# Blog rolling with mongoDB, express and Node.js

> Article and Code updated by loarabia (Larry Olson). Article and Code updated by Toby Clemson

In this article I hope to take you through the steps required to get a fully-functional (albeit feature-light) persistent blogging system running on top of node.

The technology stack that we'll be using will be node + express + mongoDB all of which are exciting, fast and highly scalable. You'll also get to use jade and stylus for driving the templated views and styling! We will be using npm to ease the package management and installation issues.

This article will be fairly in-depth so you may want to get yourself a rather large mug of whatever beverage you prefer before you settle down :)

## Getting Started / Pre-Requisites.

### mongoDB

Installation is as simple as downloading the installer from here. For this tutorial I've been using v1.8.2 on MacOSX but any recent version should work. Once installed you can just execute 'mongod' to have a local instance up and running.

### node.js

I'll assume that you already have an installed version of node.js (why else would you be looking at a how-to?! ;) ) However as node is subject to a reasonably high rate of change for the purposes of this article everything has been written to run against the 'v0.4.10' tag.

### npm

The original version of this article required various dependencies to be installed by hand from github repositories, package management has moved on within node.js since that time and we will use this to our advantage by

following the instructions on how to install the npm. Once npm is installed you should be able to execute:

```
npm search
```

From your console and see a list of node packages that can be installed, if you cannot then it would be a good plan to figure out why not before proceeding.

## Getting hold of express

Installing express on your system is as easy as:

```
// You may need to run this under sudo
npm install express -g
```

Now we can begin with the process of writing our blog, let the good times (blog)roll.

## Defining our application

We're going to build a very simple blogging application (perhaps we'll build on this in a future article?). It is going to support the reading of blog articles, posting of blog articles and commenting on them. There will be no security, authentication or authorization. Hopefully this will demonstrate enough of the technology stack to let you move forward quickly.

### The data types

Because we're dealing with a document orientated database rather than a relational database we don't need to worry about what 'tables' we will need to persist this data to the database or the relationships between records within those tables. In fact we only have 1 datatype in the application at all, the article:

```
{
  _id: 0,
  title: '',
  body: '',
  comments: [{
    person: '',
    comment: '',
    created_at: new Date()
  }],
```

```
  created_at: new Date()
}
```

There are plenty of other document configurations we could've used. For example, there's no notion of an article's authors and the 'created_at' field could probably be omitted as the default 'Primary Key factory' that mongodb-native uses generates time-based object ids. However, this datatype provides a good foundation and we'll use a 'proper' date for simplicity.

> It should be noted that one oft-reported issue with mongoDB is the size of the data on the disk. As we're dealing with a document orientated database each and every record stores all the field-names with the data so there is no re-use. This means that it can often be more space-efficient to have properties such as 't', or 'b' rather than 'title' or 'body', however for fear of confusion I would avoid this unless truly required!

## The operations

There is a discrete set of operations (or things we want to achieve) that fall within the scope of this article, they are (in the order that we will tackle them):

1. Create a new article.
2. Show the list of all the articles.
3. Show an individual article and its comments.
4. Comment on an article

Now that we know what we're trying to achieve lets try and achieve that goal in a step-by-step fashion.

## From small acorns do giant oak trees grow

*Well alright, fairly small blogging apps can grow!*

In express a 'normal' application consists of a call to `configure`, followed by a series of method calls that declare `routes` and what happens to requests that match the route followed by a call to `listen`.

Thus one of the simplest express applications could be written as follows:

simple-express.js

```
// Module dependencies.
var express = require('express');

var app = express.createServer();

// Configuration
```

```
app.configure( function() {
});

// Routes
app.get('/', function(req, res) {
    res.send('Hello World');
});

app.listen(3000);
```

The above code declares a single `route` that operates on `GET` requests to the address `/` from the browser and will just return a simple (non-HTML) text string and a response code of 200 to the client.

Now, this is one of the simplest bits of application code one can write but express is a framework which can automatically build out an application template for you including picking a styling and templating engine.

By executing the following commands:

```
mkdir blog
cd blog
express -c stylus
npm install -d
```

You will have:

1. Made the blog directory for your project (might be a good idea to put this under source control)
2. Asked express to generate an application using the jade template engine and the stylus css engine (jade is the default html template engine)
3. Asked npm to download and locally install any dependencies required by this express application.

If you were to now take simple-express.js an put it into the blog folder as 'app.js' you should be able to execute it.

```
node app.js
```

When you browse to localhost:3000 you should see that old favourite 'Hello World!'. This file is the starting point of the blogging application and we shall build on it now :)

## A chapter in which we build on our humble

## beginnings

Now that we have a fully working web server we should probably look at doing something with it. In this section we will learn how to use jade to render our data and create forms to post the data back to the server, initially we will store this in memory.

The layout of express applications is fairly familiar and is usually of the form:

```
express                 /* The top level folder containing our app  */
|-- app.js              /* The application code itself              */
|-- lib                 /* Third-party dependencies                 */
|-- public              /* Publicly accessible resources            */
|   |-- images
|   `-- javascripts
|   `-- stylesheets
`-- views               /* The templates for the 'views'            */
```

## Of providers and data

Because the intention of this article is to show how one might use a persistent approach in node.js we shall start with an abstraction: provider. These 'providers' are going to responsible for returning and updating the data. Initially we'll create a dummy in-memory version just to bootstrap us up and running, but then we'll move over to using a real persistence layer without changing the calling code.

articleprovider-memory.js

```
var articleCounter = 1;

ArticleProvider = function(){};
ArticleProvider.prototype.dummyData = [];

ArticleProvider.prototype.findAll = function(callback) {
  callback( null, this.dummyData )
};

ArticleProvider.prototype.findById = function(id, callback) {
  var result = null;
  for(var i =0;i<this.dummyData.length;i++) {
    if( this.dummyData[i]._id == id ) {
      result = this.dummyData[i];
      break;
    }
  }
  callback(null, result);
```

```
};

ArticleProvider.prototype.save = function(articles, callback) {
  var article = null;

  if( typeof(articles.length)=="undefined")
    articles = [articles];

  for( var i =0;i< articles.length;i++ ) {
    article = articles[i];
    article._id = articleCounter++;
    article.created_at = new Date();

    if( article.comments === undefined )
      article.comments = [];

    for(var j =0;j< article.comments.length; j++) {
      article.comments[j].created_at = new Date();
    }
    this.dummyData[this.dummyData.length]= article;
  }
  callback(null, articles);
};

/* Lets bootstrap with dummy data */
new ArticleProvider().save([
  {title: 'Post one', body: 'Body one', comments:[{author:'Bob', comment:
  {title: 'Post two', body: 'Body two'},
  {title: 'Post three', body: 'Body three'}
], function(error, articles){});

exports.ArticleProvider = ArticleProvider;
```

If the above code is saved to a file named `articleprovider-memory.js` in the same folder as the `app.js` we created earlier and `app.js` is modified to look as follows:

app-simple.js

```
var express = require('express');
var ArticleProvider = require('./articleprovider-memory').ArticleProvid

var app = module.exports = express.createServer();

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
```

```
  app.use(express.methodOverride());
  app.use(require('stylus').middleware({ src: __dirname + '/public' }));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});

var articleProvider= new ArticleProvider();

app.get('/', function(req, res){
  articleProvider.findAll(function(error, docs){
      res.send(docs);
  });
})

app.listen(3000);
```

If the app is re-run and you browse to localhost:3000 you will see the object structure of 3 blog posts that the memory provider starts off with, magic!

## A view to a kill

Now we have a way of reading and storing data (patience, memory is only the beginning!) we'll want a way of displaying and creating the data properly. Initially we'll start by just providing an index view of all the blog articles. Fortunately the `express` command we executed earlier to bootstrap our blogging application has already constructed a layout and an index page for us to use. The layout file is Ok as it is, but we need to adjust the index page to look as follows: (be very careful about the indentation, that first lines should be up against the left-hand margin!):

views/index.jade

```
h1= title
#articles
    - each article in articles
      div.article
        div.created_at= article.created_at
        div.title
            a(href="/blog/"+article._id)!= article.title
        div.body= article.body
```

Next change your `get('/')` routing rule in your `app.js` to be as follows:

2/app.js#root

```
app.get('/', function(req, res){
    articleProvider.findAll( function(error,docs){
        res.render('index.jade', { locals: {
            title: 'Blog',
            articles:docs
            }
        });
    })
});
```

Now you should be able to restart the server and browser to localhost:3000. Et voila! We'll not win any design awards, but you should now see a list of 3 very 'functional' blog postings (don't worry we'll come back to the style in a moment).

There are two important things to note that we've just done;

The first is the change to our application's routing rules. What we've done is say that for any browser requests that come in to the route ('/') we should ask the data provider for all the articles it knows about (a future improvement might be 'the most recent 10 posts etc.') and to 'render' those returned articles using the jade template `index.jade`.

The second is the usage of a 'layout' jade file `layout.jade`. This file will be used whenever a call to 'render' is made (unless over-ridden in that particular call) and provides a simple mechanism for common style across all page requests.

> If you're familiar with jade then you may want to skip this section, otherwise please read-on! jade is yet-another templating language, however this one is driven by the rule that 'Markup should be beautiful'. It provides a lightweight syntax for declaring markup with a bare minimum of typed characters.
>
> Reading a jade template is simple. The hierarchy of elements is expressed as indentation on the left hand-side; that is, everything that starts in a given column shares the same parent. Each line of jade represents either a new element in the (eventual) HTML document or a function within jade (which offers conditions and loops etc). Effectively jade takes a JSON object and binds it to any `literal` text in the jade template, applies the rules that define jade and then processes the resulting bag of stuff to produce a well-formed and valid HTML document of the specified DOCTYPE. (Yay!)

As is probably obvious we need a little styling to be applied here, fortunately we can see that the default layout requests a stylesheet already:

views/layout.jade

```
!!!
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body!= body
```

Not only that, but the default stylesheet has already been populated for us in `public/styesheets/style.styl', however sadly the contents don't fully meet our requirements so lets change that file to look more like this:

public/stylesheets/style.styl

```
body
  font-family "Helvetica Neue", "Lucida Grande", "Arial"
  font-size 13px
  text-align center
  text-stroke 1px rgba(255, 255, 255, 0.1)
  color #555
h1, h2
  margin 0
  font-size 22px
  color #343434
h1
  text-shadow 1px 2px 2px #ddd
  font-size 60px
#articles
  text-align left
  margin-left auto
  margin-right auto
  width 320px
  .article
    margin 20px
    .created_at
        display none
    .title
        font-weight bold
        text-decoration underline
        background-color #eee
    .body
        background-color #ffa
```

Again after restarting your app and browsing to localhost:3000 you should see the posts, with a little more style (admittedly not much more!).

Something to notice here:

> Stylus is to CSS as jade is to HTML. However reading stylus can be a little more complex as the hierarchy that is being described is really individual selectors. Lines that start at the same column are rules. Rules are applied to the hierarchy that they're found under, for example in the above stylus example, the bottom most line of stylus `background-color #ffa` is equivalent to the CSS `#articles .article .body {background-color: #ffa;}` this equivalence is due to the position of the start of this line relative to its parent lines :) (Easy really!)

## Great, so how do I make my first post?

Now we can view a list of blog posts it would be nice to have a simple form for making new posts and being re-directed back to the new list. To achieve this we'll need a new view (to let us create a post) and two new routes (one to accept the post data, the other to return the form).

views/blog_new.jade

```
h1= title
form( method="post")
    div
        div
            span Title :
            input(type="text", name="title", id="editArticleTitle")
        div
            span Body :
            textarea( name="body", rows=20, id="editArticleBody")
        div#editArticleSubmit
            input(type="submit", value="Send")
```

Add two new routes to app.js

2/app.js#blog

```
app.get('/blog/new', function(req, res) {
    res.render('blog_new.jade', { locals: {
        title: 'New Post'
    }
    });
});

app.post('/blog/new', function(req, res){
    articleProvider.save({
        title: req.param('title'),
        body: req.param('body')
    }, function( error, docs) {
        res.redirect('/')
```

```
    });
});
```

Upon restarting your app if you browse to new post you will be able to create new blog articles, awesome! Looking at the post route we can see that upon successfully saving we redirect back to the index page where all the articles are displayed.

If I've lost you along the way you can get a zip of this fully working (but non-persisting) blog here: Checkpoint 1. (please be aware that upon extracting the zip file you will need to re-perform the `npm install -d` command within the folder to install the project dependencies.)

## Adding permanent persistence to the mix

I promised that by the end of this article we'd be persisting our data across restarts of node, I've not yet delivered on this promise but now I will ..hopefully ;)

To do this we need to install a dependency on node-mongodb-native, which will allow our burgeoning application to access mongoDB. Once again our friend npm comes to the rescue. The dependencies of node applications can be expressed with a small JSON file in the root of the application `package.json` the `npm` tool understands how to read this file and install the dependencies on our behalf! (Yay! Go Tools!)

To add a dependency on mongoDB you need to make your `package.json` look as follows:

package.json

```
{
    "name": "application-name"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.4.3"
    , "stylus": ">= 0.0.1"
    , "jade": ">= 0.0.1"
    , "mongodb": ">= 0.9.6-7"
  }
}
```

Once you have saved the file you then just need to execute the following command to have `npm` talk to the internet-tubes, download and then install the nodejs client for mongoDB.

```
npm install -d
```

Now we need to replace our old memory based data provider with one thats capable of using mongodb:

articleprovider-mongodb.js

```
var Db = require('mongodb').Db;
var Connection = require('mongodb').Connection;
var Server = require('mongodb').Server;
var BSON = require('mongodb').BSON;
var ObjectID = require('mongodb').ObjectID;

ArticleProvider = function(host, port) {
  this.db= new Db('node-mongo-blog', new Server(host, port, {auto_reconn
  this.db.open(function(){});
};


ArticleProvider.prototype.getCollection= function(callback) {
  this.db.collection('articles', function(error, article_collection) {
    if( error ) callback(error);
    else callback(null, article_collection);
  });
};

ArticleProvider.prototype.findAll = function(callback) {
    this.getCollection(function(error, article_collection) {
      if( error ) callback(error)
      else {
        article_collection.find().toArray(function(error, results) {
          if( error ) callback(error)
          else callback(null, results)
        });
      }
    });
};


ArticleProvider.prototype.findById = function(id, callback) {
    this.getCollection(function(error, article_collection) {
      if( error ) callback(error)
      else {
        article_collection.findOne({_id: article_collection.db.bson_seria
          if( error ) callback(error)
          else callback(null, result)
        });
      }
```

```
    });
};

ArticleProvider.prototype.save = function(articles, callback) {
    this.getCollection(function(error, article_collection) {
      if( error ) callback(error)
      else {
        if( typeof(articles.length)=="undefined")
          articles = [articles];

        for( var i =0;i< articles.length;i++ ) {
          article = articles[i];
          article.created_at = new Date();
          if( article.comments === undefined ) article.comments = [];
          for(var j =0;j< article.comments.length; j++) {
            article.comments[j].created_at = new Date();
          }
        }

        article_collection.insert(articles, function() {
          callback(null, articles);
        });
      }
    });
};

exports.ArticleProvider = ArticleProvider;
```

We will also require a minor change to `app.js` to use this new replacement provider.

app.js

```
/**
 * Module dependencies.
 */

var express = require('express');
var ArticleProvider = require('./articleprovider-mongodb').ArticleProvi


var app = module.exports = express.createServer();

// Configuration

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
```

```
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(require('stylus').middleware({ src: __dirname + '/public' }));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});

var articleProvider = new ArticleProvider('localhost', 27017);
// Routes

app.get('/', function(req, res){
    articleProvider.findAll( function(error,docs){
        res.render('index.jade', {
            locals: {
                title: 'Blog',
                articles:docs
            }
        });
    })
});

app.get('/blog/new', function(req, res) {
    res.render('blog_new.jade', { locals: {
        title: 'New Post'
    }
    });
});

app.post('/blog/new', function(req, res){
    articleProvider.save({
        title: req.param('title'),
        body: req.param('body')
    }, function( error, docs) {
        res.redirect('/')
    });
});

app.get('/blog/:id', function(req, res) {
    articleProvider.findById(req.params.id, function(error, article) {
```

```
        res.render('blog_show.jade',
        { locals: {
            title: article.title,
            article:article
        }
        });
    });
});

app.post('/blog/addComment', function(req, res) {
    articleProvider.addCommentToArticle(req.param('_id'), {
        person: req.param('person'),
        comment: req.param('comment'),
        created_at: new Date()
       } , function( error, docs) {
           res.redirect('/blog/' + req.param('_id'))
       });
});

app.listen(3000);
console.log("Express server listening on port %d in %s mode", app.address
```

As you can see we had to make only the smallest of changes to move away from a temporary in-memory JSON store to the fully persistent and highly scalable mongoDB store.

Let us pause for a second to take a look at two of the methods we've just written to access mongoDB, it is perhaps not immediately obvious what is happening as there are a *lot* of different things going on:

```
ArticleProvider.prototype.getCollection= function(callback) {
  this.db.collection('articles', function(error, article_collection) {
    if( error ) callback(error);
    else callback(null, article_collection);
  });
};
```

1. Declares the `getCollection` method on the provider's `prototype`. This method only accepts one mandatory argument, a function that will be called back with the the results upon completion (or error in the case of an error.) This approach is a common idiom in node but can be confusing to look at initially.
2. In mongoDB there are no tables as such, (hence schema-less) but there are `collections`. A `collection` is a logical grouping of similar documents, but there are very few constraints on what types of document is put in these `collections`. For our purpose we will have a single `collection` called

`articles`. By calling `collection` on the `db` object and passing in our collection name `articles` and a callback to deal with the response mongoDb will quietly create the collection from scratch and return it if there wasn't a collection of that name already or it will just return a reference to an existing collection. (This behaviour can actually be controlled by configuring mongoDB to be `strict`.)
3. If an error is passed back then we need to propagate it back to the previously passed callback.
4. Otherwise pass the collection that came from mongoDB back to the previously passed callback.

and

```
ArticleProvider.prototype.findById = function(id, callback) {
    this.getCollection(function(error, article_collection) {
      if( error ) callback(error)
      else {
        article_collection.findOne({_id: article_collection.db.bson_seria
          if( error ) callback(error)
          else callback(null, result)
        });
      }
    });
};
```

1. Declares the `findById` method on the provider's `prototype`. This method is going to take in one argument the `id` of the article we wish to retrieve and a callback that will receive the data.
2. We call the previously defined method `getCollection` to retrieve our collection of records from the mongoDB server, this method is asynchronous so we have to pass in the callback that will be called when it completes.
3. If there was an error we give up here and pass it back to the callback.
4. An `else` keyword, if I need to explain this I'm very impressed you've made it this far before falling asleep :)
5. The `article_collection` that is passed in the callback from the previous call to `getCollection(...)` (think `FROM clause`) exposes various methods for manipulating the data stored inside of the database. Here we've chosen to use `findOne` which when given some criteria to search on will return the sole record that matches those criteria, but there are others such as `find` which returns a `cursor` that can be iterated over etc. 5a. This line also contains the `specification` argument (think `criteria` or `WHERE clause`) used by the `findOne` method, here we're using the passed in `id` (which is a hexadecimal string ultimately coming from the browser so needs to be converted to the real type that our `_id` fields are being stored as.) It basically states 'Find me the document in the collection who has a property named `_id` and a value

equivalent to an `ObjectId` constructed with the passed in hexadecimal string.
6. If there was an error we give up here and pass it back to the callback.
7. Now we have the record we searched for in the database we pass it back to the callback we originally passed into the (in our case this callback would do the page rendering.)
8. ,9,10 & 11. Meh! some brackets and stuff :)

I hope this explains a little better what is now going on inside our new provider code.

## Adding comments

We're about halfway through the set of (4) operations we defined earlier but you'll be pleased to know that we've completed the majority of the work, everything from here on in is just minor improvements :)

Just to re-cap over we've done so far:

- Create a new article.
- Show the list of all the articles.

and we still need to do:

- Show an individual article and its comments.
- Comment on an article.

So, lets crack on!

## Showing an individual article and its comments

Displaying an individual article isn't much different to displaying one of the articles within the list of articles that we've already done, so we'll pinch some of that template. In addition to displaying the title and body though we will also want to render all the existing comments and provide a form for readers to add their own comment.

We'll also need a new route to allow the article to be referenced by a URL and we'll need to tweak the rendered list so our titles on the list can now be hyperlinks to the real article's own page.

> One thing that we should touch on here is surrogate vs natural keys. It seems that with document orientated databases it is encouraged where possible to use natural keys however in this case we've not got any sensible one to use (*unless you fancy title to be unique enough you crazy fool*.)

> Normally this wouldn't be that much of an issue as surrogate keys are usually fairly sane things like auto-incremented integers, unfortunately the *default* primary key provider that we're using generates universally unique (and universally opaque) binary objects / large numbers in byte arrays. These 'numbers' don't really translate well into HTML so we need to use some utility methods on the `ObjectId` class to translate to and from a hex-string into the id that can located on the database.

We need to update the index page's view:

views/index-final.jade

```
h1= title
#articles
    - each article in articles
      div.article
        div.created_at= article.created_at
        div.title
            a(href="/blog/"+article._id.toHexString())!= article.title
        div.body= article.body
```

The page that shows a single blog entry:

views/blog_show-final.jade

```
h1= title
div.article
    div.created_at= article.created_at
    div.title= article.title
    div.body= article.body
    - each comment in article.comments
      div.comment
        div.person= comment.person
        div.comment= comment.comment
    div
      form( method="post", action="/blog/addComment")
        input( type="hidden", name="_id", value=article._id.toHexString()
        div
          span Author :
          input( type="text", name="person", id="addCommentPerson")
        div
          span Comment :
          textarea( name="comment", rows=5, id="addCommentComment")
        div#editArticleSubmit
          input(type="submit", value="Send")
```

The stylesheet that renders these pages:

public/stylesheets/style-final.styl

```
body
```

```
  font-family "Helvetica Neue", "Lucida Grande", "Arial"
  font-size 13px
  text-align center
  text-stroke 1px rgba(255, 255, 255, 0.1)
  color #555
h1, h2
  margin 0
  font-size 22px
  color #343434
h1
  text-shadow 1px 2px 2px #ddd
  font-size 60px
#articles
  text-align left
  margin-left auto
  margin-right auto
  width 320px
  .article
    margin 20px
    .created_at
        display none
    .title
        font-weight bold
        text-decoration underline
        background-color #eee
    .body
        background-color #ffa
.article
  .created_at
    display none
  input[type =text]
    width 490px
    margin-left 16px
  input[type =button], input[type =submit]
    text-align left
    margin-left 440px
  textarea
    width 490px
    height 90px
```

We also need to add a new rule to `app.js` for serving these view requests:

app-final.js#getBlogs

```
app.get('/blog/:id', function(req, res) {
    articleProvider.findById(req.params.id, function(error, article) {
        res.render('blog_show-final.jade',
        { locals: {
```

```
            title: article.title,
            article:article
        }
        });
    });
});
```

Now if you browse to localhost:3000 the previous articles you added (click new post to create a new one if you have none) should now be visible (apologies for the lack of style once again!) The titles of these articles are now hyperlinks to individual pages that display the article in all itself original glory, comments and all (but alas no comments have been added so far.)

## Comment on an article

Commenting on an article is a simple extension upon everything we've already gone through, the only minor complexity is in the style of 'update' we use on the back-end.

Transactions are largely non-existent in mongoDB but there are several approaches to achieving atomicity in certain scenarios. For comment addition we're going to use a `$push` update that allows us to add an element to the end of an array property of an existing document atomically (which is absolutely perfect for our needs!)

All the views/stylesheet changes we need were made in the last set of changes but we need to add in a new route to handle the POST:

app-final.js#addComment

```
app.post('/blog/addComment', function(req, res) {
    articleProvider.addCommentToArticle(req.param('_id'), {
        person: req.param('person'),
        comment: req.param('comment'),
        created_at: new Date()
       } , function( error, docs) {
           res.redirect('/blog/' + req.param('_id'))
       });
});


app.listen(3000);
console.log("Express server listening on port %d in %s mode", app.address
```

and a method to our provider to make the change on the persistent store:

articleprovider-mongodb-final.js#addCommentToArticle

```
ArticleProvider.prototype.addCommentToArticle = function(articleId, comm
```

```
  this.getCollection(function(error, article_collection) {
    if( error ) callback( error );
    else {
      article_collection.update(
        {_id: article_collection.db.bson_serializer.ObjectID.createFromHe
        {"$push": {comments: comment}},
        function(error, article){
          if( error ) callback(error);
          else callback(null, article)
        });
    }
  });
};
```

After restarting and browsing to a blog article (any one will do) you should now be able to add comments to your articles ad-infinitum. How easy was that?

If you've made it to this point without any issues then congratulations! Otherwise this Checkpoint 2 archive should contain all the code as I have it now!

## Where next

Clearly this blogging application is very rough and ready (there is no style to speak of for starters) but there are several clear directions that it could take, depending on feedback I'll either leave these as exercises for the reader or provide additional tutorials over time:

- Markup language support (HTML, Markdown etc. in the posts and comments)
- Security, authentication etc.
- An administrative interface
- Multiple blog-support.
- Decent styling <g> (inc. themes)

I hope this helps at least someone out there get to grips with how you might start actually writing web apps with node, express and mongoDB.

Good luck! :)

**Fin.**

View the discussion thread.

snippets used in the examples are in the public domain.
Wheat v2 running on node v0.11.13