**Introduction to MongoDB and Node.js**

Throughout this course, you'll be building a small shopping cart application using the MEAN

Stack. Users of this application will be able to browse through products, add products to their cart, and even check out using fake payment.

Throughout this course, you'll build a rest API, a desktop web client using AngularJS. The app will look something like this. As a new user, you'll be able to log in with Facebook. Then you'll be able to search for an Asus laptop.

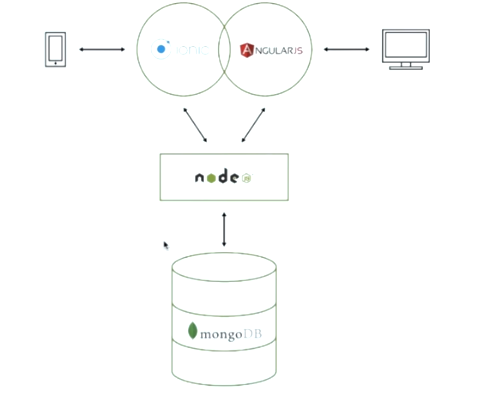
You'll be able to add the laptop to your cart, and then check out with a fake credit card. Once you're done with this course, you should have a high level understanding of how all the components of the MEAN Stack fit together and how you can leverage them to quickly build out web apps.

Ready?

Let's get started.

**Recipe1: Starting a standalone mongod and connecting with shell**

At a high level, the architecture for this application looks like this graphic.



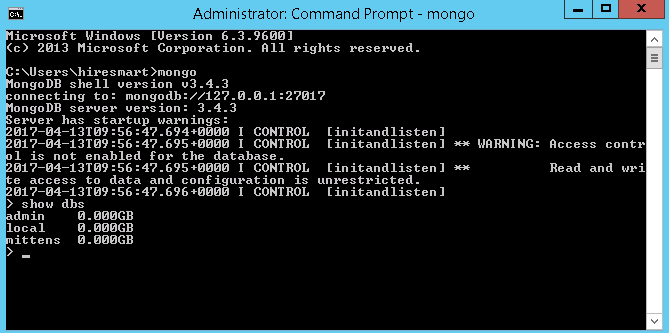
Here Node.js + MongoDB is your server. The server is responsible for storing user data. The server will be written in Node.js. And your Node.js code will store data like what products are for sale and what products are in a user's cart in MongoDB.

Since the MongoDB is at the base of the MEAN stack, you'll start by working with MongoDB and work your way up the stack. Now, MongoDB is a powerful database that enables you to store documents which are roughly equivalent to a JavaScript object or a Java LinkedHashMap rather than rows and columns like a traditional SQL database.

For the purposes of this course, you'll use MongoDB to store and query for JavaScript Object Notation, or JSON for short.

To open MongoDB database, Open Command prompt and type mongod.

Open another Command prompt and type mongo after that type show dbs.



Here mongod, which is the MongoDB core server, and mongo, which is the MongoDB shell that's useful for talking to the mongod server.

Create Database: To create database type below command in command line: use databaseName

e.g. use test

For instance, you can insert a single document. So this document has a key, hello, and a value, world. You can insert it, and then you can query it.

C:\Users\462439\Desktop\2.PNG

Now, queries in the MongoDB shell are expressed as JSON objects. So for instance, when you specify key, hello, and value, world, MongoDB will return all documents that have the key hello equal to world.



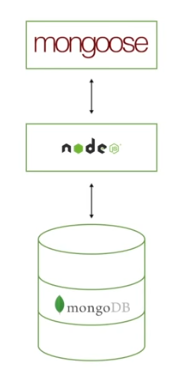
So in this example, the db.test.insert function tells MongoDB to insert the documents into the collection named test. And then db.test.findOne tells MongoDB to search for documents in the collection named test.

**Recipe2: Connecting to mongodb with node.js**

Now that you've got a MongoDB instance running and node.js set up, it's time to connect to MongoDB using node.js, so you could start building out the server part of your mean stack application.

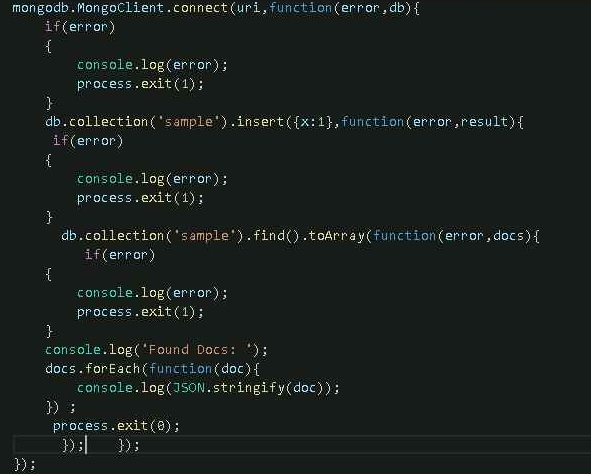
The MongoDB NPM package is the officially supported MongoDB node.js driver. The driver lets you write node.js code to talk to MongoDB.

In this course you'll primarily use Mongoose to interface with MongoDB. Mongoose is an object document mapper, or ODM for short, which provides functionality like schema validation on top of the MongoDB node.js driver.



Open index.js file and type code as below.

C:\Users\462439\Desktop\6.PNG



This MongoClient helper is what you will use to create a connection to MongoDB. This MongoClient.connect function takes a MongoDB connection string, which is a URI that tells the driver which MongoD to connect to. And in this case the URI is MongoDB://localhost port 27017, and the example database.

The MongoClient.connect function also takes a callback function. The callback function gets called when either an error occurred, or the driver successfully connected. The callback function takes two parameters. A potentially null error and a potentially null db handle. In this simple example, if an error occurs, we just log it and exit.

On your terminal, run nodemon index.js ,once you run this program, you should see that MongoDB successfully inserted a document, and then query to back.

C:\Users\462439\Desktop\5.PNG

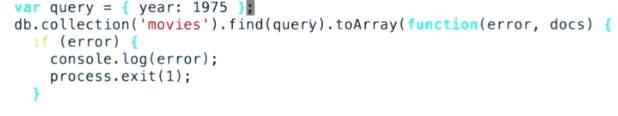
**Recipe3: Inserts and queries using the node.js driver**

Here you can also insert JSON object into a mongodb collection. For example



So this code right here inserts a document representing the movie entitled Jaws into the collection named movies.

For instance, find() query will return all documents in the collection. However, if I were to create a JSON document with a single key year equal to the number 1975 and pass that to the find function, MongoDB will return all documents in the collection named movies that have the key year equal to the number 1975.



Now, MongoDB also lets you store and query for nested JSON documents, arrays, nested documents within arrays, and other really complex hierarchical objects. For example, rather than a flat document, imagine that the data model is also used for ratings and screenplay authors. Note that this document contains a nested document for ratings and then array field for the screenplay authors. Queries on such fields are fully supported in MongoDB.



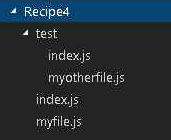
Find query on this, you like all documents where the audience rating was 90% or greater.



**Recipe4: require() basics**

The require function is Node.js's mechanism for breaking up large projects into small, manageable files. The require function lets you include functions from external modules and other files in a clean and elegant way.

You already used the require function once before in order to include the MongoDB Node.js driver. In this recipe you'll learn how to integrate your own code with the require function.

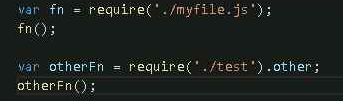


In this example you have two files called index.js and myfile.js, and the directory called Test. The index.js file will be the main entry point for your program-- that is, you'll run node index.js.

The test directory contains two more files, index.js and myotherfile.js. The top level index.js file will access code from all of these files.

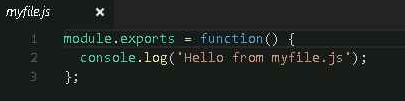
**Let's take a look at the top level index.js file**

**#index.js**



In this index.js file, you call require on myfile.js and the test directory.

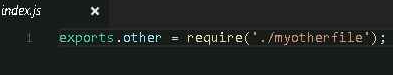
**First let's take a look at what the myfile.js file looks like.**



In this case, calling require on myfile.js gives you a function that prints this hello from myfile.js message.

Now let's consider what happens when you call require of test. Recall that test is a directory, so when you call require on a directory, Node.js looks for an index.js file in that directory and uses that file. In other words, require of ./test is equivalent to require on ./text/index.js.

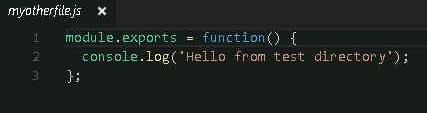
**Now let's take a look at test/index.js.**



Now, this file introduces two subtleties about require. First, notice that this file uses exports.other instead of module.exports. The exports variable is a convenient shorthand for module.exports. This file would do the same thing if you used module.exports.other, rather than exports.other. The only difference is that you can't directly assign to the exports variable.

The other subtlety is that this file calls require on myotherfile.js without specifying the test directory. This is because require resolves file names relative to the current files directory. So in other words, the current working directory when you call require on myotherfile within test/index.js is the test directory itself.

**Finally, let's take a look at the myotherfile.js file that's in the test directory.**



This file uses the same module.exports pattern that you saw in myfile.js, however it prints a slightly different message through the screen.

Now that you've taken a look through all the files, let's run node index.js.

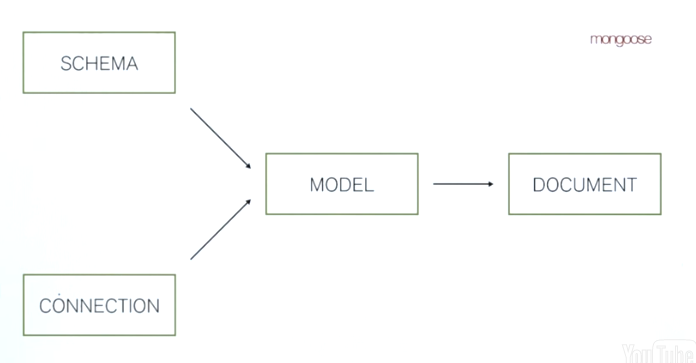


So as you can see, this file prints the message from myfile.js, followed by the message from otherfile.js from the test directory.

**Recipe5: Introduction to mongoose**

In this recipe you will learn about Mongoose, which is the most popular object document mapper, or ODM for short for MongoDB and node.js. Mongoose provides features like schema validation, pseudo joins, and numerous other features on top of the MongoDB node.js driver.

Mongoose's API consists of four primary data types.



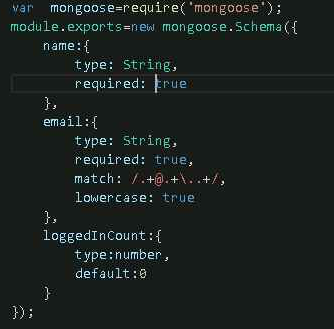
A schema is a set of rules that defines what fields a document may have and what properties the document must satisfy to be considered valid.

A connection is an object representing one or more sockets that are connected to the MongoDB server.

At a high level, a model is a combination of a schema and a connection. A model is a wrapper around a single MongoDB collection and uses its associated schema to make sure documents inserted into the collection satisfy the schema's constraints.

A document is a single object from a collection. It has an associated model and a dot save function that you can use to persist that document to MongoDB.

Suppose you wanted your user documents to have three fields. A name, an email address, and the number of times the user has logged in. Create schema.js file as below.



This is how you would express these requirements in Mongoose. The username is a string and is required, email is of type string, also required, and must match this particular regular expression. And the logged in count is a number that defaults to zero. Also, the user's email has this lowercase property. This tells Mongoose to convert the email string to lowercase before saving which enables emails to be case insensitive.

In order to use this schema, you first need to make Mongoose connect to MongoDB. This mongoose.connect function is the most concise way to make Mongoose connect to this mongod server. So for this create index.js file and type below code.

var mongoose=require('mongoose');

var schema=require('./schema');

mongoose.Promise = global.Promise;

mongoose.connect('mongodb://localhost:27017/example');

//parameters are: model name, schema, collection name

var User=mongoose.model('User',schema,'users');

var user = new User({

name:'John Smith', email:'John@Smith.io'

});

user.save(function(error){

if(error)

{ console.log(error);

process.exit(1);

}

User.find({email:'john@smith.io'},function(error,docs){

if(error)

{ console.log(error);

process.exit(1);

}

console.log(require('util').inspect(docs));

process.exit(0);

});

});

Once you have called connect, you can then use the mongoose.model function to create a user model from the schema and collection. The user model has several nice properties. For instance, you can use the user model to create a new user document with name John Smith and email john@smith.io, and then you can save the document.

Now once you run this script, you should see that Mongoose creates this new user, saves it to the database, and then successfully queries for the user document again and prints it to the screen.

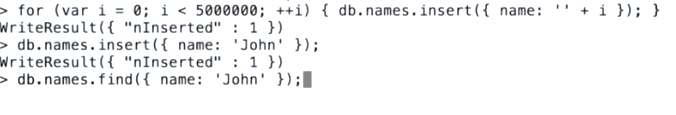


**Recipe7: MongoDB indexes**

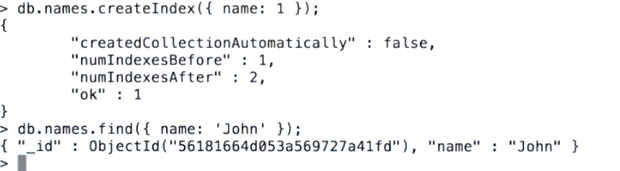
Indexes are a way to get consistent performance from your queries as your data grows. Indexes are key to getting good performance from MongoDB. Now, if you're familiar with indexes in SQL databases, MongoDB indexes are pretty similar.

This is OK for small data sets, but collection scans become more and more expensive as your data grows. Now, when you create an index by calling the create index function in the shell, MongoDB creates a data structure that maps the values of the name field to documents that have that value of the name field.

Now, let's see how this works for a larger data set. So, for instance, in this case we've inserted five million documents and then one document with name John. Without an index, you'll notice that there's a slight delay in executing the query.



So let's create an index and see what happens. So now we created an index on name, and let's re-run that query. As you'll see, the query is pretty instantaneous.



**Recipe8: Retail application schema: product**

Create product.js file as below to create product schema.

var mongoose = require('mongoose');

var Category = require('./category');

var CategoryModel = mongoose.model('CategoryModel', Category);

var productSchema = { name: { type: String, required: true },

// Pictures must start with "http://"

pictures: [{ type: String, match: /^http:\/\//i }],

price: { amount: { type: Number, required: true },

// Only 3 supported currencies for now

currency: { type: String, enum: ['USD', 'EUR', 'GBP'], required: true }

},

category: [{type: String, ref: 'CategoryModel'}]

};

module.exports = new mongoose.Schema(productSchema);

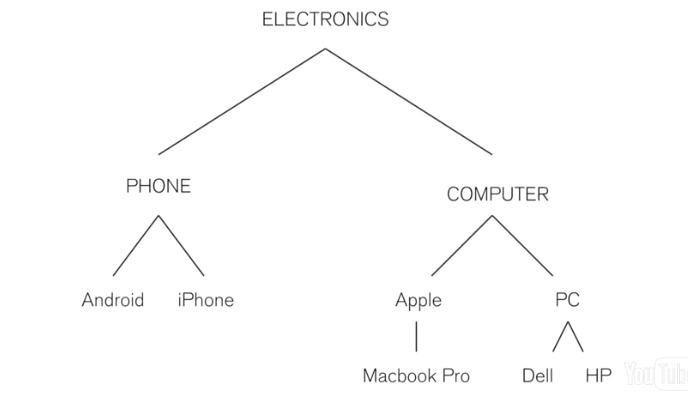
module.exports.productSchema = productSchema;

You have to design a set of schemas for the retail MEAN stack application. Recall that there will be three schemas-- product, category, and user.

The most basic schema, the one that we'll talk about first, is the product schema. The general idea is this schema represents what you will display on an individual product view, so the product's name, a list of pictures, how much the product costs, and the category that the product belongs to.

**Recipe9: Retail application schema: category**

One way your retail application will enable users to discover new products is through a category hierarchy.



Your user will be able to query for a list of all products under the category phones, which will include results for both Android and iOS. Your user will also be able to drill down into more specific categories like phones that run iOS or the more general category of electronics. With this in mind, you need to design your category schema in order to be able to efficiently answer the question which products are in the electronics category, as well as the question which products are in the iOS category.

So let’s create category.js file as below.

var mongoose=require('mongoose');

var categorySchema={

\_id:{type: String},

parent:{ type: String, ref: 'Category' },

ancestors:[{ type: String, ref: 'Category' }]};

module.exports=new mongoose.Schema(categorySchema);

module.exports.categorySchema=categorySchema;

**To test it create server.js file as below and test it using postman**

var express = require('express');

var bodyParser = require('body-parser');

// Mongoose import

var mongoose = require('mongoose');

// Mongoose connection to MongoDB

mongoose.connect('mongodb://localhost:27017/retail');

// Mongoose Schema definition

var Schema = mongoose.Schema;

var productSchema = require('./product');

var categorySchema=require('./category');

// Mongoose Model definition

var ProductModel = mongoose.model('ProductModel', productSchema);

var CategoryModel = mongoose.model('CategoryModel', categorySchema);

// Bootstrap express

var app = express();

//Add Json Support

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: false }));

// URLS management

/\*\*Create new product with its category

\* Call the following from a Rest-Client (postman):

http://localhost:3000/products/create

Header: Content-Type:application/json

Body:

{ "name": "iphone6",

"price":{"amount":100,"currency":"USD"},"category": ["iOS"] }

\*/

app.post('/products/create', function (req, res) {

console.log("body content: " + req.body);

var productModelJson = req.body;

var productModel = new ProductModel(productModelJson);

productModel.save(function saveProduct(error, savedProductModel) {

if (error) {

console.log("Error, ProductModel not saved");

return;

}

console.log("Success: ProductModel saved");

return res.json(savedProductModel);

});});

/\*\* Create new category

\* Call the following from a Rest-Client (postman):

http://localhost:3000/category/create

Header: Content-Type:application/json

{"\_id":"iOS", "parent": "Phones", "ancestors": ["Electronics","Phones","iOS"] }

\* \*/

app.post('/category/create', function (req, res) {

console.log("in /category/create");

var categoryModelJson = req.body;

var categoryModel = new CategoryModel(categoryModelJson);

categoryModel.save(function(error) {

if(error) {

console.log(error);

return res.json({msg: "error"});

}

res.json(categoryModel);

});});

app.get('/category', function (req, res) {

console.log("in category");

CategoryModel.find({}, function (err, docs) {

res.json(docs);

});});

app.get('/products', function (req, res) {

ProductModel

.findOne({ name:'iphone6' })

.populate('category')

.exec(function (err, docs) {

res.json(docs);"

});});

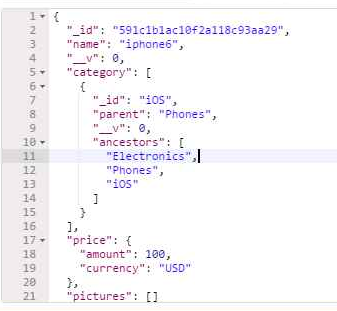
// Start the server

var serverApp = app.listen(3000, function () {

console.log('Express server listening on port 3000 ');

});

Final product is look like as below



**Recipe10: Retail application schema: user and cart**

The third and final schema that you will use in the retail application is the user schema. This schema defines the data that you'll store about individual users. The user document will contain the user's username, their profile picture, their Facebook oauth ID, and the list of products in their cart.

The Facebook oauth ID is the string that will serve as a unique identifier for this user's Facebook account. This will allow your users to log in with Facebook. User and product have a many-to-many relationship.

So let’s create user.js file as below.

var mongoose = require('mongoose');

module.exports = new mongoose.Schema({

profile: { username: { type: String, required: true, lowercase: true },

picture: { type: String, required: true, match: /^http:\/\//i } },

data: { oauth: { type: String, required: true }, cart: [{

product: { type: mongoose.Schema.Types.ObjectId },

quantity: { type: Number, default: 1, min: 1 }

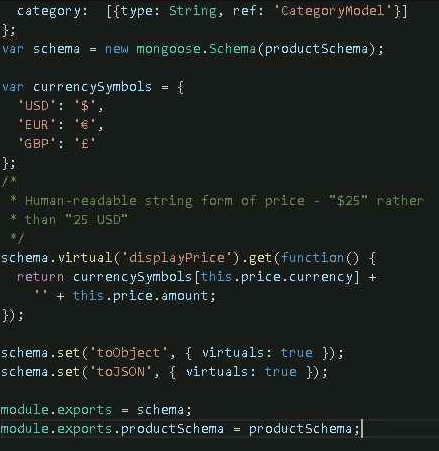
}] }});

**Recipe11: Advanced mongoose features: virtuals**

Mongoose has numerous handy features that make it an indispensable tool for web development to Node.js. Virtuals are one such feature. Virtuals are properties that are typically computed from other properties. They are not persisted to the database, but they can be accessed just like any other property.

Displaying a price as, say, 25 USD is not a very good choice for user experience. 25 preceded by a dollar sign is a more professional looking choice.

To declare a virtual you use this virtual function on your schema.



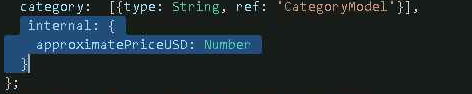
Save and run, output will be look like this.



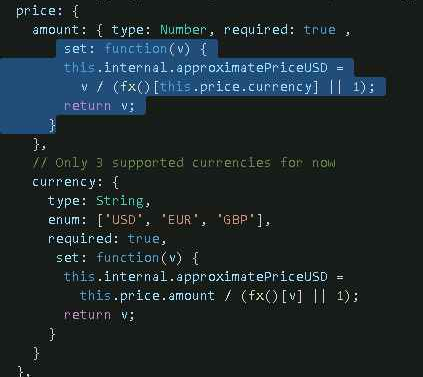
**Recipe12: Advanced mongoose features: custom setters + sorting by currency**

Your product schema supports different currencies. Products can list their price in either US dollars, euro dollars, or British pound sterling. You may be wondering, how will you be able to ensure a consistent sort order when products have different currencies?

Your REST API will be able to account for these differences and ensure that products are sorted using a recent exchange rate. In order to ensure consistency when sorting by price, what you want is a numeric field that represents the price of the product in a fixed currency. This approximatePriceUSD field will represent the product's price in US dollars.

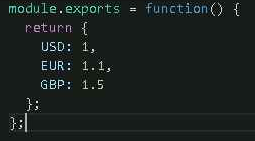


The Mongoose feature that will enable you to do this is custom setters. With custom setters you can tell Mongoose to perform certain operations every time the value of a certain field is set.



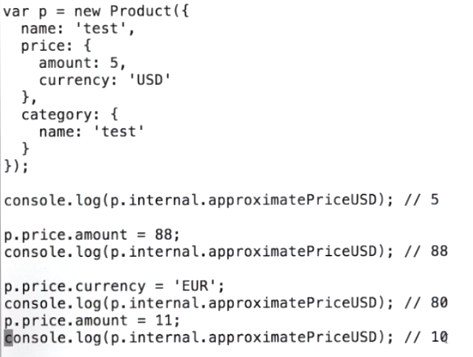
In this case, this function updates the internal.approximatePriceUSD property to reflect the changes in the price dot amount. The way your schema computes the internal.approximatePriceUSD field is by taking the provided price and dividing it by the exchange rate defined by this fx function.

The exchange rate is pulled from this fx function. And the fx function is required in from a file called fx.js. Let's take a look at what fx.js looks like.



As you can see, this fx file exports a function that returns some hard coded exchange rates that are reasonable approximations. In the REST API session, you will make this dynamic. But for this example, the hard coded rates are sufficient.

Let’s start manipulating the price in currency and see what happens to the internal.approximatePriceUSD field.



The power of custom setters is that you don't have to do anything special in the code that uses the product schema. You don't have to call any functions or even be aware of the fact that internal.approximatePriceUSD is changing. The Mongoose schema abstracts away that layer of business logic. So all you're doing is setting properties, and Mongoose takes care of the rest.