****

**Node Js**

**Submitted By:**

**Shaikh Sameer**

**Introduction**

* **Event Driven**
* **Non Blocking I/O Model**
* **Light weight and Efficient**
* **Package System – npm**
* **Open Source and Completely free**

**Uses**

* **I/O Application**
* **Data Streaming Application**
* **Data Intensive Application**
* **JSON APIs based Application**
* **Single Page Application**

**Mongoose**

## Importing

*// Using Node.js `require()`*

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

*// Using ES6 imports*

**import** mongoose **from** 'mongoose';

## Installation

First install [node.js](http://nodejs.org/) and [mongodb](https://www.mongodb.org/downloads). Then:

$ npm install mongoose

### Connecting to MongoDB

First, we need to define a connection. If your app uses only one database, you should use mongoose.connect. If you need to create additional connections, use mongoose.createConnection.

Both connect and createConnection take a mongodb:// URI, or the parameters host, database, port, options.

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

 mongoose.connect('mongodb://localhost/my\_database');

Once connected, the open event is fired on the Connection instance. If you're using mongoose.connect, the Connection is mongoose.connection. Otherwise, mongoose.createConnection return value is a Connection.

**Note:** If the local connection fails then try using 127.0.0.1 instead of localhost. Sometimes issues may arise when the local hostname has been changed.

**Important!** Mongoose buffers all the commands until it's connected to the database. This means that you don't have to wait until it connects to MongoDB in order to define models, run queries, etc.

### Defining a Model

Models are defined through the Schema interface.

const Schema **=** mongoose.Schema;

const ObjectId **=** Schema.ObjectId;

 const BlogPost **=** **new** Schema({

  author**:** ObjectId,

  title**:** String,

  body**:** String,

  date**:** Date

});

### Accessing a Model

Once we define a model through mongoose.model('ModelName', mySchema), we can access it through the same function

const myModel **=** mongoose.model('ModelName');

Or just do it all at once

const MyModel **=** mongoose.model('ModelName', mySchema);

The first argument is the singular name of the collection your model is for. **Mongoose automatically looks for the plural version of your model name.** For example, if you use

const MyModel **=** mongoose.model('Ticket', mySchema);

Then Mongoose will create the model for your **tickets** collection, not your **ticket** collection.

Once we have our model, we can then instantiate it, and save it:

const instance **=** **new** MyModel();

instance.my.key **=** 'hello';

instance.save(function (err) {

*//*

});

You can also findOne, findById, update, etc. For more details check out [the docs](http://mongoosejs.com/docs/queries.html).

**Important!** If you opened a separate connection using mongoose.createConnection() but attempt to access the model through mongoose.model('ModelName') it will not work as expected since it is not hooked up to an active db connection. In this case access your model through the connection you created:

const conn **=** mongoose.createConnection('your connection string');

const MyModel **=** conn.model('ModelName', schema);

const m **=** **new** MyModel;

m.save(); *// works*

vs

const conn **=** mongoose.createConnection('your connection string');

const MyModel **=** mongoose.model('ModelName', schema);

const m **=** **new** MyModel;

m.save(); *// does not work b/c the default connection object was never connected*

### Embedded Documents

In the first example snippet, we defined a key in the Schema that looks like:

comments: [Comment]

Where Comment is a Schema we created. This means that creating embedded documents is as simple as:

*// retrieve my model*

var BlogPost **=** mongoose.model('BlogPost');

*// create a blog post*

var post **=** **new** BlogPost();

*// create a comment*

post.comments.push({ title**:** 'My comment' });

post.save(function (err) {

**if** (**!**err) console.log('Success!');

});

The same goes for removing them:

BlogPost.findById(myId, function (err, post) {

**if** (**!**err) {

    post.comments[0].remove();

    post.save(function (err) {

*// do something*

    });

  }

});

#### Intercepting and mutating method arguments

You can intercept method arguments via middleware.

For example, this would allow you to broadcast changes about your Documents every time someone sets a path in your Document to a new value:

schema.pre('set', function (next, path, val, typel) {

*// `this` is the current Document*

  this.emit('set', path, val);

*// Pass control to the next pre*

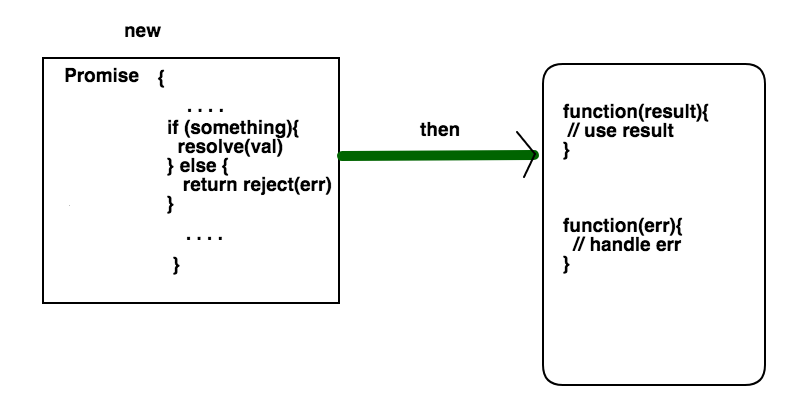
  next();

});

### What is a Promise?

A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers to an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of the final value, the asynchronous method returns a promise for the value at some point in the future.

We can create a promise in JavaScript and use it as an upcoming fact to describe few actions. Promises are kind of design patterns to remove the usage of unintuitive callbacks.



Promise creation and usage

As the picture depicts, these are the steps for creating and using promises

### Creating a Promise

We can create a promise in our Node JS program using the **new** constructor. For all the examples I use **Node v6.5.0**. You should install Node JS on your machine before beginning with this tutorial. Even though promises can be used in browsers, this article mainly focuses on writing asynchronous code on Node.

var myPromise = new Promise(function(resolve, reject){  
 ....  
})

Now let us create a **main**function where we get the Promise for above function and attach a function callback in the **then**function.

|  |
| --- |
| var request = require("request"); |
|  | var userDetails; |
|  |  |
|  | function initialize() { |
|  | // Setting URL and headers for request |
|  | var options = { |
|  | url: 'https://api.github.com/users/narenaryan', |
|  | headers: { |
|  | 'User-Agent': 'request' |
|  | } |
|  | }; |
|  | // Return new promise |
|  | return new Promise(function(resolve, reject) { |
|  | // Do async job |
|  | request.get(options, function(err, resp, body) { |
|  | if (err) { |
|  | reject(err); |
|  | } else { |
|  | resolve(JSON.parse(body)); |
|  | } |
|  | }) |
|  | }) |
|  |  |
|  | } |
|  |  |
|  | function main() { |
|  | var initializePromise = initialize(); |
|  | initializePromise.then(function(result) { |
|  | userDetails = result; |
|  | console.log("Initialized user details"); |
|  | // Use user details from here |
|  | console.log(userDetails) |
|  | }, function(err) { |
|  | console.log(err); |
|  | }) |
|  | } |
|  |  |
|  | main(); |

Output looks like this.

Initialized user details

{  
 "login": "narenaryan",  
 "id": 5425726,  
 "avatar\_url": "<https://avatars3.githubusercontent.com/u/5425726?v=3>",  
 "gravatar\_id": "",  
 "url": "<https://api.github.com/users/narenaryan>",  
 "html\_url": "<https://github.com/narenaryan>",  
 "followers\_url": "<https://api.github.com/users/narenaryan/followers>",  
 "following\_url": "[https://api.github.com/users/narenaryan/following{/other\_user](https://api.github.com/users/narenaryan/following%7B/other_user)}",  
 "gists\_url": "[https://api.github.com/users/narenaryan/gists{/gist\_id](https://api.github.com/users/narenaryan/gists%7B/gist_id)}",  
 "starred\_url": "[https://api.github.com/users/narenaryan/starred{/owner}{/repo](https://api.github.com/users/narenaryan/starred%7B/owner%7D%7B/repo)}",  
 "subscriptions\_url": "<https://api.github.com/users/narenaryan/subscriptions>",  
 "organizations\_url": "<https://api.github.com/users/narenaryan/orgs>",  
 "repos\_url": "<https://api.github.com/users/narenaryan/repos>",  
 "events\_url": "[https://api.github.com/users/narenaryan/events{/privacy](https://api.github.com/users/narenaryan/events%7B/privacy)}",  
 "received\_events\_url": "<https://api.github.com/users/narenaryan/received_events>",  
 "type": "User",  
 "site\_admin": false,  
 "name": "Naren Arya",  
 "company": "Citrix R&D India",  
 "blog": "[http://narenarya.in](http://narenarya.in/)",  
 "location": "Banglaore",  
 "email": null,  
 "hireable": true,  
 "bio": "A Software Development Engineer with expertise in Python and JavaScript. Coding in Golang and Reading books are his hobbies .",  
 "public\_repos": 69,  
 "public\_gists": 41,  
 "followers": 134,  
 "following": 7,  
 "created\_at": "2013-09-10T09:01:57Z",  
 "updated\_at": "2017-04-24T04:39:04Z"  
}

## What is Callback?

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

## Blocking Code Example

Create a text file named **input.txt** with the following content −

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Create a js file named **main.js** with the following code −

var fs = require("fs");

var data = fs.readFileSync('input.txt');

console.log(data.toString());

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Program Ended

## Non-Blocking Code Example

Create a text file named input.txt with the following content.

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Update main.js to have the following code −

var fs = require("fs");

fs.readFile('input.txt', function (err, data) {

if (err) return console.error(err);

console.log(data.toString());

});

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

**Sample Application Path**

https://github.com/sameers25/Node-Js/tree/master/Nodejs-Express-MongoDb-API

**API Path**

1. http://localhost:8080/home/addStudent

body : {

"title" : "Tester","description" : "Web Mobile"

}

1. http://localhost:8080/home/getStudentListbyId

body :{

“UserId”:” 5bfcf99f48cf2b2e604699f5”,"title" : "Tester","description" : "Web Mobile"

}

1. http://localhost:8080/home/getStudentList

body : {

}

1. http://localhost:8080/home/updateStudentListById

body : {

“UserId”:” 5bfcf99f48cf2b2e604699f5”

}

1. http://localhost:8080/home/deleteStudentListByIdt

body : {

“UserId”:” 5bfcf99f48cf2b2e604699f5”

}