Import & export in NodeJs:

Logger.js file

Function sayHello(message){

Console.log(‘message”);

}

module.exports.sayHello = sayHello;

main.js file

const logger = require(‘./logger’);

logger.sayHello(‘vahid’);

Built-in Modules in NodeJs:

There are some useful built-in modules in Node-JS that provide facilities for us. Using these Built-in modules are the samae as regular module and done with require function. For example, “path” module:

Const path = require(‘path’);

Var pathObj = path.parse(\_\_fileName);

------------------------------------------------------

Another built-in module is os, which includes a lot of useful function related to operating system. For example:

Const os = require(‘os’);

Console.log(`free memory: ${os.totalmem()}`);

Another built-in module is fs which work with files in nodejs. It is worth noting that in fs module, each function has two version: sync version and async version. In async version each function has another extra parameter as callback function, that has two arguments. One for result and another for error value.example:

const fs = require('fs');

fs.readdir('./' , function(err , files){

    if (err) console.log('Error:' , err);

    else console.log(files);

});

Events: events are signals that emitted and another function listen to them and based on the content of those events, do special job. Raising an event done with “emit” command and Registering and listener doing with “on” command. Each listener has a callback function which do an specific job. Example:

const EventEmitter = require('events');

const emitter = new EventEmitter();

//Register a listener

Emitter.on(‘messageLogged’ , function(){

Console.log(‘message logged’);

});

//Raise an Event

Emitter.emit(‘messageLogged’);

We can send an arguments to an event such like this:

//Register a listener

Emitter.on(‘messageLogged’ , function(arg){

Console.log(‘message logged’ , arg);

});

//Raise an Event

Emitter.emit(‘messageLogged’ , {id: 1 , url: ‘https://www.google.com’});

HttpModules: these modules are another type of events which listen to different ports and urls. For example:

const Logger = require('./logger');

const http = require('http');

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

    if (req.url === '/'){

        res.write('Hello World');

        res.end();

    }

    if (req.url === '/api/courses'){

        res.write(JSON.stringify([1 , 2 , 3]));

        res.end();

    }

});

serevr.listen(3000);

console.log('Listening to Port 3000');

To see all of the built-in modules in Node-js you can see website nodejs.org in “Docs” section.

NPM (Node Package Manager)

There are a lot of useful packages and classes which are written by third-party providers and we can use them for free. They are in npmjs.com website. To use them first we have to create package-json in our project using this code:

npm init

when we type this command in command prompt, some of questions asked us to build that. To ignore these questions we use this command when create package.json:

npm init –yes

Tip: when node show a require command, analyze the meaning of that in 3 guesses.

1. It’s may be a core module (internal utilities such as fs, http, ….)
2. It’s may be a file or folder
3. Or it’s may be a node module (third-party functions which in npmjs.com)

Underscore module

It’s installed with this command: npm i underscore

It has a lot of useful functions, such as “contains”, which works like this:

Var \_ = require(‘underscore’);

\_. contains ([1, 2, 3], 2) => true

Solve the problem of moving big and heavy projects

In real world we have projects which has a heavy node-modules folder with bigger than hundreds of megabytes and we don’t want to upload or send the project with its heavy node-modules folder. To solve this problem, we can don’t send this folder with project or even we can delete this folder and there is not something to worry about it. Because we can retrieve this folder in destination using package-Json config file and by typing this command: npm i.

Another issue accured when we use git repository and when we use git status, all the content of node-modules appeared and it says that you have to commit this folder. To solve this problem, we should do these steps:

1. Create a file with this name in the root of project: .gitignore.
2. List all the files and folders which we want to ignore in git changes. For example node\_modules/
3. Its finished and when we write the git status command, this list wont appear in the changed list.

Semantic Version

The versions of installed modules in package.json shows like this:

^ 8.3.4: the first number on the left is major, second is minor and third called patch.

Patch changed when the error is exists and fixes. The minor changes when the update is accured but it does not have side effect on previous versions . and major changes when a big change is accured and causes previous versions.

Carot character ^: it means that this module can work with any versions of modules with the same major number. In this example 8.x

Tilde ~: it means that this module can work just with the versions with the same major and minor number. In this example 8.3.x

When we write in package.Json without any character, it means that this project just work with the exactly same version of this module.

All of this subjects, say the “npm i” command that which version of modules should be installed in destination.

npm list

with this command can see all the installed packages with their versions and dependencies. If you want to see just main packages try this command: npm list –depth=0

npm view

with this command can see all the information about a specific module like this:

npm view mongoose

and if we just see the specific modules dependencies, use this command:

npm view mongoose dependencies.

Tip: if you want to install a specific version of a module, do it like this:

npm I [mongoose@4.2.1](mailto:mongoose@4.2.1)

npm outdated and check-update

with “npm outdated” command can see the package’s status in project, that which of them is updated or not.

With the command npm update, we can update all the packages to the latest version of this package but with the same major.

To force all the packages to update with the latest version, first you have to update package.Json use this command:

npm i -g npm-check-updates

and then install the newest version, use this command:

npm i

Development dependencies

Sometimes there are some packages that use only in development and debug mode and they aren’t related to product. Such as test related packages and jshint package. To segregate these packages in package.json with another packages, you should install them in different way like this:

npm i jshint –save-dev

Uninstall modules

You can uninstall each module with this command: npm un mongoose

Then its uninstall and remove from package.json and node\_modules folder.

Restful API

It is a method to connect between client and server. It has five method:

* Get all: it returns all the data of specific table
* Get Specific row: it sends an id of the specific data and get the full info of it
* PUT: it sends an id of the specific data in the header and the content of data in body and update that data according to sent data
* POST: it sends the content of data in body and insert data according to sent data
* DELETE: it sends an id of the specific data in the header and delete that data in database

express

it’s a useful and popular framework which simplify working with server and maintaining big projects with a lot of routs and using it help us to get rid of complicated if statements.

To use it, first you have to install it with this command: npm i express

Then use it like this:

const express = require('express');

const app = new express();

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

    res.send('hello world');

});

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

    res.send([1 , 2 , 3]);

});

app.listen(3000 , () => console.log ('listening to port 3000 ...'));

nodemon

It stands for node monitor and such a useful tool to get rid of tedious need to run our program after each change. Its just need to install nodemon and run it one time. After that nodemon monitor your program and every time you update your program and save it, nodemon monitor your change and automatically run again your program. To install it use this command: npm i -g nodemon

And for first time type this command: nodemon app.js and just it.

Tip: in real production environment and hosting environment, there is dynamic port number and port 300 may be not exist and free. To solve this problem, we have to use dynamic port number based on hosting port number. So act like this:

Const port = process.env.port;

app.listen(port , () => console.log(`listening to port ${port} …`));

Get specific data using express

To retrieve specific data from db or an array, as i mentioned before, it needs to send an id in our request and check it in get to find it in array or db. For example:

const app = new express();

const courses = [

{id: 1 , name: 'math'},

{id: 2 , name: 'history'},

{id: 3 , name: 'geography'}

];

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

    res.send('hello world!!');

});

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

    res.send([1 , 2 , 3]);

});

app.get('/api/courses/:id' , (req , res) => {

    const course = courses.find(c => c.id === parseInt(req.params.id));

    if(!course) res.status(404).send('the course is not found');

    res.send(course);

});

const port = process.env.port || 3000;

app.listen(port , () => console.log (`listening to port ${port} ...`));

Post request

To run and test post request need to install postman and login to it. And the code like this.

const express = require('express');

const app = new express();

app.use(express.json());

app.post('/api/courses' , (req , res) =>{

    const course ={

        id: courses.length + 1 ,

        name: req.body.name

    };

    courses.push(course);

    res.send(course);

});

Error Handling

We can not rely on client that send us a true value and we have to check it before save it or manipulate it. There are some useful modules to error handling and one of them is joi. To install joi use this command: npm i joi

And to use it, act like this:

app.post('/api/courses' , (req , res) => {

    const schema = Joi.object({

        name: Joi.string()

            .min(3)

            .required()});

    const result = schema.validate(req.body);

    if(result.error){

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

        return;

    }

    const course ={

        id: courses.length + 1 ,

        name: req.body.name

    };

    courses.push(course);

    res.send(course);

});

PUT request

To update any course, have to use put method. In put method, like a get specific method, first should find a course based on its id and if it doesn’t find, return not found error. After that you have to validate the value that you want to update that row with it. And if there is not any problem, update that row. The way is like this:

app.put('/api/courses/:id' , (req , res) => {

    const course = courses.find(c => c.id === parseInt(req.params.id));

    if(!course)

        res.status(404).send('the course is not found');

    const { error } = validateCourse(req.body);

    if(error){

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

        return;

    }

    course.name = req.body.name;

    res.send(course);

});

function validateCourse(course){

    const schema = Joi.object({

        name: Joi.string()

            .min(3)

            .required()});

    return schema.validate(course);

}

DELETE request

To delete a specific row, first you have to find it and if its not exist raise a 404 error, and if you find it, delete it from array and return the deleted row. Like this:

app.delete('/api/courses/:id' , (req , res) => {

    const course = courses.find(c => c.id === parseInt(req.params.id));

    if(!course) return res.status(404).send('the course is not found');

    const index = courses.indexOf(course);

    courses.splice(index , 1);

    res.send(course);

});

Middleware

Middleware’s are functions which take parameters and do a specific request and return custom values or send their returns to another function. For example in previous section we can categorize app.use(express.Json()) as a middleware.

Middlewares are execute sequential and in order. When you have a next() command in your middleware, after finishing the execution of that middleware, next middleware is execute, and if there is not any middleware after that, the execution is hanging.

Built-in Middlewares

There are some built-in middlewares in node-js that do different tasks for us. For example we can mention few of them:

Express.urlencoded({ extended: true }); // which can parse traditional key:value url patterns even with complex params such as arrays or objects.

Express.static(‘public’); // with this url we can parse static files such as txt files which in public folder, without the need to mention /public folder in url.

Morgan(‘tiny’); // log all the api called in console.log

Helmet(); // secure http headers

Running environment

The program might be in different environments, for example in production, debugging, … . we can get the current environment using app.get() middleware and decide to choose which function is called or not. For example decide that morgan call only in development mode. Like this:

If (app.get(‘env’) === ‘development’){

App.use(morgan(‘tiny’));

Console.log(‘morgan enabled’);

}

But the above approach has a difficulty in real use. For example you might be a lot of this conditions and logs which you don’t want them in production mode. In such situation you have to comment all the commands before publishing your code and conversely if you want back to development mode again, you have to uncomment all these codes and absolutely you should worry about this if you forget any of these comment and uncomment functions.

A better and safer way to do this, using a debug built-in module. First you have to install it using npm i debug and then import it in your app like this:

const startupDebugger = require('debug')('app:startup');

const dbDebugger = require('debug')('app:db');

After that you should set your environment to this values in command prompt like this:

Set DEBUG=app:startup or set DEBUG=app:db

Or even: set DEBUG=app:\*

And replace console.log with debug command.

Pug template strings

Sometime we need to send html to clients not only raw data. To do this have to use template strings. There are different template strings, for example:

* Pug
* Mustache
* EJS

To use pug first install it using npm i pug command.

Then set template engine in our code to pug: app.set(‘view engine’ , ‘pug’);

After that create a folder with view name in source and set views to that: app.set(‘views’ , ‘./views’);

Then create a file with index.pug name in view folder and write this code on it:

html

    head

        title = title

    body

        h1 = message

then back to app.js and replace the simple app.get(‘/’ ) with this code:

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

    res.render('index' , { title: 'My express app' , message: 'hello world'});

});

MongoDb CRUD

To work with database, node has different modules for each database. One of these modules which uses to connect and work with mongodb database, is mongoose.

To use mongoose first you have to install it like this: npm i mongoose

Then should go to mongodb.com and install mongodb.

Then go to advanced system setting and add the address in which mongodb installed in our computer in environmental path.

Then we have to require mongoose and connect to database like this:

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/playground')

.then(() => console.log('connecting...'))

.catch((err) => console.error('could not connect' , err));

Then you have to create a schema based on your desired database in such a class format, like this:

const courseSchema = new mongoose.Schema({

    name: String,

    author: String,

    price : Number,

    tags: [ String ],

    date: { type: Date , default: Date.now },

    isPublished: Boolean

});

Then you have to create a model based on that schema and choose a name for that like this:

const Course = mongoose.model('course' , courseSchema);

Now everything is ready to start your work with database and have 4 main operations on it. Before start the crud operation, I have to mention that retrieving and writing data to database are operations which may take a long time to complete and to increase the performance of runtime, nodejs use promise structure to handle database operations.

First of all we check the insert query. To insert a row in database, first you have to create an object of schama class and then write it in database like this:

async function createCourse(){

    const course = new Course({

        name: 'zaban',

        author: 'ali',

        tags: ['lor' , 'pesar'],

        isPublished: true,

        price: 51

    });

    const result =await course.save();

    console.log(result);

}

createCourse();

For retrieve data from database, use find operation in mongoose like this:

async function getCourses(){

    const courseList = await Course.find({ name: 'vahid'}) // find rows which have the name property with ‘vahid’ value

    .sort({ name: 1}) //sort returned data(1 means ascending and -1 means descending order

    .select({ name: 1, author: 1}); //choose specific fields

    console.log(courseList);

};

There are a lot of comparison operators in mongodb that can use to create diverse and complex queries. For example :

gt: greater than

lt: less than

gte: greater than or equal

lte: less than or equal

in and nin: say that if a value in list of values or not

The example of using these operators in queries like are like this:

async function getComparisonCourses(){

    const courseList = await Course.find({ price:{ $nin: [20, 51]} })

    .sort({ price: 1})

    .select({ name: 1, price: 1});

    console.log(courseList);

};

Moreover we can use logical operators like “or” and “and” to create more complex queries like this:

async function getLogicalCourses(){

    const courseList = await Course.find()

    .or([ { author: 'Mosh' } , { isPublished: true }])

    .sort({ price: 1})

    .select({ name: 1, price: 1});

    console.log(courseList);

};

And so we can use regular expressions to compare an input string with different values. For example these patterns:

* /^value/: means retrieve all the patterns which start with value
* /value$/: means retrieve all the patterns which end with value
* /.\*value.\*/: means retrieve all the patterns which contain value

For example see this code:

async function getRegularExpressionSearch(){

const courseList = await Course.find({ name: /.\*Course.\*/i })

    .sort({ price: 1})

    .select({ name: 1, price: 1});

    console.log(courseList);

};

We can have aggregation functions too, like this:

async function getCourseCount(){

    const courseList = await Course.find({ name: /.\*Course.\*/i })

        .sort({ price: 1})

        .countDocuments()

        console.log(courseList);

    };

And also we can handle pagination subject with queries and retrieve specified part of data, like this:

async function getPaginationCount(pageNumber, pageSize){

    const courseList = await Course.find()

        .sort({ price: 1})

        .skip((pageNumber - 1) \* pageSize)

        .limit(pageSize);

        console.log(courseList);

    };

To implement update operation we have two approaches, query approach and update first approach. In query approach we have to first select data and then check the existence or other situations and then decide to update or not. Like this:

async function updateCourse(id){

    const course = await Course.findById(id);

    if (!course) return;

    course.isPublished = false;

    course.author = 'Madaeni';

    course.save();

    return course;

}

But in update-first approach we certain about existence of data and just want to update it. Like this:

async function updateCourse2(id){

    const course = await Course.findByIdAndUpdate(id,{

        $set:{

            isPublished: false,

            author: 'Jason'

        }

        } );

    return course;

}

async function updateCourse3(id){

    const course = await Course.updateOne({ \_id: id},{

        $set:{

            isPublished: false,

            author: 'Jason'

        }

        } );

    return course;

}

To delete a specified data from database using its key, we don’t have a hard work and dot it such an easy like this:

async function deleteCourse(id){

    const course = await Course.findByIdAndDelete(id);

    return course;

}

Important and final Tip: MongoDB is not a relational database like SQL server. We don’t have tables and rows and relations; beside this we have collections (equivalent to table) and documents (equivalent to rows). And moreover, we don’t have such a strict rules about the structure of data and MongoDB (and another no-SQL database) is more flexible.

Mongoose data validation

As I said before, MongoDB is very flexible and does not prevent any un-structured data to write in database. So we have to validate our data with another tools. For example in the previous course model, imagine that you want user to insert a course with required value. So you do that like this:

const courseSchema = new mongoose.Schema({

    name: {type: String, required: true}, //in this line set name property in required state

    author: String,

    price : Number,

    tags: [ String ],

    date: { type: Date , default: Date.now },

    isPublished: Boolean

});

const Course = mongoose.model('course' , courseSchema);

async function createCourse(){

    const course = new Course({

        name: 'zaban',

        author: 'ali',

        tags: ['lor' , 'pesar'],

        isPublished: true,

        price: 51

    });

    try{

        const result =await course.save();

        console.log(result);

    }

    catch(err){

        console.log(err.message);

    }

We can have more complex and conditional validators in mongoose which related to more than one field. For example we can have the condition which say that each published course must have a price. We control this condition like this:

const courseSchema = new mongoose.Schema({

    name: {type: String, required: true},

    author: String,

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

    tags: [ String ],

    date: { type: Date , default: Date.now },

    isPublished: Boolean

});

Moreover we can have another validators for string and Enum fields like this:

const courseSchema = new mongoose.Schema({

    name: {

        type: String,

        minlength: 5,

        maxlength: 255,

        required: true

    },

    category:{

        type: String,

        enum: ['web' , 'mobile' , 'network']

    },

    author: String,

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

    tags: [ String ],

    date: { type: Date , default: Date.now },

    isPublished: Boolean

});

Sometime there are situations that can not validate them using such a simple built-in validation. In these cases, we have to implement custom validators. For example, about tag property which is an array, simple validations doesn’t work and we have to implement complex instruction like this:

const courseSchema = new mongoose.Schema({

    name: {

        type: String,

        minlength: 5,

        maxlength: 255,

        required: true

    },

    category:{

        type: String,

        enum: ['web' , 'mobile' , 'network']

    },

    author: String,

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

    tags: {

        type: Array,

        validate: {

            validator: function(v){

                return v && v.length > 0;

            },

            message: 'A course should have at least one tag'

        }

    },

    date: { type: Date , default: Date.now },

    isPublished: Boolean

});

Sometimes we need to have async validator when for example we have decide validation based on the value that retrieve from database. In this situation we use like this:

tags: {

        type: Array,

        validate: {

            isAsync: true,

            validator: function(v , callback){

                setTimeout(() => {

                    const result= v && v.length > 0;

                    callback(result);

                }, 4000);

            },

            message: 'A course should have at least one tag'

        }

    },

As mentioned before, error is an object which have different fields and have different instance for each occurred error. To test and investigate that, act like this:

try{

        const result =await course.save();

        console.log(result);

    }

    catch(err){

        for (field in err.errors)

            console.log(err.errors[field].message);

    }

We have additional validators in properties for each type. For example for string type we have uppercase function which convert string to uppercase and trim which remove extra spaces from string. And for number type we have validators such as round operator. We use this validators like this:

 category:{

        type: String,

        enum: ['WEB' , 'mobile' , 'network'],

        uppercase: true,

        trim: true

    },

    author: String,

    price : {

        type: Number,

        required: function (){

            return this.isPublished

        },

        get: v => Math.round(v),

        set: v => Math.round(v)

    },

Modeling relationships

In real world applications we face with models and classes which has some different fields, but its necessary to have a relation between them. Imaging two entities course and author. Author can be a separate entity that has a lot of fields itself. There are two approach to save data of course and author in database, which include:

* Using references (Normalizations)
* Using embedded documents (Denormalization)

In first approach the data of author entity save in separate object and just its id saved in course object to maintain a relation between two entity. Like this:

let author = {

  id: 1,

  name:'vahid',

  date: '1393/01/01',

  job: 'engineer'

}

let course = {

  name: "algorithm",

  author: 1

}

In the second approach all the two entities data with their complete fields saved in an individual entity and in other world we use nested objects, like this:

let course = {

  name: "algorithm",

  author : {

    id: 1,

    name:'vahid',

    date: '1393/01/01',

    job: 'engineer'

  }

}

In the first approach we have consistency, because if one of the author fields value may be change, with a change in just one place in author entity value, we can solve the problem in all of the program, but in second approach we need multiple and inaccurate change.

But in second approach we acquire query performance, because when we need the complete information of author and course, just one query is needed, but in second approach, two query is needed. So choice between two approaches is a tradeoff between consistency and query references.

There are third approach called hybrid approach and work like this, which some important information of author saved in course entity and other information remain in author entity. Some ecommerce websites uses this approaches, for example about products.

let author = {

  id: 1,

  name:'vahid',

  date: '1393/01/01',

  job: 'engineer'

}

let course = {

  name: "algorithm",

  author: {

id: 1,

name: ‘vahid’

}

In real application use models and create reference in first approach, like this:

const Author = mongoose.model('Author', new mongoose.Schema({

  name: String,

  bio: String,

  website: String

}));

const Course = mongoose.model('Course', new mongoose.Schema({

  name: String,

  author: {

    type: mongoose.Schema.Types.ObjectId,

    ref: 'Author'

  }

}));

async function createCourse(name, author) {

  const course = new Course({

    name,

    author

  });

  const result = await course.save();

  console.log(result);

}

When we want to list courses and retrieve author and other related entities data, act like this:

async function listCourses() {

  const courses = await Course

    .find()

    .populate('author' , 'name') // says that retrieve authors entities data too and just show the name field

    .populate('category' , 'name -\_id') // says that retrieve category entities data too and just show the name field and don’t show the id field

    .select('name');

  console.log(courses);

}

Embedding documents

In the embedding approach we don’t have any reference and embed author in course directly like this:

const Author = mongoose.model('Author', authorSchema);

const Course = mongoose.model('Course', new mongoose.Schema({

  name: String,

  author: authorSchema

}));

And remaining code stay unchanged.

One important tip about embedding approach that you have to know is that, in this approach subdocument (author in this lecture) doesn’t save separately and just embed in course object. So, when you want to update authors of a specific course, do it like this:

async function updateCourse(courseId){

  const course = await Course.findById(courseId);

  course.author.name = 'Mosh';

  course.save();

}

And you write course.author.save() and not say author.save(). Because the author doesn’t exists separately.

In update first approach we can do like this:

async function updateCourse(courseId){

  const course = await Course.updateOne({ \_id: id },{

    $set:{

      'authors.name': 'saeed2'

    }

  });

}

In this approach simply we can remove author object from each course like this:

async function updateCourse(courseId){

  const course = await Course.updateOne({ \_id: id },{

    $unset:{

      'authors': ''

    }

  });

}

Array of subdocuments

If we want to have multiple subdocuments in each document, we can have it as an array. First, we need to define our schema like an array and then embed it on document like this:

const authorSchema = new mongoose.Schema({

  name: String,

  bio: String,

  website: String

});

const Course = mongoose.model('Course', new mongoose.Schema({

  name: String,

  authors: [authorSchema]

}));

Then you can update course document by adding and removing elements to this array like this:

async function addAuthor(courseId , author){

  const course = await Course.findById(courseId);

  course.authors.push(author);

  course.save();

}

async function removeAuthor(courseId , authorId){

  const course = await Course.findById(courseId);

  const author = course.authors.id(authorId);

  author.deleteOne();

  course.save();

}

Transaction and two face commits

In relational database like SQL, we have the concept which says that there are multiple commands that must execute together, in other word, Either all of them should be executed or none of them should be executed. This concept called transaction. In no\_SQL database like mongodb we don’t have such a concept, but mongodb implement this requirement using two face commits. There is a module called fawn in npm which do the transaction need for us. First you have to install and require fawn like this:

npm i fawn

const Fawn = require('fawn');

Fawn.init(mongoose);

Then package all the operation which should be execute in transaction way like this:

new Fawn.Task()

  .save('rentals' , rental)

  .update('movies' , { \_id: movie.\_id } , {

    $inc:{ numberInStock: -1 }

  })

  .run();

ObjectId in mongodb

There is a field in each table in mongodb which plays the role of unique id in relational database like SQL. It is a 24 character (12 byte) string which with a very high rate commits the uniqueness of id.

They consist of 12 bytes:

- 4 bytes: timestamp // show the current date

- 3 bytes: machine identifier

- 2 bytes: process identifier

- 3 byes: counter

This objectId created by mongo driver before the row is inserted in database. To validate the objectId can do it like this:

mongoose.Types.ObjectID.isValid(id);

another way is using Joi-objectId module. Which install and use like this:

npm i joi-objectid

Joi.objectId = require('joi\_objectid')(Joi);

customerId: Joi.objectId().required

Authentication and authorization

Authentication is the process of identifying if the user is who claim they are and authorization determine if the login user have the right permission or not.

For authentication we have to create a model first like this:

const User = mongoose.model('User', new {

    name: {

      type: String,

      required: true,

      minlength: 5,

      maxlength: 50,

      required: true

    },

    email: {

      type: String,

      required: true,

      minlength: 5,

      maxlength: 255,

      required: true,

      unique: true

    },

    password: {

      type: String,

      required: true,

      minlength: 5,

      maxlength: 1024,

      required: true

    }

});

Then as before write CRUD api’s in routes files like this:

router.post('/', async (req, res) => {

  const { error } = validate(req.body);

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

  let user = await User.findOne({ email: req.body.email });

  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

  });

  await user.save();

  res.send(user);

});

Lodash module

There is a problem with above code, that show password in output to users. There is a useful and simple module, called lodash which halp us to handle it and simplify and shortened our code. First you have to install lodash in npm command and import it in your code. Loadash has a useful function called pick which shorten our code. Using from pick and loadash like this:

const \_ = require('lodash');

router.post('/', async (req, res) => {

  const { error } = validate(req.body);

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

  let user = await User.findOne({ email: req.body.email });

  if (user) return res.status(400).send('User already registered.');

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

  await user.save();

  res.send(\_.pick(user , ['name' , 'email' , '\_id']));

});

There is another useful module related to joi module which can control complexity of passwords with that. Use this module like this:

const passwordComplexity = require("joi-password-complexity");

const complexityOptions = {

  min: 5,

  max: 30,

  lowerCase: 1,

  upperCase: 1,

  numeric: 1,

  symbol: 1,

  requirementCount: 2,

};

if(passwordComplexity(complexityOptions).validate(req.body.password).error.details.length){

return res.status(400).send(passwordComplexity(complexityOptions)

.validate(req.body.password).error.details[0].message);

}

bcrypt module

saving password in plain shape is not a secure method. The better way is to hash and encode them and then save them to database. Use like this:

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

const salt = await bcrypt.genSalt(10);

user.password = await bcrypt.hash(user.password, salt);

await user.save();

res.send(\_.pick(user , ['name' , 'email' , '\_id']));

There is a command in bcrypt module which uses for authentication by comparing plain password with hashed password. That works like this:

const validPassword = await bcrypt.compare(req.body.password , user.password);

  if(!validPassword) return res.status(400).send('Invalid email or password');

So, now we have to create a separate route file for authentication the login user. For that, first create a file with auth.js file in route folder. Then write this code for checking user like this:

router.post('/', async (req, res) => {

  const { error } = validate(req.body);

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

  let user = await 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);

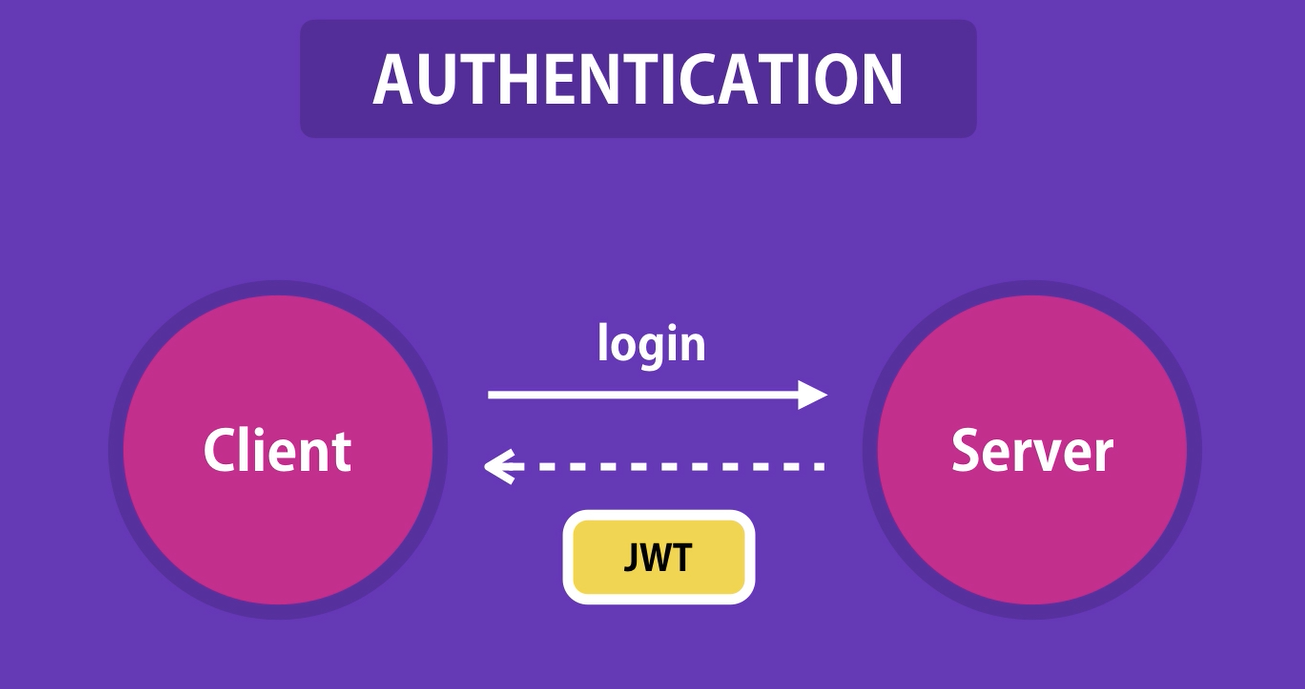
  if(!validPassword) return res.status(400).send('Invalid email or password');

  res.send(true);

});

Bcrypt.compare use for comparing a plain password with hashed password and return Boolean value.

So, now we have to modify the output of auth route. We use json web token which is the long string like driver license which uses to compare and identify user.



This token saves in the local storage in web application. To see the structure and build online json web token see the jwt.io website. Each jwt has three part:

* Header: include algorithm and type
* Payload: include main info
* Verify signature: uses another value as a private key to produce a token

To use jwt, first install it: npm I jsonwebtoken, then import it in jwt module. And go to auth route handler and use it like this:

const token = jwt.sign({ \_id: this.\_id }, 'jwtPrivateKey');

Actually, with sign command, we create a token which give two parameter, first as a payload and second as a verify signature.

So, the next step is to set verify signature uses environment variables.

To do that, first we have to create a config folder and create two files called custom-environment-variables.js and default.js. in default.js we define the pattern of key like this:

{

  "jwtPrivateKey": ""

}

And in custom-environment-variables.js tell the program that value the variable with which environment variable like this:

{

  "jwtPrivateKey": "vidly\_jwtPrivateKey"

}

Then, change the code of auth.js to get the value of jwtPrivateKey using config.get like this:

const token = jwt.sign({ \_id: this.\_id }, config.get('jwtPrivateKey'));

  return token;

And then we have to set vidly\_jwtPrivateKey in cmd like this:

Set vidly\_jwtPrivateKey=mySecureKey.

And finally we have to check the existence of this variable in the index.js like this:

if (!config.get('jwtPrivateKey')) {

  console.error('FATAL ERROR: jwtPrivateKey is not defined.');

  process.exit(1);

}

Another step is to prepare a suitable return type with a correct header like this:

 res

      .header("x-auth-token", token)

      .send(\_.pick(user, ["\_id", "name", "email"]));

As you say to create token we use only \_id parameter and in future, it may be change. So, the better way is to encapsulate this logic in user model. To do this, its better to add it to user model like this:

userSchema.methods.generateAuthToken = function() {

  const token = jwt.sign({ \_id: this.\_id }, config.get('jwtPrivateKey'));

  return token;

}

And change the user route like this:

 const token = user.generateAuthToken();

In a real project, we have several functions which manipulate data and all of them need authorization. To avoid repetition, implement this control in a middleware function and use it several times. Also reusing is simpler in this case.

module.exports = function(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(400).send('Invalid token ...')

    }

}

This middleware like another middlewares have three parameters which one of them is next and means that, when the mission of middleware complete, the execution flow to next operation. So, to apply this middleware selectively in specific functions, act like this:

const auth = require('../middleware/auth');

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

  const { error } = validate(req.body);

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

  let genre = new Genre({ name: req.body.name });

  genre = await genre.save();

  res.send(genre);

});

As I told before, middleware place in the middle of a function and control the execution flow. In above code auth is placed in the route of post method.

Role Based authorization

Assume that there is an operation that only specific roles can do it. For example, just admin can delete a data. In such a situation, you need another middleware to handle that. First you have to create a new middleware like this:

module.exports = function(req, res, next){

    if(!req.user.isAdmin) return res.status(403).send('Access Denied');

    next();

}

Then you have to add isAdmin field to the model and everywhere which you validate fields like this:

const userSchema = new mongoose.Schema({

  name: {

    type: String,

    required: true,

    minlength: 5,

    maxlength: 50

  },

  email: {

    type: String,

    required: true,

    minlength: 5,

    maxlength: 255,

    unique: true

  },

  password: {

    type: String,

    required: true,

    minlength: 5,

    maxlength: 1024

  },

  isAdmin: Boolean

});

function validateUser(user) {

    const schema = Joi.object({

        name: Joi.string()

            .min(5)

            .max(50)

            .required(),

        email: Joi.string()

            .min(5)

            .max(50)

            .required()

            .email(),

        password: Joi.string()

        .min(5)

        .max(50)

        .required(),

        isAdmin: Joi.boolean()

     });

  return schema.validate(user);

}

And absolutely place it in token payload like this:

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

    const salt = await bcrypt.genSalt(10);

    user.password = await bcrypt.hash(user.password, salt);

    await user.save();

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

    const token = user.generateAuthToken();

    res

      .header("x-auth-token", token)

      .send(\_.pick(user, ["\_id", "name", "email", "isAdmin"]));

  }

Then use it in noticed function beside another middlewares, like this:

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

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

  if (!genre) return res.status(404).send('The genre with the given ID was not found.');

  res.send(genre);

});

In above code, first auth middleware get the control and check the authorization and then control flow into the admin middleware and check the role permission control. In situations like this, which we have more than one middleware, we place middlewares in an array in the route of execution like above ([auth, admin]).

Important Tip: 401 status uses for un-authorization token, in situations like tampered tokens.

403 status uses in forbidden situations which current user doesn’t have permission to do a specific operation.

Handling and logging

In real applications everything is not execute properly and a lot of problems may be accured. And it is necessary to handle this errors and show the meaningful errors to clients and log them in database.

One of the errors is the disconnection mongodb services. In such a situation you have to handle this error like this:

router.get('/', async (req, res) => {

  try{

    const genres = await Genre.find().sort('name');

    res.send(genres);

  }

  catch(ex){

    res.status(500).send('Something failed. ');

  }

});

It worth noting that “500” status code is for server errors.

This problem is repetitive and may be occurred in several methods. To avoid repetition its better to convert it to a separate middleware and use it several times. To do that first create a file with error.js in middleware folder and write in it these codes:

module.exports = function(err , req , res , next){

    res.status(500).send('Something failed. ');

}

And add these codes to index.js:

const error = require('../middleware/error');

app.use(error);

and change the genre.js like this:

router.get('/', async (req, res, next) => {

  try{

    const genres = await Genre.find().sort('name');

    res.send(genres);

  }

  catch(ex){

    next(ex);

  }

});

This implementation is correct but not efficient enough, because this try- catch code repeat in each router and moreover the base logic that do main operation lost among many other codes. To prevent this, first create a new middleware called async.js and write these codes:

module.exports = function (handler){

    return async(req, res, next) => {

      try{

        await handler(req, res)

      }

      catch(ex){

        next(ex);

      }

    }

  }

Which get the main function as a parameter and put it among try- catch block. And then change the route handler like this:

const asyncMiddleware = require('../middleware/async');

router.get('/', asyncMiddleware (async (req, res, next) => {

    const genres = await Genre.find().sort('name');

    res.send(genres);

  }));

This approach is good, but a better solution is using async-express-errors module, which install with this command: npm i async-express-errors

And all you need is to import it in index.js like this: require(‘async-express-errors’).

And remove asyncmiddleware from code like this:

router.get('/', async (req, res, next) => {

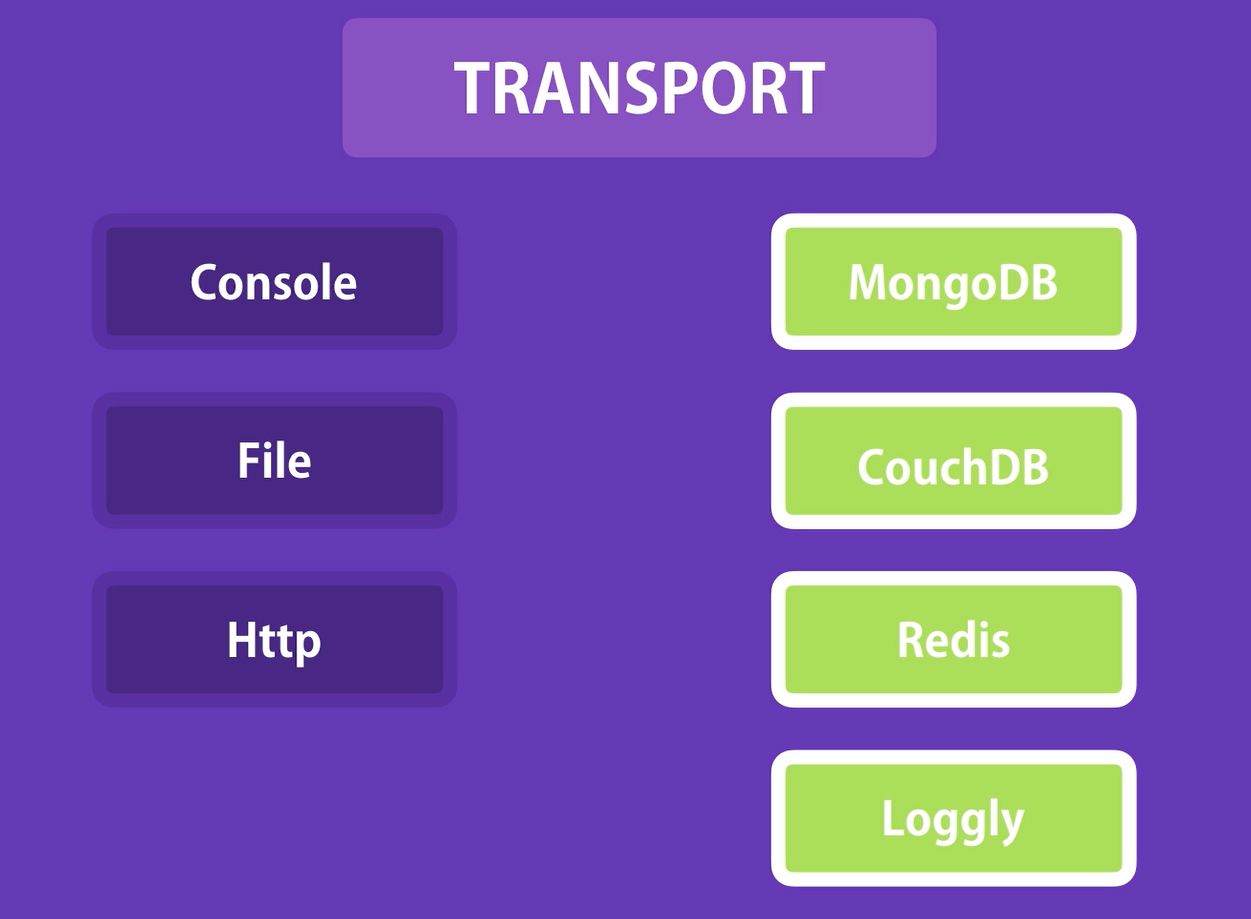
    const genres = await Genre.find().sort('name');

    res.send(genres);

  });

Logging with winston

There are several ways to store logs and exceptions in projects, and some projects has their own logging method. But for small and medium scale projects, its better to use standards modules to handle that. One of the standard modules is Winston module, which have to install it first.



As above picture Winston has such a structure. Console use to show messages, file use to store errors and http use to route to specific url if needed.

To use Winston, first you have to import it in index.js and define your logger like this:

nst logger = winston.createLogger({

  level: 'error',

  format: winston.format.json(),

  defaultMeta: { service: 'user-service' },

  transports: [

    new winston.transports.File({ filename: 'logfile.log', level: 'error' })

  ]

});

logger.add(new winston.transports.Console({

  format: winston.format.simple(),

}));

Then add this code to error middleware:

const winston = require('winston');

module.exports = function(err , req , res , next){

    winston.error(err.message, err);

    res.status(500).send('Something failed. ');

}

And that it. If error is occurred, its logged and saved in logfile.log.

If you want to store logs in mongodb, use Winston-mongodb module. First you have to install it with npm i winston-mongodb.

Then add this code to index.js instead of other code used to store log in file:

winston.add(new winston.transports.MongoDB({

  db: 'mongodb://localhost/vidly',

  level: 'error'

}));

There are some situations in which, the error is occurred out of the scope of routers and express. In previous method, these errors are not logged and just showed in server console and if the project is on the production mode, we don’t have any log. To solve this problem use process event Emitter and add these codes to index.js:

process.on('uncaughtException' , (ex) => {

  console.log('WE GOT An error');

  winston.error(ex.message, ex);

});

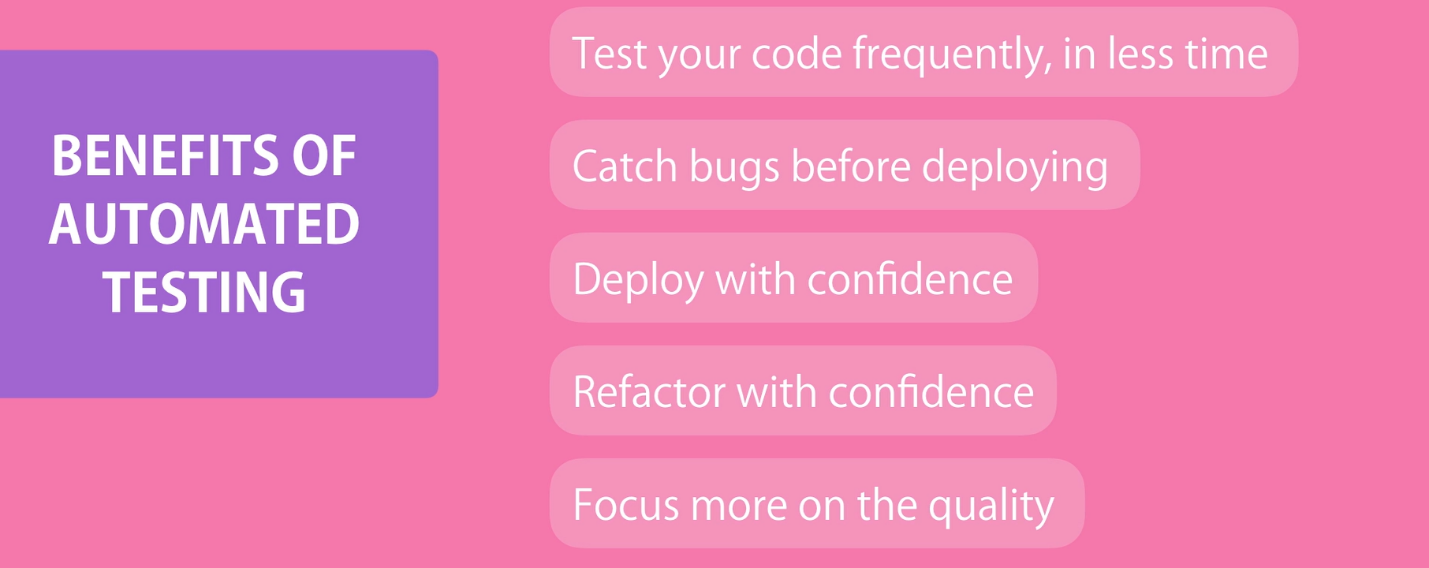
This solution is good, but it works just for synchronous codes and it doesn’t work for async codes like promise based codes. To handle async code errors like unhandled promise rejection, we have to use another event emitter of process, like this:

 process.on('unhandledRejection', (ex) => {

    throw ex;

  });

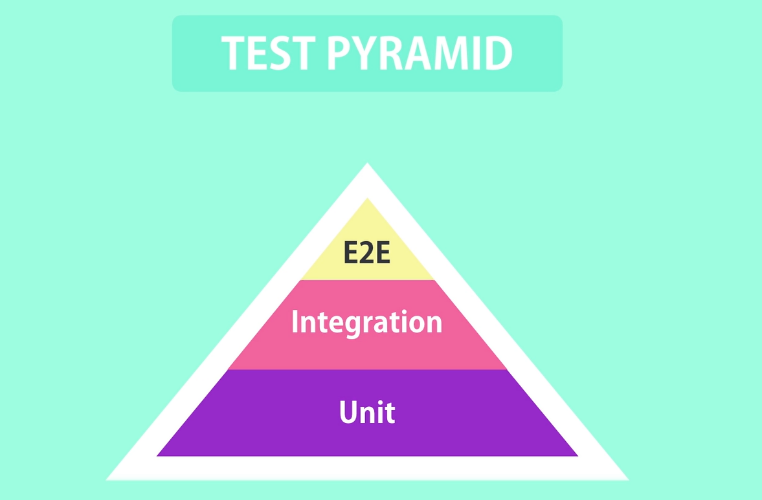
Unit Test



There are three types of testing:

* Unit test: which test class or classes, regardless of its external dependencies like db, …. These tests are cheap and fast, but its not reliable and don’t give enough confidence about program health.
* Integration Test: which test program with its external dependencies. These tests are slow to execute and expensive, but they are more reliable and give more confidence.
* End-To-End Test: Drives an application through its UI. These tests give you the greatest confidence, but they are very slow and very brittle and fragile against any changing.

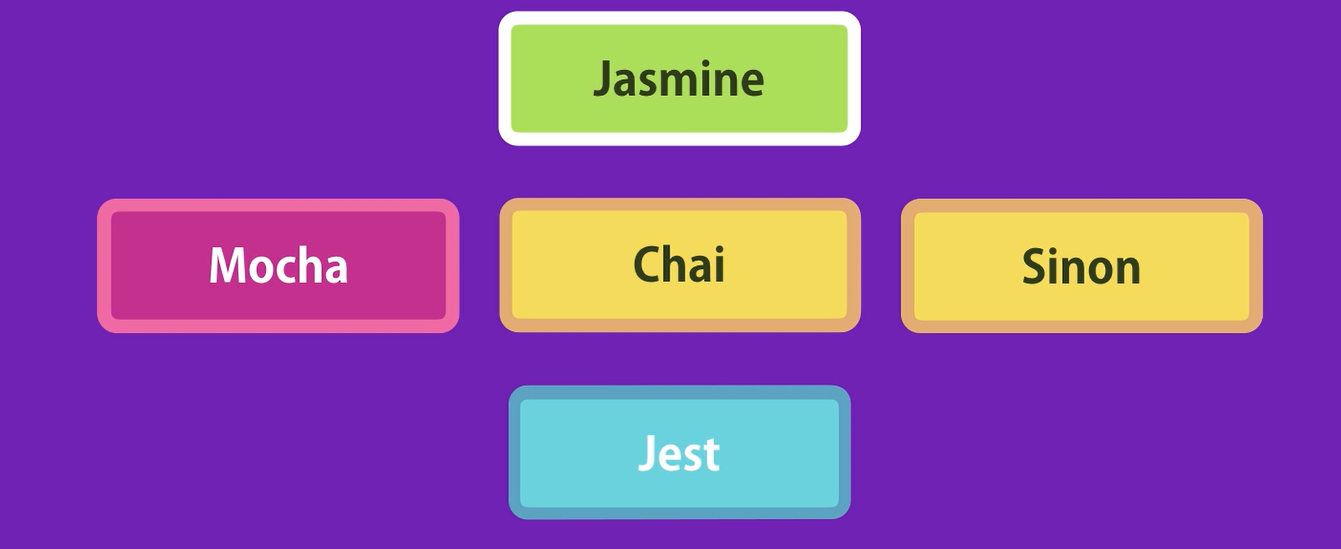
Now, the question is: which of these test type should be use in your application? And the answer is, all of them with following pyramid:



And follow these instructions:



There are several tools to implement your test scenarios. The most popular tools are:



That Jest is the newest of them which doesn’t need any extra framework.

Write our First Unit Test

To write and run test, should follow these steps:

* Install jest module: npm i jest –save-dev
* Create a folder with “test” name
* Create a test file in this folder for each js file that we want to write test for it. With its name like this: lib.test.js
* Change the package.json file test script: “test” : “jest”
* Write your test in lib.test.js like this:

test('Our First Test', () => {

});

* Run program with npm test command

Numeral Test

And now let’s see a real example which test an absolute function, which return a positive version of each number. As you know this function has three ways to execute, positive number, negative number and zero. So we have to write at least three test for it:

const lib = require('../lib');

    test('absolute - should return positive value if input is positive', () => {

        const result = lib.absolute(1);

        expect(result).toBe(1);

    });

    test('absolute - should return positive value if input is negative', () => {

        const result = lib.absolute(-1);

        expect(result).toBe(1);

    });

    test('absolute - should return 0 if input is 0', () => {

        const result = lib.absolute(0);

        expect(result).toBe(0);

    });

One necessary advice that you have to notice is that test scripts are first citizens code and important as the main source code and as you worry about the structure of your source code, you have to care about your test scripts too. So for example to make your test code prettier, you can to group them with describe function like this and eliminate extra and lone names:

const lib = require('../lib');

    describe('absolute' , () => {

        it('should return positive value if input is positive', () => {

            const result = lib.absolute(1);

            expect(result).toBe(1);

        });

        it('should return positive value if input is negative', () => {

            const result = lib.absolute(-1);

            expect(result).toBe(1);

        });

        it('should return 0 if input is 0', () => {

            const result = lib.absolute(0);

            expect(result).toBe(0);

        });

    });

String Test

And now let’s see a string test example:

describe('greet' , () => {

        it('should return a message', () => {

            const result = lib.greet('Mosh');

            //expect(result).toMatch(/Mosh/);

            expect(result).toContain('Mosh');

        });

    });

Important Tip: Don’t use neither so specific nor general. Because if your test be so specific it may return wrong result with a minimum change in base function and on the other hand, if your test be very general it may be so optimism about errors and may don’t catch them.

Array Test

And now let’s see an Array test example:

 describe('getCurrencies' , () => {

   it('should return the list of currencies', () => {

     const result = lib.getCurrencies();

     expect(result).toEqual(expect.arrayContaining(['EUR' , 'AUD' , 'USD']));

        });

    });

Object Test

And now let’s see an Object test example:

    describe('getProduct' , () => {

        it('should return the object', () => {

            const result = lib.getProduct(1);

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

            expect(result).toHaveProperty('id', 1);

            expect(result).toHaveProperty('price', 10);

        });

Exception Test

Imaging the function which if the username would be valid should return that and otherwise if it has falsy values, it should throw an error like this:

module.exports.registerUser = function(username) {

  if (!username) throw new Error('Username is required.');

  return { id: new Date().getTime(), username: username }

}

So there are two execution path, So at least two unit test is required. Like this:

describe('registerUser' , () => {

        it('should throw an error if username is falsy', () => {

            const args = [null , undefined , '' , 0 , NaN , false];

            args.forEach((a) => {

                expect(() => {

                    lib.registerUser(a)

                }).toThrow();

            });

        });

        it('should return the object if user is valid', () => {

            const result = lib.registerUser('mosh');

            expect(result).toMatchObject({ username: 'mosh'});

            expect(result.id).toBeGreaterThan(0);

        });

    });

Continuously running test

In real applications with more than 100 functions its so tedious that whenever the changes in functions or tests occurred, we forced to run npm test command. To avoid this tedious work, change the package.json file like this:

  "scripts": {

    "test": "jest --watchAll"

  }

So, like a nodemon this command watch any change in our program and if any change is occurs, it runs and monitor it.

Mock Function

When you want to unit test a module which has an external dependency, you can not do it. Because unit test is not involved in external dependencies and it may return false response wrongly. To avoid this problem, we have to simulate functions with external dependencies, like db or network functions, with mock functions which simulate that return value. For example imagine that you have a function which connect to database and retrieve data from database like this:

Const customer =db.getCustomerSync(order.customerId);

To write a unit test you have to remove any real connect to external dependencies, So you can rewrite this function like this:

module.exports.getCustomerSync = function(id){

Return { id: id, points: 11};

}

So, with this technique every connection to external dependencies are removed and can test another parts of program.

Let you know, that there is a tool in jest module to write mock function like this:

db.getCustomerSync = jest.fn().mockReturnValue({ email: 'a' });

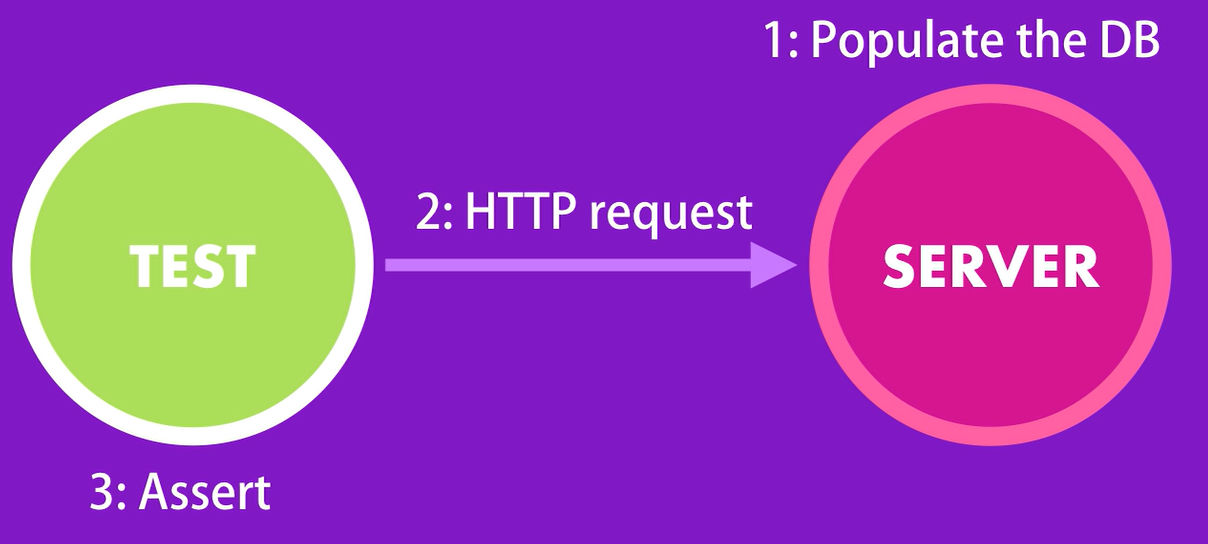
mail.send = jest.fn();

lib.notifyCustomer({ customerId: 1 });

expect(mail.send).toHaveBeenCalled();

Integration Test

In integration test, we have to test and simulate external sources like this:



Before start to implement integration test, you have to make these changes in program:

* Remove require (‘winston-mongodb’) from logging file
* Add this command to package.json: “test”: “jest –watchAll –verbose”

The Next step is to config test environment and test database to use in test operation. Because we have to avoid filling real database with test values. To do that we have to do these steps:

* Create a new file in config folder with test.json name and write this code on it:

{

    "jwtPrivateKey": "1234",

    "db": "mongodb://localhost/vidly\_tests"

  }

* Change the default.json file in config folder like this:

{

  "jwtPrivateKey": "",

  "db": "mongodb://localhost/vidly"

}

* Change the db.js file in startup folder like this:

const winston = require('winston');

const mongoose = require('mongoose');

const config = require('config');

module.exports = function() {

  const db = config.get('db');

  mongoose.connect(db)

    .then(() => console.log(`Connected to ${db}...`));

}

* Write this command in cmd: test NODE\_ENV=test
* And run the project, that’s it.

To use integration test in express, you have to install supertest module to simulate running manually the project with postman.

Next step is to export server which created in index.js to supertest like this:

const port = process.env.PORT || 3000;

const server = app.listen(port, () => console.log(`Listening on port ${port}...`));

module.exports = server;

After that, create a folder with integration in test folder, and create a file with genres.test.js in it. And write these codes on it:

const request = require('supertest');

const {Genre} = require('../../models/genre');

let server;

describe('/api/genres' , () => {

    beforeEach(() => {

        server = require('../../index');

    });

    afterEach(async () => {

        server.close();

        await Genre.deleteMany({});

    });

    describe(' all GET/' , () => {

        it('should return all genres' , async () => {

            await Genre.collection.insertMany([

                {name: "genre1"},

                {name: "genre2"}

            ]);

           const res = await request(server).get('/api/genres');

           expect(res.status).toBe(200);

           expect(res.body.length).toBe(2);

           expect(res.body.some(g => g.name === 'genre1')).toBeTruthy();

           expect(res.body.some(g => g.name === 'genre2')).toBeTruthy();

        });

    });

});

So, with this test two record insert in database and then check if these records exist in database or not.

Another test for get specific object like this:

describe('GET /:id' , () => {

        it('should return specific genres' , async () => {

            const genre = new Genre({ name: 'Genre1'});

            await genre.save();

           const res = await request(server).get('/api/genres/' + genre.\_id);

           expect(res.status).toBe(200);

           expect(res.body).toHaveProperty('name' , genre.name );

        });

    });

Another test for get specific row is to check if the id which send to get url is valid or not. To check that, write this code:

  describe('GET /:id' , () => {

        it('should return 404 if invalid id is passed' , async () => {

            const genre = new Genre({ name: 'Genre1'});

            await genre.save();

            const res = await request(server).get('/api/genres/1');

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

        });

    });

But before that, we have to change genre route to send 404 when the id is not as a correct format like this:

router.get('/:id', async (req, res) => {

  if(!mongoose.Types.ObjectId.isValid(req.params.id))

  return res.status(404).send('Invalid ID..');

});

Now its post request’s turn to test. So first you have to test if the user logged in or not. And if its not logged in, return 401 error. Like this:

describe('POST /' , () => {

        it('should return 404 if user is not logged in' , async () => {

            const res = await request(server)

                .post('/api/genres/')

                .send({ name: 'genre1'});

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

        });

    });