fields

To add certain fields to be introduced into the new documents created through the pipeline calculations.

{

  $addFields: { <new-field-name>: '$<pipeline-field>' },

#### Projecting

Used to determine whether certain fields should or should not be displayed in the final results of a pipeline.

{

  $project: {

    \_id: 0,

  },

},

**NOTE | for referring to field names in Matching, Sorting and Projecting stages, we don’t use the $ sign.**

#### Limiting

Used to limit the number of results displayed in the response.

{

  $limit: <number>,

},

#### Geospatial calculations

Used to perform calculations on geospatial data. The only geospatial aggregation pipeline stage that exists is ‘geoNear’, which is used to calculate distances of geospatial data from a specific point. The document field containing the geospatial data should have [geospatial index](#_Geospatial_index) on it.

The geoNear stage receives an object with the ‘near’ property set to the point from which distances will be calculated. So the near property will accept an object with the type field set to ‘Point’, and the ‘coordinates’ set to an array with longitude first and latitude second.

After the near property, we should define the ‘distanceField’ property with the field name in which the calculated distance would be stored.

Other optional fields can be added. One of the useful ones is ‘distanceMultiplier’ which helps us easily convert the distance unit. This property receives a number, with which it will multiply the calculated distance before storing it into the specified field.

{

  $geoNear: {

    near: {

      type: 'Point',

      coordinates: [lng, lat],

    },

    distanceField: '<field-name>',

distanceMultiplier: <number>,

},

**NOTE | This aggregation stage should always be implemented as the first stage.**

### Operators on values

Some operators act on values. Here is a list:

#### $toUpper

Probably mostly used in the grouping stage to determine how the different group names will appear in the response.

{

     $group: {

       \_id: { $toUpper: '$<field-name>' },

},

#### $ne

Probably mostly used in the matching stage to exclude documents with certain values of a specific field. Stands for Not Equal.

{

   $match: { \_id: { $ne: <value> }},

},

#### $gt / $gte / $lt / $lte

Probably mostly used in the matching stage to filter for a range of values.

{

  $match: {

    <field-name>: {

      $gte: <number>,

    },

  },

},

#### Date expression operators ($month)

If the value of a certain field is of the type Date, this operator can be used to take out the month from it. Most probably used in the matching and grouping stages.

{

  $group: {

    \_id: { $month: '$<Date-field>' },

  },

},

#### $push

Probably mostly used in the grouping stage where other process can be done also, with the aim of pushing values of a certain field of each document into a newly created array.

      {

        $group: {

          <new-field-name>: { $push: '$<field-name> },

        },

      },

## Mongoose middleware

Mongoose has 4 types of middleware used as pre/post-hooks.

### Document middleware

Document middleware can act on a currently processed document. Document middleware are defined on the schema.

#### Document pre-hook

This can act right before the currently processed document is saved into the database. The ‘pre’ method should be called on the schema. This method accepts first the event upon which it should act, and then the callback function which determines what should be done, with the this keyword referring to the current document, and also having access to the ‘next’ function.

<schema-name>.pre('save', function (*next*) {});

#### Document post-hook

The callback function passed into the ‘post’ method chained on the schema, has access to the next function, along with the document that was just persisted into the database.

<schema-name>.post('save', function (*doc*, *next*) {});

**NOTE | in case we need to refer to the Model of the current document, we can use ‘this.constructor’, where ‘this’ is the current document. We would not be able to simply use the Model name, because at the point where we declare pre/post-hooks in the code, the Model is not yet created out of the schema. If we place the pre/post-hook middleware after the Model declaration, the Model would not contain that specific pre/post-hook.**

**NOTE | the ‘save’ pre-hook will only be triggered with ‘.save()’ and ‘.create()’ commands.**

### Query middleware

Allows us to run functionalities before of after a certain query is executed.

#### Query pre-hook

The this keyword in the callback function here refers to the current query.

<schema-name>.pre('find', function (*next*) {});

**NOTE | the query pre-hook is usually used to act on all kinds of ‘find’ events, including ‘findOne’. Use regular expressions to include all events starting with ‘find’.**

<schema-name>.pre(/^find/, function (*next*) {});

**NOTE | one of the most popular use cases of a query pre-hook is to perform** [**populating processes**](#_Populating_(child_referencing))**.**

**NOTE | in order to access the current document in a query pre-hook, we can execute the current query (this). It will return with the current document.**

<schema-name>.pre(/^find/, async function (*next*) {

  const <current-doc> = await this.findOne();

next();

});

**NOTE | by executing the query and accessing the current document, we can, in turn, access the Model using the ‘constructor’ on the current document.**

<schema-name>.post(/^find/, async function () {

  <current-doc>.constructor.<static-method>(<argument>);});

**NOTE | we can add properties to the query object in the pre-hook, and these properties would be accessible in the post-hook, where the query would not be executable anymore. In the example above, if the current document needs to be accessed in a post-hook query middleware, it should be stored as a property on the current query (this).**

<schema-name>.pre(/^find/, async function (*next*) {

  this.<current-doc> = await this.findOne();

  next();

});

#### Query post-hook

The callback function of the query post-hook has access to the next function, along with all the documents found by the query.

<schema-name>.post(/^find/, function (*docs*, *next*) {});

**Note | in the query post-hook we still have access to the current query through the ‘this’ keyword, but since it has already executed, we can no longer execute it. We can just use its properties. These properties might be added to the current query (this) in the pre-hook query middleware.**

**NOTE | in order to make a query middleware run for document update or delete, we should use a regular expression that finds any event starting with ‘findOneAnd’, because Mongoose ‘findByIdAndUpdate’ and ‘findByIdAndDelete’ functions actually run MongoDB ‘findOneAndUpdate’ and ‘findOneAndDelete’ functions behind the scenes.**

<schema>.pre(/^findOneAnd/, function (*next*) {});

<schema>.post(/^findOneAnd/, function (*docs*, *next*) {});

### Aggregation middleware

Allows us to implement a functionality for all aggregate events.

#### Aggregate pre-hook

The callback function of an aggregate pre-hook has access to the next function, with the this keyword referring to the current aggregation object, on which we have access to the ‘pipeline’ method. Calling the pipeline method on the current aggregation object, returns the array of stages defined in the aggregation pipeline. This array can be treated as a regular JavaScript array, into which we can **push** or **unshift** additional stages.

tourSchema.pre('aggregate', function (*next*) {});

## Mongoose errors

Some errors are generated by Mongoose in a couple of different situations. We want these errors to be generated through an extended error class where they are marked as operational errors.

### Invalid ID format

This happens when Mongoose cannot convert the inputted ID to a valid MongoDB ID. This kind of error has a name property of ‘CastError’ with which it can be detected in the production environment of the global error handler middleware. This error always includes ‘path’ and ‘value’ properties, indicating in which field invalid data has been inputted.

### Duplicate keys

This happens when the client attempts to create a document with a value that should be unique, but it already exists in one of the other documents. This error usually contains a ‘code’ field of 11000, with which it can be detected in the production environment of the global error handler middleware.

### Validation

This happens when the client attempts to update data of a document with a value that goes against the validators implemented in the schema. This error contains the ‘name’ property of ‘ValidationError’. For each invalid inputted data in the request body, an error object is created where related error message defined in the schema validators is returned and stored in the ‘message’ property. These error objects are all stored in an ‘errors’ object inside the main ‘error’ object. We usually need to take our all the messages in this object and append them to a formatted error message before sending to the client.

# Authentication

## JSON Web Token

This is a stateless solution for performing authentication measures. All the communication through which the authentication process is executing should be performed over HTTPS.

### JWT creation

After a user’s login credentials are considered valid, a unique JWT is created using a constant secret string stored on the server. This process is called signing the JWT. This JWT is sent back to the client in a cookie. At this point the user is already logged in.

### JWT structure

The JWT structure consists of 3 parts which are used for JWT validation.

* Header: contains metadata about the token. This metadata is encoded, not encrypted.
* Payload: contains any data we want. This data is encoded, not encrypted.
* Verify signature: this unique part is created using the header, the payload, and the secret string stored on the server.

### JWT validation

From now on, the client will send the JWT back to the server along with each request. Now the server should check if the JWT is valid. If so, the requested data will be sent back to the client. The validation process contains 4 steps.

#### Check if token exists in the request

We should first check if the token has been sent by the client, to see if there actually is any token.

#### Validate token

If there is a token, this step basically checks

1. **If the token is not manipulated**: the ‘verify’ function takes the header and the payload, and attempts to create a test signature with the secret string stored on the server. It this test signature matches the verify signature that already exists within the JWT, then the JWT is considered valid. If anything was modified in the header or the payload, the test signature would not match the verify signature, and an [error](#_JWT_verification_error) would be generated.
2. **If the token has not expired**: the verify method will automatically check if the token has not expired yet. It uses the expiring option set on the token when it was signed. If the token was expired, an [error](#_JWT_verification_error) would be generated.

#### Check if the user still exists in database

We should check if the user has been deleted after the token was signed. If the user was deleted, the token would still exist, but we don’t want to let anyone use the token. This would typically use the ID stored in the JWT to find the user document with that ID, and if there is a user with that ID in the database, this step is checked.

#### Check if user changed password after JWT was signed

If someone gets access to the account of a user, the user might change their password to protect themselves against that attack. So if this happened after a JWT was signed, we should consider that JWT as invalid. This check is typically performed through an [instance method](#_Instance_methods) that compares two timestamps: one representing the time when the token was signed, accessible at the [‘iat’](#_Jwt.verify) property of the decoded payload, and one representing the time when the password was changed, if it was changed ever.

#### Put fresh user on the request object

After all steps are successfully verified, then the user document found in the process is inserted into the request object, where it can be found in other middleware, like the [authorization](#_Authorization) middleware, protected by these stages, gathered together as a protection middleware.

### JWT library

Requires NPM installation:

npm install jsonwebtoken

Also needs to be required in a file:

const jwt = require('jsonwebtoken');

#### Jwt.sign

used to sign a JWT. This method accepts first an object containing the payload, which is usually the ID of the user document. Then the secret string, stored on the ‘config.env’ file on the server, should be passed into this method. Afterwards, an object of options can be passed into the sign method.

The secret string should be unique string consisting of at least **32 characters** long, allowing us to use **HAS256** encryption. The token’s header will be generated automatically.

Into the object of options, we can specify the JWT expiration time. This time is normally stored in the configuration file.

Configuration file:

JWT\_SECRET=hot-butter-popcorn-cold-cheese-chips

JWT\_EXPIRES\_IN=90d

Signup or login functionality:

const token = jwt.sign({ id: <user-doc>.\_id }, process.env.<secret-string-env-variable>, {

    expiresIn: process.env.<expiration-env-variable>,

  });

#### Jwt.verify

This method receives first the token that should be verified, then the secret string stored on the server, then a callback function which will run once the verification process is done. This method performs an asynchronous task and it should be awaited. We may need to [promisify](#_10_–_Util) this method to follow the async/await code paradigm.

const decoded = await promisify(jwt.verify)(token, process.env.JWT\_SECRET);

The result of the verify method is the decoded payload of the JWT. This payload includes an ‘iat’ property containing a timestamp representing the time when the JWT was signed.

### JWT verification error

A JWT might be manipulated or expired. In case the token is manipulated and therefore invalid, an error will be generated with the ‘name’ property of ‘JsonWebTokenError’. In case the token is expired, the ‘name’ property would be ‘TokenExpiredError’.

## Password encryption

Password encryption is a serious measure that should be taken care of as a user is created or an existing user’s password is updated. To implement encryption we use the Bcrypt package. Password encryption consists of two stages: salting, hashing.

### Bcrypt

Requires NPM installation:

npm install bcryptjs

Needs to be required in a file:

const bcrypt = require('bcryptjs');

Now there are methods available on bcrypt.

#### Bcrypt.hash

Performs Password encryption, which is usually implemented in a [Mongoose document pre-hook](#_Document_pre-hook), if the password is modified. The ‘hash’ method accepts first the document field that contains the password, and then a cost parameter, determining how CPU intensive the salting operation will be. This method performs an asynchronous task and it should be awaited.

userSchema.pre('save', async function (*next*) {

  if (!this.isModified('password')) return next();

  this.password = await bcrypt.hash(this.password, 12);

  this.passwordConfirm = undefined;

next();

});

**NOTE | after encryption, we usually get rid of the password confirm field by setting it to undefined. We don’t need to persist this field into the database.**

#### Bcrypt.compare

Performs a comparison between an inputted plain password and an encrypted password stored in the database, and finally returns a Boolean. This method is used to confirm the password that the user has inputted to log in. This method performs an asynchronous task and it should be awaited.

const compare = await bcrypt.compare(<input-password>, <encrypted-password>);

## Authorization

This will allow certain users to perform certain actions based on their user roles. The middleware taking care of this process should be able to receive arguments, while normally, middleware functions cannot receive custom arguments. Typically, the concept of a **wrapper function** is used here. This wrapper function returns another function, ready to act as a normal middleware that would be called by Express sometime in future. But this returned middleware function would have access to the arguments that the wrapper function received when it was called and executed. This access is possible through JavaScript closures.

## Password reset

This process typically consists of 2 steps, each performed on different routes:

1. **User claims that they have forgotten their password:** at this step, the user would have to provide their email address. After checking if the email address actually exists in the database, a password reset token would be created and sent to the email. The token is usually created by an [instance method](#_Instance_methods) defined on the user documents. The reset token doesn’t have to be as cryptographically strong as a password hash. Instead, it can simply be generated using the [randomBytes](#_Producing_a_random) function from the [crypto module](#_11_–_Crypto), and then encrypted using the [createHash](#_Encrypt_a_random) method. The plain version should be used to form a URL leading the user to a page where the new password can be set. This URL is sent back to the user, usually through email. The encrypted version of the reset token would be stored on the user document, normally along with a short expiration limit property. Remember to ‘[.save()](#_Document.save)’ this update to the user document using a bypass over schema validators. If there is an error in the process of sending the email, the token and its expiry that was persisted to the user document should be set to undefined, and again persisted to the database bypassing the schema validators.
2. **The user sends the new password along with the password reset token:** at this step, the user would have to click on the URL sent to their email which will lead them to the page where they can set their new password. The unencrypted token is in this URL and it should be encrypted so that we can then find a user document on which the same encrypted token is stored and that the token’s expiration time is $gt than the current time. Then if a user is found, the inputted password and passwordConfirm values in the request body should be inserted and finally persisted into the user document, without bypassing the schema validators. Then typically a property that holds the time when the password of a user document is changed should be updated, normally through a [document pre-hook](#_Document_pre-hook) middleware that allows this update only if the [modified](#_Document.isModified) property is the password property or if the user document is not [newly created](#_Document.isNew). At the end, a new JWT should be signed and sent back to the client.

**NOTE | a possible problem is reported suggesting that updating the passwordChangedAt timestamp into the database might take longer than usual, and therefore making the newly signed JWT already expired because of a condition implemented in the authentication algorithm that checks if the user has changed their password after the JWT was signed. Although this doesn’t seem likely to happen since the updating process is awaited in the password reset functionality, it is suggested to update the passwordChangedAt property with 1 second before the current time.**

## Logout functionality

Since the login functionality performs authentication by signing and sending an HTTP-only cookie, logging users out cannot be done with deleting cookies. We can’t actually delete HTTP-only cookies. But we can send a new cookie with the exact same name but with no JWT in it. This would then replace the previous cookie. All subsequent requests will contain this new cookie with which the user will not be identified as logged in.

## Security measures

There are some measures that should be taken to protect an application against some types of attacks.

### Compromised database

This is when an attacker has somehow found a way to penetrate our database.

Measures:

* Always encrypt passwords and password reset tokens.

### Brute-force attacks

This is when the attacker tries to guess a password by trying millions of random passwords until they find the right one.

Measures:

* Make the login request slow. The Bcrypt package does that to some extent.
* Implement rate limiting: limit the number of requests coming from one IP. Rate limiter is usually implemented as a global middleware. NPM package called Express Rate Limit is usually used. This requires NPM installation:

npm install express-rate-limit

And needs to be required into a file:

const rateLimit = require('express-rate-limit');

The rateLimit variable now holds a function which can be called, and its result can be stored in another variable, and this variables can then be used in a global middleware. The function will receive an object of options where the ‘max’ property determines the number of requests that are acceptable from one IP during a certain amount of time, which should be defined in the ‘windowMs’ property. A ‘message’ property can also be defined, which will be displayed as an error message when an IP exceeds the limit.

const limiter = rateLimit({

  max: 100,

  windowMs: 60 \* 60 \* 1000,

});

The ‘limiter’ variable can then be used in a global middleware with ‘app.use’.

app.use(limiter);

It is also possible to apply the limiter to a certain route:

app.use('<route>', limiter);

This will insert 2 items added to the response headers: RateLimit-Limit displaying the value of ‘max’ property, and Rate-Limit-Remaining displaying the number of remaining requests allowed to be sent from that specific IP. Once this limit is exceeded, a 429 (Too Many Requests) error will be sent back to the client.

* Implement maximum number of login requests and a waiting time.

### Cross-site scripting attacks

This is when the attacker tries to inject scripts into our webpage to run a malicious code.

Measures:

* Never store JWT in local storage. JWT should only be stored in an HTTP-only cookie.
* Sanitize user input data and set security HTTP headers: helmet package. This package requires NPM installation.

npm install helmet

Needs to be required into a file:

const helmet = require('helmet');

The functionality should be implemented as a global middleware, usually right after the rate [limiter middleware](#_Brute-force_attacks), at the beginning of the App file in order to make sure the security headers would be set. The implementation only involves calling the helmet function, as it is defined by the variable name, in an ‘app.use’.

app.use(helmet());

Calling helmet will return another function that would sit inside ‘app.use’ waiting to be called. This would then add some fields to the response headers. The browser understands these headers and will act on them.

Visit Helmet documentation on: <https://github.com/helmetjs/helmet>

### Denial-of-Service (DOS) attacks

This is when the attacker sends so many requests to a server that it breaks down the application.

Measures:

* Implement rate limiting.
* Limit the amount of payload data that can be sent in a request body. This should be set as an option in the [Express body parser](#_Express_body_parser) middleware.

app.use(express.json({ limit: '10kb' }));

* Avoid using evil regular expressions that take an exponential time to run.

### NoSQL query injection attacks

This is when an attacker tries to inject some query, instead of valid data, in order to create query expressions that will translate to a true Boolean.

Measures:

* Implement a well-defined Mongoose schema
* Sanitize user input data: this should usually be implemented as a middleware after the [Express body parser](#_Express_body_parser) middleware. To sanitize input data we can use 2 NPM packages: Express Mongo Sanitize, and XSS-Clean. Both require NPM installation.

npm install express-mongo-sanitize

npm install xss-clean

They should be required into a file:

const mongoSanitize = require('express-mongo-sanitize');

const xss = require('xss-clean');

and after the Express body parser middleware, we implement the Mongo Sanitize middleware:

app.use(mongoSanitize());

This would look into the request body, the query string, and request parameters to filter out all the $ signs and dots, since they are used to write MongoDB operators. We then use the XSS-Clean middleware:

app.use(xss());

This will clean input data from malicious HTML code, by converting all HTML symbols.

### Cross-site request forgery

This is when the attacker forces a user to execute unwanted actions on a web application in which they are logged in.

Measures:

* Use the Csurf package

### Best security practices

* All communications between server and client has to happen over HTTPS.
* Always create random password reset tokens, not generated based on dates or anything like that. Always give password reset tokens an expiration limit.
* Make JWTs invalid after a user changes their password.
* Never commit a configuration file to version control systems like Git.
* Never send the entire error object to the client. Always try to hide the stack trace from the client.
* Require the user to re-authenticate before performing high-value actions, like payments or deleting documents.
* Implement a blacklist functionality that tracks untrusted tokens.
* Confirm the email address when an account is first created.
* Implement refresh tokens to remember users in order to keep them logged in forever or until they choose to log out.
* Implement two-factor authentication.
* Prevent parameter pollution: the HPP package can be used to implement this. It requires NPM installation:

npm install hpp

needs to be required into a file:

const hpp = require('hpp');

it should be implemented as a global middleware, usually after the XSS-clean middleware.

app.use(hpp());

Now if the client hits this route, on which our API might not be ready to act:

{{URL}}api/v1/tours?sort=duration&sort=price

Duplicate parameters will trigger the HPP middleware and it will only act on the last parameter. However, with a similar route like this:

{{URL}}api/v1/tours?duration=5&duration=9

We actually want the application to work for this and other document fields. To do this, we can define the whitelist option in the hpp method. This option can receive an array of field names that should be allowed to be inserted as duplicates into the route.

app.use(

  hpp({

    whitelist: [

      'duration',

      '<field-name2>',

      '<field-name3>',

    ],

  })

);

# Global Node error handling

In addition to error handling measures that are regularly implemented for Mongoose and Express, some errors that occur outside Mongoose and Express should not be left unhandled. These errors include unhandled promise rejections, and uncaught exceptions.

## Unhandled promise rejections

It is a common practice to handle all promise rejections globally, since it might be hard to keep track of all promises throughout the code. For each unhandled promise rejection, an ‘unhandledRejection’ event is emitted by the ‘process’ object. This handler, which is also called a safety net, is implemented in the server file after the ‘server’ variable is defined. With promise rejections, we have the option of closing the server and shutting down the application.

process.on('unhandledRejection', (*err*) => {});

This might be used for:

* Handling the rejected promise returned by the attempt to establish connection between Mongo database and Express application.

## Uncaught exceptions

All errors or bugs in the synchronous code are called uncaught exceptions. When such errors happen, an ‘uncaughtException’ event is emitted by the ‘process’ object. This handler is implemented at the beginning of the server file, even before requiring the App. Since uncaught exceptions put the Node application in an unclean state, we have no choice but to shut down the application. We also don’t need to account for a graceful shutdown.

process.on('uncaughtException', (*err*) => {

    process.exit(1);

});

# Exporting and Importing Custom Modules

In Node JS, every single file is treated as a module. To export a custom un-named functionality from one file:

## Module.exports

It is used to export one single variable, like one class or one function. This export can be done un-named.

module.exports = (*<arguments>*) => {};

Then to import this functionality in another file we can give it a name:

const <function-name> = require("<file-path>");

The file path will start with ‘./’.

## Exports

It is used to export multiple named variables.

exports.<function-name1> = (*<arguments>*) => {};

exports.<function-name2> = (*<arguments>*) => {};

Then to import this functionality in another file:

const <custom-name> = require("<file-path>");

Then that exported function will be accessible on this custom-named variable:

const result = <custom-name>.<function-name>(*<arguments>*)

De-structuring can also be used when requiring.

const {<function-name1>, <function-name2>, <function-name3>} = require("./test-module-2");

# Node Application File Structure

It is a good practice to have a nice separation of concern between different functionalities in a Node application.

## App

All code related to Express should usually be declared in an ‘app.js’ file. This file includes all the Express settings. The ‘app’ variable which holds the result of calling express, should be exported, so that it would be accessible from the server file.

**CODE STRUCTURE-APP:** In a separate router (sub-application) file we have in order:

1. Express required (along with all other modules if necessary)
2. Middleware stack implemented
3. Routers mounted
4. App exported

## Server

All code related to server should usually be declared in a ‘server.js’ file. This is the starter file and from here requests will be listened to. This file needs to import the app variable from the ‘app.js’ file in order to be able to listen to requests. After establishing this server file, Nodemon should execute this, instead of app.

**CODE STRUCTURE-SERVER:**

1. App required
2. Server started and listening
3. MIGHT BE ADDED: database configuration, error handling, environment variables

# Environment Variables

These are variables used to define the environment in which a Node app is running. Node JS has a lot of environment variables defined in ‘env’ object of the ‘process’ core module. Environment variables are also used as configuration settings for the Node application.

Environment variables are usually defined in a configuration file called ‘config.env’ in the root folder. Different variables environments including the conventional ‘NODE\_ENV’ can be defined:

NODE\_ENV=development

Sensitive data about the application’s configuration can also be stored in this file. This configuration file should then be connected to Node application using the ‘dotenv’ NPM package.

npm intall dotenv

It should then be required in the **server** file, which is the starter file.

const dotenv = require('dotenv');

**NOTE | this should be required before requiring the ‘app’ variable, Otherwise, the application will run without having access to the environment variables.**

Then the ‘config’ method should be used on ‘dotenv’ variable, with the configuration file path passed into it through an object of options.

dotenv.config({ path: './config.env' });

**NOTE | Nodemon should restart after defining and connecting the config file.**

With the environment variables defined, middleware can be used based on them. For example:

if (process.env.NODE\_ENV === 'development') {

  app.use(morgan('dev'));

}

**NOTE | ‘process’ is, by default, accessible in all modules of Node application.**

# Server-side rendering with PUG

In order to build or render webpages on a server, we should use a template engine. There are a couple of template engines like Handlebars, EGS, and Pug. The template engine allows us to create template files, and then fill them with data. Pug is the most commonly used template engine with Express.

Pug templates makes writing HTML code easier. It also allows us to put all kinds of variables into pug files, through which we can inject data from the backend.

For rendering the webpages of a website, we usually implement a special separate router and controller file. In the router, we usually only ever use the GET method for each route. This is all about sending back HTML pages as response. This router is usually implemented in the App file as the first router, getting engaged by the root URL:

app.use('/', viewRouter);

app.use('/api/v1/tours', tourRouter);

app.use('/api/v1/users', userRouter);

app.use('/api/v1/reviews', reviewRouter);

## Setting up template engine

We should start by setting up the Pug template engine in Express. This should usually be implemented at the beginning of the App file, right after the ‘app’ variable is defined by calling Express.

app.set('view engine', 'pug');

PUG is another NPM package that requires NPM installation, but doesn’t need to be required in any file.

npm install pug

Then we should define where the template files are stored in the file system. It is best to set it with an absolute path using ‘\_\_dirname’ variable. This would mean that we would have to place the ‘views’ folder inside any directory where our App file is located. We can also use the ‘join’ method of the Path Node module. Pug templates are called ‘views’ in Express. All template files should be stored on a ‘views’ folder, since we usually follow the MVC architecture.

app.set('views', path.join(\_\_dirname, './views'));

## Routes for rendering webpages

Routes that are assigned to rendering Pug templates are normally defined in App file. Just like other routes defined here, special routes can be defined, where the Pug template is sent back to the client as a response, in order to render the webpage on the browser. This is done by calling the ‘render’ method on the response object. The render method accepts the name of the Pug template file.

app.get('/', (*req*, *res*) => {

  res.status(200).render('base');

});

The template engine setup implemented by the ‘set’ method, directs the ‘render’ method too look for the target template file in the ‘views’ folder and nowhere else.

This route definition and the callback function declared in it, would preferably be implemented in a separate **router** file and a separate **controller** file.

### Passing variables into a Pug file

In order to pass variables into a ‘base’ template file, we can insert an object after the template name into the render method.

app.get('/', (*req*, *res*) => {

  res.status(200).render('base', {

    <variable1>: <value1>,

<variable2>: <string>,

<variable3> //defined elsewhere

  });

These variables will then be accessible in the pug file using the [buffered code](#_Buffered_code) syntax or the [interpolation](#_Interpolation) syntax.

### Passing variables into all Pug files

In order to make some variables accessible to all our Pug files, we can put that variable on the ‘locals’ property of the response object.

res.locals.<variable> = <value>;

This variable with its value is now accessible in any pug file

body

    h1= <variable>

**WARNING | This didn’t work fine last time used. The pug file didn’t recognize the variable.**

## Pug templates file structure

Usually, we have a base template, upon which all other templates will be based.

### Base

This pug template usually contains a header and a footer code that would simply repeat in all pages of the website. The header and footer code itself however, would be written in separate ‘header’ and ‘footer’ files, and those files will be **included** in the base template file. The main content of each webpage would also be built dynamically in separate files according to different routes and the data received from the server. Then the resulting template files would be included into the base template file.

### Include a Pug file into another

To include a pug file into another file, use the ‘include’ command:

include <template-name>

**NOTE | The name for template files that only serve for being included usually contains an underscore in the beginning. For instance, if we want to have a separate template file for the header code, we would call it ‘\_header.pug’ and use this name when including it in a base pug file:**

include \_header

**NOTE | if this included file is supposed to be replaced with an HTML code inside another element, then consider indentation carefully.**

<element>

  include <template-name>

### Extending a base template

The ‘extends’ feature acts kind of the opposite of ‘include’ feature. If we extend the base template file in other template files, we are, in a sense, including the base file in the other template files. So we would be able to use the base template file in every other template file. So in a template file that needs to extend the base template file we use the ‘extends’ command:

extends base

Files that extend a base template file, usually contain the HTML code for the main content of a requested page. This code should be inserted in a named block in these template files. This code would then automatically be injected into a block with the same name in the base template file.

Extending file:

extends base

block content

<Pug code>

Extended file (base):

body

    // HEADER

    include \_header

    // CONTENT

    block content // will be replaced by <Pug code>

    // FOOTER

    include \_footer

the two ‘content’ blocks will connect to each other because we have the ‘extends’ command in the extending file.

**NOTE | each file can only extend 1 other file. However, each file can have multiple named blocks. So an EXTENDING file can fill in multiple blocks of an EXTENDED file.**

**NOTE | implementing blocks like this will make it so that the whole code of the block will be replaced by the code written in the blocks of extending files. However, we can also make blocks to insert additional code in the beginning or in the end of a block.**

**Extending file:**

extends base

block append content

<Pug code>

**Extended file:**

body

    include \_header

    block content // will be replaced by <Pug code>

    include \_footer

**The pug code in the content append block will be added to the end of the content block in the extended file. We can also use ‘prepend’.**

## Pug syntax

### General syntax

Pug is a simple whitespace-sensitive syntax for writing HTML. This means that all we use to write HTML elements is their name and indentation in our code. Indentations represent the parent-child relationship between HTML elements inserted into the Pug file.

body

    h1 The Park Camper

    p This is just some text

**NOTE | To define a div element, we can ignore writing the ‘div’ element name at the beginning. Div is a standard HTML element and it can simply be written with only its class name:**

.<class>

**NOTE | to define the parent-child relationship for an element inside another empty element we can use the : sign.**

<element1>: <element2> <content>

This would translate in normal Pug syntax to:

<element1>

    <element2> <content>

This is mostly used when defining the items of an ordered or unordered list. For instance:

ul.<class>

  li: a(href='#') <content1>

  li: a(href='#') <content2>

  li: a(href='#') <content3>

### Classes

A class name should be defined right after the HTML element beginning with a dot. Multiple classes can be defined following this pattern.

<element>.<class1>.<class2>

**NOTE | in case we need to generate dynamic class names (e.g. containing numbers), we would have to declare classes using the attributes declaration syntax.**

<element>.(class=`picture-box\_\_img--${<variable>}`)

### Attributes

Attributes of an element should be defined in () right after the element with no space in between. Multiple attributes should be separated by commas.

<element>.<class>(<attribute>='<value>')

This pattern can be followed both in the Head and the Body HTML elements. For instance, in Head:

link(rel='stylesheet', href='css/style.css')

link(rel='shortcut icon', type='image/png' href='img/favicon.png')

**NOTE | these asset requests to the routes defined in the href attribute are automatically directed toward the ‘public’ folder where we have set the** [**Express static middleware**](#_Express_static_middleware) **to serve static files from.**

### Buffered code

Variables passed into a Pug file would be accessible in the Pug file using a buffered code.

body

    h1= <variable>

it is also possible to include some JavaScript code as a buffered code.

body

    h2= <variable>.toUpperCase()

as another example:

span= <variable>.<date-property>.toLocaleString('en-us', {month: 'long', year: 'numeric'})

### Un-buffered code

This is a kind of regular JavaScript code written in Pug files that don’t add anything to the output HTML code. It just produces some value that can then be used in a buffered code. Un-buffered code always starts with a dash. It can use all JavaScript syntax, even conditionals and really everything.

- const x = 9;

h2= 2 \* x

#### JavaScript conditions + Pug

To insert a JavaScript conditional syntax in Pug, again we start the conditional statement by a dash. In the next line, with one level of indentation, we insert the Pug HTML code that we want to be rendered if that condition returns true.

- if(<variable> === <value1>)

    <el>.<class> <content1>

- if(<variable > === <value1>)

    <el>.<class> <content2>

### Pug conditions

Pug conditions are very simple conditions, and complex things cannot be done with them. So when we need simple conditions we can use them. For instance, if we want to check if a variable is defined we can use:

    if <variable>

      a.nav\_\_el.nav\_\_el--logout Log out

    else

      a.nav\_\_el(href='/login') Log in

### Interpolation

This is usually used to combine a regular HTML code with another input coming from a variable passed into the Pug file.

head

    title Natours | #{<variable>}

### Pug loops

When a variable (e.g. tours), passed into a Pug template file through a ‘render’ method, contains an array of values, we can use Pug loops to loop over that array and render a certain HTML code for each element.

block content

  main.main

    .card-container

      each tour in tours

.card

where ‘tour’ would be each tour document in the ‘tours’ array. So with ‘tour’ we would have all the tour document fields in the database available here in the Pug file.

**Note | the code that should be generated for each iteration of the loop should be indented one level further inside the loop command.**

#### Loop with counter index

The loop can be implemented along with a zero-based counter index. So on the first iteration the index will hold zero, and it will increase by 1 for each iteration.

each <variable>, <index> in <array-variable>

  .<class>

### Template strings

Template strings also work in Pug. For instance:

<el>.<class>(<attribute>=`${<variable>.<property>}`)

Or

<el>.<class>= `${<variable1>.<property>}<string>`

### Mixins

This feature is usually used when we have to include several pieces of code, that are very similar to each other, with only some few parts varying between them. Mixins are defined like functions that receive arguments, then they should be called also like functions, by passing arguments into them.

#### Defining a mixin

We use the ‘mixin’ command, then after a space, we define the name of the mixin, then without a space, we define its arguments in (), then in the next line with indentation we write the code of the mixin.

mixin <mixin-name>(*<argument1>, <argument2>*)

  .<class>

  .overview-box\_\_detail

    span.overview-box\_\_label= <argument2>

#### Calling a mixin

To call a mixin we start by a + sign, then without a space, the mixin name, then without a space, we should pass in the arguments.

+<mixin-name>(<argument1>, <argument2>)

# NPM

## Package versioning and updates

Package version number includes 3 parts:

<major version>.<minor version>.<patch version>

Note that in the package JSON file, at the beginning of each version definition, we have a symbol:

* ~ means that for updating, we will only accept patch updates.
* ^ means that for updating, we will accept patch and minor updates.
* \* means that for updating, we will also accept major updates.

To list the outdated packages in command line:

npm outdated

# Frontend NPM packages

## 1 – Axios

Popularly used on frontend to perform HTTP requests. Needs NPM installation:

npm install axios

Requires ES6 import command into a frontend JS file.

import axios from 'axios';

To make an HTTP request, we call the axios function, and pass an object into it. This object accepts a couple of properties, including the ‘data’ property which takes in an object containing the data that should be send with the request to the server. Remember that the axios function returns a promise, and its result should be awaited.

await axios({

    method: '<HTTP-method>',

    url: '<URL>',

    data: {

      <data>

    },

  });

The data passed into the axios will be automatically inserted into the request body, where it can then be received and processed by the backend code.

**NOTE | A function that uses Axios to make API calls, should usually be wrapped in a try/catch block.**

## 2 – Parcel

Used to bundle all frontend JavaScript files into one file. Requires NPM installation:

npm install parcel@1.12.3 --save-dev

we usually have to implement some scripts in the package JSON file.

"watch:js": "parcel watch <path-entry-file> --out-dir <output-directory> --out-file <output-file-name>"

Then a script tag should be added to the end of the HTML body element.

script(src='/js/<output-file-name>')

Running the script defined above in a separate terminal will run parcel, and make it watch the entry file for any changes in order to perform a re-bundle.

## 3 – Babel polyfill

Used to make some of the newer JavaScript features work in older browsers. Requires NPM installation:

npm install @babel/polyfill

Needs to be imported into a frontend JS file, usually the entry file of Parcel.

import '@babel/polyfill';

# Frontend JavaScript tips

## Reloading a page

This reload is usually performed when logging a user in or out. When logging them in, we probably want to redirect them from the login page to the homepage. This is done by calling the ‘assign’ method on the ‘location’ object. The assign method accepts a route string, which determines where the user should redirected to.

location.assign('/');

When logging users out, we probably want to just reload the page, without redirecting them to anywhere else. We usually want the current page to just reload. Passing ‘true’ into the reload method will force the reload to be performed on the server, and not on the browser cache.

location.reload(true);

# Start up a backend development workflow

Follow the steps:

1. Create a basic file structure that defines the root folder of the project.
2. Include a ‘.prettierrc’ file in order to prepare the VS code Prettier extension.
3. Create three files: ‘server.js‘, ‘app.js’, ‘config.env’
4. Initiate NPM and Install Express, dotenv, mongoose.
5. Require Express into the app file. Create the server by calling Express and store it into the app variable. Then, export the app variable to make it accessible in the server file, where we would start the server.
6. Require dotenv, app, and mongoose in the server file.
7. Make dotenv connect to the config file right at the beginning. Now all environment variables are accessible through ‘process.env’.
8. Establish a Mongo database on Atlas. Get your password and connection string and put them into the config file, along with the port number and the NODE\_ENV variable.
9. In the server file, connect the application to your database using Mongoose. Call the ‘connect’ method on Mongoose.
10. In the server file, finally start up the server by calling the ‘listen’ method on the app.
11. Start Postman, set up a new collection along with a development environment for the project. The environment would include a URL variable holding the http host address, along with the port number.
12. Create a ‘routes’ folder and create separate router files. Into these router files, require Express, then create a router in each file, and then export the router.
13. Create a ‘models’ folder and create separate model files. Into these router files, require Mongoose, in order to be able to model the data.
14. Create a ‘controllers’ folder and create separate controller files.
15. Create a global Async catch block.
16. Create a global Error class extending the native error class.
17. Create a global error handler middleware.