2. Node Module System

Global Object

Console object of *console.log()* is global object. We have other objects and functions that are globally available in node. For eg *setTimeout()*. In browser we have *window* objects that represents global scope. But in node we don't have *window* instead we have *global.* Create *app.js*

```
var message = "This is a message.";

console.log(global.message);
```

If we see in terminal we will get *undefined* because variable *message* has scope only in *app.js.*

Every file in node application is considered *module.* The variables or functions defined in that module are within the scope of that file. To make it publicly available outside that module, we should export it.

Every node application has at least one file or one module which we call the main module. If we *console.log(module)* we see

Module {

id: '.',

exports: {},

parent: null,

filename: '/home/surya/Documents/Node/Node Module System/app.js',

loaded: false,

children: [],

paths:

['/home/surya/Documents/Node/Node Module System/node_modules',

'/home/surya/Documents/Node/node_modules',

'/home/surya/Documents/node_modules',

'/home/surya/node_modules',

'/home/node_modules',

'/node_modules'] }

Now create a new module to this application add *logger.js.* In above we saw *exports* as an empty object.

Loading a Module

To load a module, we need a require function. In *logger.js:*

```
var url = "http://mylogger.io/log";

function log(message) {
    console.log(message);
}

module.exports.log = log;
```

In *app.js:*

```
const logger = require("./logger");

logger.log("Message");
```

Here, instead of exporting an object from a module, we wanna export only a single function.

```
module.exports = log;
```

Now in *app.js, logger* is no longer an object. It is a function.

```
logger("Message");
```

Press f2 to rename all at once in VSCode.

Module wrapper function

Lets make a syntactical error

```
var error = ;
```

In console, we will see

(function (exports, require, module, __filename, __dirname) { var error = ;

Node doesn't execute the code in *app.js* directly. It wraps it inside of a function

Path module

```
const path = require("path");
```

Here *path* is an object with bunch of useful methods.

```
const pathObj = path.parse(__filename);

console.log(pathObj);
```

In console:

```
{ root: '/',
  dir: '/home/surya/Documents/Node/Node Module System',
  base: 'app.js',
  ext: '.js',
  name: 'app' }
```

File System Module

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

Synchronous approach:

```
const file = fs.readdirSync("./");
console.log(file);
```

Asynchronous approach:

```
fs.readdir("./", (err, res) => {
      if (err) console.log(erdr);
      else console.log(res);
});
```

We can name argument anyway we like:

```
fs.readdir("./", (error, files) => {
      if (error) console.log(error);
      else console.log(files);
});
```

Always prefer to use Asynchronous method.

Event

A signal that indicates something has happened. Several classes in node raises different type of events.

```
const EventEmitter = require("events");
```

Here *EventEmitter* starting is uppercase because this is a convention that indicates this is a class. Class is a container for related methods and properties.

```
const emitter = new EventEmitter();

// register a listener
emitter.on("message", () => {
    console.log("Listener called");
});

// Raise an event
emitter.emit("message");
```

Here, the order is important. If we register after calling the emit method, when we call the *emit* method, the emitter iterates over all the registered listeners and calls them synchronously.

Event Arguments

```
emitter.on("message", e => {
    console.log("Listener called", e);
});

emitter.emit("message", { id: 1, url: "http://" });
```

Extending EventEmitter

Instead of using *emitter* object directly, we will create a class that has all the capability of *EventEmitter*. Here we will raise event in *logger* and will listen to this event in *app*

app.js:

```
const EventEmitter = require("events");
const emitter = new EventEmitter();
// register a listener
emitter.on("message", e => {
    console.log("Listener called", e);
});
const log = require("./logger");
log("message");
```

logger.js:

```
const EventEmitter = require("events");
const emitter = new EventEmitter();
var url = "http://mylogger.io/log";
function log(message) {
    console.log(message);
    // Raise an event
    emitter.emit("message", { id: 1, url: "http://" });
}
module.exports = log;
```

Now when we run this application, we will only see *message* in the console or the event listener will not be called. Because here we are working with two different *EventEmitters* object each in *app* and *logger*. Here event listener is only registered to *EventEmitter* object of *app.js*. So we will create a class that has all the capabilities of *EventEmitter* and additional capabilities

app.js

```
const Logger = require("./logger");
const logger = new Logger();
logger.on("message", e => {
    console.log("Listener called", e);
});
logger.log("message");
```

logger.js

```
const EventEmitter = require("events");
class Logger extends EventEmitter {
    log(message) {
        console.log(message);
        // Raise an event
        this.emit("message", { id: 1, url: "http://" });
    }
}
module.exports = Logger;
```

HTTP Module

HTTP Module is for creating networking applications. Now with ***http.createServer()*** we can create a web server. This server is an *event-emitter*.

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

const server = http.createServer();

server.on("connection", (socket) => {
     console.log("New connection");
});

server.listen(3000);

console.log("Listening to port 3000");
```

But in real world scenario, we won't be using ***connection*** event. We pass callback function to ***createServer()*** method.

```
const server = http.createServer((req, res) => {
     if (req.url == "/") {
          res.write("<h1> Hello World </h1>");
          res.end();
     }

     if (req.url == "/api/courses") {
          res.write(JSON.stringify([1, 2, 3]));
          res.end();
     }
```

Again in real world, we are not going to use http module to build back-end service for our application. Instead we use framework called **Express**. Express is build in top of **http** module in node.

3. Node Package Manager

Install a specific version of npm:

npm i for install ***-g*** for global ***npm@5.5.1***

Before we add any node packages to our application, we need to create a file that is called *package.json* which is basically a json file that includes some basic information about our project or metadata. To create *package.json:* ***npm init***
It will ask to fill some properties with default value.

There is a faster way to create *package.json* by ***npm init –yes***

Installing a package

Go to www.npmjs.com to get some information about the packages we want to install. For example search for underscore in npm homepage and theres the information about installation command, current version, collaborators, github repository, etc. In terminal ***npm i*** shortcut for ***underscore***

Now when we run this, in *package.json* we see a new property "*dependencies*" and under we can see *underscore.* So in *package.json* we specify all the dependencies of our project and their version. When we run ***npm install*** or ***i*** npm is going to download the latest version from the registry and it will store it inside a folder called *node_modules* inside we have *underscore* folder where one of the files is *package.json* where we can find some info or metadata about this package. In older version of npm we need to explicitly contain *–**save*** flag to list this package in dependencies of our project's *package.json.* But in recent version of npm we don't need this.

Using a package

In node modules there are other various folders other than the package we installed because the package comes with dependencies. If a package uses different version of same dependency then that dependency will be installed locally on that package. For example our project uses *async v-1* and *mongoose* uses *async v-2,* then *async v-2* will be installed locally, ie mongoose folder.

Source Control

Delete *node_modules* folder. But we can restore from terminal by ***npm i*** because npm looks in our *package.json* file for all dependencies and restores it. So we can ignore these files for taking memory space from our projects. We can exclude this *node_modules* folder from *git.* Initialize git repository in our projects folder by ***git init*** and if we ***git status*** we will see files that we need to add to git repository. In root folder of our project create *.gitignore* where we can list all the file and folders we need to exclude.

.gitignore

```
node_modules/
```

Now *git status* will exclude *node_modules.* To add all ***git add .*** And ***git commit -m "Our first commit"***

Semantic Versioning

In *package.json* in *"dependencies"* we will get version of packages with *caret* or "^" character. In semantic versioning, the version of node package has three components like
4.13.6 //Major.Minor.Patch where *Patch* is used for bug fixes. Like if we find a but in 4.13.6, the developers will fix the bug and release as 4.13.7. Minor version is used for adding new features that don't break the existing API. For eg in *mongoose* if the developers team add a new feature that could potentially break the existing applications that depends upon this version of this *mongoose* then they will increase the major version. Now this *caret* character tells *npm* that we are interested in any version of *mongoose* as long as the major version is same. Instead of *caret* we can see *tilda* or ~ referring that we are interested in any *patch* version as long as *major* and *minor* is same.

Listing the installed packages

From above we can conclude that there may be difference in version of package in *package.json* and actual installed package. To see this we can run ***npm list*** so we see all dependencies and their dependencies. If we want to list dependencies of only our application then ***npm list –depth=0 .*** In *npmjs.com* we can see dependencies of package or *npm view package_name* for eg
npm view mongoose where we can see *package.json* file of *mongoose* and there is list of *dependencies.* We can see dependencies directly ***npm view mongoose dependencies.*** To see the history of versions of a package ***npm view mongoose versions.***

Updating the local packages

There might be newer versions of dependencies so we can quickly find out what packages have been outdated and what are the new versions ***npm outdated*** there is *current*, ***wanted*** is latest version *caret* allow to use, ***latest*** is latest version available, ***npm update*** will update to wanted version. To updated to latest version available we need to install ***ncu*** first ***npm i -g npm-check-updates*** Now run ***npm-check-updates*** we will see all outdated packages and their new versions. To update run ***ncu*** (*a shorthand*) ***-u***(*upgrade*) to upgrade *package.json* to latest release of dependencies. Now to upgrade packages according to *package.json*, ***npm i***

Dev Dependencies

Sometimes we have dependencies that we only need during development. Like tools for running *unit tests*, for running analysis on our code, bundling our JS code and should not go in production environment. Lets install *jsHint* ***npm i jshint –save-dev*** Now in *package.json* we will get new options "*devDependencies*".

Uninstall a Package

npm un*(for uninstall)* ***mongoose***

4. Restful API's Using Express

Restful Services

Communication between ***Client*** and ***Server*** using HTTP protocol. REST stands for **Re**presentational **S**tate **T**ransfer. It is convention for building these HTTP services which use simple HTTP protocol principles to provide support to *Create*, *Read*, *Update* and *Delete* data. We refer to these operations as **CRUD** operations. HTTP methods: ***GET*** for getting data, ***POST*** for creating data, ***PUT*** for updating data, ***DELETE*** for deleting data. For example

GET: */api/customers* Getting all list of customers

GET: */api/customers/1* Getting a specific customer with id

PUT: */api/customers/1* Updating a specific customer with body of request by referencing *id*

DELETE: */api/customers/1* Deleting a specific customer by referencing *id*

POST: */api/customers* Creating a customer with body of request

Using Express

npm i express

Create *index.js*

```
const express = require("express");
```
It returns a function *express()*

```
const app = express();
```
app is an object of type express; *app* represents our application; *app* has bunch of useful methods;

app.get(), app.post(), app.put() and ***app.delete()***

app.get() has two arguments first argument is path or url. Second is callback function which have two arguments *req* and *res; req* object has properties that gives us information about the incoming request.

```
app.get("/", (req, res) => {
    res.send("Hello World");
});
```
The callback function is also called the route handler.

To avoid stopping and starting on terminal every-time we change the code, we use *nodemon* which is shorthand for *node monitor* ***npm i -g nodemon***

```
const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`Listening to port ${port}...`));
```
To set an environment variable ***export PORT=5000*** in terminal.

Route Parameters

```
app.get("/api/courses/:id")
```
Here *id* is a parameter. We can give any name but by convention we use *id*. In order to read this parameter, we use *req.params.id* `res.send(req.params.id);`

It is also possible to have multiple parameters in the route. Like */api/posts/:year/:month* to list all the post of given month and year.

We can also get querystring parameters. For example we want to sort by name to all the post in 2018 January.

```
app.get("/api/posts/:year/:month", (req, res) => {
    res.send(req.query);
});
```

In browser: *http://localhost:3000/api/posts/2018/4?sortBy=name*

Result: *{"sortBy":"name"}*

Handling Get requests

```
const courses = [
    { id: 1, name: "Science" },
    { id: 2, name: "Math" },
    { id: 3, name: "English" }
];

app.get("/api/courses/:id", (req, res) => {
    const course = courses.find(e => e.id === parseInt(req.params.id));
    if (!course)
        res.status(404).send("The course you are searching could not be found");
    res.send(course);
});
```

To ensure the status code is 404, inspect and in the network click *all* and refresh the page.

Handling Post request

```
app.post("/api/courses", (req, res) => {
// posting to collection of courses
    const course = {
        id: courses.length + 1,
        name: req.body.name // for this line to work, we need to enable parsing of JSON
                                object in body of request
    };
    courses.push(course);
    res.send(course); // client needs to know the id of the object.
});
```

To take post request, we are adding a piece of middleware

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

Input validation

```
app.post("/api/courses", (req, res) => {
    if (!req.body.name || req.body.name.length < 3) {
        //400 is Restful convention of bad request
        res.status(400).send("Name is required and should be minimum 3 character");
        return; //if error then we don't want rest of the function to be executed.
    }
```

However, in real world application, working with complex objects, we use Node package ***npm i joi***

```
const Joi = require("joi"); // Pascal naming because this module returns class.
```

With *Joi* we need a schema. Schema defines shape of our object.

```
app.post("/api/courses", (req, res) => {
    const schema = Joi.object({ name: Joi.string().min(3).max(30).required() });
    const result = schema.validate(req.body);

    if (result.error) {
        res.status(400).send(result.error);
    }
```

If we just want to show specific error result:

```
res.status(400).send(result.error.details[0].message);
```

Handeling Put Request

```
app.put("/api/courses/:id", (req, res) => {
    // Look up the course. If not existing return 404
    const course = courses.find((e) => e.id === parseInt(req.params.id));
    if (!course)
        res.status(404).send("The course you are searching could not be found");

    //Validate. If invalid, return 400-Bad request
    const result = validateCourse(req.body);

    if (result.error) {
        res.status(400).send(result.error.details[0].message);
        return;
    }

    //Update the course
    course.name = req.body.name;

    //Return the updated course
    res.send(course);
});
```

We are putting course validation in a single function.

```
function validateCourse(course) {
    const schema = Joi.object({ name: Joi.string().min(3).max(30).required() });
    return schema.validate(course);
}
```

Handling Delete Request

```
app.delete("/api/courses/:id", (req, res) => {
    // Look up the course
    // Not existing, return 404
    const course = courses.find(c => c.id === parseInt(req.params.id));
    if (!course)
        return res
        .status(404)
        .send("The course you are searching could not be found!!");
        // return because we don't want to go further if course not found.

    // Delete
    const index = courses.indexOf(course);
    courses.splice(index, 1); // We go to index and remove 1 object.

    // Return the same course
    res.send(courses);
});
```

5. Express Advanced

Middleware

Middleware is a function that takes *request* object and either returns the response to the client or passes control to another middleware function. In *Express* every route handler function(*get*, *post*) is technically middleware function. When we call *express.json()*, this method returns, a middleware function. The job of this function is to read the request and if there is a *json* object in the body of the request, it will parse the body of the request into a *json* object and will set *req.body* proprety.

Building a custom Middleware

```
// For Logging
    app.use(function(req, res, next) {
    // we call this method to install a middleware function in request processing pipeline.
    console.log("Logging...");
    next(); // The request will hang if we didnt implement this
});

//For authentication
app.use(function(req, res, next) {
    console.log("Authenticating....");
    next();
});
```

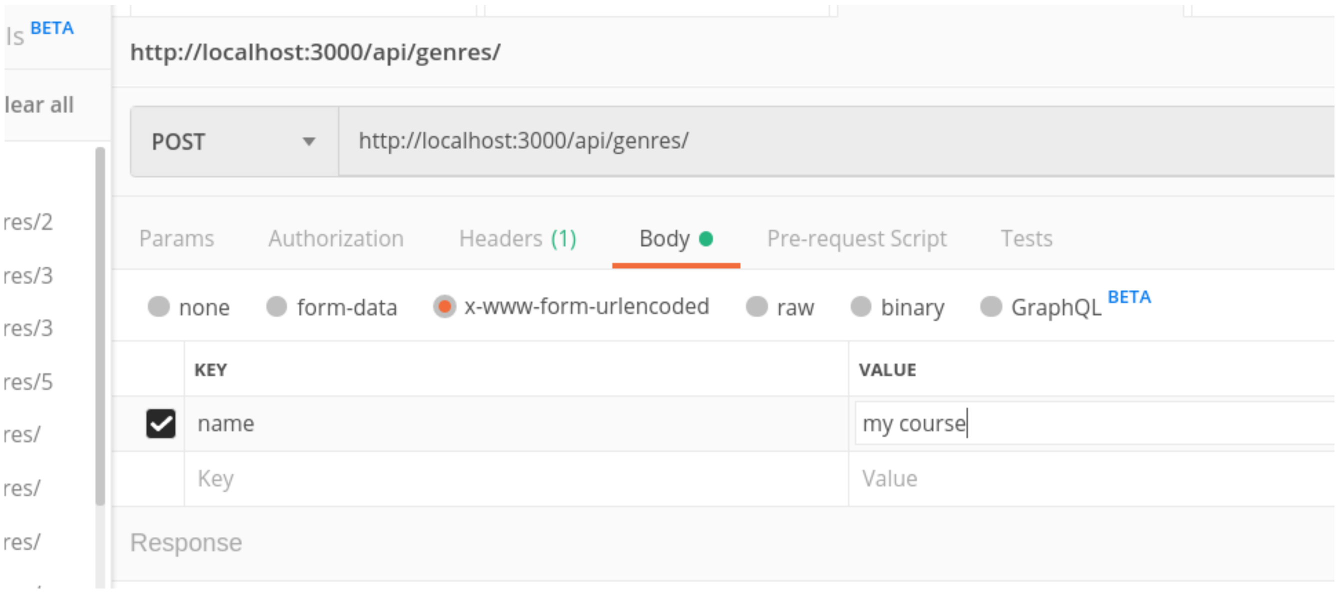
For clean code, we want to put middleware in different files. Create *logger.js*

```
function log(req, res, next) {
    console.log("Logging...");
    next();
}

module.exports = log; // Because this module exports a single function
```

Built-in Middlewares

```
app.use(express.urlencoded());
```

It parses incoming requests with url-encoded payloads. Lets send a post request in Postman.

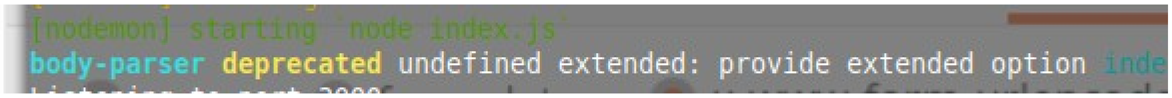


http://localhost:3000/api/genres/

POST http://localhost:3000/api/genres/

KEY	VALUE
✓ name	my course
Key	Value

If we look in the terminal:



```
[nodemon] starting `node index.js`
body-parser deprecated undefined extended: provide extended option inde
```

For this:

```
app.use(express.urlencoded({ extended: true }));
```

Another built-in middleware:

```
app.use(express.static("public"));
```

We are putting all our static assets like *css, images* inside this folder. Create file *public/readme.txt*
Now with this middleware function we can go to browser with url: *localhost:3000/readme.txt* to get the text. So here, static content are served from the root of the site.

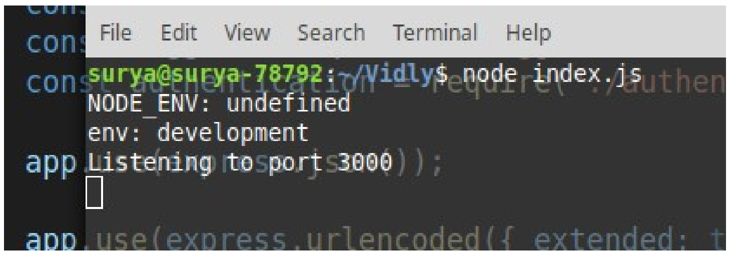
Environments

Based on which environment we are working upon(development, testing, staging, production), we need to enable or disable certain features; ***process*** is a global object in node which gives us access to the current process. This has property called ***env*** which gives us the environment variable. ***NODE_ENV*** returns the environment for this node application. To see which environment we are in:

```
console.log(`NODE_ENV: ${process.env.NODE_ENV}`);
```

We have another way to get the current environment. In this, if we haven't set environment variable, it will be *development* by default.

```
console.log(`env: ${app.get("env")}`);
```



```
surya@surya-78792:~/Vidly$ node index.js
NODE_ENV: undefined
env: development
Listening to port 3000
```

If we want logging of HTTP request only in development (not in production, testing,..) we can:

```
if (app.get("env") === "development") {
    app.use(morgan("tiny"));
    console.log("Morgan enabled...");
}
```

To set environment to production, in terminal: ***export NODE_ENV=production***

Configuration

We will set configuration setting and overwrite them in each environment. Package ***npm rc*** is popular for managing configuration. But better package will be ***npm i config***

Create a file called *config/default.json* where we have configuration for default settings.

```
{
    "name": "My Express Application"
}
```

We can overwrite the default configuration in development from *development.json*

```
{
    "name": "My Express App - Development",
    "mail": {
        "host": "dev-mail-server"
    }
}
```

And other config files like *production.json*, etc.

In *index:* `const config = require("config");`

To get the settings:

```
console.log(`Application Name: ${config.get("name")}`);
```

```
console.log(`Mail Server: ${config.get("mail.host")}`);
```

In terminal: ***export NODE_ENV=development.*** Now *development.json* will overwrite the settings.

Now: we should not store the application secrets like *passwords*, in these configuration files. We store them in environment variables. In terminal, define environment variable for storing password of a mail server: ***export password=1234.*** To prevent clashing of environment variables, prefix with name of our application: ***export app_password=1234***

Create *config/custom-environment-variables.json* where we define the mapping of configuration settings to environment variables.

```
{
    "mail": {
        "password": "app_password"
    }
}
```

In *index:* `console.log(`Mail Password: ${config.get("mail.password")}`);`

config.get() looks at various sources to get value for this configuration.

*Please read the documentation https://www.npmjs.com/package/config in detail.

Debugging

We will look for a replacement for *console.log()* for debugging and other helpful functionalities. ***npm i debug*** And we will use environment variable to enable debugging in development and disabling it in production. We can also customize the level of debugging information.

The ***require*** function returns a function we call this function by giving it an argument which is an arbitrary namespace that we define for debugging.

```
const startupDebugger = require("debug")("app:startup");
```

The second bracket in *require* function is giving argument to this function. We can give any name to this argument.

We can have another debugger for debugging database related messages:

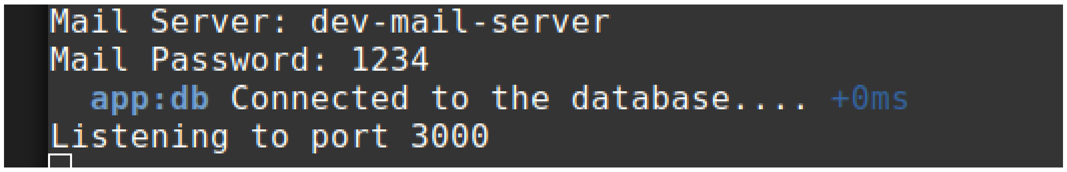
```
const dbDebugger = require("debug")("app:db");
```

We can use *debugger* like:

```
//LOGGING FOR DATABASE WORK
dbDebugger("Connected to the database....");
```

Now in terminal, we use environment variable to determine what kind of debugging info we wanna see in the console. Terminal: ***DEBUG=app:db***

Now we will see only the debugging messages that are part of the namespace *app:db*



```
Mail Server: dev-mail-server
Mail Password: 1234
  app:db Connected to the database.... +0ms
Listening to port 3000
```

Or we want to see debugging messages of both **db** and **startup**: ***export DEBUG=app:startup,app:db***

Or we will register all types of debug in terminal: ***export DEBUG=app:*.*** It color codes debugger of different namespaces. We can set the environment variable at the time of running the application so we wont have to explicitly use ***export*** command:

DEBUG=app:db nodemon index.js

In real world scenario, multiple debugging functions are not required in the same file or module. So we can change the name of *dbdebugger* to just *debug* because in file related to database, we only work on databases.

```
const debug = require("debug")("app:startup");
if (app.get("env") === "development") {
    debug("Morgan Enabled....");
}
```

Templating Engines

There are various templating engine for express applications. Popular ones are **Pug, Mustache, EJS.**
Install pug: ***npm i pug***

We need to set the view engine. So in *index.js*

```
app.set("view engine", "pug");
```

In this way express will internally load this module ie. we don't need ***require*** syntax to load module.
Optional: If we want to overwrite path to a template: `app.set("views", "./views"); //Default path`

Create /*views/index.pug*. Inside *index.pug:*

```
html
    head
        title= title
    body
        h1= message
```

Pug will convert the above syntax to html markup.

```
app.get("/", (req, res) => {
    res.render("index", {
        title: "My express App",
        message: "Hello, This is the message",
    });
});
```

The first argument of ***render*** is the name of our view ie *index.pug*. Second argument is an object with values for the parameters we have defined in that template.

Structuring Express Application

Take out all routes that include */api/genres* and include them in *routes/genres.js*. When we separate the route in separate module,

```
const express = require("express");
const router = express.Router();
```

Then rename all *app* to *route*

```
module.exports = router;
```

In index

```
const genres = require("./routes/genres");
```

and

```
app.use("/api/genres", genres);
```

This code handles *.../api/genres/* part of url so in *genres.js* we could eliminate this part.

6. Asynchronous Javascript

Sync vs Async

```
console.log("Before");
setTimeout(() => {
    console.log("Reading a user from a database...");
}, 2000);
console.log("After");
```

Async Patterns

Put the *timeout* in a different function.

```

function getUser(id) {
    setTimeout(() => {
        console.log("Reading a user form the database. ");
        return { id: id, gitHubUsername: "Mosh" };
    }, 1000);
}
```

Now when we call this function,

```
console.log("before");
const user = getUser(1);
console.log(user);
console.log("after");
```

We will get *undefined* in the console. Because the arrow function we pass to ***setTimeout*** is executed 2 seconds after.

Callbacks

```
function getUser(id, callback) {
    setTimeout(() => {
        console.log("Reading user from a database...");
        callback {id: id, gitHubUsername = "Mosh"} // we call back with this result
    }, 2000);
}
```

callback is a function that, we call when result of an async operation is ready.

Now ***getUser*** needs a second argument, that is function that need to be called ***function(user)*** because we are passing ***user*** object in *callback*. So we can display ***user*** on console.

```
console.log("Before");
    getUser(1, user => {
    console.log("User", user);
});
console.log("After");
```

```
function getUser(id, callback) {
    setTimeout(() => {
        console.log("Reading data from a database...");
        callback({ id: id, gitHubUsername: "Mosh" });
    }, 2000);
}
```

Callback Hell

Callback inside callback:

```
console.log("Before");
    getUser(1, user => {
        getRepositories(user.gitHubUsername, gitUser => {
        console.log("Github Username:", user.gitHubUsername, gitUser);
    });
});
console.log("After");
```

```
function getUser(id, callback) {
    setTimeout(() => {
        console.log("Reading data from a database...");
        callback({ id: id, gitHubUsername: "Mosh" });
    }, 2000);
}
```

```
function getRepositories(username, callback) {
    setTimeout(() => {
        callback(["repo1", "repo2", "repo3"]);
    }, 2000);
}
```

Nested Callback

```
getUser(1, user => {
    getRepositories(user.gitHubUsername, gitUser => {
        getCommites(repo, commits => {
            console.log(commits);
        });
    });
});
```

We can refer this as ***Callback Hell***

If this was synchronous

```
const user = getUser(1);
const repo = getRepositories(user.gitHubUsername);
const commit = getCommites(repo[0]);
console.log(commit);
```

One simple solution to *callback hell* is ***Named Function***

Promises

Promise holds the eventual result of an asynchronous operation. It promises that it will either give value or error. It has three state: *Pending(is in async operation), Fulfilled, Rejected.*

Create *promise.js*

```
const p = new Promise((resolve, reject) => {
    // Promise is a Constructor function
    //Kick off some async work
    // .....
    resolve(1);
    reject(new Error("Message"));
});
```

Resolve and *Reject* are functions. We can call *resolve* to pass a value which is result of async operation. Alternatively we use *reject* to return error, where we pass error object.

Now somewhere we will consume this promise. We have two methods for ***p*** object, ***catch*** and ***then.*** *Catch* for catching any errors and *then* for getting the result.

```
p.then(result => {
    console.log("Result: ", result);
});
```

Async in promise:

```
const p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(1);
    }, 2000);
    // reject(new Error("Message"));
});
```

We can also chain *catch* with *then*

```
const p = new Promise((resolve, reject) => {
    setTimeout(() => {
        // resolve(1);
        reject(new Error("This is error message"));
    }, 2000);
});

p
    .then(result => console.log("Result: ", result))
    .catch(err => console.log("Error: ", err.message)
    // Each error object that we have in JS has message property
```

Replacing Callbacks with Promises

```
function getUser(id) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Reading data from a database...");
            resolve({ id: id, gitHubUsername: "Mosh" });
        }, 1000);
    });
}
```

```
function getRepositories(username) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(["repo1", "repo2", "repo3"]);
        }, 1000);
    });
}
```

```
function getCommits(repo) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(["commit1"]);
        }, 1000);
    });
}
```

Consuming Promise

```
const p = getUser(1);
p.then(user => console.log(user));
```

or

```
getUser(1).then(user => console.log(user));
```

So nested callback becomes:

```
getUser(1)
    .then(user => getRepositories(user.gitHubUsername))
    .then(repos => getCommits(repos[0]))
    .then(commit => console.log(commit));
    .catch(err => console.log("Error", err));
```

Now it will catch error for any of these promises.

Creating Settled Promises

Create *promise-api.js* which will be a playground file. Promise that is already resolved is particularly when writing a unit test.

```
const p = Promise.resolve(1);
p.then(result => console.log(result));
```

or

```
const p = Promise.resolve({ id: 1 });
```

Sometimes we wanna create a Promise that is already rejected.

```
const p = Promise.reject(new Error("Reason for rejection:"));
p.catch(error => console.log(error));
```

When we call reject with *Error* object we will get a call-stack as return.

Parallel Promises

All is a method of Promise class (not object) where we give array of promises. This method will return a new Promise that will be resolved when all the promises are resolved.

```
const p1 = new Promise(resolve => {
    setTimeout(() => {
        console.log("Async operation 1...");
        resolve(1);
    }, 1000);
});
```

```
const p2 = new Promise(resolve => {
    setTimeout(() => {
        console.log("Async operation 2...");
        resolve(1);
    }, 1000);
});
```

```
Promise.all([p1, p2]).then(result => {
    console.log(result);
});
```

If one Promise in array is rejected then the final Promise is also considered rejected.

```
const p1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        console.log("Async operation 1...");
        reject(new Error("Because something failed"));
    }, 1000);
});
```

```
Promise.all([p1, p2])
    .then(result => {
console.log(result);
})
    .catch(error => console.log("Error:", error.message));
```

If we wanna do something as soon as first async operation completes. Here when 1 promise is fulfilled the final Promise is considered fulfilled.

```
Promise.race([p1, p2]).then((result) => console.log(result));
```

Async and Await

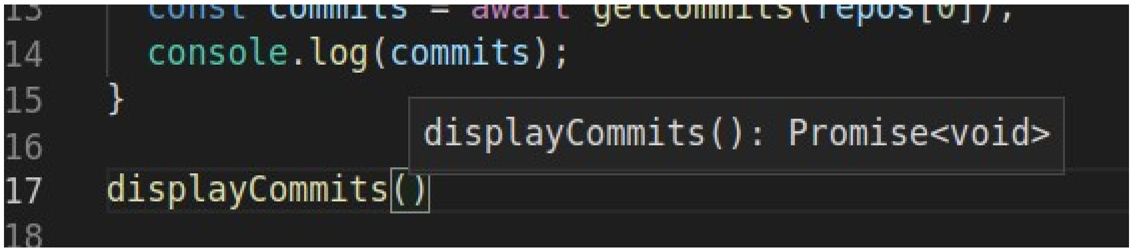
It helps us like Async code like Sync code. Anytime we’re calling a function that returns a Promise, we can await the result of that function and get the actual result just like calling a synchronous function.

```
const user = await getUser(1);
const repos = await getRepositories(user.gitHubUsername);
const commits = await getCommits(repos[0]);
console.log(commits);
```

Whenever we use *await* operator in a function, we need to decorate the function with *async* modifier.

```
async function displayCommits() {
    const user = await getUser(1);
    const repos = await getRepositories(user.gitHubUsername);
    const commits = await getCommits(repos[0]);
    console.log(commits);
}
```

```
displayCommits()
```



```
    const commits = await getCommits(repos[0]);
    console.log(commits);
}

displayCommits()
```

displayCommits(): Promise<void>

This function is returning a promise of *void* that means the promise once fulfilled doesn’t result in a value. Its telling us that *async and await* approach are built on top of promises.

When we use *async and await* we don’t have catch method. The way we get the errors is using *try-catch* block.

```
async function displayCommits() {
    try {
        const user = await getUser(1);
        const repos = await getRepositories(user.gitHubUsername);
        const commits = await getCommits(repos[0]);
        console.log(commits);
    } catch (err) {
        console.log("Error:", err.message);
    }
}
```

Exercise

Before:

```
getCustomer(1, (customer) => {
    console.log('Customer: ', customer);
    if (customer.isGold) {
        getTopMovies((movies) => {
            console.log('Top movies: ', movies);
            sendEmail(customer.email, movies, () => {
                console.log('Email sent...')
            });
        });
    }
});
```

```
function getCustomer(id, callback) {
    setTimeout(() => {
        callback({
            id: 1,
            name: 'Mosh Hamedani',
            isGold: true,
            email: 'email'
        });
    }, 1000);
}
```

```
function getTopMovies(callback) {
    setTimeout(() => {
        callback(['movie1', 'movie2']);
    }, 1000);
}
```

```
function sendEmail(email, movies, callback) {
    setTimeout(() => {
        callback();
    }, 1000);
}
```

After:

```
async function displayCustomer() {
    try {
        const costumer = await getCustomer(1);
        console.log("Customer", costumer);
        if (costumer.isGold) {
            const topMovies = await getTopMovies();
            console.log("Top movies:", topMovies);
            await sendEmail(costumer.email, topMovies);
            console.log("Email sent");
        }
    } catch (err) {
        console.log(err.message);
    }
}
displayCustomer();
```

```
function getCustomer(id) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve({
                id: 1,
                name: "Mosh Hamedani",
                isGold: true,
                email: "email"
            });
        }, 1000);
    });
}
```

```
function getTopMovies() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(["movie1", "movie2"]);
        }, 1000);
    });
}
```

```
function sendEmail(email, movies) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve();
        }, 1000);
    });
}
```

7. MongoDB CRUD Application

First go to the official website of **mongodb** and see the installation process for your operating system. After mongodb is installed, install node package **mongoose: *npm i mongoose***. It gives simple API to work with mongodb database.

When querying a data, we will get JSON object form MongoDB.

Create *index.js*

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

    mongoose.connect("mongodb://localhost/playground") // This is for development
    environment. We will use different connection string for production environment.
    .then(() => console.log("Connected to the databse."))
    .catch(err => console.log("Could not connect to databse", err));
```

In real application, the connection string should come from a configuration file. Mongodb will automatically create **playground** database for if there isn't already. This returns a promise.

In *Mongoose*, we have a concept called schema which is specific to *mongoose (not mongoDB).* We use Schema to define shape of documents (rows) in MongoDB collection (table). To create a schema:

```
const courseSchema = new mongoose.Schema({
    name: String,
    author: String,
    tags: [String], // Array of strings type
    date: {
        type: Date,
        default: Date.now
    },
    isPublished: Boolean
});
```

List of types we can use when creating a Schema are: **String, Number, Date, Buffer** used for storing binary data**, Boolean, ObjectID** used for asigning unit identifiers**, Array**

Models

```
const Course = mongoose.model("Course", courseSchema);
```

Here *Course* is a class from which we can create objects based on this class. Here we are using *Pascal Case* for *Class* and *Camel Case* for *Object.*

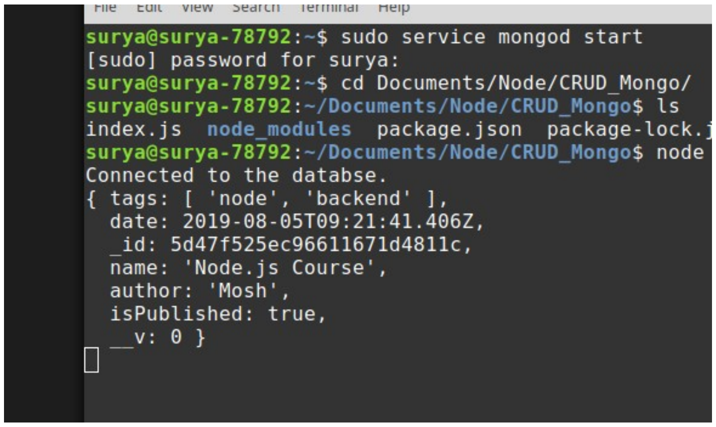
```
const course = new Course({
    name: "Node.js Course",
    author: "Mosh",
    tags: ["node", "backend"],
    // Not setting date because it has default value
    isPublished: true
});
```

```
course.save();
```

With *save()* we are dealing with *async* operation which returns promise. So we can await it.

```
const result = await course.save();
console.log(result);
```

This ***result*** is the actual course object, that is saved in the database. When using **await,** the code should be inside of an async function.



```
surya@surya-78792:~$ sudo service mongod start
[sudo] password for surya:
surya@surya-78792:~$ cd Documents/Node/CRUD_Mongo/
surya@surya-78792:~/Documents/Node/CRUD_Mongo$ ls
index.js  node_modules  package.json  package-lock.j
surya@surya-78792:~/Documents/Node/CRUD_Mongo$ node
Connected to the databse.
{ tags: [ 'node', 'backend' ],
  date: 2019-08-05T09:21:41.406Z,
  _id: 5d47f525ec96611671d4811c,
  name: 'Node.js Course',
  author: 'Mosh',
  isPublished: true,
  __v: 0 }
```

Now create another document:

```
const course = new Course({
    name: "Angular Course",
    author: "Mosh",
    tags: ["Angular", "Front-End"],
    isPublished: true
});
```

Querying Documents

```
async function getCourses() {
    const courses = await Course.find();
    console.log(courses);
}

getCourses();
```

We can use filter in **find()**:

```
const courses = await Course.find({ author: "Mosh", isPublished: true });
```

Other methods

```
const courses = await Course.find({ author: "Mosh", isPublished: true })
    .limit(10)
    .sort({ name: 1 }) // Sorting the documents by their name and 1 for ascending, -1 for
    descending
    .select({ name: 1, tags: 1 }); // Only want name and tags to show
    console.log(courses);
```

Comparison Query Operators

eq : equal, **ne:** not equal, **gt:** greater than, **gte:** greater than or equal, **lt:** less than, **lte, in, nin** not in

Lets suppose we wanna get all courses that are more than 10 dollars.

```
Course.find({ price: { $gt: 10 } })
```

With key, value of an object we cannot express the concept of greater than or smaller. So we need to pass an object in place of value. This object is again container for key, value pairs. The **$** sign indicates that we are working with an operator.

```
const courses = await Course.find({ price: { $gte: 10, $lte: 20 } })
```

Courses that are between 10 and 20 dollar. If we want courses that are either $10, $15, $20:

```
const courses = await Course.find({ price: { $in: [10, 15, 20] } })
```

Logical Query Operators

Courses authored by Mosh **OR** courses that are published. We use **find()** without any filters.

```
.find().or([{ author: "Mosh" }, { isPublished: true }])
```

The **and** logical operator is exactly the same.

Regular Expressions

```
// Starts with "Mosh"
.find({ author: /^Mosh/ })

//Ends with "Hamedani"
.find({ author: /Hamedani$/i }) // "i" if we want case insensitive

//Can contain "Mosh" anywhere
.find({ author: /.*Mosh.*/ });
```

To count:

```
const courses = await Course.find()
.or([{ author: "Mosh" }, { isPublished: true }])
.limit(10)
.sort({ name: 1 })
.select({ name: 1, tags: 1 })
.count();
```

It counts the documents which matches its criteria.

Pagination

In a real world application, we pass this values as querystring parameters for restful APIs.

```
/api/courses?pageNumber=2&pageSize=10
```

Here we use **skip** method to implement pagination in console. In order to implement pagination, we need to skip all the documents in the previous pages.

```
async function getCourses() {
    const pageNumber = 2; // Hardcoded just for reference.
    const pageSize = 10;

    const courses = await Course
        .find({ author: "Mosh", isPublished: true })
        .skip((pageNumber - 1) * pageSize)
        .limit(10)
        .sort({ name: 1 })
        .select({ name: 1, tags: 1 })
        .count();
    console.log(courses);
```

Another way of using Query Operators

```
async function getCourses() {
    return await Course.find({
        isPublished: true,
        tags: { $in: ["frontend", "backend"] }
    })
    .sort("-price")
    .select("name author");
}
```

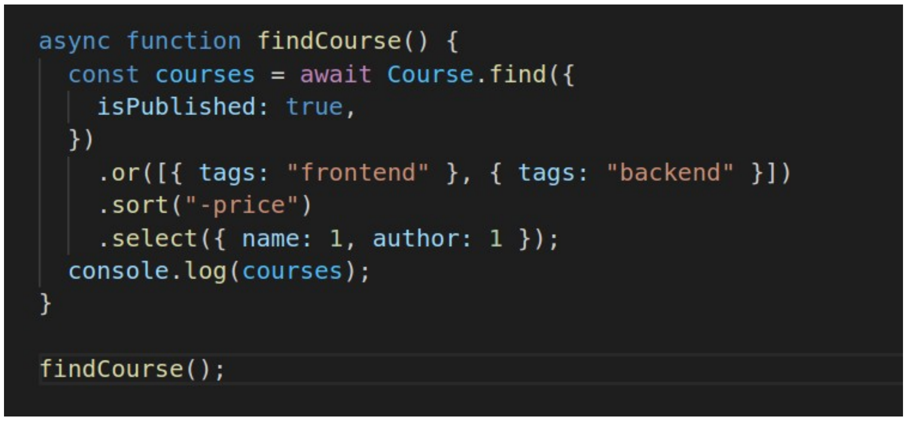
Exercise 1

Command to import JSON file into a new database**:**

mongoimport –db mongo-exercises –collection courses –file exercise-data.json --jsonArray

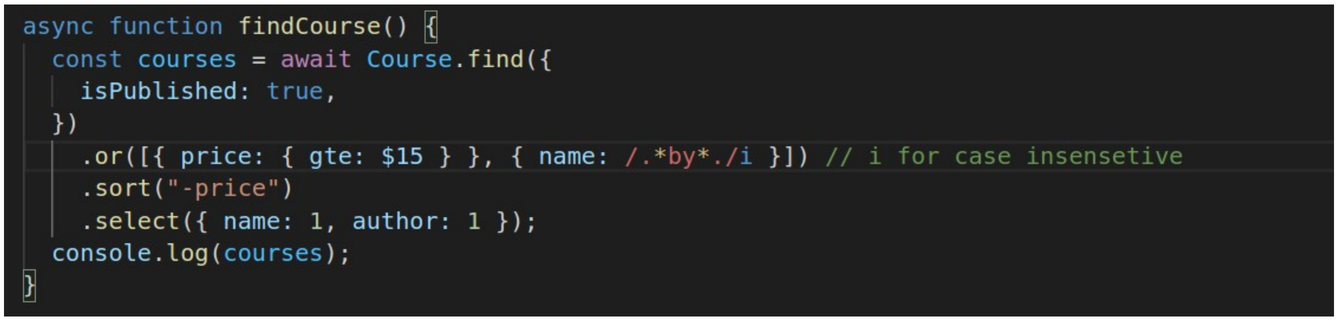
mongoimport: To import data into mongo database
flags: *--db* (specifying name of database), ***mongo-exercises***(we are using separate database for exercies), ***--collection***(we set this to *courses*), ***--file***(JSON file we want to import), ***--jsonArray***(Because the data that we have in this json file is represented using an array.)

Now getting all the published frontend orbackend courses, sorting by their price on descending order, picking only by name and author and displaying them:



```
async function findCourse() {
  const courses = await Course.find({
    isPublished: true,
  })
    .or([{ tags: "frontend" }, { tags: "backend" }])
    .sort("-price")
    .select({ name: 1, author: 1 });
  console.log(courses);
}

findCourse();
```

Now getting all the published courses that are $15 or more, or have the word ‘by’ in their title.



```
async function findCourse() {
  const courses = await Course.find({
    isPublished: true,
  })
    .or([{ price: { gte: $15 } }, { name: /.*by*./i }]) // i for case insensetive
    .sort("-price")
    .select({ name: 1, author: 1 });
  console.log(courses);
}
```

Updating Documents

There are two ways to update documents. One approach is *Query First*. Here, we find a document, then modify its properties and finally call ***save()***.

```
async function updateCourse(id) {
    const course = await Course.findById(id);
    if (!course) return;
    course.set({
        isPublished: true,
        author: "Another Author"
    });
    course.save();
}
updateCourse("12345555");
```

Or we can update **course** by direct assigning

```
course.isPublished = true;
course.author = "Another Author";
```

Now this is query first approach. This approach is useful, if we receive an input from the client and make sure update is a valid operation. But sometimes we don't wanna receive input from client and update anyways.

```
const course = await Course.update({ isPublished: true });
```

With this we can update multiple documents in one go. If we want to update course for particular id. Also we are using update operators of mongodb. Search **mongodb update operators** in google.

```
const course = await Course.update(
    { _id: id },
    {
        author: "Another Author",
        isPublished: false
    }
);
```

If we want to get document before it was updated in return then

```
const course = await Course.findByIdAndUpdate(id, {
    $set: {
        author: "Another Author",
        isPublished: false
    }
});

console.log(course);
```

Removing Documents

DeleteOne() takes filter, finds the first one fulfilling the requirement and deletes it.

```
async function removeCourse(id) {
    Course.deleteOne({ isPublished: true });
}
```

8. Mongoose Data Validation

Validation

```
const courseSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
author: String,
```

If we run, we will get error like *Unhandled Promise Rejection Warning*. We are dealing with a promise with rejection but we haven't handled rejection. So we are putting this code in **try-catch** block.

```
try {
    const result = await course.save();
    console.log(result);
} catch (err) {
    console.log(err.message);
}
```

We can also manually trigger validation.

```
try {
    await course.validate();
```

There is design flaw of monggose where **validate()** returns void. If it would have returned a Boolean then we can have conditional if valid or not valid. For this to work we could pass callback in **validate().** We have **catch** block if there was error but this will be messy.

Now the validation we added above(***required: true***) is only meaningful for **mongoose** where **MongoDB** doesn't care. It's good to have validation defined at the database level. But we have **joi** which complements **mongoose** on validation.

Built-in Validators

Required is one of the built in validators in mongoose. We can conditionally make a property required or not. Here **price** is only required if the course is published.

```
isPublished: Boolean,
price: {
    type: Number,
    required: function() {
        return this.isPublished;
    }
}
```

In this case we cannot replace this function with arrow function because there is a function somewhere in **mongoose** which will call this function. So **this** will reference that function, not the course object we are dealing here.

With string, we have **minlength, maxlength**. We also have **match** where we can pass *regular expression(**/pattern/**)*. We also have **enum** where we specify the set of valid strings.

```
category: {
    type: String,
    required: true,
    enum: ["web", "mobile", "network", ""]
},
```

Here, **enum** means category should be either **web, mobile,** or **network.** Nothing other than that

Custom Validators

When we look at the **tags** property, which is array of string. If we want to enforce that all courses should have at-least one tag, we cannot use the required validator coz we can simply pass an empty array.

```
tags: {
    type: Array,
    validate: {
        validator: function(v) {
            return v.length > 0;
        },
        message: "A course should have at least one tag."
    }
},
date: {
    type: Date,
    default: Date.now
```

Now if we exclude property **tags** when creating data:

```
const course = new Course({
    name: "Angular Course",
    author: "Mosh",
    // tags: ["Angular", "Front-End"],
    isPublished: true
});
```

We will get the same error message: *A course should have at least one tag.* Because when we are setting property **type** to **Array,** mongoose will initialize it to an empty array.

Now if we set **tags** to **null**:

```
name: "Angular Course",
author: "Mosh",
tags: null,
isPublished: true
```

Error is: *Cannot read property 'length' of null.* So we will change conditional as:

```
validator: function(v) {
    return v && v.length > 0;
},
```

Async Validators

Sometimes validation logic may involve reading something from a database or remote HTTP server where we don't have answer straight away. In this case we need **async validator**.

```
validate: {
    isAsync: true,
    validator: function(v) {

```

We will add callback as second argument of validator function.

```
validator: function(v, callback) {
    setTimeout(() => {
        const result = v && v.length > 0;
        callback(result);
    }, 2000);
},
```

Validation Errors

The exception we get in the catch block, has a property called **errors**. It has separate error property for each invalid property for **Course** object. If we enter invalid data in **tags** and **categories,** we can access like

```
ex.error.tags,
ex.error.category
```

```
try {
    await course.validate();
    } catch (ex) {
    for (field in ex.errors) console.log(ex.errors[field].message);
}
```

Now we have two validation error messages.

Schema Type Options

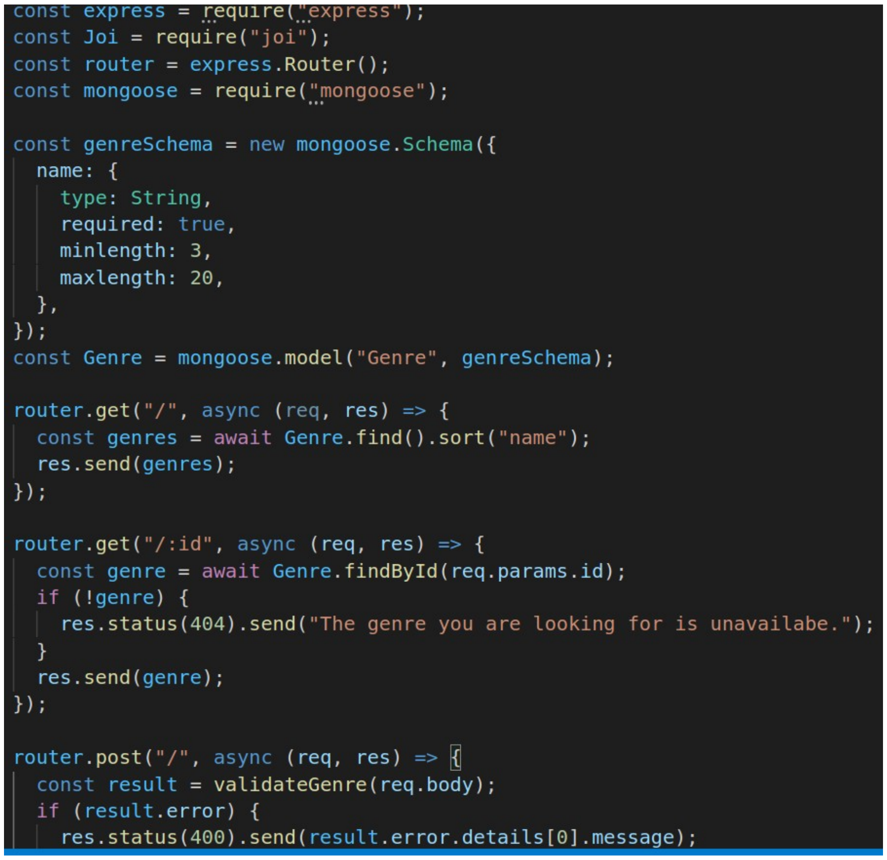
When defining a Schema, we can set the *type* of the property directly like `name: String,` or using object. This object have few properties like **type, required, enum,** etc. We have few more useful properties. For **strings** we have additional properties like **lowercase:** *true* so mongoose will automatically convert the value of this **category** property to lowercase. Similarly **uppercase, trim:** *true* which will trim the whitespaces.

We have some properties which will be implemented irrespective of its type. Like getter **get** and setter **set.** Suppose we always wanna round-off the value of the **price.** It takes arrow function with **v** as argument.

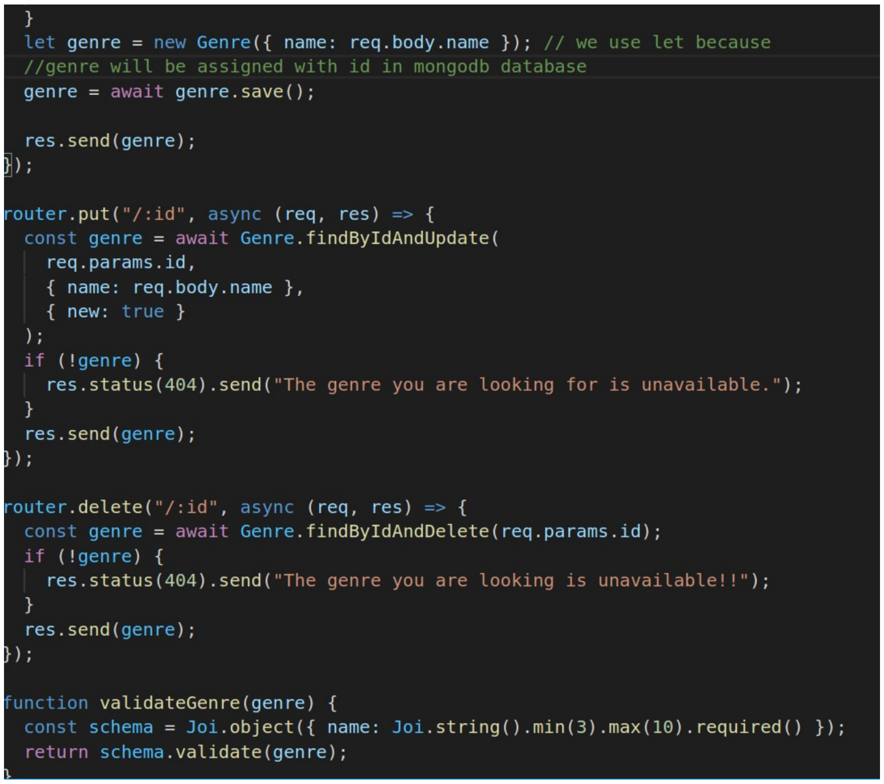
```
price: {
    type: Number,
    required: function() {
        return this.isPublished;
    },
    get: v => Math.round(v),
    set: v => Math.round(v)
}
```

When we set the **price, set** function will be called and set to rounded value to the database. If we go to database and manually change the **price** from *integer* to *double* and it’s value to *15.89*. Now when we access this data, **get** will be called and round that value.

Project: Integrate Database in Genres



```javascript
const express = require("express");
const Joi = require("joi");
const router = express.Router();
const mongoose = require("mongoose");

const genreSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 3,
    maxlength: 20,
  },
});
const Genre = mongoose.model("Genre", genreSchema);

router.get("/", async (req, res) => {
  const genres = await Genre.find().sort("name");
  res.send(genres);
});

router.get("/:id", async (req, res) => {
  const genre = await Genre.findById(req.params.id);
  if (!genre) {
    res.status(404).send("The genre you are looking for is unavailabe.");
  }
  res.send(genre);
});

router.post("/", async (req, res) => {
  const result = validateGenre(req.body);
  if (result.error) {
    res.status(400).send(result.error.details[0].message);
```

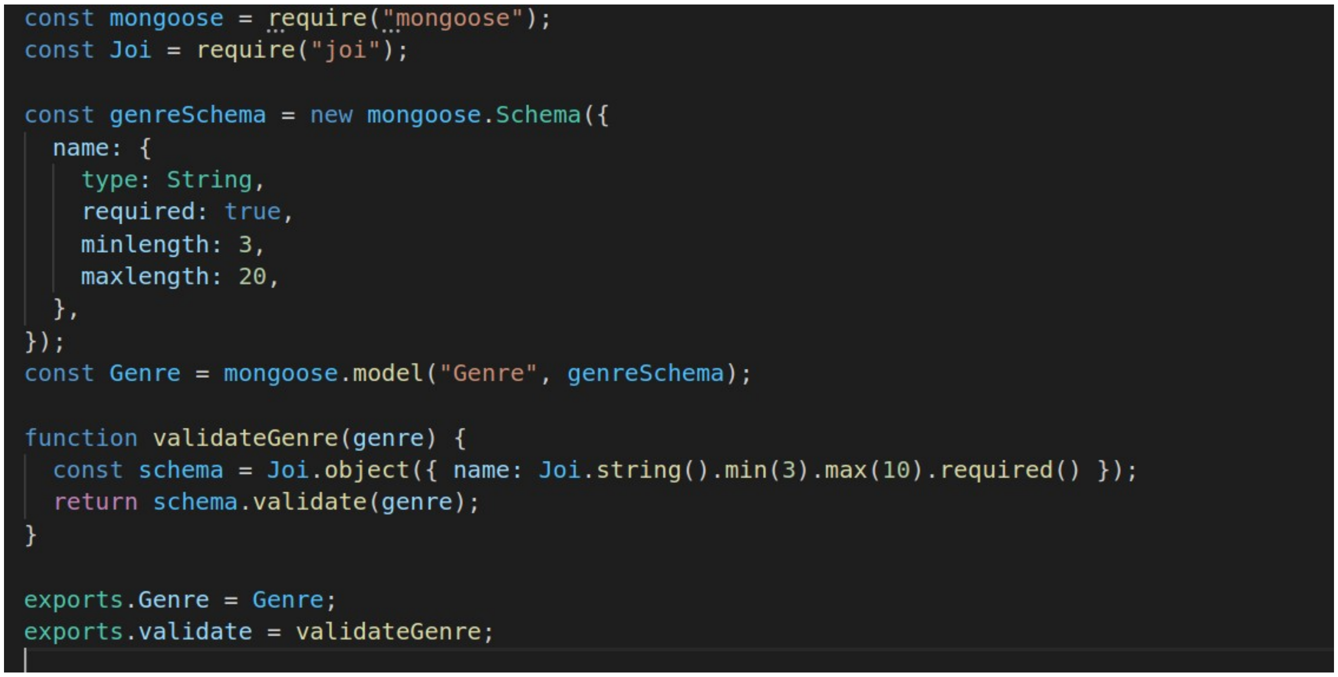


```javascript
  }
  let genre = new Genre({ name: req.body.name }); // we use let because
  //genre will be assigned with id in mongodb database
  genre = await genre.save();

  res.send(genre);
});

router.put("/:id", async (req, res) => {
  const genre = await Genre.findByIdAndUpdate(
    req.params.id,
    { name: req.body.name },
    { new: true }
  );
  if (!genre) {
    res.status(404).send("The genre you are looking for is unavailable.");
  }
  res.send(genre);
});

router.delete("/:id", async (req, res) => {
  const genre = await Genre.findByIdAndDelete(req.params.id);
  if (!genre) {
    res.status(404).send("The genre you are looking is unavailable!!");
  }
  res.send(genre);
});

function validateGenre(genre) {
  const schema = Joi.object({ name: Joi.string().min(3).max(10).required() });
  return schema.validate(genre);
}
```

Restructuring the project:

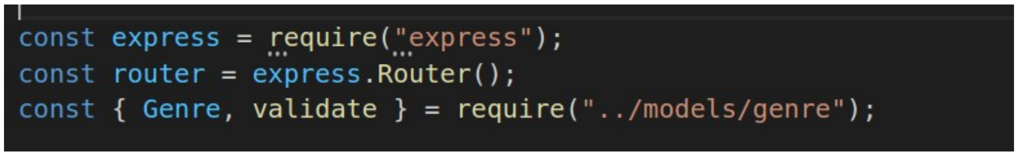
According to single responsibility principle, to keep our applications maintainable, we must ensure each module is responsible for only one thing. If we look at *genres.js*, it is the module responsible only for routes. The definition of **genre** object doesn't belong in this module. So all object & modules will be inside *models* folder.

So *models/genre.js:*



```
const mongoose = require("mongoose");
const Joi = require("joi");

const genreSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 3,
    maxlength: 20,
  },
});
const Genre = mongoose.model("Genre", genreSchema);

function validateGenre(genre) {
  const schema = Joi.object({ name: Joi.string().min(3).max(10).required() });
  return schema.validate(genre);
}

exports.Genre = Genre;
exports.validate = validateGenre;
```

In *routes/genres.js:*



```
const express = require("express");
const router = express.Router();
const { Genre, validate } = require("../models/genre");
```

9. Modeling Relationships Between Data

Until now we have worked with single self contained documents. We can have a **Course** document which has an **author.** We can have collection of **author**s document with it's properties like *name, website, image,* etc.

We have two approach to work with related objects. Using References (**Normalization**) and Using Embedded Documents (**Denormalization**).

Normalization:

```
let author = {
    name: "Mosh",
    country: "United States"
};
```

```
let course = {
    name: "Node.js",
    author: "id_of_author_document_collections"
};
```

Denormalization

Here we are embedding a document inside of another document.

```
let course = {
    name: "Node.js",
    author = {
        name: "Mosh",
        country: "USA"
    },
    tags: ["backend", "javascript"]
}
```

Within them we have trade-off between *query performance* vs *consistency*.
In normalization we have single place to modify author in future. That results in **consistency.** But when querying a course, we need to do extra query to load **author**. Hence **Normalization: *Consistency*** and **Denormalization: *Performance.*** Now we have the third approach called **hybrid** approach.

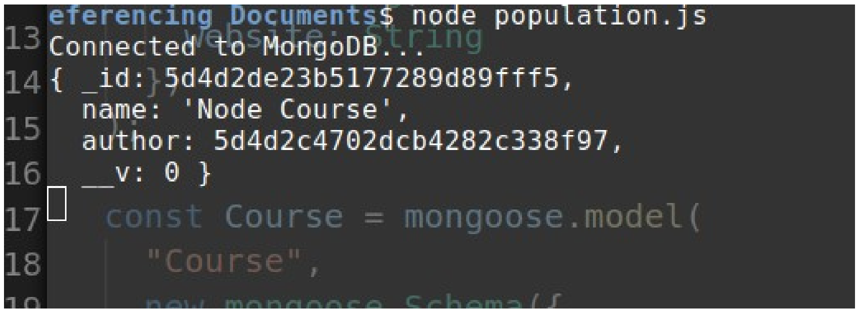
```
let author = {
    name: "Mosh"
    // 50 other properties
};
```

```
let course = {
    name: "Node.js",
    author: {
        id: "reference_id",
        name: "Mosh"
    }
```

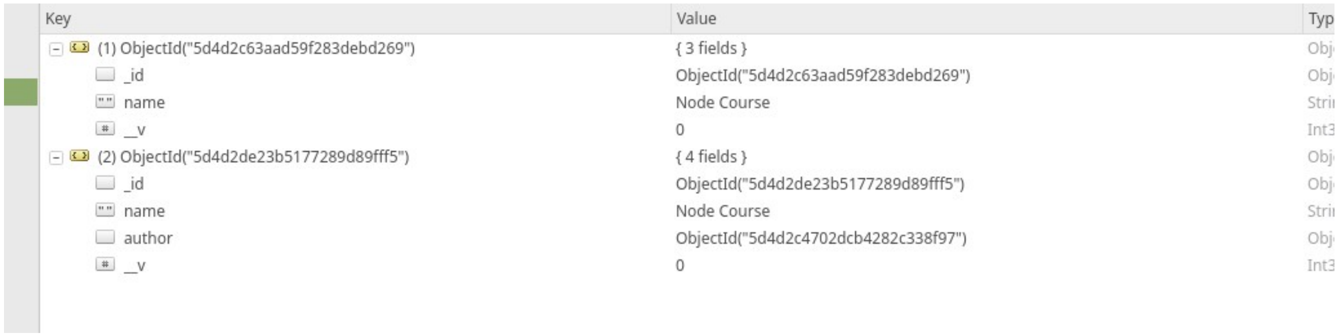
Referencing Documents

We have *population.js* from the downloaded file. Here we have two models **Author** and **Course.** For **course** to **include** author, in courseSchema, we set this to a type of Schema object.

```
const Course = mongoose.model(
    "Course",
    new mongoose.Schema({
        name: String,
        author: {
            type: mongoose.Schema.Types.ObjectId,
            ref: "Author" // Name of the target collection
        }
    })
);
```



```
eferencing Documents$ node population.js
Connected to MongoDB...
{ _id: 5d4d2de23b5177289d89fff5,
  name: 'Node Course',
  author: 5d4d2c4702dcb4282c338f97,
  __v: 0 }
```

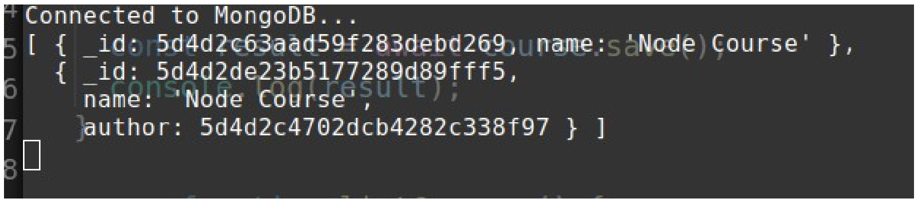


Key	Value	Typ
(1) ObjectId("5d4d2c63aad59f283debd269")	{ 3 fields }	Obj
_id	ObjectId("5d4d2c63aad59f283debd269")	Obj
name	Node Course	Stri
__v	0	Int3
(2) ObjectId("5d4d2de23b5177289d89fff5")	{ 4 fields }	Obj
_id	ObjectId("5d4d2de23b5177289d89fff5")	Obj
name	Node Course	Stri
author	ObjectId("5d4d2c4702dcb4282c338f97")	Obj
__v	0	Int3

Population

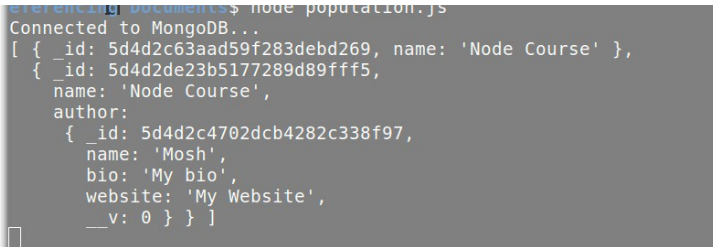
Lets get all the courses along with their authors.

```
async function listCourses() {
    const courses = await Course.find().select("name author");
    console.log(courses);
}
```



```
Connected to MongoDB...
[ { _id: 5d4d2c63aad59f283debd269, name: 'Node Course' },
  { _id: 5d4d2de23b5177289d89fff5,
    name: 'Node Course',
    author: 5d4d2c4702dcb4282c338f97 } ]
```

```
async function listCourses() {
    const courses = await Course.find()
    .populate("author")
    .select("name author");
    console.log(courses);
}
```



```
eferencing Documents$ node population.js
Connected to MongoDB...
[ { _id: 5d4d2c63aad59f283debd269, name: 'Node Course' },
  { _id: 5d4d2de23b5177289d89fff5,
    name: 'Node Course',
    author:
     { _id: 5d4d2c4702dcb4282c338f97,
       name: 'Mosh',
       bio: 'My bio',
       website: 'My Website',
       __v: 0 } } ]
```

If we just want to get **name** property of author second argument of **populate** is properties we want to include: `.populate("author", "name")`

To exclude a property we have to use dash: `.populate("author", "name country -age - _id")`

We can also populate other document:

```
const courses = await Course.find()
    .populate("author", "name -_id")
    .populate("category", "name -tags")
    .select("name author");
console.log(courses);
```

In mongoose there is no data integrity. So if we change the **id** of author there is no error and **author** property of **course** just returns *null*.

Embedding Documents

```
const authorSchema = new mongoose.Schema({
    name: String,
    bio: String,
    website: String
});
```

For course:

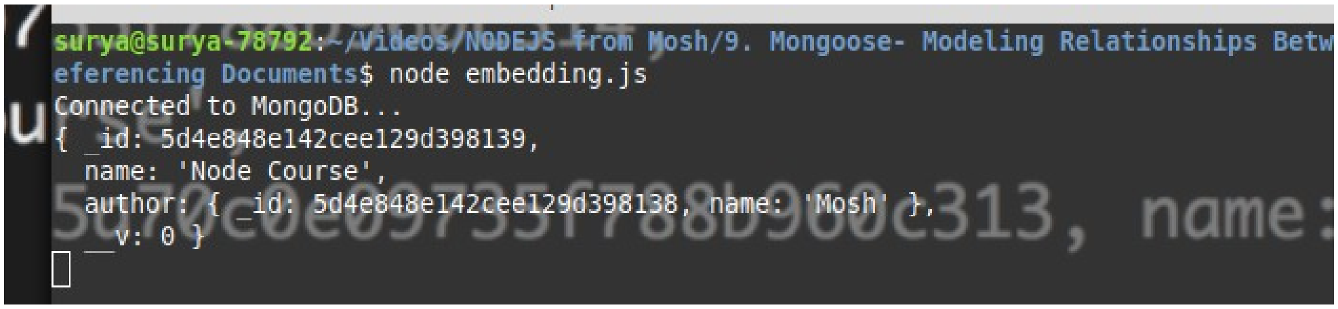
```
const Course = mongoose.model(
    "Course",
    new mongoose.Schema({
        name: String,
        author: authorSchema
    })
);
```

Function for creating course:

```
async function createCourse(name, author) {
    const course = new Course({
        name,
        author
    });
    const result = await course.save();
    console.log(result);
}
```

For creating course:

```
createCourse("Node Course", new Author({ name: "Mosh" }));
```



```
surya@surya-78792:~/Videos/NODEJS from Mosh/9. Mongoose- Modeling Relationships Betw
eferencing Documents$ node embedding.js
Connected to MongoDB...
{ _id: 5d4e848e142cee129d398139,
  name: 'Node Course',
  author: { _id: 5d4e848e142cee129d398138, name: 'Mosh' },
  __v: 0 }
```

Here the **author** is sub-document. It can use most features of normal document. We can implement validation here. But they cannot be saved on their own.

```
async function updateAuthor(courseId) {
     const result = await Course.findById(courseId);
     course.author.name = "Mosh Hamedani";
     course.save();
}
```

Here we don't need **course.author.save()**

We can also update sub-document directly with **update():**

```
async function updateAuthor(courseId) {
     const result = await Course.update(
          { _id: courseId },
          {
               $set: {
                    "author.name": "Mosh Hamedani"
               }
          }
     );
}
```

If we want to remove the sub document, we use the **unset** operator.

```
const result = await Course.update(
     { _id: courseId },
     {
          $unset: {
               "author": ""
          }
     }
);
```

We can also validate sub-documents:

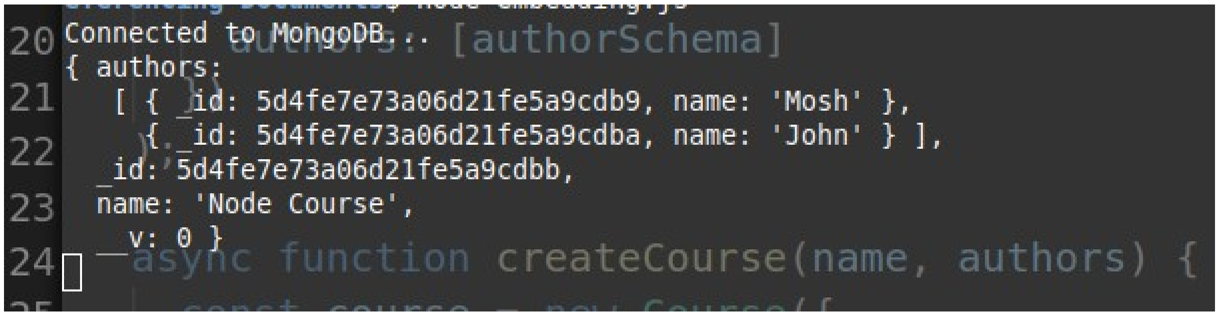
```
const Course = mongoose.model(
     "Course",
      new mongoose.Schema({
          name: String,
          author: {
               type: authorSchema,
               required: true,
          },
     })
);
```

Array of Sub-Documents

```
const Course = mongoose.model(
    "Course",
    new mongoose.Schema({
        name: String,
        authors: [authorSchema]
    })
);
```

```
async function createCourse(name, authors) {
    const course = new Course({
        name,
        authors
    });
```

```
createCourse("Node Course", [
    new Author({ name: "Mosh" }),
    new Author({ name: "John" })
]);
```



```
Connected to MongoDB...
{ authors:
   [ { _id: 5d4fe7e73a06d21fe5a9cdb9, name: 'Mosh' },
     { _id: 5d4fe7e73a06d21fe5a9cdba, name: 'John' } ],
  _id: 5d4fe7e73a06d21fe5a9cdbb,
  name: 'Node Course',
  __v: 0 }
```

We can always add an author to this array later on.

```
async function addAuthor(courseId, author) {
    const course = await Course.findById(courseId);
    course.authors.push(author);
    course.save();
}
```

```
addAuthor(
    "5d4fe7e73a06d21fe5a9cdbb",
    new Author({
        name: "Amy"
    })
);
```

To remove an author:

```
async function removeAuthor(courseId, authorId) {
	const course = await Course.findById(courseId);
	const author = course.authors.id(authorId);
	author.remove();
	course.save();
}
removeAuthor("5d4fe7e73a06d21fe5a9cdbb", "5d4feab7d2faa4215fb89533");
```

Project

Every movie object should have a title: String, genre: Object where we are embedding Genre document inside movie. *models/movie.js:*

```
const mongoose = require("mongoose");
const Joi = require("joi");
const { genreSchema } = require("./genre");
```

```
const movieSchema = new mongoose.Schema({
	title: {
		type: String,
		required: true,
		trim: true, // To get rid of any padding around
		minlength: 1,
		maxlength: 100,
	},
	genre: {
		type: genreSchema,
		required: true,
		min: 0,
		max: 255,
	},
	numberInStock: {
		type: Number,
		required: true,
		min: 0,
		max: 255,
	},
	dailyRentalRate: {
		type: Number,
		required: true,
		min: 0,
		max: 255,
	},
});
const Movie = mongoose.model("Movie", movieSchema);
```

```
function validateMovie(movie) {
	const schema = Joi.object({
		title: Joi.string().min(1).max(50).required(),
		genreId: Joi.string().required() /* We want client to
		send only id of the genre but in mongoose we have genre
		which is a complex object so mongoose schema can be
		independent of Joi schema */,
		numberInStock: Joi.number().min(0).max(255).required(),
		dailyRentalRate: Joi.number().min(0).max(255).required(),
	});
	return schema.validate(movie);
}
```

```
exports.Movie = Movie;
exports.validate = validateMovie;
```

routes/movies.js:

```
const express = require("express");
const router = express.Router();
const { Movie, validate } = require(      "../models/movie");
const { Genre } = require("../models/genre");

router.get("/", async (req, res) => {
      const movies = await Movie.find().sort("title");
      res.send(movies);
});

router.get("/:id", async (req, res) => {
      const movie = await Movie.findById(req.params.id);
      if (!movie) {
            return res.status(404).send("This movie is not available");
      }
      res.send(movie);
});

router.post("/", async (req, res) => {
      const result = validate(req.body);
      if (result.error) {
            return res.status(400).send(result.error.details[0].message);
      }

      const genre = await Genre.findById(req.body.genreId);
      if (!genre) return res.stauts(404).send("This genre is unavailable");

      let movie = new Movie({
            title: req.body.title,
            genre: {
                  _id: genre._id,
                  name: genre.name,
            } /*Here we selectively set the properties of genre because if
            we dont wanna include all propertiec by: genre: genre */,
            numberInStock: req.body.numberInStock,
            dailyRentalRate: req.body.dailyRentalRate,
      });
      movie = await movie.save();
      res.send(movie);
});

//router.put()
//router.delete()

module.exports = router;
```

Transactions

In some relational databases, we have the concept of transaction. It is a group of operation that should be performed as a unit. So either all of these operations will complete or if one operation fails, all will be rolled back. In mongoDB, we don't have transaction. So we have a technique called **Two Phase Commit** which is an advanced mongoDB course. We can find this course in https://docs.mongodb.com/v3.4/tutorial/perform-two-phase-commits/

Hopefully we have a library that gives us the concept of transaction but internally it implements this transaction using Two Phase Commit technique.

npm i fawn

In **Vidly** project */routes/rentals.js:*

This is a class which has an initialize method, which has **init()** method which we call at starting. We pass **mongoose** object that we required at top.

```
Fawn.init(mongoose);
```

Now we are not going to create the rental and update the movie explicitly as we did previously. We will create a task object which is like a transaction. There we have one or more operations which will be treated as a unit. So we can exclude the following code:

```
rental = await rental.save();
movie.numberInStock--;
movie.save();
```

So we have:

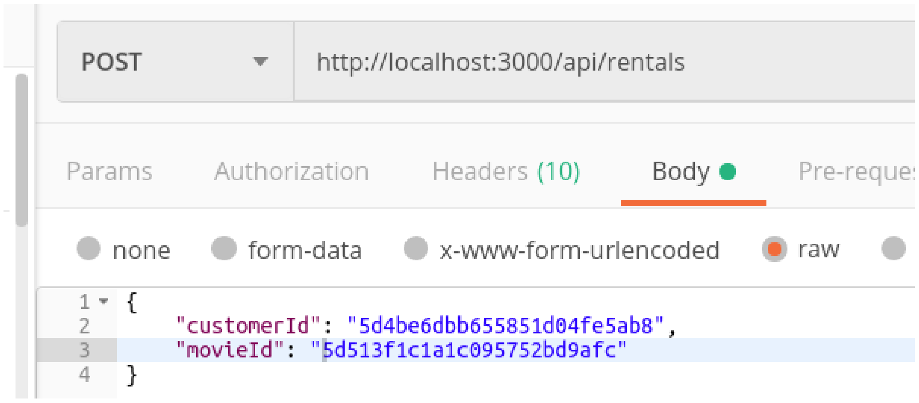
```
new Fawn.Task()
    .save("rentals", rental)
    // Here we are working directly with the collection so we need to pass the actual
    // name of the collection in the database and this name is case sensetive.
    .update(
        "movies",
        // In second argument, we pass Id of movie that should be updated.
        { _id: movie._id },
        // Third argument is what we are updating
        {
            // Here we can use the increment operator
            $inc: { numberInStock: -1 }
        }
     )
    // Finally we need to call run() to chain all these operations.
    .run();
```

We can run **remove()** and to know more features go to **GitHub** page documentation.

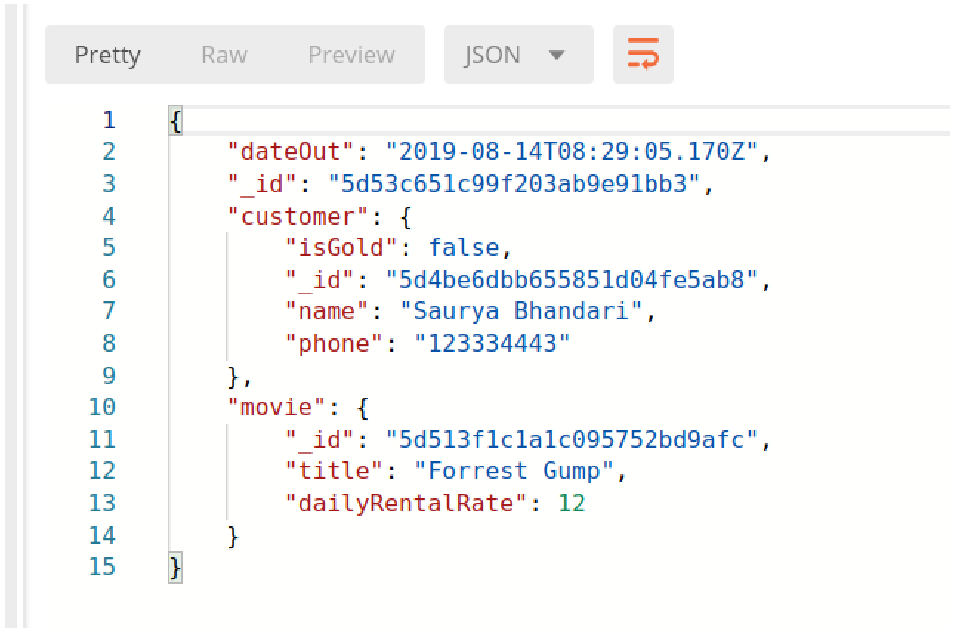
We need to wrap this in try-catch block in case something fails.

```
catch (ex) {
    res.status(500).send("Something went wrong...");
    // Status 500 means internal server error
}
```

We will run **index** and post in rental route



```
{
    "customerId": "5d4be6dbb655851d04fe5ab8",
    "movieId": "5d513f1c1a1c095752bd9afc"
}
```



```
{
    "dateOut": "2019-08-14T08:29:05.170Z",
    "_id": "5d53c651c99f203ab9e91bb3",
    "customer": {
        "isGold": false,
        "_id": "5d4be6dbb655851d04fe5ab8",
        "name": "Saurya Bhandari",
        "phone": "123334443"
    },
    "movie": {
        "_id": "5d513f1c1a1c095752bd9afc",
        "title": "Forrest Gump",
        "dailyRentalRate": 12
    }
}
```

If we refresh **movies** collection, **numberInStock** decreases. In this code, we didn't set the **_id** or **dateOut** properties. But in the body of the response, both these properties are set. So when we create a new **rental** object, mongoose sets the default values. We also see a new collection **ojlintaskcollections.** It is created by **Fawn** which uses this collection to perform two phase commits.

Validating ObjectIds

Let’s change **customerId** to *1234* which is not a valid object id. We will see error like *UnhandledPromiseRejection* and we have **CastError** *Cast to objectId failed for value “1234” at path “_id” for model “Customer”*. Word **path** can represent chain of properties like a customer can have **address** which can have **street.** We can validate object id like:

```
if (!mongoose.Types.ObjectId.isValid(req.body.customerId));
    return res.status(400).send("Invalid Customer...");
```

and we have to repeat the same for **movie** also. And this is a **bad** implementation. This logic belongs to **validate** function. There is an npm package to support validating **objectId**s in **Joi**. So go to **validateRental** function. So **npm i joi-objectid**.

Go to *models/rentals.js:*

This returns a function which we need to call and pass **Joi** module as reference.

```
Joi.objectId = require("joi-objectid")(Joi);
```

Using this in function **validateRental:**

```
function validateRental(rental) {
    const schema = Joi.object({
        customerId: Joi.objectId().required(),
        movieId: Joi.objectId().required(),
    });
    return schema.validate(rental);

}
```

Now in PostMan send request with invalid **customerId,** we get:

```
"customerId" with value "1234" fails to match the required pattern: /^[0-9a-fA-F]{24}$/
```

Now it is likely we will use this method in other places also. So we load it in **index** so **Joi.objectId** will be initiated in **index.** So in index:

```
const Joi = require("joi");
Joi.objectId = require("joi-objectid")(Joi);
```

Now ***objectId*** will be imbedded in ***Joi***. So we can use ***Joi.objectId*** in whichever file we want.

Also in **movies** when we use **genreId** we should implement this validation.

Now when posting Movie to the database mongoose talks to the mongoDB-driver and sets the Id right before saving. So we don’t need to define movie with **let** because technically we are not resetting the movie property after saving. So we can replace **let** by **const.**

10. Authentication and Authorization

Authentication is the process of identifying if the users are who they claim they are. That's done by logging in.

Authorization is determining if the user has the right permission to perform the given operation.

So in our Vidly application, only authenticated or logged in users can perform operations that modify data. Anonymous users can only read the data. For additional securities, only admin users can **delete** data.

Now we will add two endpoints:

For Register: POST to ***/api/users*** coz we are creating new User.

There should be and HTTP method for logging in. We should implement this in RESTful terms. Sometimes the operation we are dealing with doesn't have CRUD semantic. Here we are referring **Logging** as **request** or **command**. So we use POST because we are creating a new resource to ***/api/logins***

We will create **user** model **user.js** where **user** has three properties **name, email** and **password** with **validateUser** function.

Registering Users

We will create new route **/api/users**. Then we will create POST request for registering new user.

```
const { User, validate } = require("../models/user");
const express = require("express");
const router = express.Router();
```

```
router.post("/", async (req, res) => {
    const result = validate(req.body);
    if (result.error)
        return res.status(400).send(result.error.details[0].message);
```

```
    //Checking if user is not already registered
    let user = await User.findOne({ email: req.body.email });
    //findOne() is looking users by one of its properties
    if (user) return res.status(400).send("User already registered");
```

```
    user = new User({
        name: req.body.name,
        email: req.body.email,
        password: req.body.password,
    });
    user = await user.save();
    res.send(user);
});
```

```
module.exports = router;
```

Using Lodash

We don’t want to include **password** in **res.send().** There are two approach. One is to use a custom object like:

```
res.send({
     user: user.name,
     email: user.email
});
```

But here we will use a utility library that gives lots of utility functions to working with objects. **Lodash** is an optimized version of **underscore.** So **npm i lodash.**

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

We can call this anything like *lodash* but by convention we call by underscore. It has a utility method called ***pick().*** It will create new *user* object with only mentioned properties.

```
res.send(
     _.pick(user, ["name", "email"])
);
```

We can also use this in creating **user** model.

```
user = new User({
     name: req.body.name,
     email: req.body.email,
     password: req.body.password,
});
```

The above code will now be:

```
user = new User(
     _.pick(req.body, ["name", "email", "password"])
);
```

Now we wanna enforce complexity to the password. For this we have on top of **Joi** called **joi-password-complexity**. Search for this and add complexity to password.

Hashing Password

*bcrypt is not installed

For this, we are going to use very popular library called **bcrypt.** So **npm i** **bcrypt@1.0.3**

We will create a playground file for getting to know bcrypt. So create **hash.js**

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

Now, to hash a password, we need a **salt**. It is a random string that is added before or after a password so the resulting hashed password will be different each time based on the salt that is used. We have *sync* and *async* salt. We are using async salt.

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

async function run() {
    const salt = await bcrypt.genSalt(10);
    const hashed = await bcrypt.hash("hesoyam", salt);
    console.log(salt);
    console.log(hashed);
}

run();
```

So every time we run this application, we have different salt. The first part of hashed password includes **salt.** When we authenticate user we need this salt to retrieve the password. Now we will include this in **user** route.

```
user = new User(_.pick(req.body, ["name", "email", "password"]));
const salt = await bcrypt.genSalt(10);
user.password = await bcrypt.hash(user.password, salt);
res.send(_.pick(user, ["name", "email"]));
```

Authenticating Users *

Create *routes/auth.js* and use this as route /**api/auth** in index.

```
router.post("/", async (req, res) => {
    const { error } = validate(req.body);
    if (error) return res.status(400).send(error.details[0].message);

    let user = User.findOne({ email: req.body.email });
    if (!user) return res.status(400).send("Invalid email or password");

    const validPassword = await bcrypt.compare(req.body.password, user.password);
    if (!validPassword) return res.status(400).send("Invalid Email or Password");

    res.send(true); // Indicating that it is a valid login.
});
```

```
function validate(req) {
    const schema = Joi.object({
        email: Joi.string().min(5).max(255).required(),
        password: Joi.string().min(5).max(255).required(),
    });
    return schema.validate(req);
}
```

Testing the Authentication

For authentication, we are testing with authenticate user many times. So in postman we can save a request for further use.

Now in postman if we Post request with *api/auth:* with valid **email** and **password,** we should be getting **true** in response.

JSON Web Tokens

Now instead of returning **true,** we will return a JSON web token. It is a long string that identifies a user. It can be understood as a Driver’s license or Passport. When a user logs in on the server, we generate JWT to give it to the client and tell them that next time when they want to come back and call one of the endpoints, they have to show it. So on the client, we need to store this web Token which is a long string so we can send it back to the server for future API calls. The client can be a web application or a mobile application. For a web application we can use local storage. We can learn more of JWT in ***jwt.io***. There we have a debugger for working with JSON web tokens. The string has three parts which are color coded. Red part is header. We have two properties **alg** for algorithm and **typ** which is JWT. This is standard. Second part is payload. We have three properties **sub** is for user id, **name** and **admin.** This payload includes the public properties about the user. The third part is a digital signature. This is created based on the content of this JWT along with the secret or private key which is only available on the server. If a malicious user modified the admin property of payload, the digital signal will be invalid.

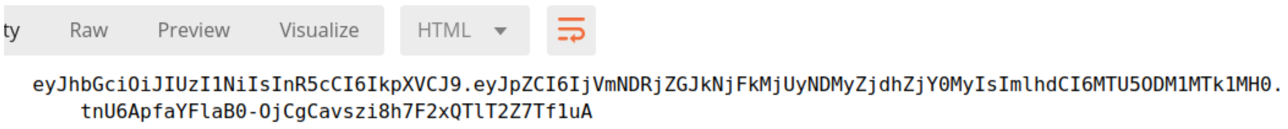
Generating Authentication Tokens

In terminal **npm i jsonwebtoken.** So in the *auth* module before returning the response, we need to create a new JWT. Import JWT as **jwt**. `const jwt = require("jsonwebtoken");`

Here we have a **sign** method where we pass the payload. What properties we put into this is completely upto us. As second argument we need to pass a **private key** which will be used to create a digital signature. We will store this key in environment variable later.

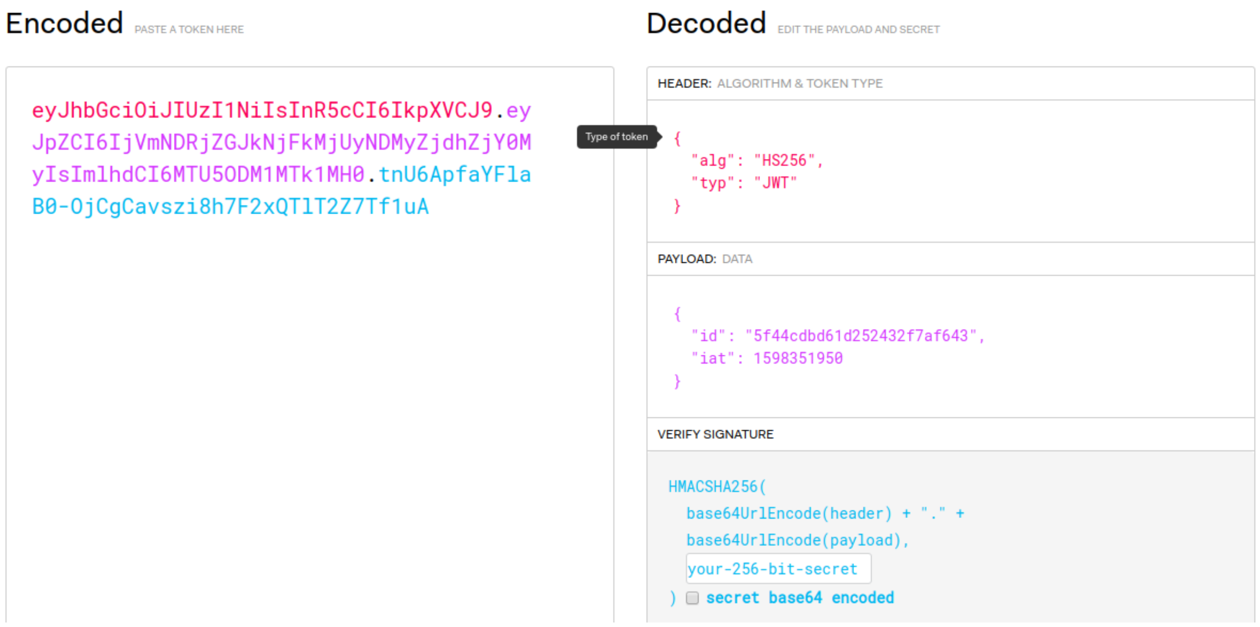
```
    const token = jwt.sign({ _id: user._id }, "somePrivateKey");/* Here we
    hardcoded jwt value but in real world application we store secrets in
    environment variables */
    res.send(token);
```

Now if we send the request in the *auth/* endpoint, we will get this token on success.



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjVmNDRjZGJkNjFkMjUyNDMyZjdhZjY0MyIsImlhdCI6MTU5ODM1MTk1MH0.
    tnU6ApfaYFlaB0-OjCgCavszi8h7F2xQTlT2Z7Tf1uA
```

If we copy and paste this token on the debugger of *jwt.io* we will get the json object. On the payload we have **iat** which is the time this token was created. We can use this to determine the age of JWT.



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JpZCI6IjVmNDRjZGJkNjFkMjUyNDMyZjdhZjY0M
yIsImlhdCI6MTU5ODM1MTk1MH0.tnU6ApfaYFla
B0-OjCgCavszi8h7F2xQTlT2Z7Tf1uA
```

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

```
{
  "id": "5f44cdbd61d252432f7af643",
  "iat": 1598351950
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Storing secrets in Environment variables

In terminal **npm i config@1.29.4.** Create new folder & file *config/default.json* where we will have several settings for our application. Here we have empty string because actual value is not here. We are just defining a template for all settings in our application.

```
{
    "somePrivateKey": ""
}
```

Create *config/custom-environment-variables.json.* Make sure to get the right spelling. In this file we specify the mapping between our application settings and environment variables.

```
{
    "somePrivateKey": "vidly_suryaPrivateKey"
}
```

We will map this setting to environment variable called `vidly_suryaPrivateKey`**.** We prefix this with application name so we don't end up one application setting overwriting another application setting. Now we go to *auth.js* and call to **config.get()** method.

```
const config = require("config");
```

Using **config:**

```
const token = jwt.sign({ _id: user._id }, config.get("jwtPrivateKey"));
```

Now this **jwtPrivateKey** is name of our application setting. Actual secret will be in our environment variable.

Now go to *index.js*. We make sure that when the application starts we make sure that our environment variable is set. So `const config = require("config");`

```
if (!config.get("jwtPrivateKey")) {
    console.error("FATAL ERROR: jwtPrivateKey is not defined...");
    process.exit(1);
}
```

Here **process** is global object in node. Method **exit()** where **0** indicates success and anything other than 0 is failure. If we run the application without setting environment variable, we will see FATAL ERROR message & app crashed. Here nodemon still running while application crashes. Now we set the environment variable from terminal:

export vidly_suryaPrivateKey=saurya@78792

Setting Response Headers

When the user registers, we wanna assume that the user is logged in. Let's imagine Vidly is an application that runs locally in a video store. So people who use this application are the people who work in a video store. In **post** method of **user** module, we are returning user object of three properties in response. Here we will return JSON web token in HTTP header.

For any custom headers that we define in our application we should prefix this with **x-** and remaining string we put as we wish. The first argument is the name of the header & the second argument is the value. So in *routes/users:*

```
const token = jwt.sign({ _id: user._id }, config.get("jwtPrivateKey"));
res
    .header("x-auth-token", token)
    .send(_.pick(user, ["_id", "name", "email"]));
```

Now when registering user form Postman, when sent, go to header tab we will see JWT with header **x-auth-token**.

Encapsulating logic in Models

We have exact same code in auth module and user module to generate JWT. In future we will add various properties to this payload. So we should have this logic at one place. To encapsulate this logic, we will be based on **Information Expert Principle.** So here **user** object is responsible for generating authentication tokens. So the function that generate auth-token should be a method in **user** object. So we should call **generateAuthToken()** like `const token = user.generateAuthToken();` from other modules. We have **userSchema** in **user** model. We will add a method in **userSchema.**

`userSchema.methods.generateAuthToken = function() {};` When we do this, then the **user** object from *auth* will have method **generateAuthToken.** We replace user with **this** to point to the object from which the method gets called. So we use regular function syntax. For *method* as a part of an *object* we should not use arrow functions.

```
userSchema.methods.generateAuthToken = function() {
    const token = jwt.sign({ _id: this._id }, config.get("jwtPrivateKey"));
    return token;
};
```

Authorization Middleware

Let *api/genres* be called only by authenticated users. We need to read request headers where we specify the name of the header where we expect JWT stored in this header. If this is valid then only we give access to *api/genres*. Otherwise we will return status 401 which is client doesn't have credentials to access this resource. We will put this logic in a middleware function. In root add a new folder *middleware* and add *auth.js.*

Here function **auth** takes three arguments **req, res** and **next** which we use to pass control to the next middleware function in the request processing pipeline. We call **jwt.verify** where we pass **token** and a *private-key (*we stored this in an environment variable) for decoding this token. We need **config** module to read this *private-key*. This method will verify JWT and if its valid, it will decode it and return the payload. If not valid, it will throw an exception. So we wrap this with try-catch block. We will add **user** property to request **req** and assign decoded payload to **req.user.** So in route handler, we can access **req.user._id.**

```
function auth(req, res, next) {
    const token = req.header("x-auth-token");
    if (!token) return res.status(401).send("Access denied. No token provided");

    try {
        const decoded = jwt.verify(token, config.get("jwtPrivateKey"));
        req.user = decoded;
        next();
    } catch (ex) {
        res.status(401).send("Invalid token");
    }
}

module.exports = auth;
```

Protecting Routes

We will apply the middleware function selectively to some endpoints. In *genres.js:*

```
router.post("/", auth, async (req, res) => {
```

So **auth** will be executed before other middleware function. Now when adding new genre, we should send genre name with JWT in header of the request.

Getting the Current User

We will add new api endpoint for getting the current user. We could have endpoint like **api/:id** but we don’t want anyone put another user’s id and get the privileges. So we can have endpoint like **api/me** which will only be available to authenticated users. And we will get id from JWT. In *api/users:* add **auth** middleware,

```
router.get("/me", auth, async (req, res) => {
    const user = await User.findById(req.user._id).select("-password"); // password excluded
    res.send(user);
});
```

Now send a POST request in Postman with endpoint **api/users/me** with **x-auth-token** in header. This will send current user information.

Logging Out a User

In *routes/auth.js* which is authentication module, we defined route for authenticating users. We need to delete this token for logging out users. We implement logging out feature on the client, not on the server.

Role Based Authorization

Go to user model and in **userSchema** add a property **isAdmin** with type Boolean. Now go to *robo3T* and add **isAdmin: true** to one user. When someone logs in we wanna include **isAdmin** in payload so we can extract this property directly from the token. To include this in payload:

```
userSchema.methods.generateAuthToken = function() {
    const token = jwt.sign(
        { _id: this._id, isAdmin: this.isAdmin },
        config.get("jwtPrivateKey")
    );
    return token;
};
```

Now we need a new middleware function to check if the user is Admin. So create *middlewares/admin.js*. We're assuming that this middleware function will be executed after authorization middleware function **(auth).** So **auth** sets **req.user** which we can access in this function. Status **403** is forbidden **401** is Unauthorized. We use unauthorized when a user tries to access the protected resource but don't supply a valid JWT. We give them re-chance to supply valid JWT. If the JWT is still valid but not allow to access the target resource then **Forbidden.**

```
module.exports = function(req, res, next) {
    if (!req.user.isAdmin) return res.status(403).send("Access denied...");
    next();
};
```

Now in delete method of *genres.js* we apply this middleware.

```
router.delete("/:id", [auth, admin], async (req, res) => {
```

11. Handling and Logging Errors

We should count for the unexpected situations and handle them properly. We should send the error message to the client and log the exception on the server. Lets stop **mongoDB** and send request on Postman. We will see *unhandledPromiseRejectionWarning.*

Suppose mongodb terminates for 1 minute and come back but we will not be able to serve clients even mongoDB restarts.

Handling Rejected Promises

Here we are dealing with asynchronous code with promise and if that promise is rejected and we have not handled the rejection properly. Go to *routes/genres* to *get* request.

```
router.get("/", async (req, res) => {
    const genres = await Genre.find().sort("name");
    res.send(genres);
});
```

Here we are awaiting the promise but we don't have *try-catch* block to handle the rejection. Its like we are calling **.then()** but not calling **.catch()** in promise syntax. So

```
router.get("/", async (req, res) => {
    try {
        const genres = await Genre.find().sort("name");
        res.send(genres);
    } catch (ex) {
        res.status(500).send("Something Failed.");
    }
});
```

Error code **500** is Internal Server Error.

First connect mongodb then start nodemon then only drop the connection and try.

Express Error Middleware

Instead of going to every route handler and logging the error, we will use a central place to handle errors. Here we are registering all the middleware functions.

```
app.use("/", home);
app.use("/api/users", users);
app.use("/api/genres", genres);
app.use("/api/customers", customers);
app.use("/api/movies", movies);
app.use("/api/rentals", rentals);
app.use("/api/auth", auth);
```

In *express*, we have special kind of middleware function called *error middleware*. We will register this middleware function after these middleware functions of using routes.

```
app.use(function(err, req, res, next) {
    // We will log the exception here
    res.status(500).send("Something failed");
});
```

In *genres.js:*

```
router.get("/", async (req, res, next) => {
    try {
        const genres = await Genre.find().sort("name");
        res.send(genres);
    } catch (ex) {
        next(ex);
    }
});
```

We call **next** to pass control to the next middleware function in the request processing pipeline. So we will end up to the previous function and the exception we pass will be the first argument to that function.

Now the logic for logging the exception might be long. So for this function in *index,* we will do *Orchestration or High Level Arrangement.* So move this middleware function in separate module.

Create *middleware/error.js* :

```
module.exports = function(err, req, res, next) {
    res.status(500).send("Something failed");
};
```

So in *index:*

```
const error = require("./middleware/error");
```

and then,

```
app.use(error);
```

Note that we are not passing **error()** because we will only pass reference to this function.

Now again the problem is we have to add *try-catch* block in every route handler.

Removing Try-Catch Block

We should move this as high-level function. The template of this function is like

```
function asyncMiddleware() {
    try {
        // Code which will vary from one route handelr to another
    } catch (ex) {
        next(ex);
    }
}
```

What if we have route handler function as argument an we call it in try block

```
function asyncMiddleware(handler) {
    try {
        await handler();
```

Now in route handler, we don't need try-catch block and **next** argument. Here we have *anonymous async* function: `router.get("/", async (req, res) => {`

We wanna pass this function as an argument as **handler.** Because **handler()** is an *async* function, we should *await* handler, having used *await* we should mark the function as *async*.

```
async function asyncMiddleware(handler) {
```

Now in route handler we will pass **asyncMiddleware** as second argument and pass the anonymous function as argument of it.

```
router.get(
    "/",
    asyncMiddleware(async (req, res) => {
        const genres = await Genre.find().sort("name");
        res.send(genres);
    })
);
```

Now there is tiny issue. The **handler** we are calling needs two arguments **req** and **res**. But nowhere in **asyncMiddleware** function we have defined **req** and **res**.

If we think deeply, we are calling **asyncMiddleware** and passing **handler(**async function**)** as argument in route handler. So we are calling **asyncMiddleware** function**.** But previously we just pass reference of async function (didn't called it) like:

```
router.get("/another", async (req, res) => {
} )
```

It's the **express framework** that calls this function and pass **req, res** and other arguments at runtime. So we will change **asyncMiddleware** function so it returns a route-handler function by referencing it. This will be like a factory function. We call it and get a new function.

```
function asyncMiddleware(handler) {
    return async (req, res, next) => {
        try {
            await handler(req, res);
        } catch (ex) {
            next(ex);
        }
    };
}
```

Here the route-handler function we are awaiting a promise but in **asyncMiddleware** function, we are nowhere awaiting promise.

We will put **asyncMiddleware** in separate module *middleware/async.js*. Now wrap all the route handlers with **asyncMiddleware**

A different approach to **asyncMiddleware** will be *npm module* which will monkeypatch our route handlers at runtime. Which means, when we send a request to this endpoint, this *npm-module* will wrap our route handler code inside *async.js* module automatically. So **npm i express-async-errors** (not *express-async-error*). Here version we use is 2.1.0.

So in *index:* `require("express-async-errors");` We don't have to store this in the constant. We can remove the call to **asyncMiddleware** and get our original route-handler implementation.

Logging Errors

We have very popular logging library calles **winston**. So **npm i winston@2.4.0.**

In *index:*

```
const winston = require("winston");
```

Here **winston** is the default logger exported from the *winston* module. We can also make a custom logger but is only relevant in large applications. This logger object have **transport** which is a storage device for our logs. Winston comes with few core transports like **Console, File, HTTP**. The default logger comes with one transport **console** which is for logging messages in console. We can add another transport for logging messages in a file. In *index:*

```
winston.add(winston.transports.File, { filename: "logfile.log" });
```

Back to our *error* middleware,

```
module.exports = function(err, req, res, next) {
    winston.log("error", err.message);
    res.status(500).send("Something failed");
};
```

The first argument to **log** is logging-level. It determines the importance of the message we're going to log. Some are *error* for errors, *warn* for warning, *info* for information, *verbose, debug, silly,* etc.

We can also use one of the helper methods like `winston.error(err.message, err);`

Optionally, we can also store metadata by passing **err** object. So every properties in the error object will also be stored. Lets throw an error in *get* route handler of *genres.js.*

```
throw new Error("Could not get to the routes...");
```

So somewhere in the application our error middleware will catch the exception. Send get request to *api/genres* to see new log file created with error message inside.

Logging to MongoDB

npm i winston-mongodb@3.0.0

We will just require it in index: `require("winston-mongodb");`

```
winston.add(winston.transports.MongoDB, { db: "mongodb://localhost/vidly" });
```

There are few properties which we can look on the documentation. We need to set **db** which we will set to our database collection *vidly*. We can separate this log to separate database which is upto us.

Now send request to postman and we will have new collection **log** in our *vidly* database. We can also add logging level to the **transports**

```
winston.add(winston.transports.MongoDB, { db: "mongodb://localhost/vidly", level: "error" });
```

Now, we will see only error message in the log. If we set this to **info** being third level, **error** and **warn** will be logged automatically. **Verbose**, **debug**, **silly** will not be logged.

Uncaught Exceptions

Error middleware that we have added only catches errors that happens as a part of request processing pipeline. So this is particular to express. So this will not be called if an error is thrown outside of the context of express. We have this **process** object which is an *event emitter* and gives method **on.** In node, we have a standard event called **uncaughtException**. This **uncaughtException** is raised when we have the exception in node process but nowhere it is handled using a *catch* block. We will now handle it with callback function.

```
process.on("uncaughtException", ex => {
    console.log("We have an uncaught exception");
    winston.error(ex.message, ex);
});
```

Now we will get uncaught exception in console as well as saved in logger file.

Unhandled Promise Rejections

The *uncaught exception* approach above only works with synchronous code. If we have promise somewhere and is rejected, then above code will not work.

```
const p = Promise.reject(new Error("Something failed in runtime!!!"));
p.then(() => console.log("done"));
```

Here we are not calling ***catch()*** so this will be a rejected promise.

In terminal we will see *UnhandledPromiseRejectionWarning.* In the future, unhandled promise rejections will terminate the Node.js process. The **process** object have another event called **unhandledRejection**

```
process.on("unhandledRejection", ex => {
    console.log("We got unhandled rejection.");
    winston.error(ex.message, ex);
});
```

As a best practice when *uncaughtException* or *unhandledRejection*, we should terminate the process and restart with clean state. For production, there are tools called process manager to restart the application.

```
process.on("uncaughtException", ex => {
    winston.error(ex.message, ex);
    process.exit(1);
});
```

Here 0 is success and anything but 0 is failure.

In another way we have a helper method **winston.handleExceptions()** where we can pass one or more transport objects.

```
winston.handleExceptions(
    new winston.transports.File({ filename: "uncaughtExceptions.log" })
);
```

When we call ***handleExceptions()*** we are specifying different transport file `uncaughtExceptions.log`

But this method will only work for *uncaughtException.* Now we will log *unhandledRejection* with ***handleExceptions()*** method.

```
winston.handleExceptions(
    new winston.transports.file({ filename: "uncaughtExceptions.log" })
);
process.on("unhandledRejection", (ex) => {
    throw ex;
});
```

Now *winston* will get this exception, catch and log it in *uncaughtExcetions.log*

Refactoring the modules to make the code clean and maintainable

We will only orchestrate different concerns like route handling, middlewares, database, errors, etc in index. The details of them will be moved to other modules. Create new file *startup/routes.js* will include requiring routes. It us a function where we add all the codes for setting up routes and other middleware. This function will take ***app*** as an argument. So in *index*:

```
require("./startup/routes")(app); // Calling the exported function with argument
```

startup/routes.js:

```
const express = require("express");
const error = require("../middleware/error");
const home = require("../routes/home");
const genres = require("../routes/genres");
const customers = require("../routes/customers");
const movies = require("../routes/movies");
const rentals = require("../routes/rentals");
const users = require("../routes/users");
const auth = require("../routes/auth");
```

```
module.exports = function(app) {
    app.use(express.json());
    app.use("/", home);
    app.use("/api/genres", genres);
    app.use("/api/customers", customers);
    app.use("/api/movies", movies);
    app.use("/api/rentals", rentals);
    app.use("/api/users", users);
    app.use("/api/auth", auth);
    app.use(error);
};
```

Now we will move all the database initialization code to the different module. Create *startup/database.js*

```
const mongoose = require("mongoose");
module.exports = function() {
    mongoose
        .connect("mongodb://localhost/vidly")
        .then(() => winston.info("Connected to the database..."));
        // .catch(err => winston.error("Could not connect to the database", err));
        // We want to terminate the process if error so this will be uncaught
        exception and it will be logged in uncaughtExceptions.log file
};
```

startup/logging.js:

```
const winston = require("winston");
require("winston-mongodb");
require("express-async-errors");
```

```
module.exports = function() {
    winston.handleExceptions(
        new winston.transports.File({ filename: "uncaughtExceptions.log" })
    );
```

```
    process.on("unhandledRejection", ex => {
        throw ex;
    });
```

```
    winston.add(winston.transports.File, { filename: "logfile.log" });
    winston.add(winston.transports.MongoDB, {
        db: "mongodb://localhost/vidly",
        level: "error"
    });
};
```

startup/config.js

```
const config = require("config");
const winston = require("winston");
```

```
module.exports = function() {
    if (!config.get("jwtPrivateKey")) {
        throw new Error("FATAL ERROR: jwtPrivateKey is not defined...");
        // We should throw error object instead of just a string.
        We will not have stack trace otherwise.
    }
};
```

Now we should move the configuration of **Joi.** We are using Joi at the api layer. Create *startup/validation.js*

```
const Joi = require("joi");
module.exports = function () {
    Joi.objectId = require("joi-objectid")(Joi);
};
```

Now if we take this application on a different machine, it terminates without telling what has happened. We are using **winston** to handle exceptions and only using file transports. To know why it crashed, we have to look at log file.

```
winston.handleExceptions(
     new winston.transports.Console({ colorize: true, preetyPrint: true }),
     new winston.transports.File({ filename: "uncaughtExceptions.log" })
);
```

So our *index.js* becomes:

```
const winston = require("winston");
const express = require("express");
const app = express();
```

```
require("./startup/logging")(); // In first, just in case we get error in logging other modules
require("./startup/routes")(app); // Calling the exported function with argument
require("./startup/database")(); // Just calling the exported function.
require("./startup/config")();
require("./startup/validation")();
```

```
const port = process.env.Port || 3000;
app.listen(port, () => winston.info(`Listening to port ${port}`));
```

12. Unit Testing

Automated Testing

Automated testing is the practice of writing code to test our code, and then run those tests in an automated fashion. So our code consists of our *application code* and *test code.*

Refactoring code means changing the structure of our code without changing it's behavior.

Three types of automated tests: **Unit test, Integration test, End-to-end test**

Unit test: Tests a unit of an application without it's <u>*external dependencies*</u> like *files, databases, web-services* etc.

Integration test: Tests the application with its <u>*external dependencies*</u>.

End-to-end test: Drives an application through its User Interface (UI). There are specific tools for this like *Selenial*. *Selenial* allows us to record the interaction of the user with our application and play it back to check if the application is providing the right result or not.

Tooling

We need a test-framework to write a test. It gives us library that includes a bunch of *utility functions* and *test runners.* The popular frameworks are **Jasmine, Mocha, Jest.**
Jasmine is early one. **Mocha** is most popular testing library in *npm.* **Jest** is newer framework and facebook users used it to test react applications. Its basically a wrapper around *Jasmine.* We are using **Jest** for this application. But tools and frameworks don't really matter. We should focus on the fundamentals of writing good unit tests.

Writing first Unit test

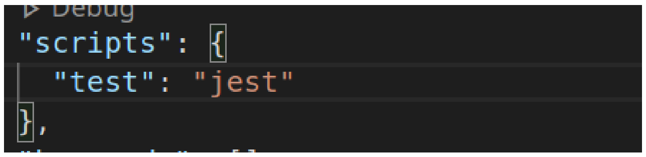
We first open the downloaded project *Writing Your First Test.*

We download Jest as: ***sudo npm i jest –save-dev*** We have flag *–save-dev* because *Jest* is a development dependency. We are using version 22.2.2.

Now if we look at *package.json, Jest* is installed under *devDependencies* because it is purely a development tool. We have property ***scripts*** under which is ***tests.***

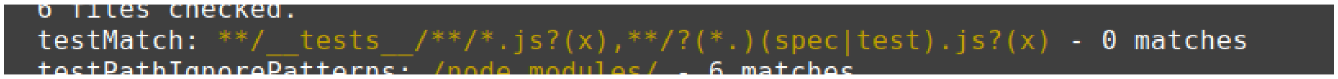
```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
},
```

These properties define commands that we can execute in the command line. To execute commands under ***tests,*** we go to terminal and type ***npm test***. So now we set:



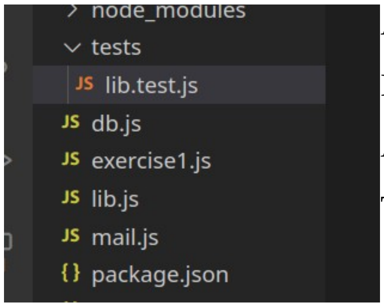
```
"scripts": {
  "test": "jest"
},
```

If we run ***npm test*** we will now get different output.



```
 6 files checked.
 testMatch: **/__tests__/**/*.js?(x),**/?(*.)(spec|test).js?(x) - 0 matches
 testPathIgnorePatterns: /node_modules/ - 6 matches
```

This is the test pattern **Jest** uses to locate our test files. Any file that ends with ***(spec|test).js*** is treates as a test file.



Add *tests* folder where we put all our tests.

Now we are going to write tests for our module *lib.js*

Add a file: *tests/lib.test.js*

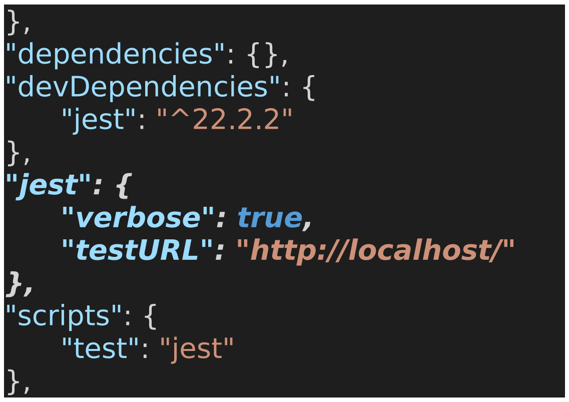
The naming convention is: /(*the module we are testing).test.(its extension)*

```
test("Our first test", () => {});
```

First argument is the name of our test. Second argument is function where we implement our test.

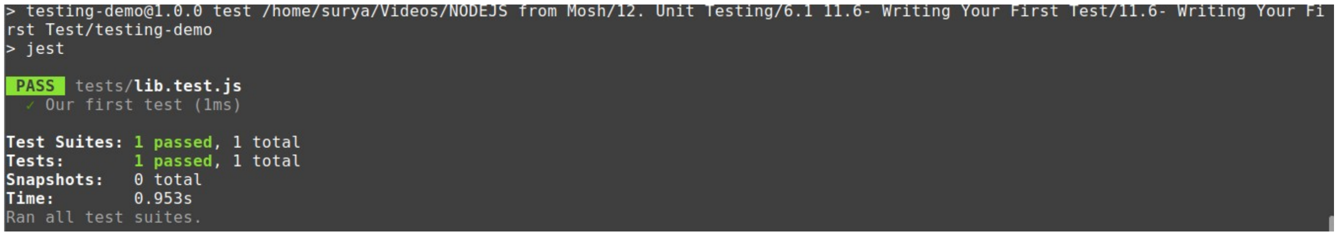
Run: ***npm test***

*Here our test failed. Needed to add following to *package.json:*

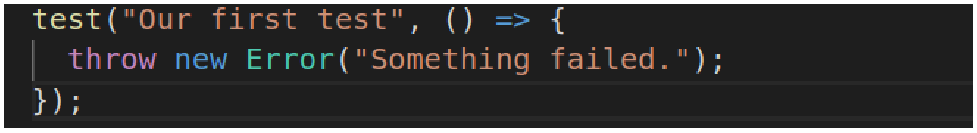


```
},
"dependencies": {},
"devDependencies": {
    "jest": "^22.2.2"
},
"jest": {
    "verbose": true,
    "testURL": "http://localhost/"
},
"scripts": {
    "test": "jest"
},
```

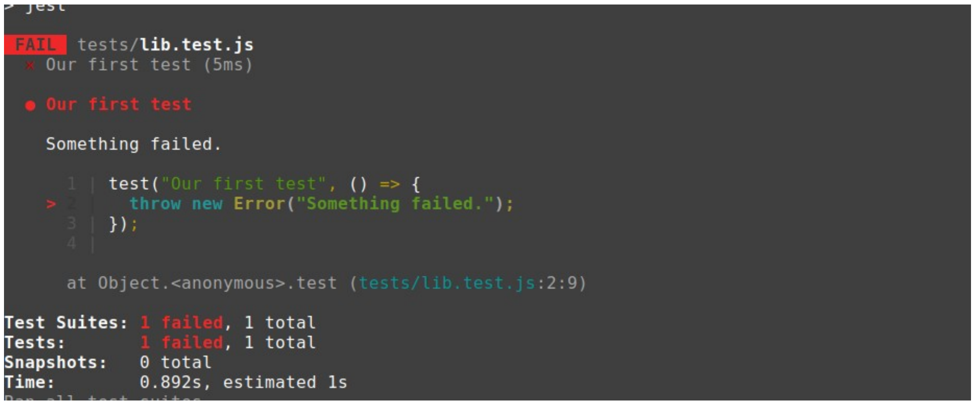
After running again result is:



```
> testing-demo@1.0.0 test /home/surya/Videos/NODEJS from Mosh/12. Unit Testing/6.1 11.6- Writing Your First Test/11.6- Writing Your Fi
rst Test/testing-demo
> jest

 PASS  tests/lib.test.js
  ✓ Our first test (1ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.953s
Ran all test suites.
```

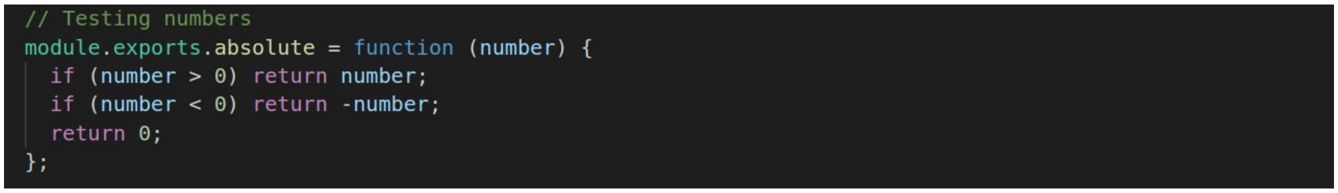


```
test("Our first test", () => {
  throw new Error("Something failed.");
});
```



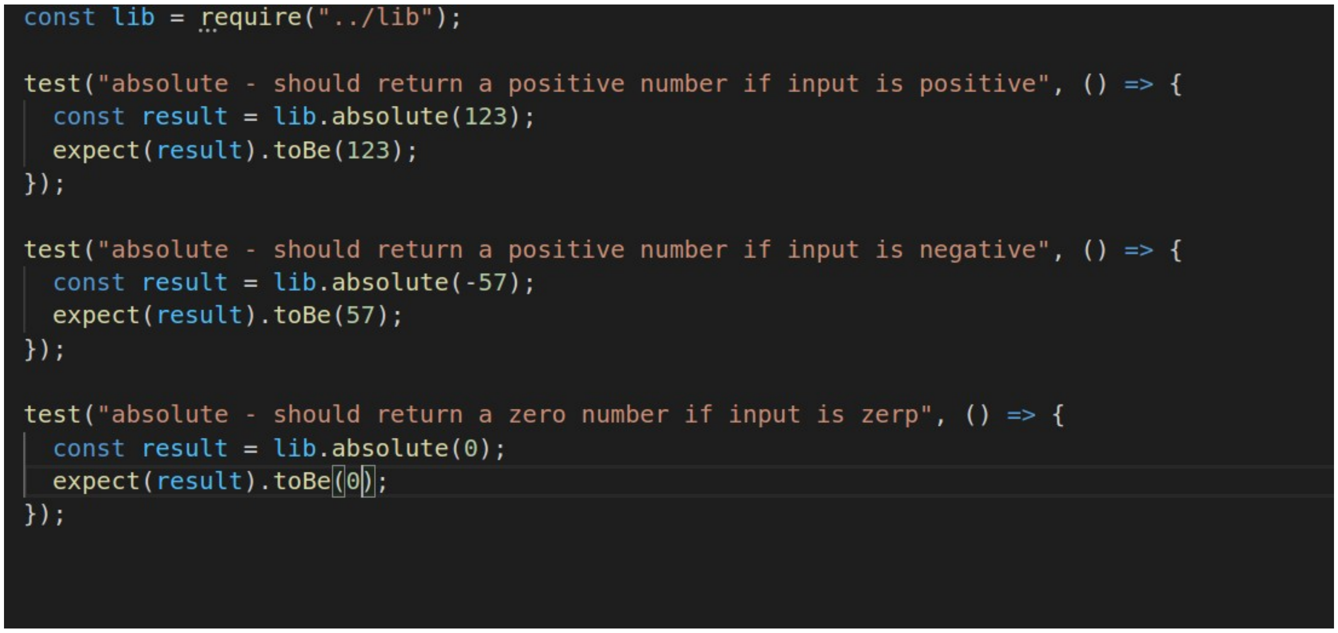
```
 FAIL  tests/lib.test.js
  ✕ Our first test (5ms)

  ● Our first test

    Something failed.

      1 | test("Our first test", () => {
    > 2 |   throw new Error("Something failed.");
      3 | });
      4 |

      at Object.<anonymous>.test (tests/lib.test.js:2:9)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.892s, estimated 1s
```

Now we are testing the following function. It returns absolute value of an integer.



```
// Testing numbers
module.exports.absolute = function (number) {
  if (number > 0) return number;
  if (number < 0) return -number;
  return 0;
};
```

Basic Guideline to writing unit test:
The number of unit test we have for a given function >= Number of execution paths.
Here we have three execution paths: if +ve, if _ve, if 0.



```
const lib = require("../lib");

test("absolute - should return a positive number if input is positive", () => {
  const result = lib.absolute(123);
  expect(result).toBe(123);
});

test("absolute - should return a positive number if input is negative", () => {
  const result = lib.absolute(-57);
  expect(result).toBe(57);
});

test("absolute - should return a zero number if input is zerp", () => {
  const result = lib.absolute(0);
  expect(result).toBe(0);
});
```

Terminal: ***npm test***

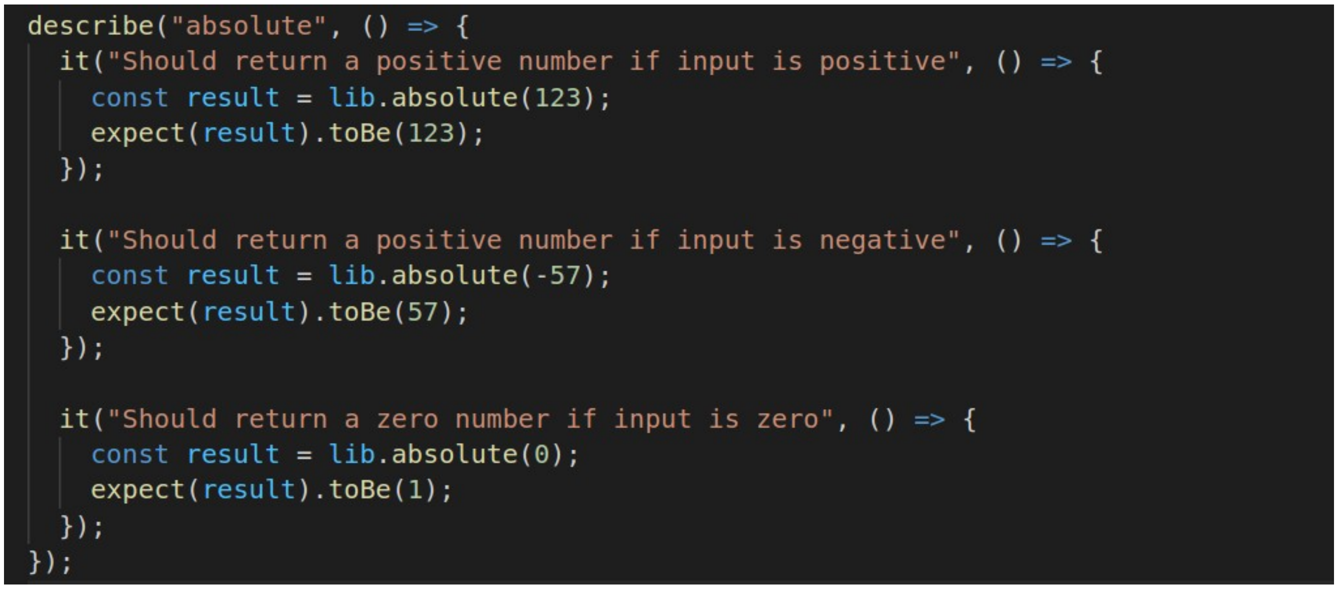
Matcher Functions:

Go to: https://jestjs.io/docs/en/using-matchers

In above example ***toBe()*** is a matcher function.

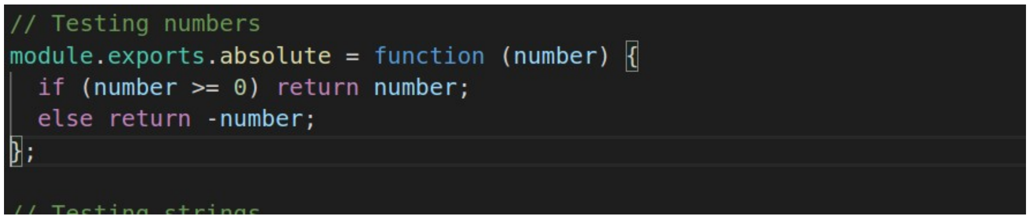
Grouping Tests

We have a function called **describe()** for grouping a bunch of related tests.



```
describe("absolute", () => {
  it("Should return a positive number if input is positive", () => {
    const result = lib.absolute(123);
    expect(result).toBe(123);
  });

  it("Should return a positive number if input is negative", () => {
    const result = lib.absolute(-57);
    expect(result).toBe(57);
  });

  it("Should return a zero number if input is zero", () => {
    const result = lib.absolute(0);
    expect(result).toBe(1);
  });
});
```

Refactoring with Confidence



```
// Testing numbers
module.exports.absolute = function (number) {
  if (number >= 0) return number;
  else return -number;
};

// Testing strings
```

Testing Strings

```
describe("greet", () => {
    it("Should return string added", () => {
        const result = lib.greet("Surya");
        expect(result).toBe("Welcome Surya");
    });
});
```

Our test should neither be too specific, nor too general. It should be at the right balance.

```
describe("greet", () => {
    it("Should return string added", () => {
        const result = lib.greet("Surya");
        expect(result).toMatch(/Surya/);// Using regular expression
    });
});
```

Testing Arrays

```
// Testing arrays
module.exports.getCurrencies = function () {
    return ["USD", "AUD", "EUR", “NPR”];
};
```

Test:

```
describe("getCurrencies", () => {
    it("Should return array of currencies", () => {
        const result = lib.getCurrencies();
        expect(result).toEqual(expect.arrayContaining(["EUR", "USD", "AUD"]));
    });
});
```

Testing Objects

```
// Testing objects
module.exports.getProduct = function (productId) {
    return { id: productId, price: 10 };
};
```

Test:

```
describe("getProduct", () => {
    it("Should return product with given ID", () => {
        const result = lib.getProduct(1);
        expect(result).toBe({ id: 1, price: 10 });
    });
});
```

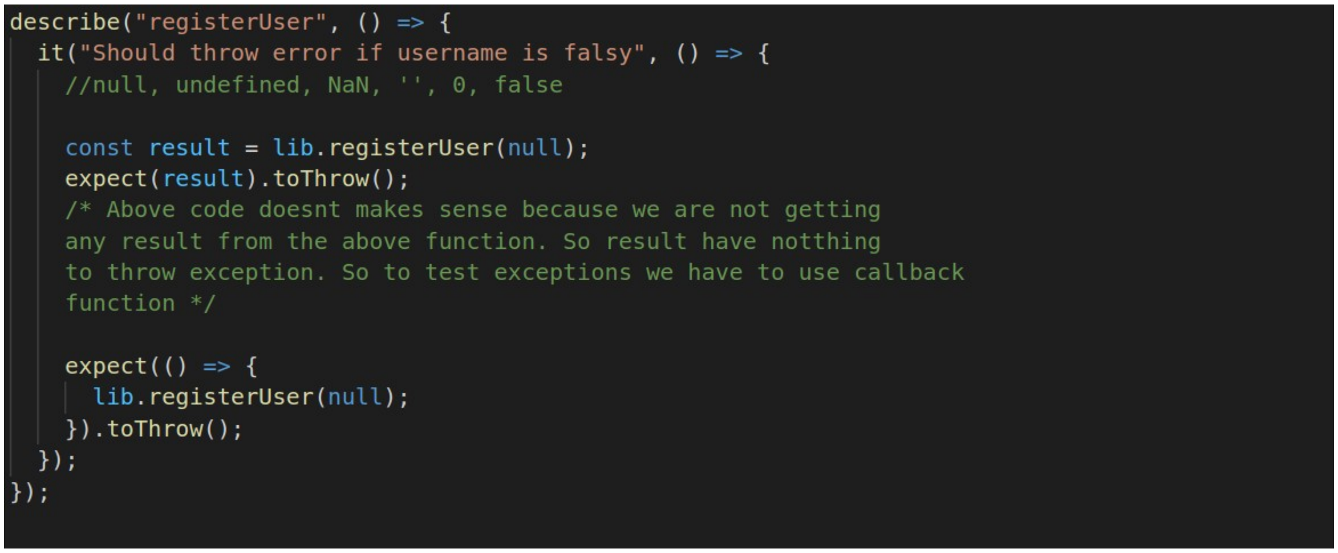
Result failed because ***toBe()*** compares references to these objects in memory. In this case ***result*** is in different memory location than the object given. So instead we use ***toEqual()*** to check object-equality.

```
expect(result).toEqual({ id: 1, price: 10 });
```

We can use other matchers with objects:
toMatchObject() It will pass as long as we have ***id*** and ***test*** matching. It there are more properties, it will pass anyway.
tohaveProperty() `expect(result).toHaveProperty("id", 1);`

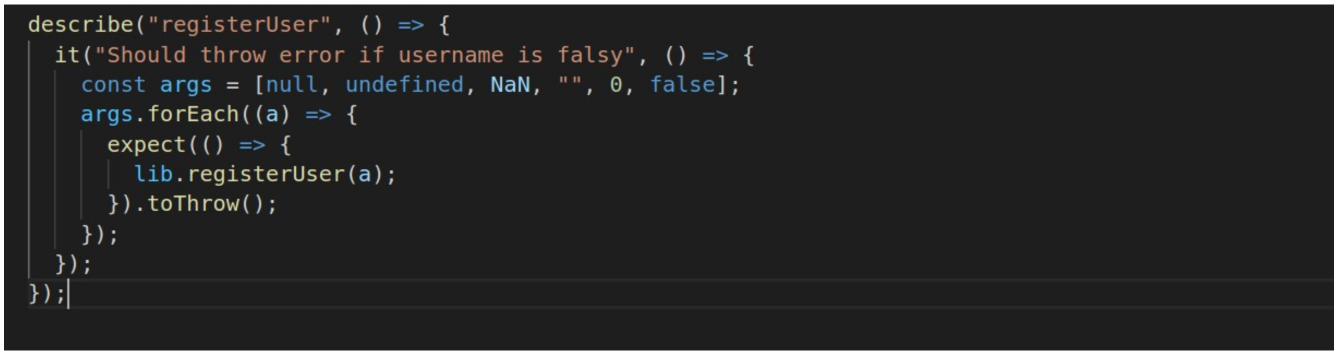
Here ***toEqual()*** is too specific. So better to use ***toMatch()*** or ***toHaveProperty()***

Testing Exceptions

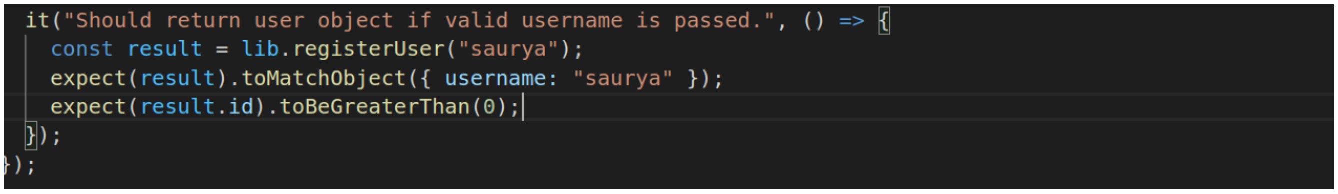


```
describe("registerUser", () => {
  it("Should throw error if username is falsy", () => {
    //null, undefined, NaN, '', 0, false

    const result = lib.registerUser(null);
    expect(result).toThrow();
    /* Above code doesnt makes sense because we are not getting
    any result from the above function. So result have notthing
    to throw exception. So to test exceptions we have to use callback
    function */

    expect(() => {
      lib.registerUser(null);
    }).toThrow();
  });
});
```

Implementing Parameterized Test:

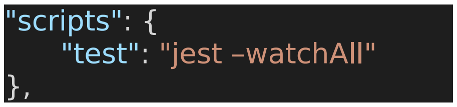


```
describe("registerUser", () => {
  it("Should throw error if username is falsy", () => {
    const args = [null, undefined, NaN, "", 0, false];
    args.forEach((a) => {
      expect(() => {
        lib.registerUser(a);
      }).toThrow();
    });
  });
});
```



```
  it("Should return user object if valid username is passed.", () => {
    const result = lib.registerUser("saurya");
    expect(result).toMatchObject({ username: "saurya" });
    expect(result.id).toBeGreaterThan(0);
  });
});
```

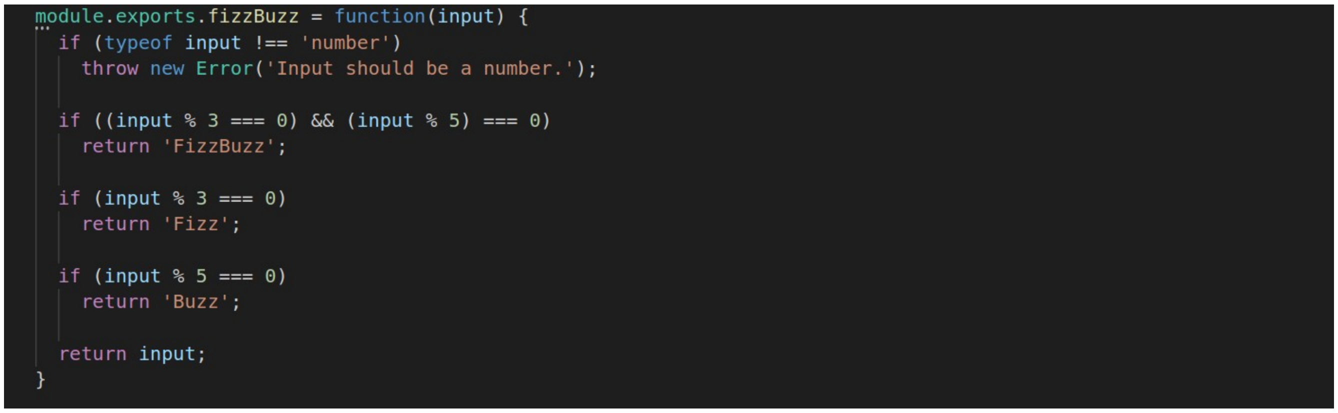
Continuously Running Tests



```
"scripts": {
    "test": "jest –watchAll"
},
```

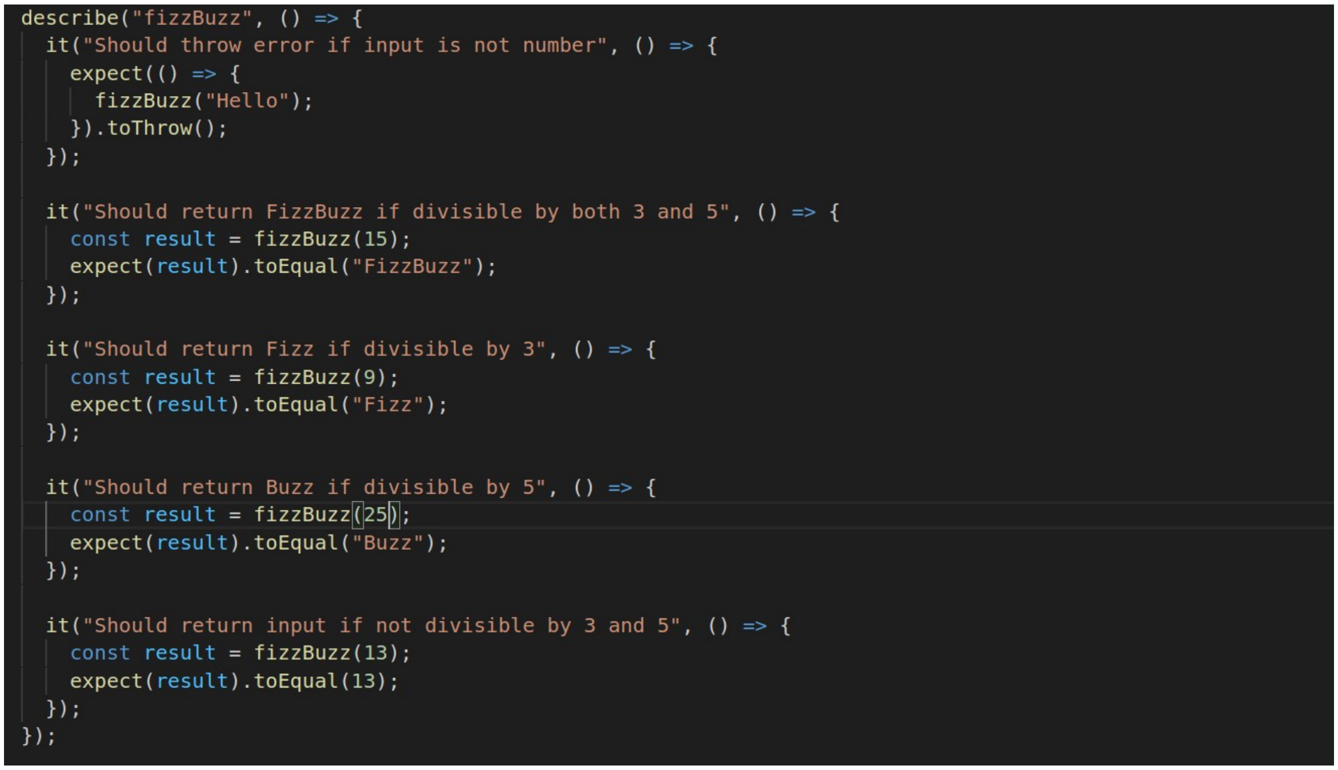
Exercise:

exercise1.js:



```
module.exports.fizzBuzz = function(input) {
  if (typeof input !== 'number')
    throw new Error('Input should be a number.');

  if ((input % 3 === 0) && (input % 5) === 0)
    return 'FizzBuzz';

  if (input % 3 === 0)
    return 'Fizz';

  if (input % 5 === 0)
    return 'Buzz';

  return input;
}
```

test/lib.test.js:



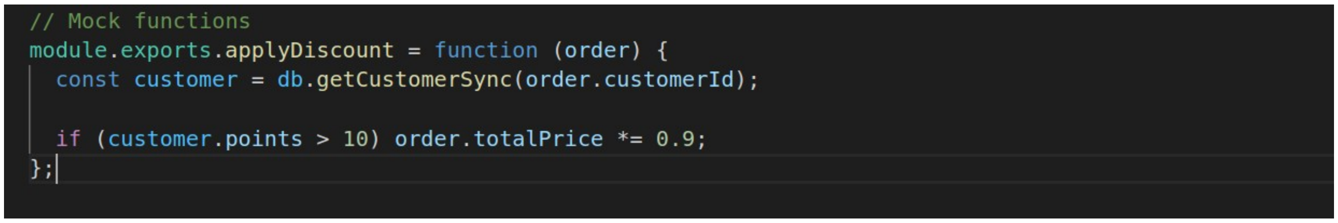
```
describe("fizzBuzz", () => {
  it("Should throw error if input is not number", () => {
    expect(() => {
      fizzBuzz("Hello");
    }).toThrow();
  });

  it("Should return FizzBuzz if divisible by both 3 and 5", () => {
    const result = fizzBuzz(15);
    expect(result).toEqual("FizzBuzz");
  });

  it("Should return Fizz if divisible by 3", () => {
    const result = fizzBuzz(9);
    expect(result).toEqual("Fizz");
  });

  it("Should return Buzz if divisible by 5", () => {
    const result = fizzBuzz(25);
    expect(result).toEqual("Buzz");
  });

  it("Should return input if not divisible by 3 and 5", () => {
    const result = fizzBuzz(13);
    expect(result).toEqual(13);
  });
});
```

```
it("Should throw error if input is not number", () => {
    expect(() => { fizzBuzz("Hello") }).toThrow();
    expect(() => { fizzBuzz(null) }).toThrow();
    expect(() => { fizzBuzz(undefined) }).toThrow();
    expect(() => { fizzBuzz(true) }).toThrow();
});
```

Creating Simple Mock Function

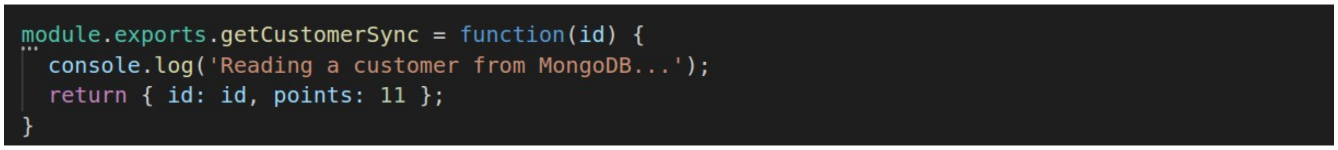
Unit testing a function that directly or indirectly talks to an external resource.

lib.js:



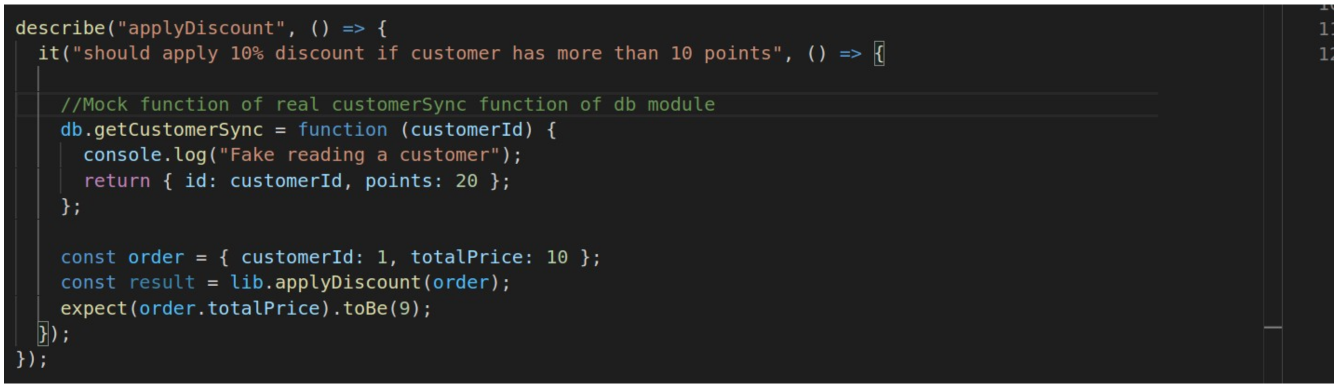
```
// Mock functions
module.exports.applyDiscount = function (order) {
  const customer = db.getCustomerSync(order.customerId);

  if (customer.points > 10) order.totalPrice *= 0.9;
};
```

db.js:



```
module.exports.getCustomerSync = function(id) {
  console.log('Reading a customer from MongoDB...');
  return { id: id, points: 11 };
}
```

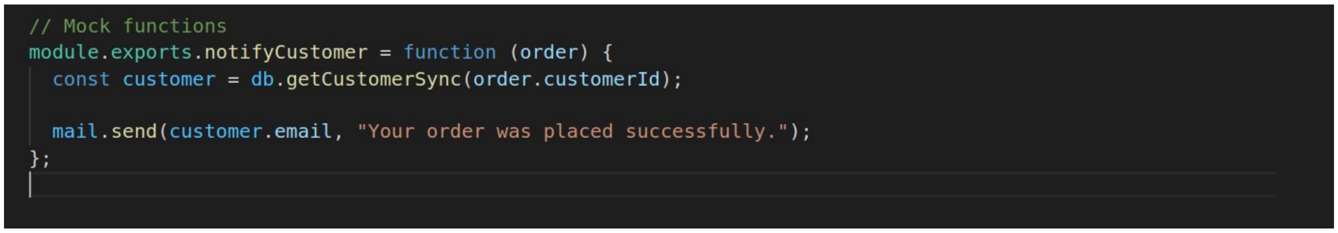
lib.test.js:



```
describe("applyDiscount", () => {
  it("should apply 10% discount if customer has more than 10 points", () => {

    //Mock function of real customerSync function of db module
    db.getCustomerSync = function (customerId) {
      console.log("Fake reading a customer");
      return { id: customerId, points: 20 };
    };

    const order = { customerId: 1, totalPrice: 10 };
    const result = lib.applyDiscount(order);
    expect(order.totalPrice).toBe(9);
  });
});
```

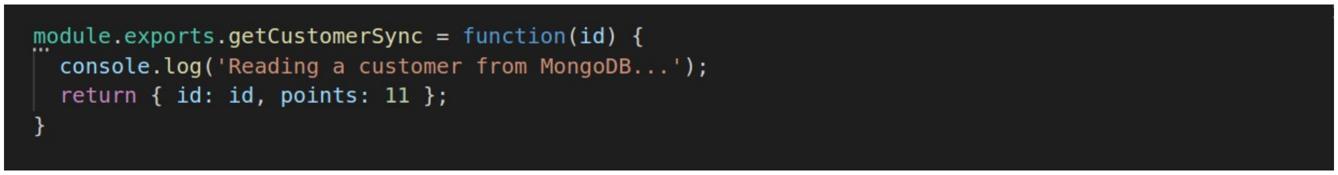
Interaction Testing

lib.js:



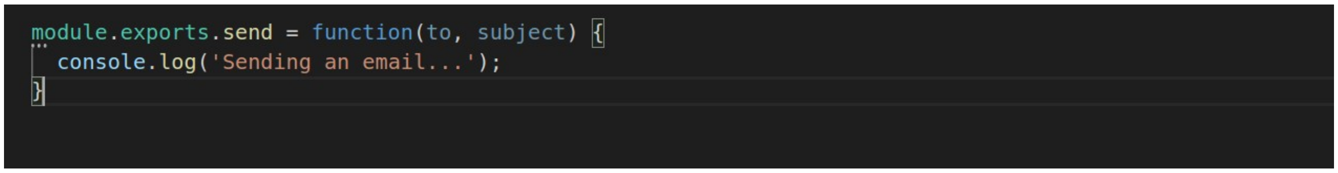
```
// Mock functions
module.exports.notifyCustomer = function (order) {
  const customer = db.getCustomerSync(order.customerId);

  mail.send(customer.email, "Your order was placed successfully.");
};
```

db.js:



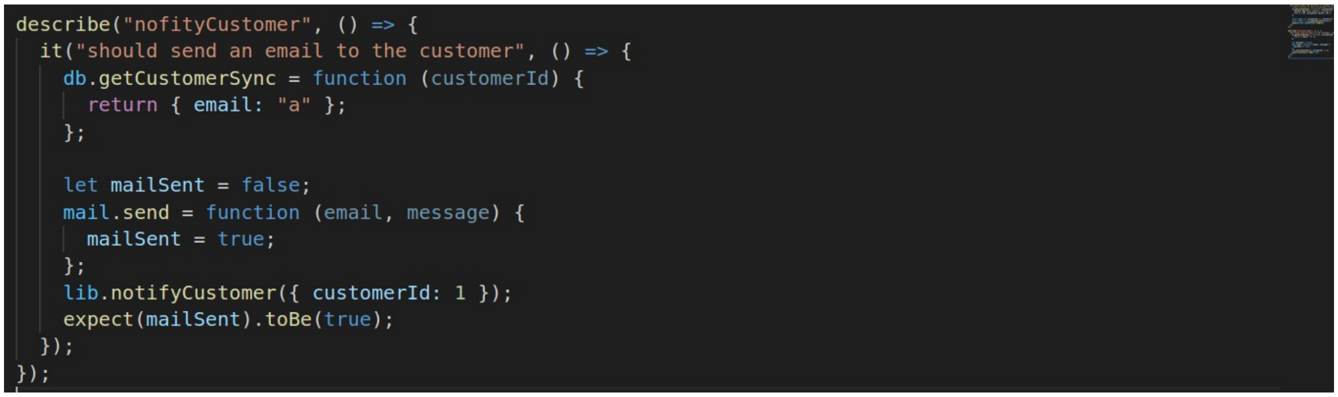
```
module.exports.getCustomerSync = function(id) {
  console.log('Reading a customer from MongoDB...');
  return { id: id, points: 11 };
}
```

mail.js:



```
module.exports.send = function(to, subject) {
  console.log('Sending an email...');
}
```

tests/lib.test.js:



```
describe("nofityCustomer", () => {
  it("should send an email to the customer", () => {
    db.getCustomerSync = function (customerId) {
      return { email: "a" };
    };

    let mailSent = false;
    mail.send = function (email, message) {
      mailSent = true;
    };
    lib.notifyCustomer({ customerId: 1 });
    expect(mailSent).toBe(true);
  });
});
```

Here we are testing the interaction of one object with another object. Here ***mail*** object of ***notifyCustomer()*** interacting with ***mail*** object of our *test* module and so on.
Now there is a better way of creating *Mock functions.*

Jest Mock Functions

In **Jest,** we have a method for creating mock functions: ***jest.fn()***
Mock function have no implementation but we can program it to return a specific value.

```
const mockFunction = jest.fn();
mockFunction.mockReturnValue(1);
const result = mockFunction();
```

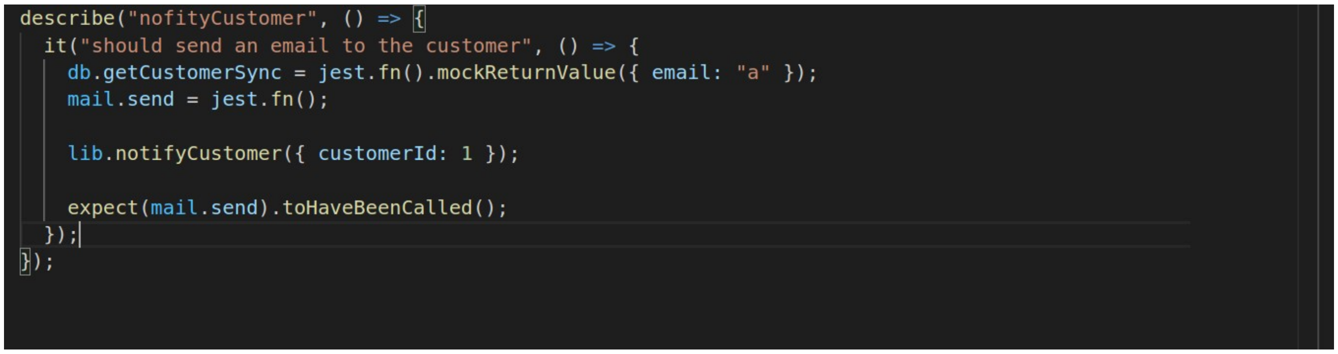
Here result will be 1.

We can also program mock function to return a *promise*.

```
mockFunction.mockResolvedValue(1);
const result = await mockFunction();
```

Simulating an error: `mockFunction.mockRejectedValue(new Error("..."));`

Rewriting interaction testing using Jest mock functions:



```
describe("nofityCustomer", () => {
  it("should send an email to the customer", () => {
    db.getCustomerSync = jest.fn().mockReturnValue({ email: "a" });
    mail.send = jest.fn();

    lib.notifyCustomer({ customerId: 1 });

    expect(mail.send).toHaveBeenCalled();
  });
});
```

To check if the arguments have been passed through this method:

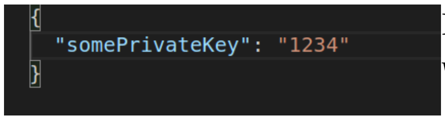
```
expect(mail.send).toHaveBeenCalled();
expect(mail.send.mock.calls[0][0]).toBe("a");
expect(mail.send.mock.calls[0][1]).toMatch(/order/);
```

calls[0][0] : first call first argument

calls[0][1] : first call second argument

Exercise:

When we run our test using **Jest**, it configures ***NODE_ENV*** to ***test.*** In our *config* folder we don't have configuration for our test file. So add *config/test.json:*



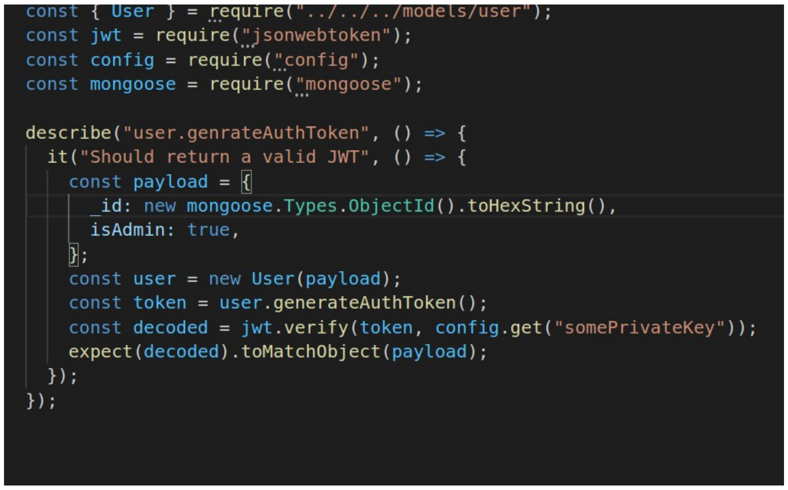
```json
{
  "somePrivateKey": "1234"
}
```

If we set *somePrivateKey* here, we don't have to set *environment variable* explicitly in command.

We cannot set ***_id*** to simple numbers like 1, 2 etc. So property of ***_id*** should be a valid objectId. Every time we created new ***_id,*** it will be different. So we don't have specific value of ***_id*** to expect. So we save this property in ***payload*** and *expect* decoded object to match ***payload***.

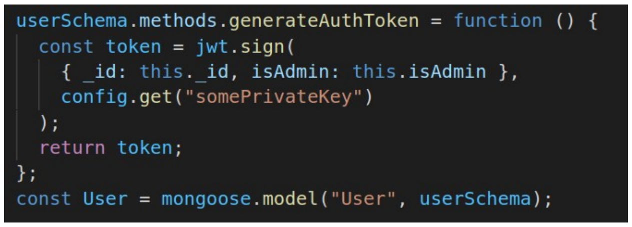
Standard ***_id*** is represented using array of numbers. But what we received in *decoded* is Hexadecimal representation. It is because when we call ***generateAuthToken***, ***jwt*** library will convert ***_id*** to hexadecimal string to store it in the *payload* of the token.

tests/unit/models/test.user.js:



```javascript
const { User } = require("../../../models/user");
const jwt = require("jsonwebtoken");
const config = require("config");
const mongoose = require("mongoose");

describe("user.genrateAuthToken", () => {
  it("Should return a valid JWT", () => {
    const payload = {
      _id: new mongoose.Types.ObjectId().toHexString(),
      isAdmin: true,
    };
    const user = new User(payload);
    const token = user.generateAuthToken();
    const decoded = jwt.verify(token, config.get("somePrivateKey"));
    expect(decoded).toMatchObject(payload);
  });
});
```

models/user.js:



```javascript
userSchema.methods.generateAuthToken = function () {
  const token = jwt.sign(
    { _id: this._id, isAdmin: this.isAdmin },
    config.get("somePrivateKey")
  );
  return token;
};
const User = mongoose.model("User", userSchema);
```

13. Integration Testing

Preparing the App

```
"scripts": {
     "test": "jest --watchAll –verbose"
},
```

This helps us troubleshoot problems.

Setting up the Test DB

config/test.json

```
{
     "somePrivateKey": "1234",
     "db": "mongodb://localhost/project-vidly_tests"
}
```

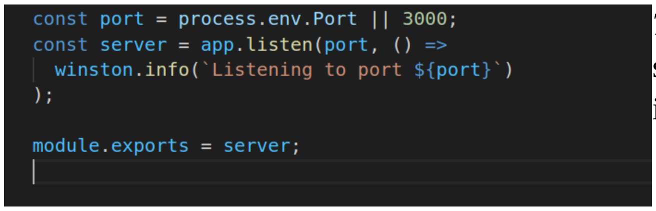
startup/database.js

```
module.exports = function () {
     const db = config.get("db");
     mongoose
          .connect(db)
          .then(() => winston.info(`Conected to the database ${db}...`));
```

Writing first Integration Test

npm i supertest –save-dev

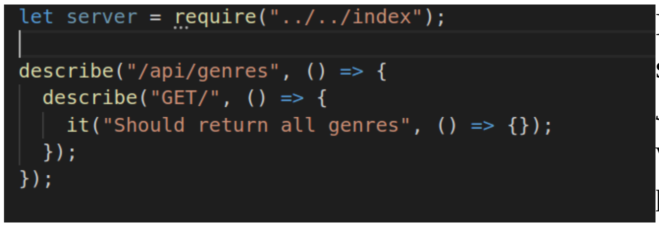
index.js:



```
const port = process.env.Port || 3000;
const server = app.listen(port, () =>
  winston.info(`Listening to port ${port}`)
);

module.exports = server;
```

*The **listen()*** method returns a server object. We should export this server so we can load this server in our test file.

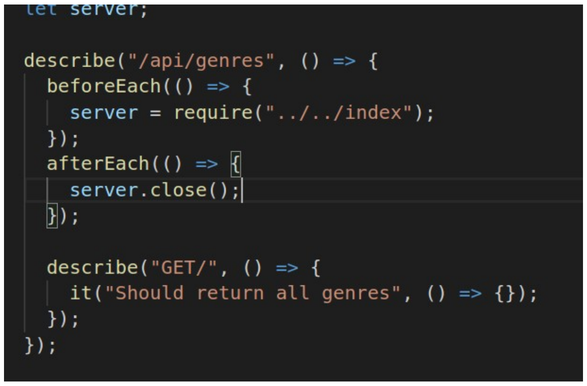
Create *tests/integration/genres.test.js:*



```
let server = require("../../index");

describe("/api/genres", () => {
  describe("GET/", () => {
    it("Should return all genres", () => {});
  });
});
```

Now first time we load this integration test, this server will listen on port 3000. Changing this code, *Jest* will re-run this code. Loading this server again, we get exception because already server running on port 3000.

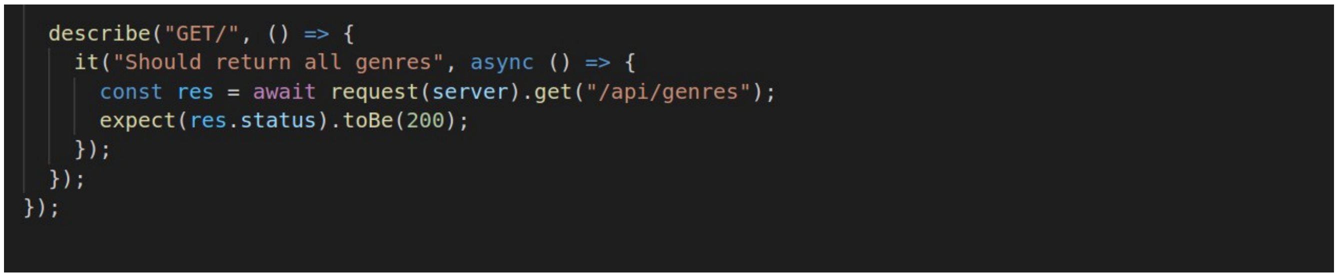
So when writing integration test, we should load the server before and close it after each test.



```
let server;

describe("/api/genres", () => {
  beforeEach(() => {
    server = require("../../index");
  });
  afterEach(() => {
    server.close();
  });

  describe("GET/", () => {
    it("Should return all genres", () => {});
  });
});
```

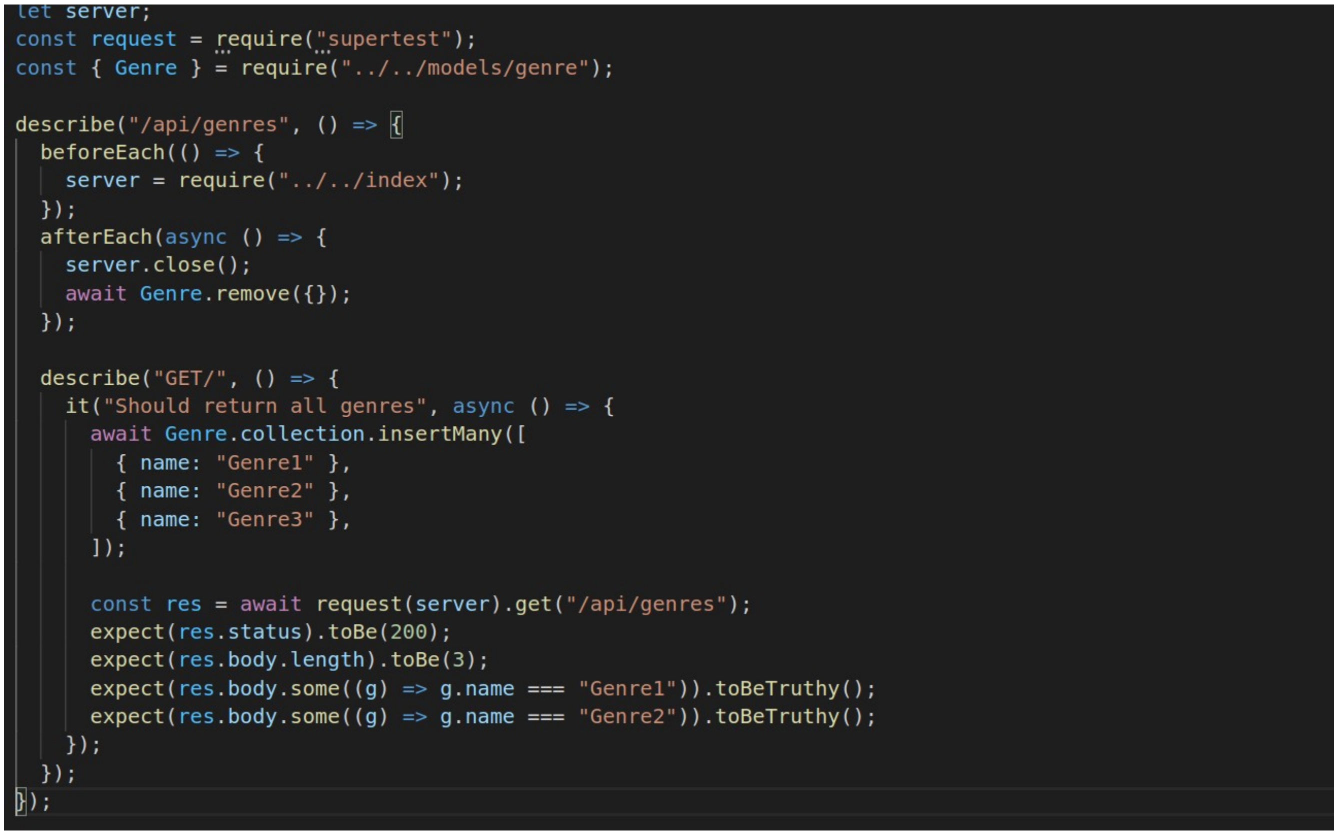
Now:

```
const request = require("supertest");
```



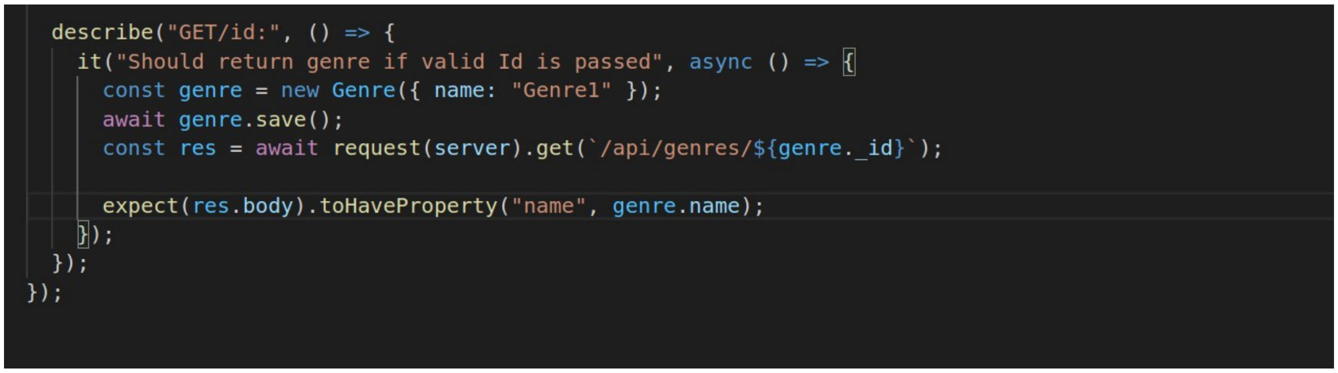
```
  describe("GET/", () => {
    it("Should return all genres", async () => {
      const res = await request(server).get("/api/genres");
      expect(res.status).toBe(200);
    });
  });
});
```

Populating the Test Database



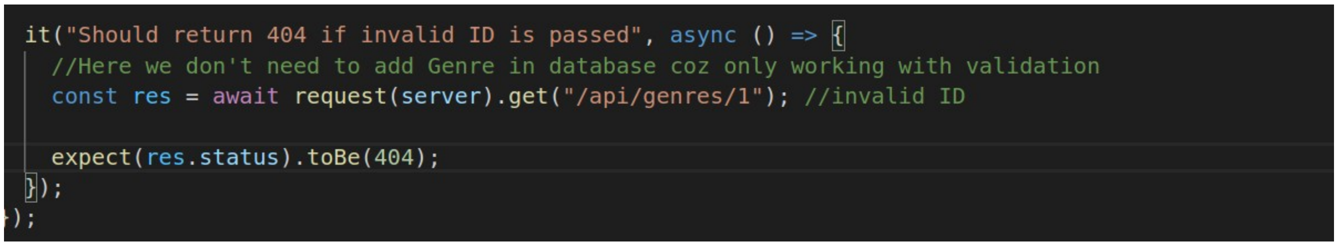
```
let server;
const request = require("supertest");
const { Genre } = require("../../models/genre");

describe("/api/genres", () => {
  beforeEach(() => {
    server = require("../../index");
  });
  afterEach(async () => {
    server.close();
    await Genre.remove({});
  });

  describe("GET/", () => {
    it("Should return all genres", async () => {
      await Genre.collection.insertMany([
        { name: "Genre1" },
        { name: "Genre2" },
        { name: "Genre3" },
      ]);

      const res = await request(server).get("/api/genres");
      expect(res.status).toBe(200);
      expect(res.body.length).toBe(3);
      expect(res.body.some((g) => g.name === "Genre1")).toBeTruthy();
      expect(res.body.some((g) => g.name === "Genre2")).toBeTruthy();
    });
  });
});
```

Testing Routes with Parameters



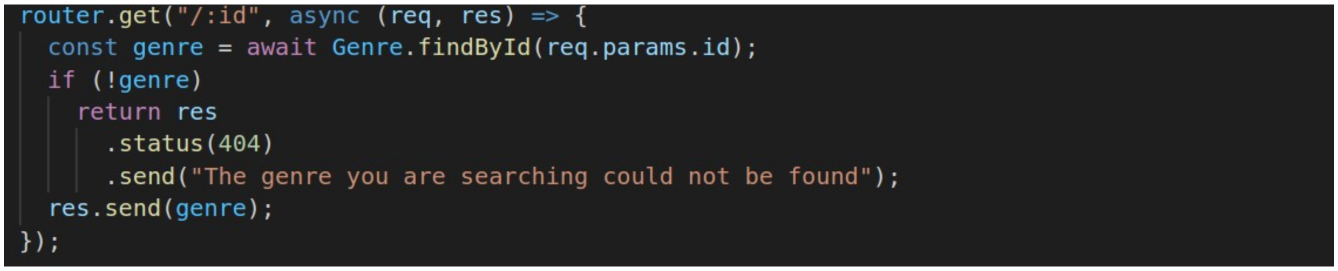
```
   describe("GET/id:", () => {
     it("Should return genre if valid Id is passed", async () => {
       const genre = new Genre({ name: "Genre1" });
       await genre.save();
       const res = await request(server).get(`/api/genres/${genre._id}`);

       expect(res.body).toHaveProperty("name", genre.name);
     });
   });
 });
```

Validating ObjectIDs



```
it("Should return 404 if invalid ID is passed", async () => {
  //Here we don't need to add Genre in database coz only working with validation
  const res = await request(server).get("/api/genres/1"); //invalid ID

  expect(res.status).toBe(404);
});
});
```

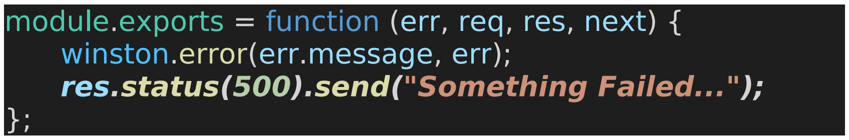
Here our test fails. Lets look at the route in *genres.js*:



```
router.get("/:id", async (req, res) => {
  const genre = await Genre.findById(req.params.id);
  if (!genre)
    return res
      .status(404)
      .send("The genre you are searching could not be found");
  res.send(genre);
});
```

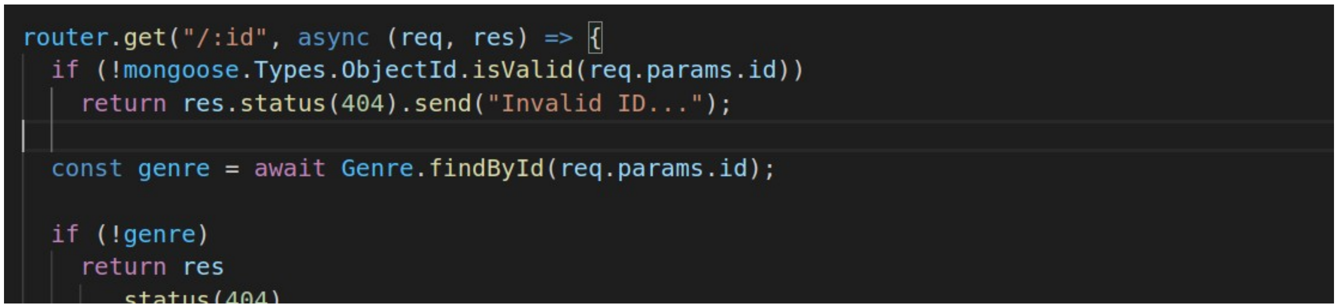
We expected 404 but received 500. We have an error coming from winston.
CastError: Cast to ObjectId failed for value "1" at path "_id" for model "Genre"

Now look at error middleware:



```
module.exports = function (err, req, res, next) {
    winston.error(err.message, err);
    res.status(500).send("Something Failed...");
};
```

Previously we validated _id using Joi but it was _id property of body of the request. But here ***_id*** is a route parameter. So before calling ***Genre.findById()*** make sure that ***req.params.id*** is valid Object ID. So in *genres.js:*

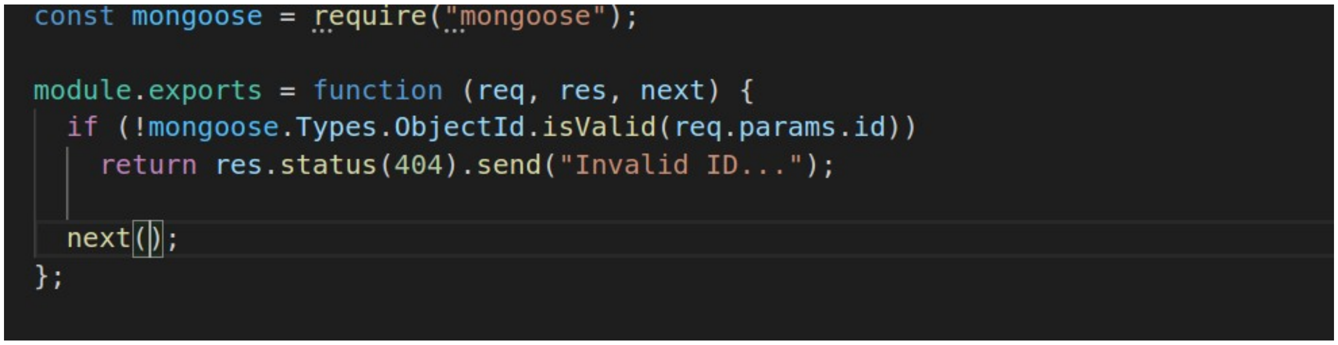


```
router.get("/:id", async (req, res) => {
  if (!mongoose.Types.ObjectId.isValid(req.params.id))
    return res.status(404).send("Invalid ID...");

  const genre = await Genre.findById(req.params.id);

  if (!genre)
    return res
      status(404)
```

Now throughout the application, we need the above logic at various endpoints which gives a single resource. So we should refactor this code and move this logic to a middleware function.

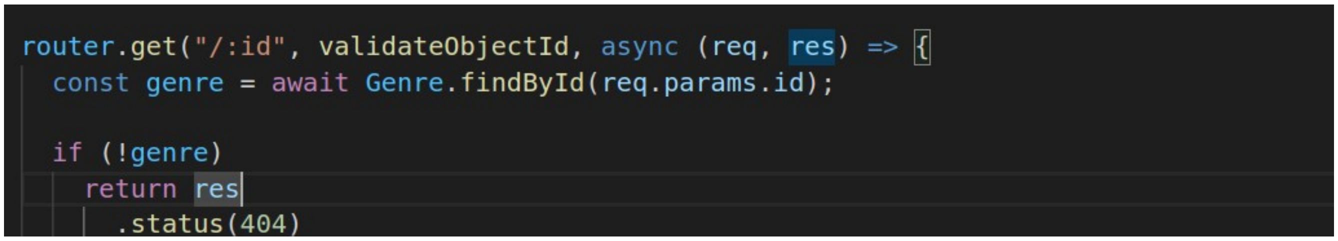
Refactoring with Confidence

Create *middleware/validateObjectId.js:*



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

module.exports = function (req, res, next) {
  if (!mongoose.Types.ObjectId.isValid(req.params.id))
    return res.status(404).send("Invalid ID...");

  next();
};
```

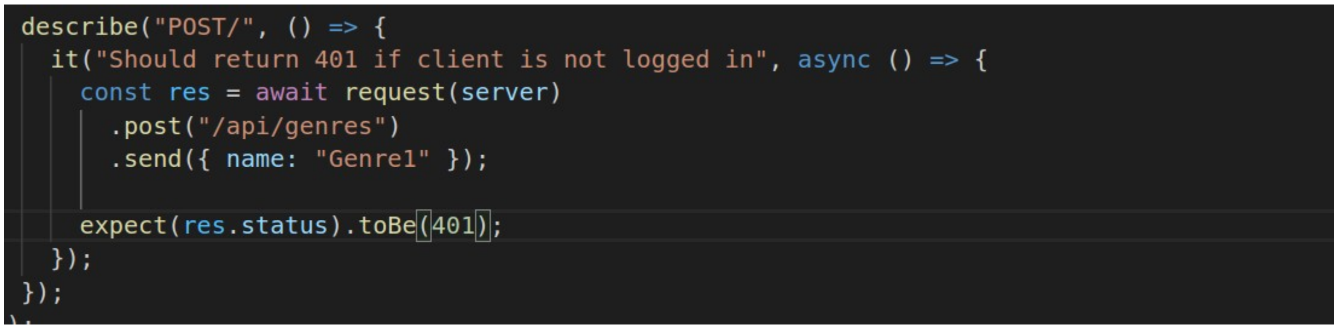
So we can use this as middleware function,



```
router.get("/:id", validateObjectId, async (req, res) => {
  const genre = await Genre.findById(req.params.id);

  if (!genre)
    return res
      .status(404)
```

Testing the Authorization

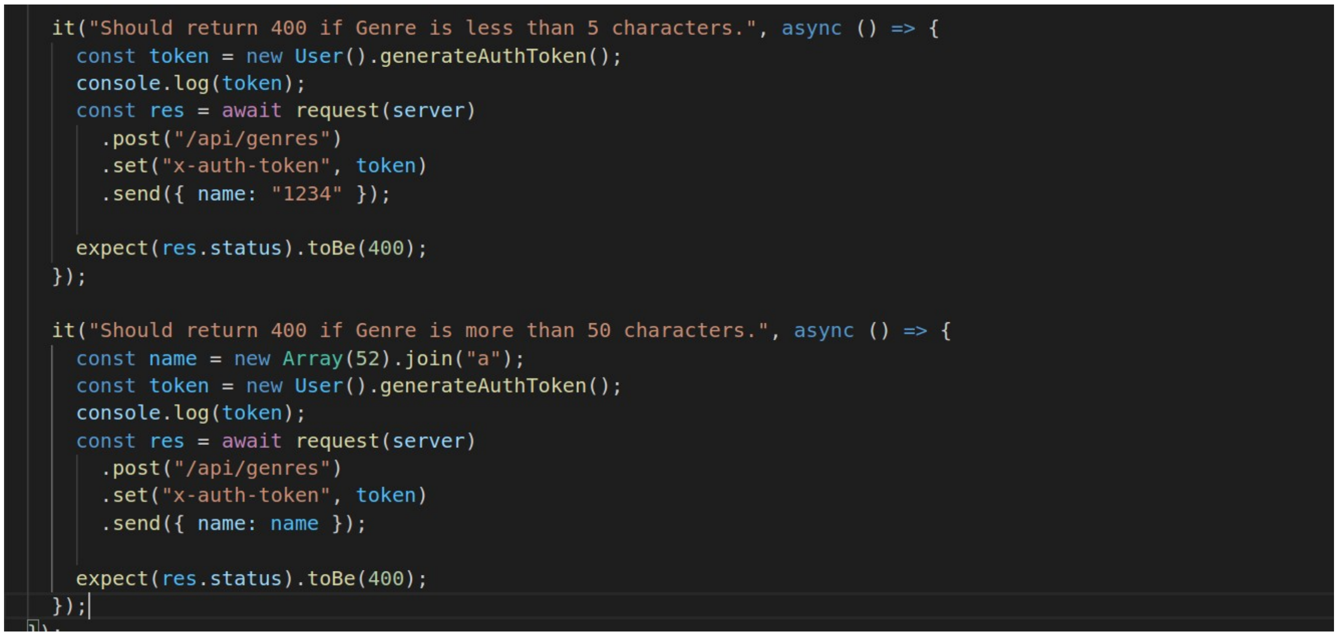
Testing the route handler for creating a new Genre:



```
describe("POST/", () => {
  it("Should return 401 if client is not logged in", async () => {
    const res = await request(server)
      .post("/api/genres")
      .send({ name: "Genre1" });

    expect(res.status).toBe(401);
  });
});
```

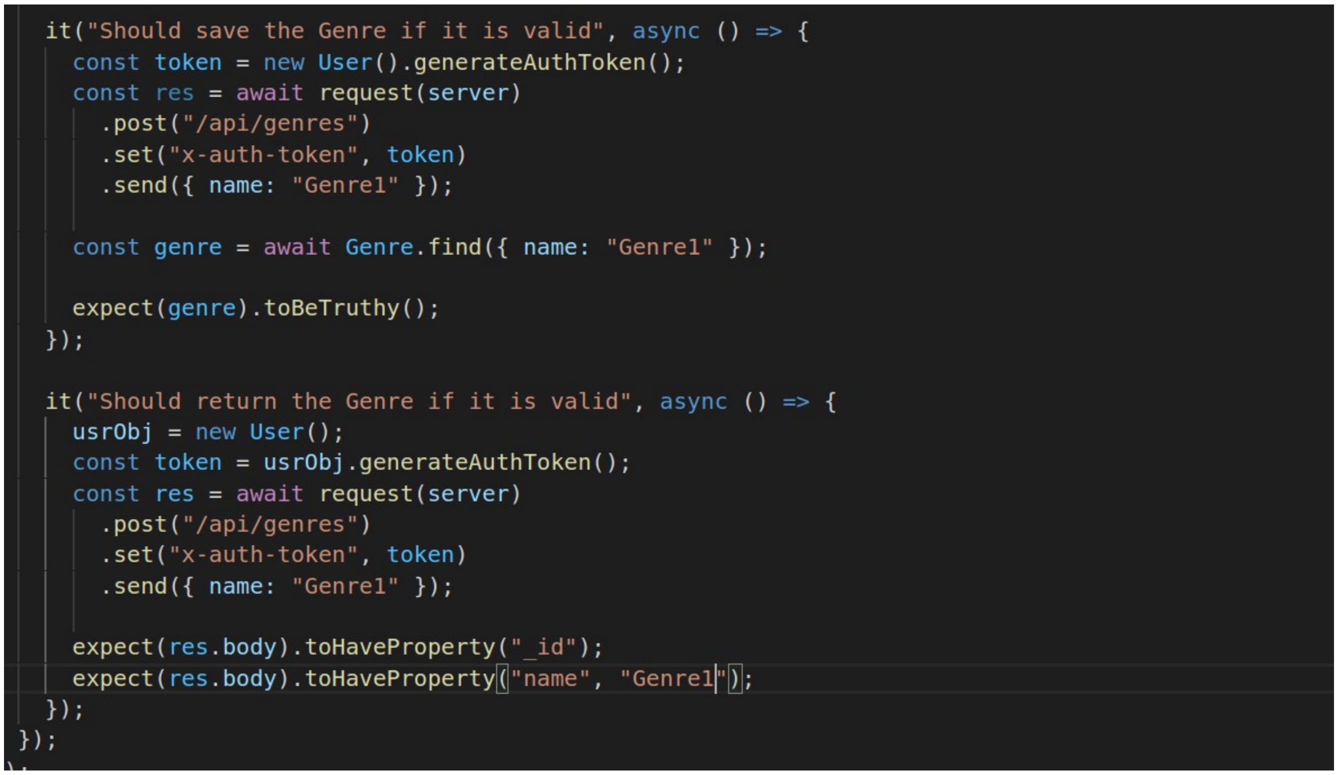
Testing Invalid Inputs

Assume client is logged in but sending invalid *Genre*.



```
  it("Should return 400 if Genre is less than 5 characters.", async () => {
    const token = new User().generateAuthToken();
    console.log(token);
    const res = await request(server)
      .post("/api/genres")
      .set("x-auth-token", token)
      .send({ name: "1234" });

    expect(res.status).toBe(400);
  });

  it("Should return 400 if Genre is more than 50 characters.", async () => {
    const name = new Array(52).join("a");
    const token = new User().generateAuthToken();
    console.log(token);
    const res = await request(server)
      .post("/api/genres")
      .set("x-auth-token", token)
      .send({ name: name });

    expect(res.status).toBe(400);
  });
```

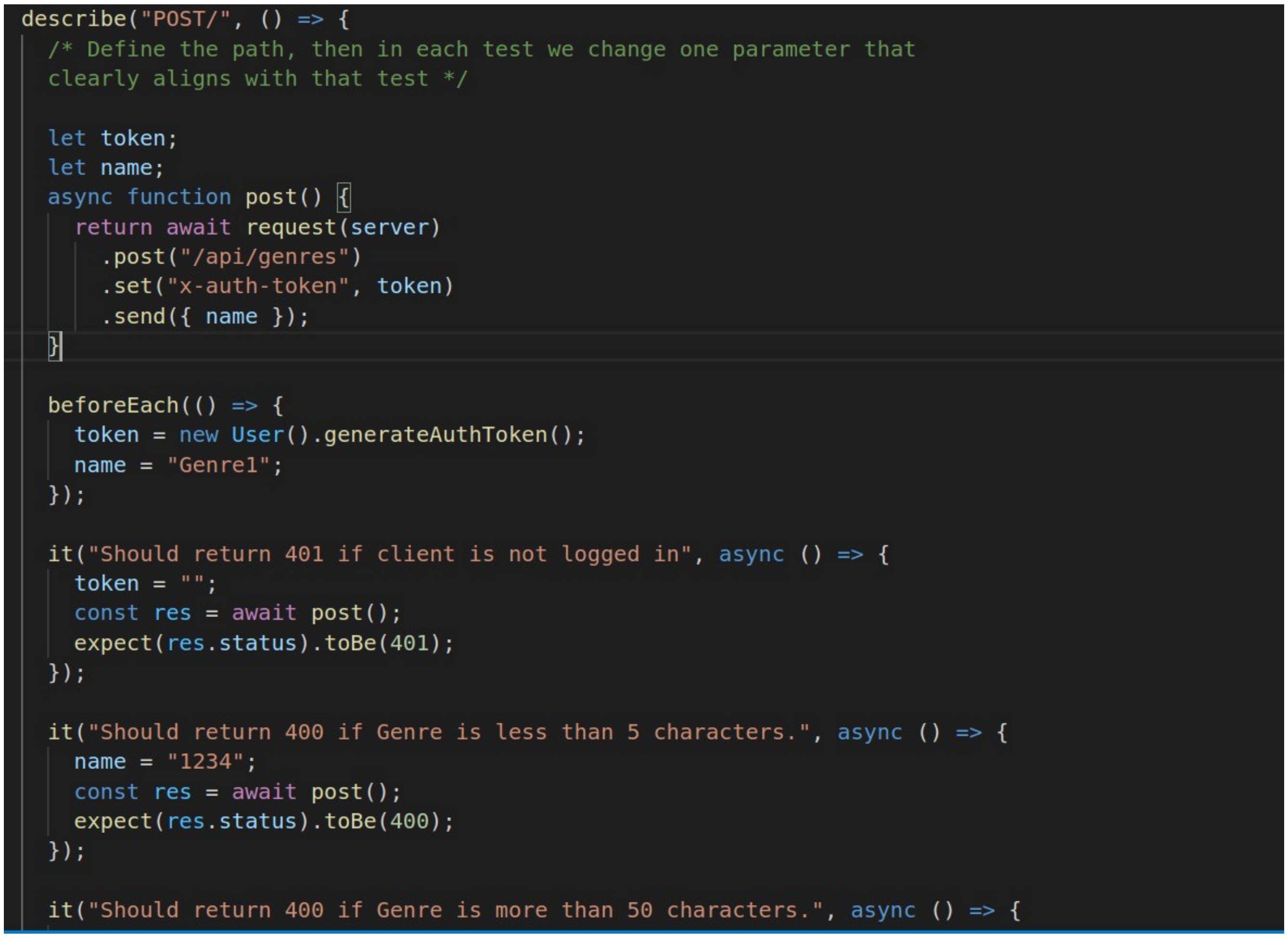
Testing Valid Inputs



```
  it("Should save the Genre if it is valid", async () => {
    const token = new User().generateAuthToken();
    const res = await request(server)
      .post("/api/genres")
      .set("x-auth-token", token)
      .send({ name: "Genre1" });

    const genre = await Genre.find({ name: "Genre1" });

    expect(genre).toBeTruthy();
  });

  it("Should return the Genre if it is valid", async () => {
    usrObj = new User();
    const token = usrObj.generateAuthToken();
    const res = await request(server)
      .post("/api/genres")
      .set("x-auth-token", token)
      .send({ name: "Genre1" });

    expect(res.body).toHaveProperty("_id");
    expect(res.body).toHaveProperty("name", "Genre1");
  });
});
```

Writing Clean Test

Many times we have repeated the same line of code.

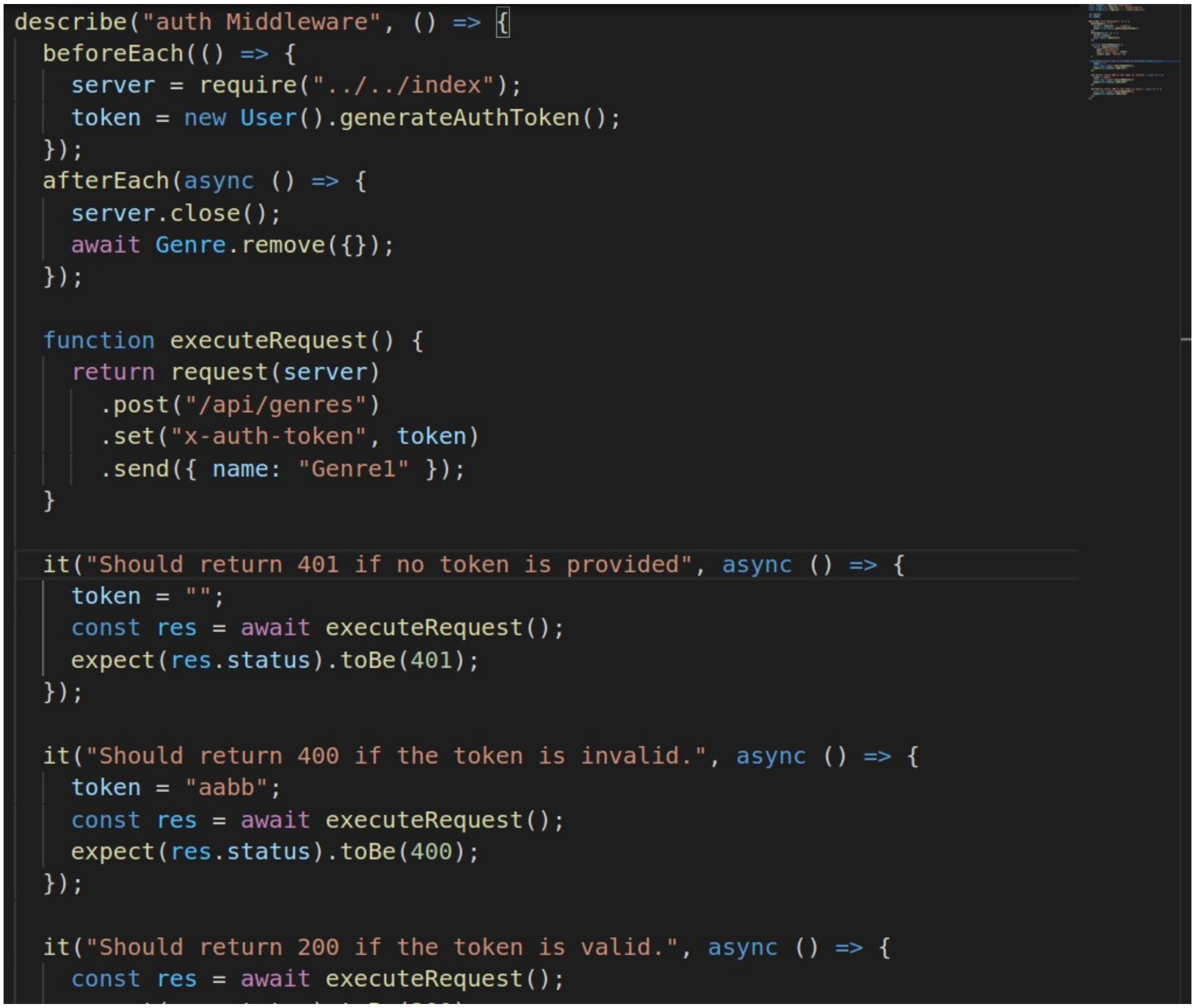


```
describe("POST/", () => {
  /* Define the path, then in each test we change one parameter that
  clearly aligns with that test */

  let token;
  let name;
  async function post() {
    return await request(server)
      .post("/api/genres")
      .set("x-auth-token", token)
      .send({ name });
  }

  beforeEach(() => {
    token = new User().generateAuthToken();
    name = "Genre1";
  });

  it("Should return 401 if client is not logged in", async () => {
    token = "";
    const res = await post();
    expect(res.status).toBe(401);
  });

  it("Should return 400 if Genre is less than 5 characters.", async () => {
    name = "1234";
    const res = await post();
    expect(res.status).toBe(400);
  });

  it("Should return 400 if Genre is more than 50 characters.", async () => {
```

Testing the Auth Middleware

Here we have tested the path where we don't have token. If we look at the Authorization middleware *middlewares/auth.js*, we have multiple execution paths. Like *not valid webToken…*

Add *integration/auth.test.js:*

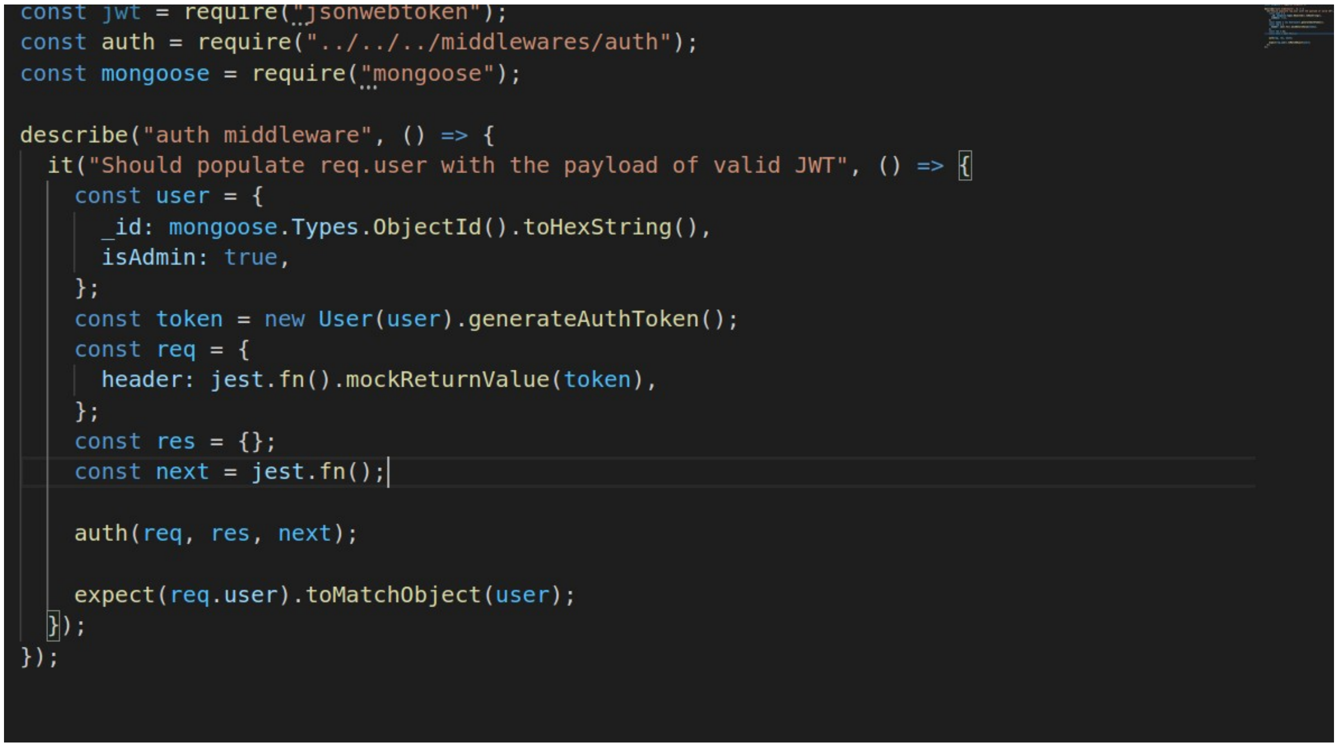


```
describe("auth Middleware", () => {
  beforeEach(() => {
    server = require("../../index");
    token = new User().generateAuthToken();
  });
  afterEach(async () => {
    server.close();
    await Genre.remove({});
  });

  function executeRequest() {
    return request(server)
      .post("/api/genres")
      .set("x-auth-token", token)
      .send({ name: "Genre1" });
  }

  it("Should return 401 if no token is provided", async () => {
    token = "";
    const res = await executeRequest();
    expect(res.status).toBe(401);
  });

  it("Should return 400 if the token is invalid.", async () => {
    token = "aabb";
    const res = await executeRequest();
    expect(res.status).toBe(400);
  });

  it("Should return 200 if the token is valid.", async () => {
    const res = await executeRequest();
```

In *middlewares/auth.js*, if the client provides a valid token, we wanna make sure that ***req.user*** is populated with payload of ***jwt.*** However *supertest* library, we don't have access to ***req*** object. So we need to write the unit test.

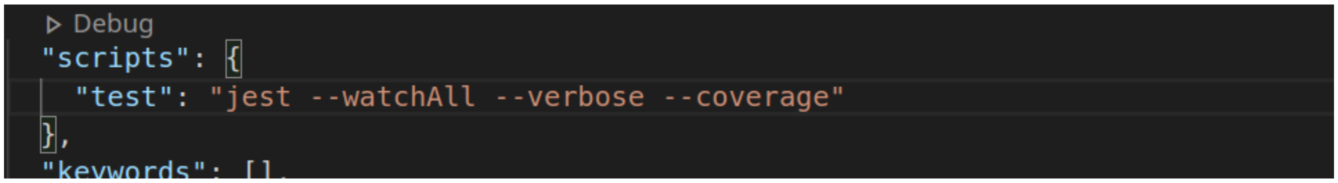
Unit Testing the Auth Middleware

Unit test and Integration test complements each other. So create *unit/middleware/auth.test.js*



```javascript
const jwt = require("jsonwebtoken");
const auth = require("../../../middlewares/auth");
const mongoose = require("mongoose");

describe("auth middleware", () => {
  it("Should populate req.user with the payload of valid JWT", () => {
    const user = {
      _id: mongoose.Types.ObjectId().toHexString(),
      isAdmin: true,
    };
    const token = new User(user).generateAuthToken();
    const req = {
      header: jest.fn().mockReturnValue(token),
    };
    const res = {};
    const next = jest.fn();

    auth(req, res, next);

    expect(req.user).toMatchObject(user);
  });
});
```

Code Coverage



```json
▷ Debug
"scripts": {
  "test": "jest --watchAll --verbose --coverage"
},
"keywords": [],
```

14. Test Driven Development