



Tutorials for iPhone / iOS Developers and Gamers

How To Write A Simple Node.js/MongoDB Web Service for an iOS App



Michael Katz on April 17, 2014

In today's world of collaborative and social apps, it's crucial to have a backend that is simple to build and easy to deploy. Many organizations rely on an application stack using the following three technologies:

- [Node.js](#)
- [Express](#)
- [MongoDB](#)

This stack is quite popular for mobile applications since the native data format is JSON which can be easily parsed by apps by way of Cocoa's [NSJSONSerialization](#) class or other comparable parsers.

In this tutorial you'll learn how to set-up a Node.js environment that leverages Express; on top of that platform you'll build a server that exposes a MongoDB database through a [REST API](#), as such:



Learn how to create your own web service for your iOS app with Node.js and MongoDB!

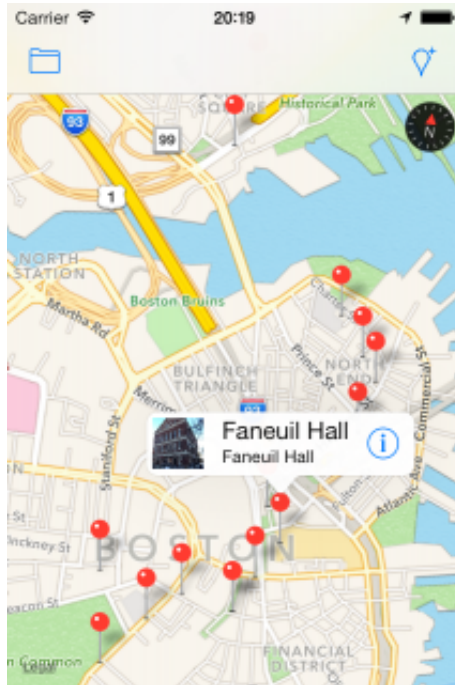


locations

name	placename	imageId	location	ca
"Boston Common"	"Boston Common"	"52f6685fd8ec644af5ba12b"	{\"type\":\"Point\",\"coordinates\":[-71.0629648,42.3555405]}	[\"Park\"]
"52f6d205890e1ff4bd000002"	"52f6d205890e1ff4bd000001"	"Massachusetts State House"	"Massachusetts State House"	{\"type\":\"Po[-71.063557
"Granary Burying Ground"	"Granary Burying Ground"	"52f6d555e3fd6321c3018c58"	{\"type\":\"Point\",\"coordinates\":[-71.0613395,42.3571132]}	[]
"King's Chapel and Burying Grounds"	"King's Chapel and Burying Grounds"	"52f6d614e3fd6321c3018c5a"	{\"type\":\"Point\",\"coordinates\":[-71.0600116,42.358029]}	[]
"52f6d6b5e3fd6321c3018c5d"	"52f6d6b5e3fd6321c3018c5c"	"Freedom Trail"	"Old South Meeting House"	{\"type\":\"Po[-71.057803
"Boston Massacre Site"	"Boston Massacre Site"	"52f6d6c2e3fd6321c3018c5e"	{\"type\":\"Point\",\"coordinates\":[-71.0570838,42.3588809]}	[]
"Faneuil Hall"	"Faneuil Hall"	"52f6d6d3e3fd6321c3018c60"	{\"type\":\"Point\",\"coordinates\":[-71.0562378,42.3600386]}	[]
			{\"type\":\"Point\",\"coordinates\":	

The backend database rendered in a HTML table

The second part of this tutorial series focuses on the iOS app side. You'll build a cool "places of interest" app to tag interesting locations so that other users can find out what's interesting near them. Here's a sneak peek at what you'll be building:



The TourMyTown main view

This tutorial assumes that you already know the basics of JavaScript and web development but are new to Node.js, Express, and MongoDB.

A Case for Node+Mongo

Most Objective-C developers likely aren't familiar with JavaScript, but it's an extremely common language among web developers. For this reason, Node has gained a lot of popularity as a web framework, but there's many more reasons that make it a great choice as a back-end service:

- Built-in server functionality
- Good project management through its package manager
- A fast JavaScript engine, known as V8
- Asynchronous event-driven programming model.

An asynchronous programming model of events and callbacks is well suited for a server which has to wait for a log of things, such as incoming requests and inter-process communications with other services (like MongoDB).

MongoDB is a low-overhead database where all entities are free-form **BSON** — "binary JSON" — documents. This lets you work with heterogeneous data and makes it easy to handle a wide variety of data formats. Since BSON is compatible with JSON, building a REST API is simple — the server code can pass requests to the database driver without a lot of intermediate processing.

Node and MongoDB are inherently scalable and synchronize easily across multiple machines in a distributed model;

this combination is a good choice for applications that don't have an evenly distributed load.

Getting Started

This tutorial assumes you have OS X Mountain Lion or Mavericks, and Xcode with its command line tools already installed.

The first step is to install [Homebrew](#). Just as CocoaPods manages packages for Cocoa and Gem manages packages for Ruby, Homebrew manages Unix tools on Mac OS X. It's built on top of Ruby and Git and is highly flexible and customizable.

If you already have Homebrew installed, feel free to skip to the next step. Otherwise, install Homebrew by opening Terminal and executing the following command:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/homebrew/go/install)"
```

Note: `cURL` is a handy tool to send and receive files and data using URL requests. You use it here to load the Homebrew installation script — later on in this tutorial you'll use it to interact with the Node server.

Once Homebrew is installed, enter the following command in Terminal:

```
brew update
```

This simply updates Homebrew so you have the most up-to-date package list.

Now, install MongoDB via Homebrew with the following command:

```
brew install mongodb
```

Make a note of the directory where MongoDB is installed as shown in the "Summary" output. You'll need this later to launch the MongoDB service.

Download and run the Node.js installer from <http://nodejs.org/download/>.

Once the installer has completed, you can test out your Node.js installation right away.

Enter the following command in Terminal:

```
node
```

This puts you into the Node.js interactive environment where you can execute JavaScript expressions.

Enter the following expression at the prompt:

```
console.log("Hello world");
```

You should receive the following output:

```
Hello world  
undefined
```

`console.log` is the Node.js equivalent of `NSLog`. However, the output stream of `console` is much more complex than `NSLog`: it has `console.info`, `console.assert`, `console.error` among other streams that you might expect from more advanced loggers such as [CocoaLumberjack](#).

The "undefined" value written to the output is the return value of `console.log` — which has no return value. Node.js always displays the output of all expressions, whether the return value is defined or not.

Note: If you've worked with JavaScript before, you need to be aware that there are a few differences between the Node.js environment and the browser environment. The `global` object is called `global` instead of `window`. Typing `global` and pressing enter at the Node.js interactive prompt displays all the methods and objects in the global namespace; however it's easier to just use [the Node.js documentation](#) as a reference. :]

The `global` object has all the pre-defined constants, functions, and datatypes available to programs running in the Node.js environment. Any user-created variables are added to the global context object as well. The output of `global` will list pretty much everything accessible in memory.

Running a Node.js Script

The interactive environment of Node.js is great for playing around and debugging your JavaScript expressions, but usually you'll use script files to get the real work done. Just as an iOS app includes `Main.m` as its starting point, the default entry point for Node.js is `index.js`. However, unlike Objective-C there is no `main` function; instead, `index.js` is evaluated from top to bottom.

Press Control+C twice to exit the Node.js shell. Execute the following command to create a new folder to hold your scripts:

```
mkdir ~/Documents/NodeTutorial
```

Now execute the following command to navigate into this new folder and create a new script file in your default text editor:

```
cd ~/Documents/NodeTutorial/; edit index.js
```

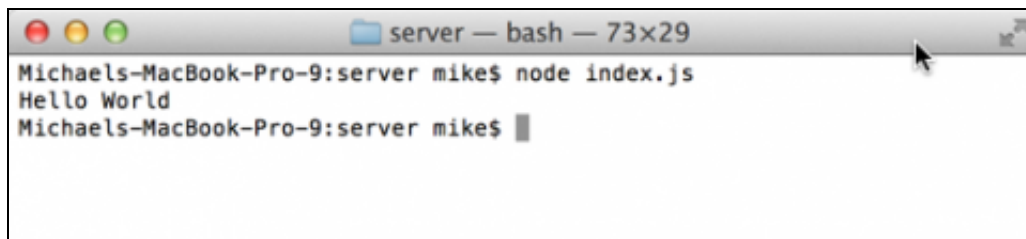
Add the following code to `index.js`:

```
console.log("Hello world.");
```

Save your work, return to Terminal and execute the following command to see your script in action:

```
node index.js
```

Once again, there's the familiar "Hello World" output. You can also execute `node .` to run your script, as `.` looks for `index.js` by default.



Admittedly, a "hello world" script does not a server make, but it's a quick way to test out your installation. The next section introduces you to the world of Node.js packages which will form the foundation of your shiny new web server!

Node Packages

Node.js applications are divided up into `packages`, which are the "frameworks" of the Node.js world. Node.js comes with several basic and powerful packages, but there are over 50,000 public packages provided by the vibrant developer community — and if you can't find what you need, you can easily make your own.

Note: Check out <https://npmjs.org/> for a list of available packages.

Replace the contents of **index.js** with the following code:

```
//1
var http = require('http');

//2
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('<html><body><h1>Hello world</h1></body></html>');
}).listen(3000);

console.log('Server running on port 3000.');
```

Taking each numbered comment in turn, you'll see the following:

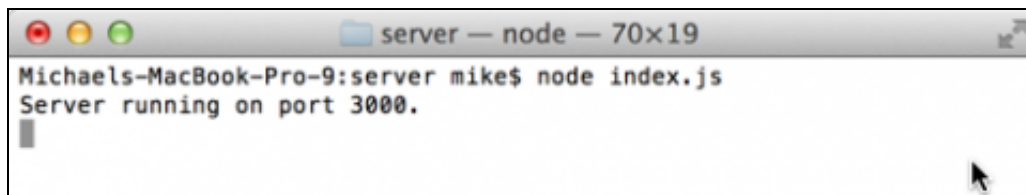
1. **require** imports modules into the current file. In this case you're importing the HTTP libraries.
2. Here you create the web service that responds to simple HTTP requests by sending a 200 response and writes the page content into the response.

One of the biggest strengths of Node.js as a runtime environment is its **event-driven model**. It's designed around the concept of callbacks that are called **asynchronously**. In the above example, you're listening on port 3000 for incoming http requests. When you receive a request, your script calls **function (req, res) {...}** and returns a response to the caller.

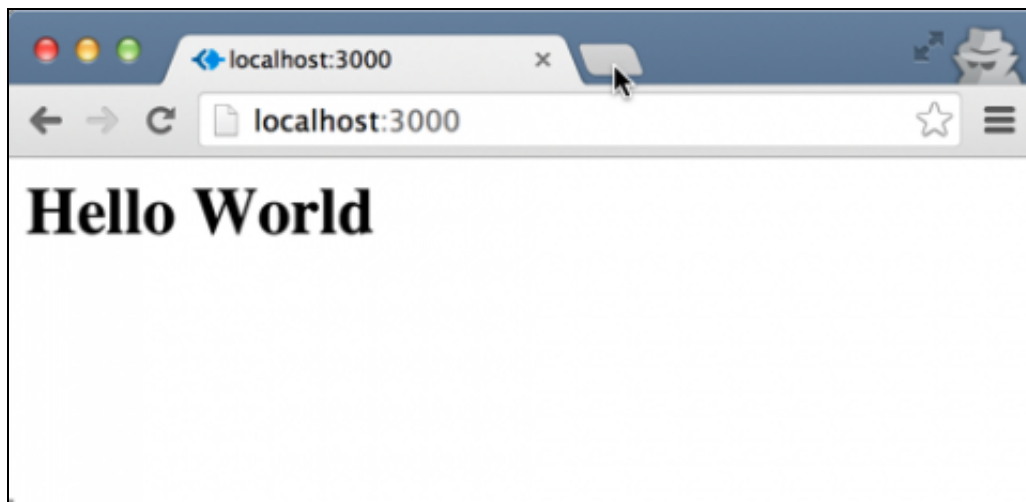
Save your file, then return to Terminal and execute the following command:

```
node index.js
```

You should see the following console output:



Open your favorite browser and navigate to **http://localhost:3000**; lo and behold, there's Node.js serving up your "Hello World" page:



Your script is still sitting there, patiently waiting for http requests on port 3000. To kill your Node instance, simply press **Ctrl+C** in Terminal.

Note: Node packages are usually written with a top-level function or object that is exported. This function is then assigned to a top-level variable by using the `require` function. This helps manage scope and expose the module's API in a sane manner. You'll see how to create a custom module later in this tutorial when you add a driver for MongoDB.

NPM – Using External Node Modules

The previous section covered Node.js built-in modules, but what about third party modules such as Express which you'll need later to provide the routing middleware for your server platform?

External modules are also imported into the file with the `require` function, but you have to download them separately and then make them available to your Node instance.

Node.js uses `npm` — the Node Packages Module — to download, install, and maintain package dependencies. If you're familiar with `Cocoapods` or ruby gems, then `npm` will feel familiar. Your Node.js application uses `package.json` which defines the configuration and dependencies of `npm`.

Using Express

Express is a popular Node.js module for routing middleware. Why do you need this separate package? Consider the following scenario.

If you used just the `http` module by itself, you'd have to parse each request's location separately to figure out what content to serve up to the caller — and that would become unwieldy in a very short time.

However, with Express you can easily define routes and chains of callbacks for each request. Express also makes it easy to provide different callbacks based on the http verb (eg. POST, PUT, GET, DELETE, HEAD, etc).

A Short Diversion into HTTP Verbs

An HTTP request includes a method — or verb — value. The default value is `GET`, which is for fetching data, such as web pages in a browser. `POST` is meant for uploading data, such submitting web forms. For web APIs, `POST` is generally used to add data, but it can also be used for remote procedure call-type endpoints.

PUT differs from **POST** in that it is generally used to replace existing data. In practical terms **POST** and **PUT** are usually used in the same way: to provide entities in the request body to be placed into a backend datastore. **DELETE** is used to remove items from your backend datastore.

POST, **GET**, **PUT**, and **DELETE** are the HTTP implementations of the CRUD model — **C**reate, **R**ead, **U**psert and **D**elese.

There are a few other HTTP verbs that are less well-known. **HEAD** acts like **GET** but only returns the response headers and not the the body. This helps minimize data transfer if the information in the response header is sufficient to determine if there is new data available. Other verbs such as **TRACE** and **CONNECT** are used for network routing.

Adding a Package to Your Node Instance

Execute the following command in Terminal:

```
edit package.json
```

This creates a new **package.json** to contain your package configuration and dependencies.

Add the following code to **package.json**:

```
{
  "name": "mongo-server",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.3.4"
  }
}
```

This file defines some metadata such as the project name and version, some scripts, and most importantly for your purposes, the package dependencies. Here's what each line means:

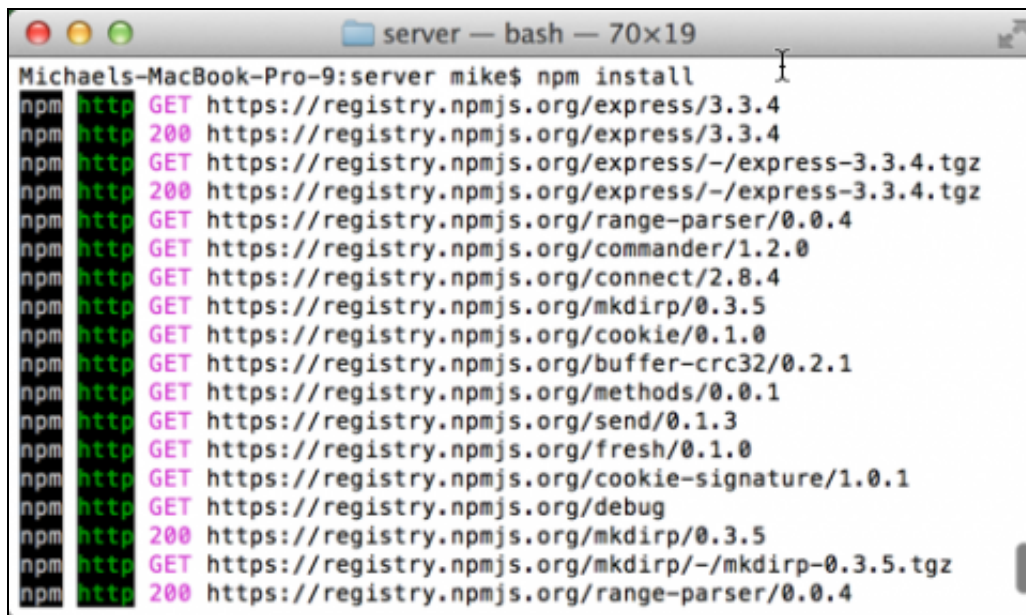
- **name** is the name of the project.
- **version** is the current version of the project.
- **private** prevents the project from being published accidentally if you set it to **true**.
- **dependencies** is a list containing the Node.js modules used by your application.

Dependencies take the form of key/value pairs of module names and versions. Your list of dependencies contains version 3.3.4 of Express; if you want to instruct Node.js to use the latest version of a package, you can use the wildcard **"*"**.

Save your file, then execute the following command in Terminal:

```
npm install
```

You'll see the following output:



```

server — bash — 70x19
Michaels-MacBook-Pro-9:server mike$ npm install
npm http GET https://registry.npmjs.org/express/3.3.4
npm http 200 https://registry.npmjs.org/express/3.3.4
npm http GET https://registry.npmjs.org/express/-/express-3.3.4.tgz
npm http 200 https://registry.npmjs.org/express/-/express-3.3.4.tgz
npm http GET https://registry.npmjs.org/range-parser/0.0.4
npm http GET https://registry.npmjs.org/commander/1.2.0
npm http GET https://registry.npmjs.org/connect/2.8.4
npm http GET https://registry.npmjs.org/mkdirp/0.3.5
npm http GET https://registry.npmjs.org/cookie/0.1.0
npm http GET https://registry.npmjs.org/buffer-crc32/0.2.1
npm http GET https://registry.npmjs.org/methods/0.0.1
npm http GET https://registry.npmjs.org/send/0.1.3
npm http GET https://registry.npmjs.org/fresh/0.1.0
npm http GET https://registry.npmjs.org/cookie-signature/1.0.1
npm http GET https://registry.npmjs.org/debug
npm http 200 https://registry.npmjs.org/mkdirp/0.3.5
npm http GET https://registry.npmjs.org/mkdirp/-/mkdirp-0.3.5.tgz
npm http 200 https://registry.npmjs.org/range-parser/0.0.4

```

`install` downloads and installs the dependencies specified in `package.json` — and your dependencies' dependencies! :] — into a folder named `node_modules` folder and makes them available to your application.

Once `npm` has completed, you can now use Express in your application.

Find the following line in `index.js`:

```
var http = require('http');
```

...and add the `require` call for Express, as below:

```
var http = require('http'),
    express = require('express');
```

This imports the Express package and stores it in a variable named `express`.

Add the following code to `index.js` just after the section you added above:

```
var app = express();
app.set('port', process.env.PORT || 3000);
```

This creates an Express app and sets its port to 3000 by default. You can overwrite this default by creating an environment variable named `PORT`. This type of customization is pretty handy during development tool, especially if you have multiple applications listening on various ports.

Add the following code to `index.js` just under the section you added above:

```
app.get('/', function (req, res) {
  res.send('<html><body><h1>Hello world</h1></body></html>');
});
```

This creates a `route handler`, which is a fancy name for a chain of request handlers for a given URL. Express matches the specified paths in the request and executes the callback appropriately.

Your callback above tells Express to match the root `/` and return the given HTML in the response. `send` formats the various response headers for you — such as `content-type` and the `status code` — so you can focus on writing great code instead.

Finally, replace the current `http.createServer(...)` section right down to and including the `console.log` line with the following code:


```
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

This is a little more compact than before. `app` implements the `function(req, res)` callback separately instead of including it inline here in the `createServer` call. You've also added a completion handler callback that is called once the port is open and ready to receive requests. Now your app waits for the port to be ready before logging the "listening" message to the console.

To review, `index.js` should now look like the following:

```
var http = require('http'),
    express = require('express');

var app = express();
app.set('port', process.env.PORT || 3000);

app.get('/', function (req, res) {
  res.send('<html><body><h1>Hello world</h1></body></html>');
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Save your file and execute the following command in Terminal:

```
node index.js
```

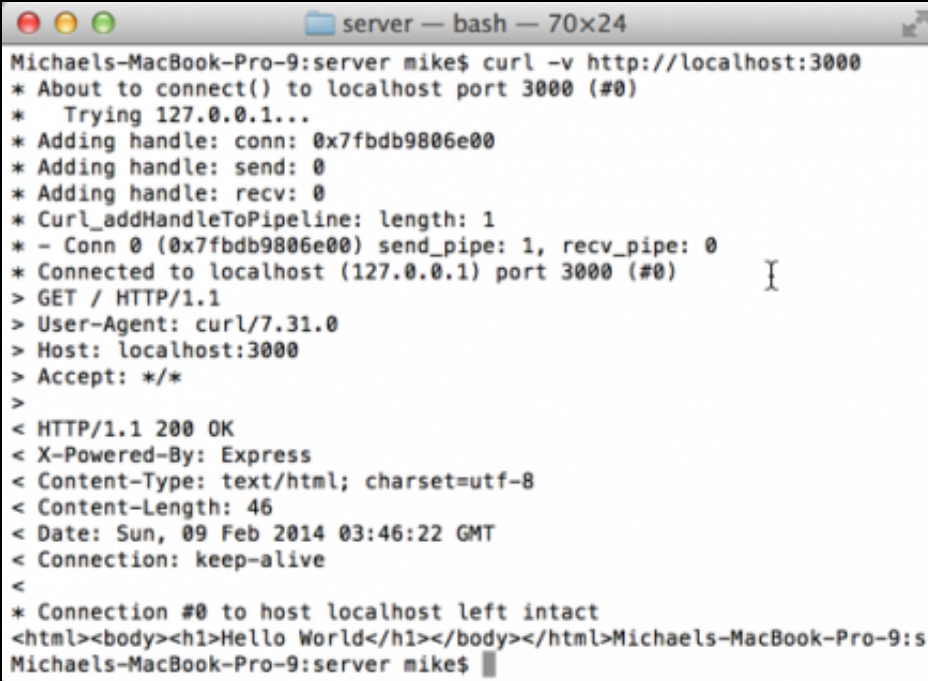
Return to your browser and reload `http://localhost:3000` to check that your Hello World page still loads.

Your page looks no different than before, but there's more than one way to see what's going on under the hood.

Create another instance of Terminal and execute the following command:

```
curl -v http://localhost:3000
```

You should see the following output:



```
server — bash — 70x24
Michaels-MacBook-Pro-9:server mike$ curl -v http://localhost:3000
* About to connect() to localhost port 3000 (#0)
*   Trying 127.0.0.1...
* Adding handle: conn: 0x7fbdb9806e00
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x7fbdb9806e00) send_pipe: 1, recv_pipe: 0
* Connected to localhost (127.0.0.1) port 3000 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.31.0
> Host: localhost:3000
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/html; charset=utf-8
< Content-Length: 46
< Date: Sun, 09 Feb 2014 03:46:22 GMT
< Connection: keep-alive
<
* Connection #0 to host localhost left intact
<html><body><h1>Hello World</h1></body></html>Michaels-MacBook-Pro-9:s
Michaels-MacBook-Pro-9:server mike$
```

`curl` spits back the response headers and content for your HTTP request to show you the raw details of what's being served up. Note the "**X-Powered-By : Express**" header; Express adds this metadata automatically to the response.

Serving up Content With Express

It's easy to serve up static files using Express.

Add the following statement to the `require` section at the top of `index.js`:

```
path = require('path');
```

Now add the following line just after the `app.set` statement:

```
app.use(express.static(path.join(__dirname, 'public')));
```

This tells Express to use the middleware `express.static` which serves up static files in response to incoming requests.

`path.join(__dirname, 'public')` maps the local subfolder `public` to the base route `/`; it uses the Node.js `path` module to create a platform-independent subfolder string.

`index.js` should now look like the following:

```
//1
var http = require('http'),
    express = require('express'),
    path = require('path');

//2
var app = express();
app.set('port', process.env.PORT || 3000);
app.use(express.static(path.join(__dirname, 'public')));

app.get('/', function (req, res) {
  res.send('<html><body><h1>Hello world</h1></body></html>');
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Using the static handler, anything in `/public` can now be accessed by name.

To demonstrate this, kill your Node instance by pressing Control+C, then execute the commands below in Terminal:

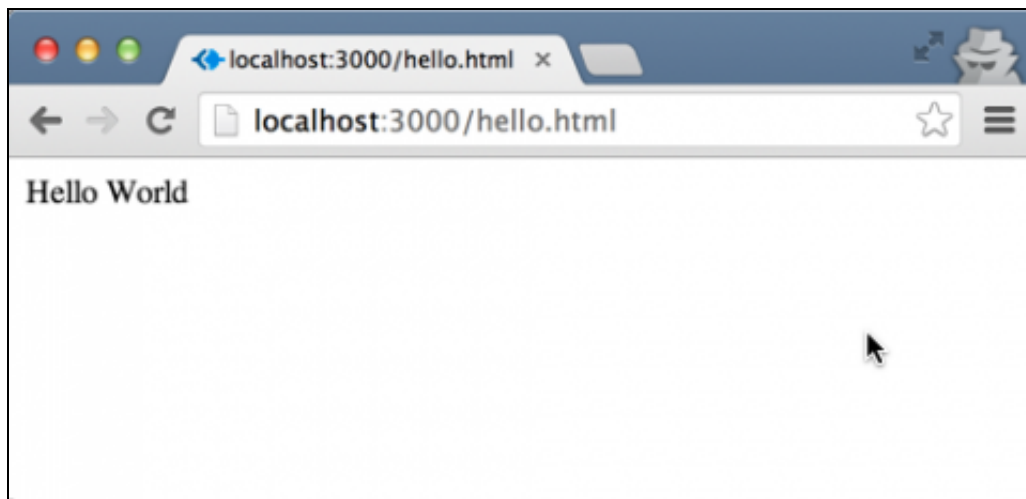
```
mkdir public; edit public/hello.html
```

Add the following code to `hello.html`:

```
<html></body>Hello world</body></html>
```

This creates a new directory `public` and creates a basic static HTML file.

Restart your Node instance again with the command `node index.js`. Point your browser to `http://localhost:3000/hello.html` and you'll see the newly created page as follows:



Advanced Routing

Static pages are all well and good, but the real power of Express is found in dynamic routing. Express uses a regular expression matcher on the route string and allows you to define parameters for the routing.

For example, the route string can contain the following items:

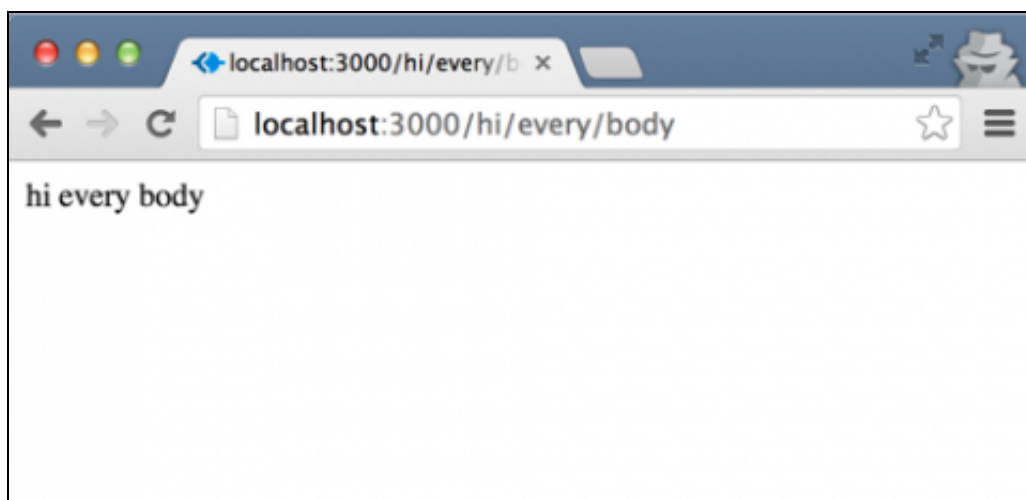
- **static terms** — `/files` only matches `http://localhost:300/pages`
- **parameters prefixed with “:”** — `/files/:filename` matches `/files/foo` and `/files/bar`, but not `/files`
- **optional parameters suffixed with “?”** — `/files/:filename?` matches both `/files/foo` and `/files`
- **regular expressions** — `/^\/people\/(\w+)/` matches `/people/jill` and `/people/john`

To try it out, add the following route after the `app.get` statement in `index.js`:

```
app.get('/:a?/:b?/:c?', function (req,res) {  
  res.send(req.params.a + ' ' + req.params.b + ' ' + req.params.c);  
});
```

This creates a new route which takes up to three path levels and displays those path components in the response body. Anything that starts with a `:` is mapped to a request parameter of the supplied name.

Restart your Node instance and point your browser to: `http://localhost:3000/hi/every/body`. You'll see the following page:



“hi” is the value of `req.params.a`, “every” is assigned to `req.params.b` and finally “body” is assigned to `req.params.c`.

This route matching is useful when building REST APIs where you can specify dynamic paths to specific items in backend datastores.

In addition to `app.get`, Express supports `app.post`, `app.put`, `app.delete` among others.

Error Handling And Templated Web Views

Server errors can be handled in one of two ways. You can pass an exception up the call stack — and likely kill the app by doing so — or you can catch the error and return a valid error code instead.

The [HTTP 1.1](#) protocol defines several error codes in the 4xx and 5xx range. The 400 range errors are for user errors, such as requesting an item that doesn’t exist: a familiar one is the common 404 Not Found error. 500 range errors are meant for server errors such as timeout or programming errors such as a null dereference.

You’ll add a catch-all route to display a **404** page when the requested content can’t be found. Since the route handlers are added in the order they are set with `app.use` or `app.verb`, a catch-all can be added at the end of the route chain.

Add the following code between the final `app.get` and the call to `http.createServer` in `index.js`:

```
app.use(function (req,res) { //1
  res.render('404', {url:req.url}); //2
});
```

This code causes the 404 page to be loaded if it is there is no previous call to `res.send()`.

There are a few points to note in this segment:

1. `app.use(callback)` matches *all* requests. When placed at the end of the list of `app.use` and `app.verb` statements, this callback becomes a catch-all.
2. The call to `res.render(view, params)` fills the response body with output rendered from a **templating engine**. A templating engine takes a template file called a “view” from disk and replaces variables with a set of key-value parameters to produce a new document.

Wait — a “templating engine”? Where does that come from?

Express can use a variety of templating engines to serve up views. To make this example work, you’ll add the popular [Jade](#) templating engine to your application.

Jade is a simple language that eschews brackets and uses whitespace instead to determine the ordering and containment of HTML tags. It also allows for variables, conditionals, iteration and branching to dynamically create the HTML document.

Update the list of dependencies in `package.json` as follows:

```
{
  "dependencies": {
    "express": "3.3.4",
    "jade": "1.1.5"
  }
}
```

Head back to Terminal, kill your Node instance, and execute the following command:

```
npm update
```

This downloads and installs the `jade` package for you.

Add the following code directly beneath the first `app.set` line in `index.js`:

```
app.set('views', path.join(__dirname, 'views'));  
app.set('view engine', 'jade');
```

The first line above specifies where the view templates live, while the second line sets Jade as the view rendering engine.

Execute the following command in Terminal:

```
mkdir views; edit views/404.jade
```

Add the following code to **404.jade**:

```
doctype html  
body  
  h1= 'Could not load page: ' + url
```

The first two lines of the Jade template create a new HTML document with a **body** element. The third line creates an **h1** element inside the **body** element due to the indent. Spacing is important in Jade! :]

The text of the **h1** element is the concatenated value of “Could not load page” and the value of the **url** parameter passed in as part of the **res.render()** in **index.js**.

As a quick check, your **index.js** file should now look like the following:

```
var http = require('http'),  
    express = require('express'),  
    path = require('path');  
  
var app = express();  
app.set('port', process.env.PORT || 3000);  
app.set('views', path.join(__dirname, 'views')); //A  
app.set('view engine', 'jade'); //B  
  
app.use(express.static(path.join(__dirname, 'public')));  
  
app.get('/', function (req, res) {  
  res.send('<html><body><h1>Hello world</h1></body></html>');  
});  
  
app.use(function (req, res) {  
  res.render('404', {url:req.url});  
});  
  
http.createServer(app).listen(app.get('port'), function(){  
  console.log('Express server listening on port ' + app.get('port'));  
});
```

Restart your instance of Node and use your browser to load the URL <http://localhost:3000/show/a/404/page>. You'll see the page below:



You now have enough starter code in `index.js` to accept incoming requests and provide some basic responses. All that you're missing is the database persistence to turn this into a useful web application that can be leveraged from a mobile app.

Introducing MongoDB

MongoDB is a database that stores JSON objects. Unlike a SQL database, a **NoSQL database** like Mongo does not support entity relationships. In addition, there is no pre-defined schema, so entities in the same collection (or "table", in relational-speak) don't need to have the same fields or conform to any predefined pattern.

MongoDB also provides a powerful querying language **map-reduce** along with support for location data. MongoDB is popular for its ability to scale, replicate and shard. Scaling and high-availability features are not covered in this tutorial.

The biggest drawbacks of MongoDB are the lack of relationship support and that it can be a memory hog as it memory-maps the actual database file. These issues can be mitigated by carefully structuring the data; this will be covered in part 2 of this tutorial.

Because of the close relationship between MongoDB documents and JSON, MongoDB is a great choice as a database for both web and mobile apps. MongoDB doesn't store raw JSON; instead, the documents are in a format called **BSON** — Binary JSON — that is more efficient for data storage and queries. BSON also has the advantage of supporting more datatypes than JSON, such as dates and C-types.

Adding MongoDB to Your Project

MongoDB is a native application and is accessed through **drivers**. There are a number of drivers for almost any environment, including Node.js. The MongoDB driver connects to the MongoDB server and issues commands to update or read data.

This means you'll need a running MongoDB instance listening on an open port. Fortunately, that's your very next step! :]

Open a new instance of Terminal and execute the following commands:

```
cd /usr/local/opt/mongodb/; mongod
```

This starts the MongoDB server daemon.

Now the MongoDB service is up and running on the default port of 27017.

Although the MongoDB driver provides database connectivity, it still has to be wired up to the server to translate incoming HTTP requests into the proper database commands.

Creating a MongoDB Collection Driver

Remember the `/:a/:b/:c` route you implemented earlier? What if you could use that pattern instead to look up database entries?

Since MongoDB documents are organized into collections, the route could be something simple like:

`/:collection/:entity` which lets you access objects based on a simple addressing system in an extremely RESTful fashion!

Kill your Node instance and execute the following commands in Terminal:

```
edit collectionDriver.js
```

Add the following code to `collectionDriver.js`:

```
var ObjectId = require('mongodb').ObjectId;
```

This line imports the various required packages; in this case, it's the `ObjectId` function from the MongoDB package.

Note: If you're familiar with traditional databases, you probably understand the term "primary key". MongoDB has a similar concept: by default, new entities are assigned a unique `_id` field of datatype `ObjectId` that MongoDB uses for optimized lookup and insertion. Since `ObjectId` is a BSON type and not a JSON type, you'll have to convert any incoming strings to `ObjectIds` if they're to be used when comparing against an `"_id"` field.

Add the following code to `collectionDriver.js` just after the line you added above:

```
CollectionDriver = function(db) {  
  this.db = db;  
};
```

This function defines the `CollectionDriver` constructor method; it stores a MongoDB client instance for later use. In JavaScript, `this` is a reference to the current context, just like `self` in Objective-C.

Still working in the same file, add the following code below the block you just added:

```
CollectionDriver.prototype.getCollection = function(collectionName, callback) {  
  this.db.collection(collectionName, function(error, the_collection) {  
    if( error ) callback(error);  
    else callback(null, the_collection);  
  });  
};
```

This section defines a helper method `getCollection` to obtain a Mongo collection by name. You define class methods by adding functions to `prototype`.

`db.collection(name, callback)` fetches the collection object and returns the collection — or an error — to the callback.

Add the following code to `collectionDriver.js`, below the block you just added:

```
CollectionDriver.prototype.findAll = function(collectionName, callback) {  
  this.getCollection(collectionName, function(error, the_collection) { //A  
    if( error ) callback(error);  
    else {  
      the_collection.find().toArray(function(error, results) { //B  
        if( error ) callback(error);  
        else callback(null, results);  
      });  
    }  
  });  
};
```



```

    }
  });
};

```

`CollectionDriver.prototype.findAll` gets the collection in line A above, and if there is no error such as an inability to access the MongoDB server, it calls `find()` on it in line B above. This returns all of the found objects.

`find()` returns a **data cursor** that can be used to iterate over the matching objects. `find()` can also accept a selector object to filter the results. `toArray()` organizes all the results in an array and passes it to the callback. This final callback then returns to the caller with either an error or all of the found objects in the array.

Still working in the same file, add the following code below the block you just added:

```

CollectionDriver.prototype.get = function(collectionName, id, callback) { //A
  this.getCollection(collectionName, function(error, the_collection) {
    if (error) callback(error);
    else {
      var checkForHexRegExp = new RegExp("^[0-9a-fA-F]{24}$"); //B
      if (!checkForHexRegExp.test(id)) callback({error: "invalid id"});
      else the_collection.findOne({'_id':ObjectID(id)}, function(error,doc) { //C
        if (error) callback(error);
        else callback(null, doc);
      });
    }
  });
};

```

In line A above, `CollectionDriver.prototype.get` obtains a single item from a collection by its `_id`. Similar to `prototype.findAll` method, this call first obtains the collection object then performs a `findOne` against the returned object. Since this matches the `_id` field, a `find()`, or `findOne()` in this case, has to match it using the correct datatype.

MongoDB stores `_id` fields as BSON type `ObjectID`. In line C above, `ObjectID()` (C) takes a string and turns it into a BSON `ObjectID` to match against the collection. However, `ObjectID()` is persnickety and requires the appropriate hex string or it will return an error: hence, the regex check up front in line B.

This doesn't guarantee there is a matching object with that `_id`, but it guarantees that `ObjectID` will be able to parse the string. The selector `{ '_id':ObjectID(id) }` matches the `_id` field exactly against the supplied `id`.

Note: Reading from a non-existent collection or entity is not an error – the MongoDB driver just returns an empty container.

Add the following line to `collectionDriver.js` below the block you just added:

```
exports.CollectionDriver = CollectionDriver;
```

This line declares the exposed, or exported, entities to other applications that list `collectionDriver.js` as a required module.

Save your changes — you're done with this file! Now all you need is a way to call this file.

Using Your Collection Driver

To call your `collectionDriver`, first add the following line to the `dependencies` section of `package.json`:

```
"mongodb": "1.3.23"
```

Execute the following command in Terminal:

```
npm update
```

This downloads and installs the MongoDB package.

Execute the following command in Terminal:

```
edit views/data.jade
```

Now add the following code to **data.jade**, being mindful of the indentation levels:

```
body
  h1= collection
  #objects
    table(border=1)
      if objects.length > 0
        - each val, key in objects[0]
          th= key
        - each obj in objects
          tr.obj
            - each val, key in obj
              td.key= val
```

This template renders the contents of a collection in an HTML table to make them human-readable.

Add the following code to **index.js**, just beneath the line **path = require('path');**:

```
MongoClient = require('mongodb').MongoClient,
Server = require('mongodb').Server,
CollectionDriver = require('./collectionDriver').CollectionDriver;
```

Here you include the **MongoClient** and **Server** objects from the MongoDB module along with your newly created **CollectionDriver**.

Add the following code to **index.js**, just after the last **app.set** line:

```
var mongoHost = 'localhost'; //A
var mongoPort = 27017;
var collectionDriver;

var mongoClient = new MongoClient(new Server(mongoHost, mongoPort)); //B
mongoClient.open(function(err, mongoClient) { //C
  if (!mongoClient) {
    console.error("Error! Exiting... Must start MongoDB first");
    process.exit(1); //D
  }
  var db = mongoClient.db("MyDatabase"); //E
  collectionDriver = new CollectionDriver(db); //F
});
```

Line A above assumes the MongoDB instance is running locally on the default port of 27017. If you ever run a MongoDB server elsewhere you'll have to modify these values, but leave them as-is for this tutorial.

Line B creates a new MongoClient and the call to **open** in line C attempts to establish a connection. If your connection attempt fails, it is most likely because you haven't yet started your MongoDB server. In the absence of a connection the app exits at line D.

If the client does connect, it opens the **MyDatabase** database at line E. A MongoDB instance can contain multiple databases, all which have unique namespaces and unique data. Finally, you create the **CollectionDriver** object in line F and pass in a handle to the MongoDB client.

Replace the first two **app.get** calls in **index.js** with the following code:

```

app.get('/:collection', function(req, res) { //A
  var params = req.params; //B
  collectionDriver.findAll(req.params.collection, function(error, objs) { //C
    if (error) { res.send(400, error); } //D
    else {
      if (req.accepts('html')) { //E
        res.render('data',{objects: objs, collection: req.params.collection});
//F
      } else {
        res.set('Content-Type','application/json'); //G
        res.send(200, objs); //H
      }
    }
  });
});

app.get('/:collection/:entity', function(req, res) { //I
  var params = req.params;
  var entity = params.entity;
  var collection = params.collection;
  if (entity) {
    collectionDriver.get(collection, entity, function(error, objs) { //J
      if (error) { res.send(400, error); }
      else { res.send(200, objs); } //K
    });
  } else {
    res.send(400, {error: 'bad url', url: req.url});
  }
});

```

This creates two new routes: `/:collection` and `/:collection/:entity`. These call the `collectionDriver.findAll` and `collectionDriver.get` methods respectively and return either the JSON object or objects, an HTML document, or an error depending on the result.

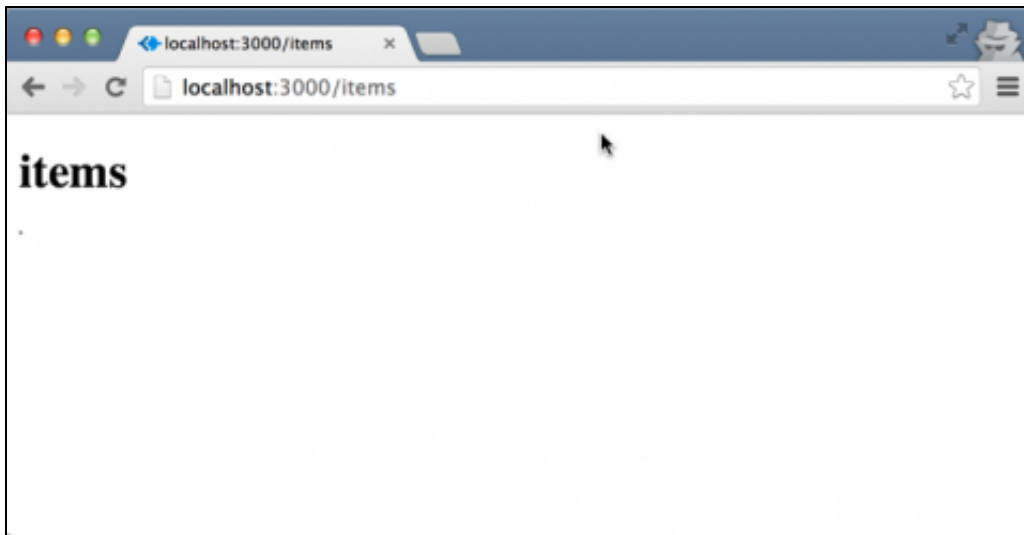
When you define the `/collection` route in Express it will match “collection” exactly. However, if you define the route as `/:collection` as in line A then it will match any first-level path store the requested name in the `req.params.collection` in line B. In this case, you define the endpoint to match any URL to a MongoDB collection using `findAll` of `CollectionDriver` in line C.

If the fetch is successful, then the code checks if the request specifies that it accepts an HTML result in the header at line E. If so, line F stores the rendered HTML from the `data.jade` template in `response`. This simply presents the contents of the collection in an HTML table.

By default, web browsers specify that they accept HTML in their requests. When other types of clients request this endpoint such as iOS apps using `NSURLSession`, this method instead returns a machine-parsable JSON document at line G. `res.send()` returns a success code along with the JSON document generated by the collection driver at line H.

In the case where a two-level URL path is specified, line I treats this as the collection name and entity `_id`. You then request the specific entity using the `get()` `collectionDriver`'s method in line J. If that entity is found, you return it as a JSON document at line K.

Save your work, restart your Node instance, check that your `mongod` daemon is still running and point your browser at `http://localhost:3000/items`; you'll see the following page:



Hey that's a whole lot of nothing? What's going on?

Oh, wait — that's because you haven't added any data yet. Time to fix that!

Working With Data

Reading objects from an empty database isn't very interesting. To test out this functionality, you need a way to add entities into the database.

Add the following method to **CollectionDriver.js** and add the following new prototype method just before the **exports.CollectionDriver** line:

```
//save new object
CollectionDriver.prototype.save = function(collectionName, obj, callback) {
  this.getCollection(collectionName, function(error, the_collection) { //A
    if( error ) callback(error)
    else {
      obj.created_at = new Date(); //B
      the_collection.insert(obj, function() { //C
        callback(null, obj);
      });
    }
  });
};
```

Like **findAll** and **get**, **save** first retrieves the collection object at line A. The callback then takes the supplied entity and adds a field to record the date it was created at line B. Finally, you insert the modified object into the collection at line C. **insert** automatically adds **_id** to the object as well.

Add the following code to **index.js** just after the string of **get** methods you added a little while back:

```
app.post('/:collection', function(req, res) { //A
  var object = req.body;
  var collection = req.params.collection;
  collectionDriver.save(collection, object, function(err, docs) {
    if (err) { res.send(400, err); }
    else { res.send(201, docs); } //B
  });
});
```

This creates a new route for the **POST** verb at line A which inserts the body as an object into the specified collection by calling **save()** that you just added to your driver. Line B returns the success code of HTTP 201 when the resource is

created.

There's just one final piece. Add the following line to `index.js` just after the `app.set` lines, but before any `app.use` or `app.get` lines:

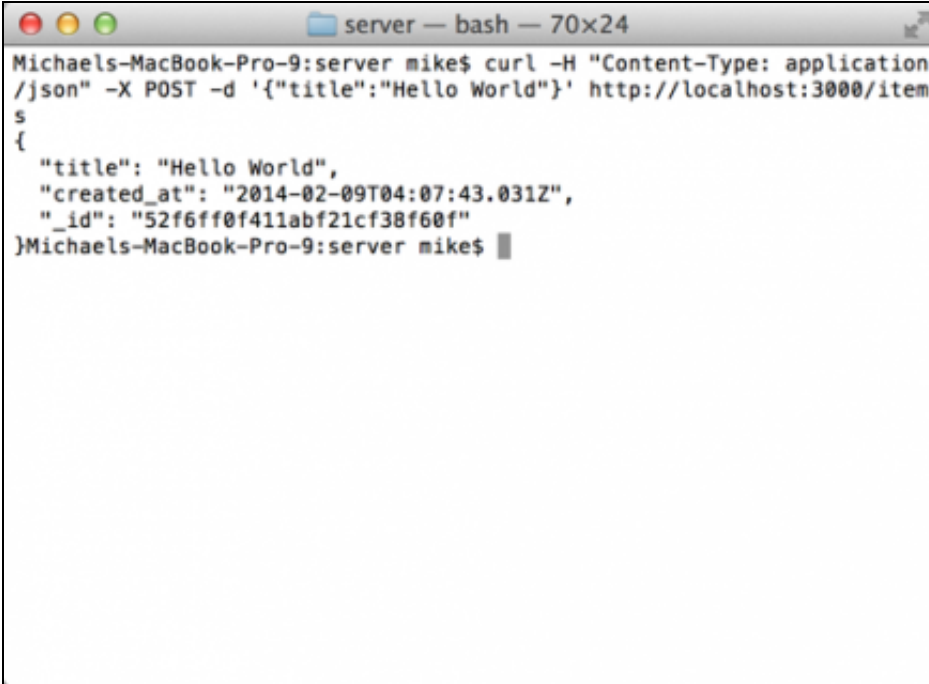
```
app.use(express.bodyParser());
```

This tells Express to parse the incoming body data; if it's JSON, then create a JSON object with it. By putting this call first, the body parsing will be called before the other route handlers. This way `req.body` can be passed directly to the driver code as a JavaScript object.

Restart your Node instance once again, and execute the following command in Terminal to insert a test object into your database:

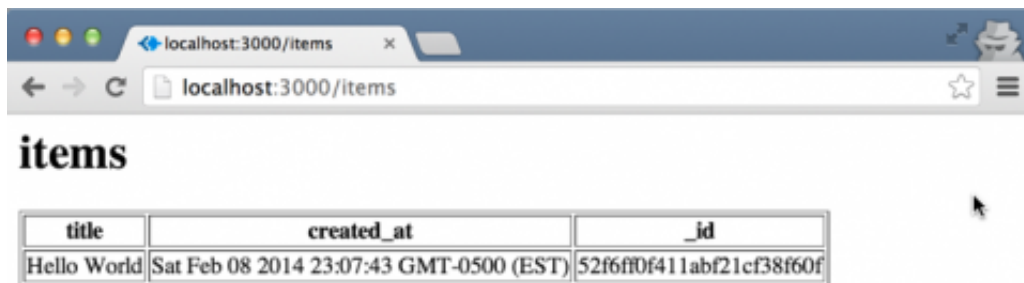
```
curl -H "Content-Type: application/json" -X POST -d '{"title":"Hello world"}' http://localhost:3000/items
```

You'll see the record echoed back to you in your console, like so:

A screenshot of a terminal window titled "server — bash — 70x24". The prompt is "Michaels-MacBook-Pro-9:server mike\$". The command entered is "curl -H 'Content-Type: application/json' -X POST -d '{\"title\":\"Hello World\"}' http://localhost:3000/items". The output is a JSON object: {"title": "Hello World", "created_at": "2014-02-09T04:07:43.031Z", "_id": "52f6ff0f411abf21cf38f60f"}.

```
Michaels-MacBook-Pro-9:server mike$ curl -H "Content-Type: application
/json" -X POST -d '{"title":"Hello World"}' http://localhost:3000/item
s
{
  "title": "Hello World",
  "created_at": "2014-02-09T04:07:43.031Z",
  "_id": "52f6ff0f411abf21cf38f60f"
}Michaels-MacBook-Pro-9:server mike$
```

Now head back to your browser and reload <http://localhost:3000/items>; you'll see the item you inserted show up in the table:



Updating and Deleting Data

You've implemented the Create and Read operations of CRUD — all that's left are Update and Delete. These are relatively straightforward and follow the same pattern as the other two.

Add the following code to `CollectionDriver.js` before the `exports.CollectionDriver` line:

```
//update a specific object
CollectionDriver.prototype.update = function(collectionName, obj, entityId, callback) {
  this.getCollection(collectionName, function(error, the_collection) {
    if (error) callback(error);
    else {
      obj._id = ObjectID(entityId); //A convert to a real obj id
      obj.updated_at = new Date(); //B
      the_collection.save(obj, function(error, doc) { //C
        if (error) callback(error);
        else callback(null, obj);
      });
    }
  });
};
```

`update()` function takes an object and updates it in the collection using `collectionDriver`'s `save()` method in line C. This assumes that the body's `_id` is the same as specified in the route at line A. Line B adds an `updated_at` field with the time the object is modified. Adding a modification timestamp is a good idea for understanding how data changes later in your application's lifecycle.

Note that this update operation replaces whatever was in there before with the new object – there's no property-level updating supported.

Add the following code to `collectionDriver.js` just before the `exports.CollectionDriver` line:

```
//delete a specific object
CollectionDriver.prototype.delete = function(collectionName, entityId, callback) {
  this.getCollection(collectionName, function(error, the_collection) { //A
    if (error) callback(error);
    else {
      the_collection.remove({'_id':ObjectID(entityId)}, function(error, doc) { //B
        if (error) callback(error);
        else callback(null, doc);
      });
    }
  });
};
```

```
});
};
```

`delete()` operates the same way as the other CRUD methods. It fetches the collection object in line A then calls `remove()` with the supplied `id` in line B.

Now you need two new routes to handle these operations. Fortunately, the `PUT` and `DELETE` verbs already exist so you can create handlers that use the same semantics as `GET`.

Add the following code to `index.js` just after the `app.post()` call:

```
app.put('/:collection/:entity', function(req, res) { //A
  var params = req.params;
  var entity = params.entity;
  var collection = params.collection;
  if (entity) {
    collectionDriver.update(collection, req.body, entity, function(error, objs) { //B
      if (error) { res.send(400, error); }
      else { res.send(200, objs); } //C
    });
  } else {
    var error = { "message" : "Cannot PUT a whole collection" };
    res.send(400, error);
  }
});
```

The `put` callback follows the same pattern as the single-entity `get`: you match on the collection name and `_id` as shown in line A. Like the `post` route, `put` passes the JSON object from the body to the new `collectionDriver`'s `update()` method in line B.

The updated object is returned in the response (line C), so the client can resolve any fields updated by the server such as `updated_at`.

Add the following code to `index.js` just below the `put` method you added above:

```
app.delete('/:collection/:entity', function(req, res) { //A
  var params = req.params;
  var entity = params.entity;
  var collection = params.collection;
  if (entity) {
    collectionDriver.delete(collection, entity, function(error, objs) { //B
      if (error) { res.send(400, error); }
      else { res.send(200, objs); } //C 200 b/c includes the original doc
    });
  } else {
    var error = { "message" : "Cannot DELETE a whole collection" };
    res.send(400, error);
  }
});
```

The `delete` endpoint is very similar to `put` as shown by line A except that `delete` doesn't require a body. You pass the parameters to `collectionDriver`'s `delete()` method at line B, and if the delete operation was successful then you return the original object with a response code of 200 at line C.


If anything goes wrong during the above operation, you'll return the appropriate error code.

Save your work and restart your Node instance.

Execute the following command in Terminal, replacing `{_id}` with whatever value was returned from the original `POST` call:

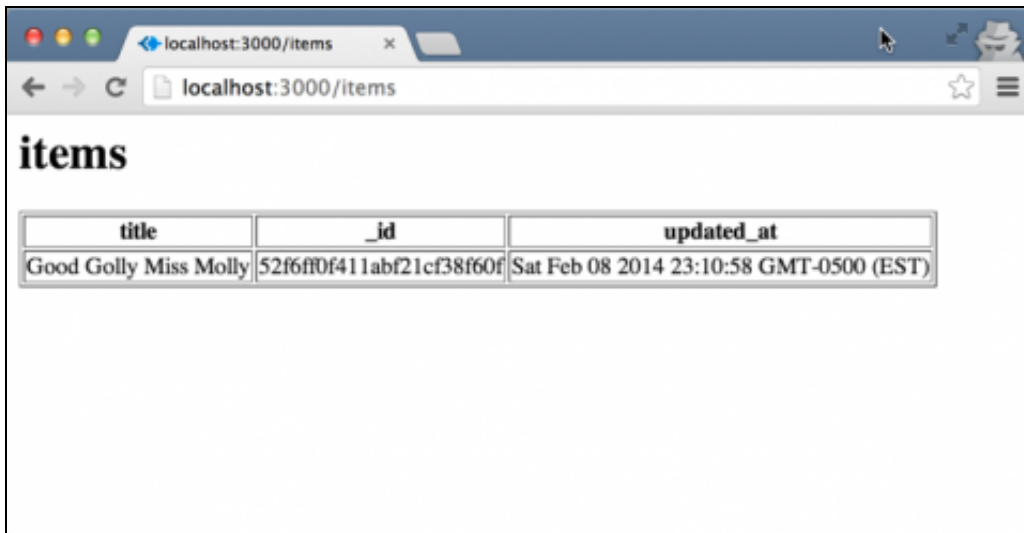

```
curl -H "Content-Type: application/json" -X PUT -d '{"title":"Good Golly Miss Molly"}' http://localhost:3000/items/{_id}
```

You'll see the following response in Terminal:



```
server — bash — 70x24
Michael's-MacBook-Pro-9:server mike$ curl -H "Content-Type: application
/json" -X PUT -d '{"title":"Good Golly Miss Molly"}' http://localhost:
3000/items/52f6ff0f411abf21cf38f60f
{
  "title": "Good Golly Miss Molly",
  "_id": "52f6ff0f411abf21cf38f60f",
  "updated_at": "2014-02-09T04:10:58.009Z"
}Michael's-MacBook-Pro-9:server mike$
```

Head to your browser and reload <http://localhost:3000/items>; you'll see the item you inserted show up in the table:

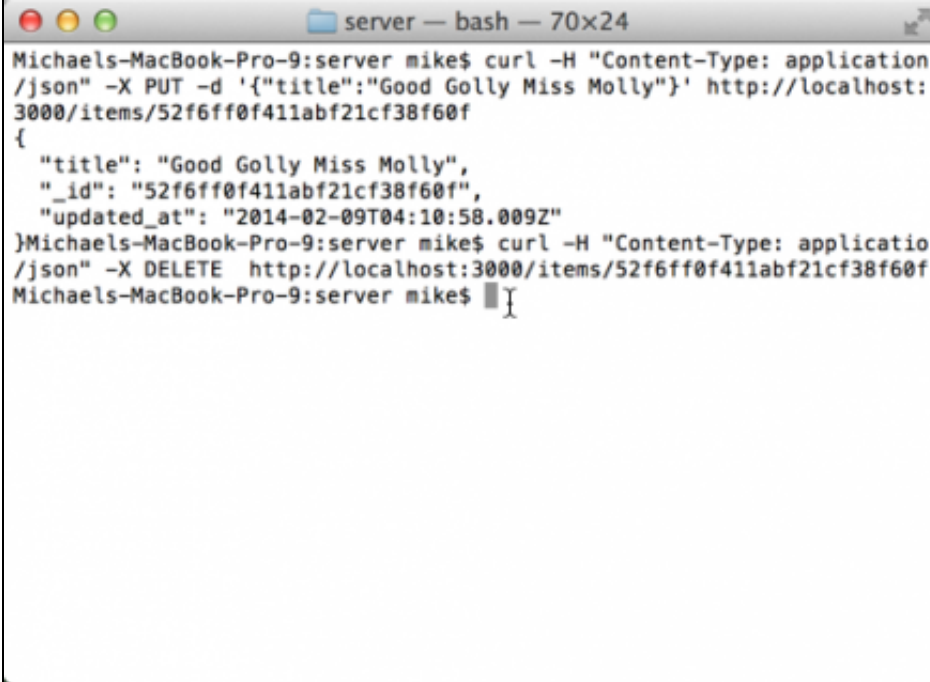


title	_id	updated_at
Good Golly Miss Molly	52f6ff0f411abf21cf38f60f	Sat Feb 08 2014 23:10:58 GMT-0500 (EST)

Execute the following command in Terminal to delete your record:

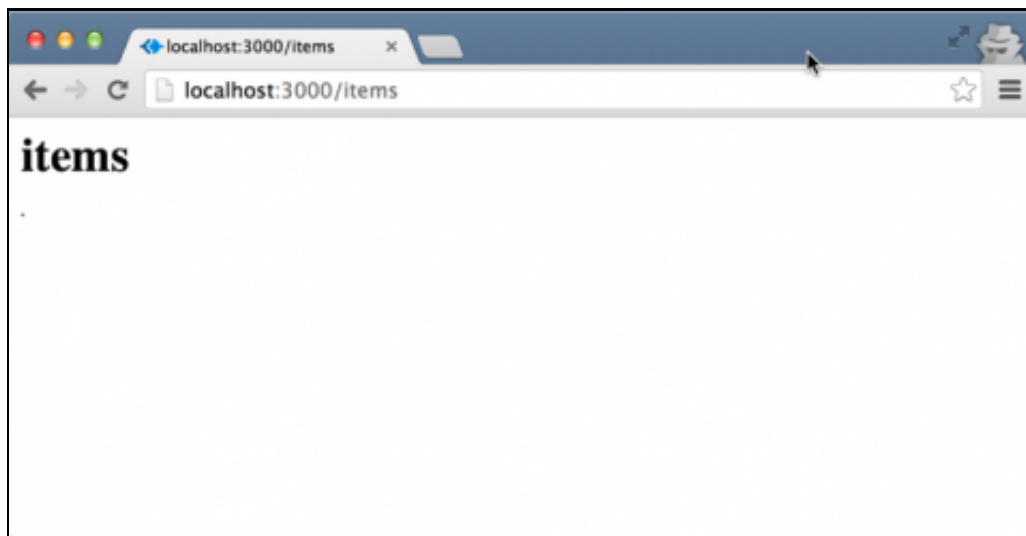
```
curl -H "Content-Type: application/json" -X DELETE http://localhost:3000/items/{_id}
```

You'll receive the following response from `curl`:

A terminal window titled 'server — bash — 70x24' on a Mac. It shows two curl commands being executed. The first is a PUT request to create a new item, and the second is a DELETE request to remove it. The output of the first command is a JSON object representing the created item.

```
Michael's-MacBook-Pro-9:server mike$ curl -H "Content-Type: application
/json" -X PUT -d '{"title":"Good Golly Miss Molly"}' http://localhost:
3000/items/52f6ff0f411abf21cf38f60f
{
  "title": "Good Golly Miss Molly",
  "_id": "52f6ff0f411abf21cf38f60f",
  "updated_at": "2014-02-09T04:10:58.009Z"
}Michael's-MacBook-Pro-9:server mike$ curl -H "Content-Type: applicatio
/json" -X DELETE http://localhost:3000/items/52f6ff0f411abf21cf38f60f
Michael's-MacBook-Pro-9:server mike$
```

Reload <http://localhost:3000/items> and sure enough, your entry is now gone:



And with that, you've completed your entire CRUD model using Node.js, Express, and MongoDB!

Where to Go From Here?

Here is the completed [sample project](#) with all of the code from the above tutorial.

Your server is now ready for clients to connect and start transferring data. In the next part of this tutorial series, you'll build an iOS app to connect to your new server and take advantage of some of the cool features of MongoDB and Express.

For more information MongoDB, check out the [official MongoDB documentation](#).

If you have any questions or comments about this tutorial, please join the discussion below!



Michael Katz

Michael Katz envisions a world where mobile apps always work, respect users' privacy, and integrate well with their users' life. To further that end, he's a Senior Software Engineer at Quanttus. When not coding, he can be found with his family playing board games, brewing, gardening, and watching the Yankees. Michael Katz has a B.S. and M.Eng in Electrical Engineering from Cornell University and a M.S. in Neuroscience from Brandeis University.