# What is a Node Web App

...

Dr Ruairi O'Reilly

# The focus of this lecture

We'll learn more about:

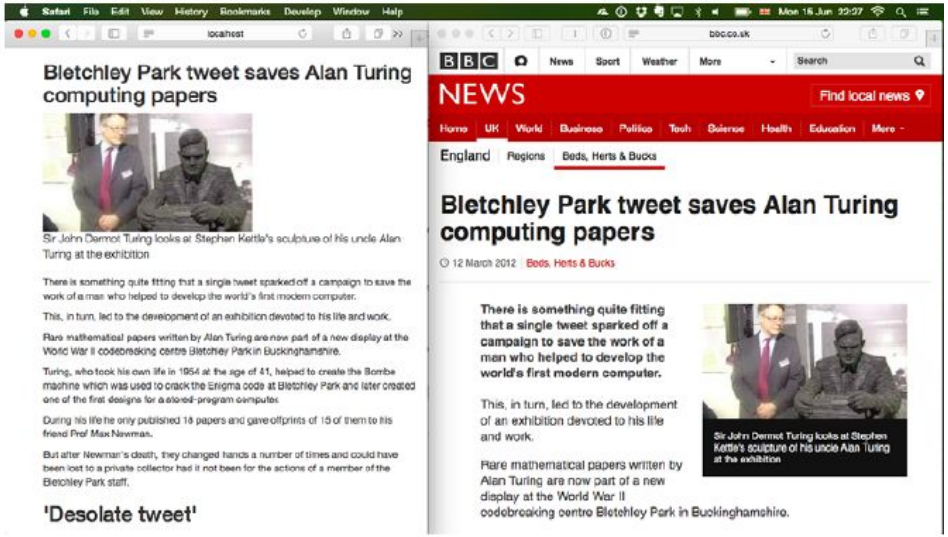
- Creating a new web application
- Building RESTful services
- Persisting data
- Working with templates

# *later*

You're going to build a web application called later that's inspired by popular read-it-later websites such as Instapaper (www.instapaper.com) and Pocket (getpocket.com). This involves starting a new Node project, managing dependencies, creating a RESTful API, saving data to a database, and making an interface with templates.

That might sound like a lot, but you'll explore each of the ideas in this chapter again in subsequent chapters.

The read-it-later page on the left has stripped away all of the navigation from the target website, preserving the main content and title. More significantly, the article is permanently saved to a database, which means you can read it at a future date when the original article may no longer be retrievable.



# Understanding a Node web application's structure

A typical Node web application has the following components:

- Package.json — list of dependencies, and the command that runs the application
- public/ — A folder of static assets, such as CSS and client-side JavaScript
- node_modules/ — The place where the project's dependencies get installed.

The application code is often further subdivided as follows:

- app.js or index.js — The code that sets up the application
- models/ — Database models
- views/ — The templates that are used to render the pages in the application
- controllers/ or routes/ — HTTP request handlers
- middleware/ — Middleware components

# Starting a new web app

To create a new web app, you need to make a new Node project. Refer to L4 if you want to refresh your memory, but to recap, you need to create a directory and run npm init with defaults:

*mkdir later*
*cd later*
*npm init -fy*

Now you have a fresh project; what's next? Most people would add a module from npm that makes web development easier. Node has a built-in http module that has a server, but it's easier to use something that reduces the boilerplate required for command web development tasks. Let's see how to install Express.

# ADDING A DEPENDENCY

To add a dependency to a project, use npm install. The following command installs Express:

*npm install --save express*

Now if you look at package.json, you should see that Express has been added. The following snippet shows the section in question:

```
"dependencies": {
    "express": "^4.14.0"
}
```

The Express module is in the project's node_modules/ folder. If you wanted to uninstall Express from the project, you would run npm rm express --save. This removes it from node_modules/ and updates the package.json file.

# A simple server Express

aSimpleServerExpress.js

```
1  const express = require('express');
2  const app = express();
3  const port = process.env.PORT || 3000;
4  app.get('/', (req, res) => {
5      res.send('Hello World');
6  });
7  app.listen(port, () => {
8      console.log(`Express web app available at localhost: ${port}`);
9  });
```

# NPM SCRIPTS

To save your server start command (node index.js) as an npm script, open package.json and add a new property under scripts called start:

```
"scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
},
```

npm install -g nodemon
Use “nodemon” instead of “node”

Now you can run your application by typing npm start. If you see an error because port 3000 is already being used on your machine, you can use a different port by running PORT=3001 npm start. People use npm scripts for all kinds of things: building client-side bundles, running tests, and generating documentation. You can put anything you like in there; it’s basically a mini-script invocation tool.

# Comparing other platforms - PHP equivalent

```
<?php echo '<p>Hello World</p>'; ?>
```

It fits on one line and is easy to understand, so what benefits does the more complex Node example have? The difference is in terms of programming paradigm: with PHP, your application is a page; in Node, it's a server.

The Node example has complete control over the request and response, so you can do all kinds of things without configuring a server. If you want to use HTTP compression or URL redirection, you can implement these features as part of your application logic.

You don't need to separate HTTP and application logic; they become part of your application.

# Building a RESTful web service

Your application will be a RESTful web service that allows articles to be created and saved in a similar way to Instapaper or Pocket. It'll use a module that was inspired by the original Readability service (www.readability.com) to turn messy web pages into elegant articles that you can read later.

When designing a RESTful service, you need to think about which operations you need and map them to routes in Express. In this case, you need to be able to save articles, fetch them so they can be read, fetch a list of all of them, and delete articles you no longer want.

# That maps to these routes

- POST /articles—Create a new article
- GET /articles/:id—Get a single article
- GET /articles—Get all articles
- DELETE /articles/:id—Delete an article

Before getting into issues such as databases and web interfaces, let's focus on creating RESTful resources with Express. You can use cURL to make requests to a sample application to get the hang of it, and then move on to more complicated operations such as storing data to make it like a real web application.

# RESTful routes example

(i) Gets all articles

(ii) Create an article

(iii) Gets a single article

(iv) Deletes an article

The following listing is a simple Express app that implements these routes by using a JavaScript array to store the articles.

Listing 3.1

```
const express = require('express');
const app = express();
const articles = [{
    title: 'Example'
}];
app.set('port', process.env.PORT || 3000);
app.get('/articles', (req, res, next) => {
    res.send(articles);
});
app.post('/articles', (req, res, next) => {
    res.send('OK');
});
app.get('/articles/:id', (req, res, next) => {
    const id = req.params.id;
    console.log('Fetching:', id);
    res.send(articles[id]);
});
app.delete('/articles/:id', (req, res, next) => {
    const id = req.params.id;
    console.log('Deleting:', id);
    delete articles[id];
    res.send({
        message: 'Deleted'
    });
});
app.listen(app.get('port'), () => {
    console.log('App started on port', app.get('port'));
});
module.exports = app;
```

# RESTful routes example

The listing has a built-in array of sample data that's used to respond with JSON for all articles (i) by using the Express *res.send* method. Express will automatically convert the array to a valid JSON response, so it's perfect for making quick REST APIs. This example can also respond with a single article by using the same principle (iii). You can even delete an article (iv) by using the standard JavaScript *delete* keyword and a numerical ID specified in the URL. You can get values from the *URL* by putting them in the route string *(/articles/:id)* and then getting the value with *req.params.id.*

The Listing can't create articles (ii), because for that it needs a request body parser; you'll look at this in the next section. First, let's look at how you can use this example with cURL (http://curl.haxx.se). curl http://localhost:3000/articles/0

# But why did we say you couldn't create articles?

The main reason is that implementing a POST request requires body parsing. Express used to come with a built-in body parser, but there are so many ways to implement it that the developers opted to make it a separate dependency.

A body parser knows how to accept MIME-encoded (Multipurpose Internet Mail Extensions) POST request bodies and turn them into data you can use in your code. Usually, you get JSON data that's easy to work with. Whenever you've submitted a form on a website, a body parser has been involved somewhere in the server-side software.

To add the officially supported body parser, run the following npm command:
*npm install --save body-parser*

Listing 3.2

```
10  app.set('port', process.env.PORT || 3000);
11
12  app.use(bodyParser.json());
13  app.use(bodyParser.urlencoded({
14      extended: true
15  }));
16  app.post('/articles', (req, res, next) => {
```

(i) Supports request bodies encoded as JSON

(ii) Supports form-encoded bodies

CODE OMITTED See next slide

This adds two useful features: JSON body parsing (i) and form-encoded bodies (ii). It also adds a basic implementation for creating articles: if you make a POST request with a field called title, a new article will be added to the articles array! Here's the cURL Command:

curl --data "title=Example 2" http://localhost:3000/articles

Now you're not too far away from building a real web application. You need just two more things: a way to save data permanently in a database, and a way to generate the readable version of articles found on the web.

Listing 3.2

```
13  app.use(bodyParser.json());
14  app.use(bodyParser.urlencoded({
        extended: true
    }));
17
18  app.post('/articles', (req, res, next) => {
19      const article = {
20          title: req.body.title
21      };
22      articles.push(article);
23      res.send(article);
24  });
```

(i) Supports request bodies encoded as JSON

(ii) Supports form-encoded bodies

# Adding a database

There's no predefined way to add a database to a Node application, but the process usually involves the following steps:

1) Decide on the database you want to use.
2) Look at the popular modules on npm that implement a driver or object relational mapping (ORM).
3) Add the module to your project with npm --save.
4) Create models that wrap database access with a JavaScript API.
5) Add the models to your Express routes.

# Before adding a database

let's continue focusing on Express by designing the route handling code from step 5. The HTTP route handlers in the Express part of the application will make simple calls to the database models. Here's an example:

```
app.get('/articles', (req, res, err) => {
        Article.all(err, articles) => {
                if (err) return next(err);
                res.send(articles); }); });
```

Here the HTTP route is for getting all articles, so the model method could be something like Article.all. This will vary depending on your database API; typical examples are Article.find({}, cb), and Article.fetchAll().then(cb). Note that in these examples, cb is an abbreviation of callback.

# Making your own model API

Articles should be created, retrieved, and deleted. Therefore, you need the following methods for an Article model class:

- Article.all(cb)—Return all articles.
- Article.find(id, cb)—Given an ID, find the corresponding article.
- Article.create({ title, content }, cb)—Create an article with a title and content.
- Article.delete(id, cb)—Delete an article by ID.

You can implement all of this with the sqlite3 module. This module allows you to fetch multiple rows of results with db.all, and single rows with db.get. First you need a database connection.

Listing 3.3 - “db.js”

- An object is created called Article that can create, fetch, and delete data by using standard SQL and the sqlite3 module.
- First, a database file is opened by using *sqlite3.Database* (i), and then an articles table is created (ii). The IF NOT EXISTS SQL syntax is useful here because it means you can rerun the code without accidentally deleting and re-creating the articles table.
- When the database and tables are ready, the application is ready to make queries. To fetch all articles, you use the sqlite3 all method (iii). To fetch a specific article, use the question mark query syntax with a value (iv); the sqlite3 module will insert the ID into the query. Finally, you can insert and delete data by using the run method (v).
- For this example to work, you need to have installed the sqlite3 module with npm install --save sqlite3. (version 3.1.8 )
- Now that the basic database functionality is ready, you need to add it to the HTTP routes. The next listing shows how to add each method except for POST. (You'll deal with that separately, because it needs to use the readability module)

Listing 3.4

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const Article = require('./db').Article;
app.set('port', process.env.PORT || 3000);
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}));
app.get('/articles', (req, res, next) => {
    Article.all((err, articles) => {
        if (err) return next(err);
        res.send(articles);
    });
});
app.get('/articles/:id', (req, res, next) => {
    const id = req.params.id;
    Article.find(id, (err, article) => {
        if (err) return next(err);
        res.send(article);
    });
});
app.delete('/articles/:id', (req, res, next) => {
    const id = req.params.id;
    Article.delete(id, (err) => {
        if (err) return next(err);
        res.send({
            message: 'Deleted'
        });
    });
});
app.listen(app.get('port'), () => {
    console.log('App started on port', app.get('port'));
});
module.exports = app;
```

(i) Loads the db module

(ii) Fetches all articles

(iii) Finds a specific article

(iv) Deletes an article

# Adding the Article model to the HTTP routes

Listing 3.4 is written assuming that you've saved listing 3.3 as db.js in the same directory.  Node will load that module (i) and then use it to fetch each article (ii), find a specific article (iii), and delete an article (iv).

The final thing to do is add support for creating articles. To do this, you need to be able to download articles and process them with the magic readability algorithm. What you need is a module from npm.

# Making articles readable and saving them for later

Now that you've built a RESTful API and data can be persisted to a database, you should add code that converts web pages into simplified "reader view" versions. You won't implement this yourself; instead, you can use a module from npm.

If you search npm for readability, you'll find quite a few modules. Let's try using node-readability (which is at version 1.0.1 at the time of this writing). Install it with npm install node-readability --save.

The module provides an asynchronous function that downloads a URL and turns the HTML into a simplified representation. The following snippet shows how node-readability is used; if you want to try it, add the snippet to index.js in addition to listing 3.5:

# Making articles readable and saving them for later

```
const read = require('node-readability');
const url = 'http://www.manning.com/cantelon2/';
read(url, (err, result)=>  {
  // result has .title and .content
});
```

The node-readability module can be used with your database class to save articles with the `Article.create` method:

```
read(url, (err, result) => {
  Article.create(
    { title: result.title, content: result.content },
    (err, article) => {
      // Article saved to the database
    }
  );
});
```

To use this in the application, open the index.js file and add a new app.post route handler that downloads and saves articles. Combining this with everything you learned about HTTP POST in Express and the body parser gives the example in the following listing.

# Generating readable articles and saving them

```
11  const read = require('node-readability');
12
13  // ... The rest of the index.js from listing 3.4
14
15  app.post('/articles',(req,res,next)=> {
16    const url = req.body.url;
17
18    read(url, (err, result) => {
19      if(err || !result) res.status(500).send('Error downloading article');
20      Article.create(
21        {title: result.title, content: result.content},
22        (err, article) => {
23          if(err) return next (err);
24          res.send(ok);
25        }
26      );
27    });
28  });
```

(i) Gets the URL from the POST body

(ii)Uses the readability mode to fetch the URL

(i) After saving the article, sends back a 200

# Generating readable articles and saving them

Here you first get the URL from the POST body (i) and then use the node-readability module to get the URL (ii). You save the article by using your Article model class. If an error occurs, you pass handling along the Express middleware stack (iii); otherwise, a JSON representation of the article is sent back to the client.
You can make a POST request that will work with this example by using the --data Option:

curl --data "url=http://manning.com/cantelon2/" http://localhost:3000/articles

Over the course of the preceding section, you added a database module, created a JavaScript API that wraps around it, and tied it in to the RESTful HTTP API. That's a lot of work, and it will form the bulk of your efforts as a server-side developer.

# Adding a user interface

Adding an interface to an Express project involves several things. The first is the use of a template engine; we'll show you how to install one and render templates shortly. Your application should also serve static files, such as CSS. Before rendering templates and writing any CSS, you need to know how to make the router handlers from the previous examples respond with both JSON and HTML when necessary.

# Supporting multiple formats

So far you've used *res.send()* to send JavaScript objects back to the client. You used cURL to make requests, and in this case JSON is convenient because it's easy to read in the console. But to really use the application, it needs to support HTML as well. How can you support both? The basic technique is to use the *res.format* method provided by Express. It allows your application to respond with the right format based on the request. To use it, provide a list of formats with functions that respond the desired way:

```
res.format({
  html: () => {
    res.render('articles.ejs', { articles: articles });
  },
  json: () => {
    res.send(articles);
  }
});
```

In this snippet, res.render will render the articles.ejs template in the views folder. But for this to work, you need to install a template engine and create some templates.

# Rendering templates

Many template engines are available, and a simple one that's easy to learn is EJS (Embedded JavaScript). Install the EJS module:  npm install ejs --save

Now res.render can render HTML files formatted with EJS. If you replace res.send (articles) in the app.get('/articles') route handler from listing 3.4, visiting http://localhost:3000/articles in a browser should attempt to render articles.ejs. Next you need to create the articles.ejs template in a views folder. The next listing shows a full template that you can use.

(i) Includes another template

(ii) Loops over each article and renders it

(iii) Includes the article's title as the link text

(i) Includes another template

head.ejs

```
html>
<head>
    <title>Later</title>
</head>
<body>
    <div class="container">
```

articles.ejs

```
<% include head %>
    <ul>
        <% articles.forEach((article) => { %>
            <li>
                <a href="/articles/<%= article.id %>">
                    <%= article.title %>
                </a>
            </li>
            <% }) %>
    </ul>
    <% include foot %>
```

foot.ejs

```
    </div>
</body>
</html>
```

The article list template uses a header (i) and footer template (i) that are included as snippets in the following code examples. This is to avoid duplicating the header and footer in every template. The article list is iterated over (ii) by using a standard JavaScript forEach loop, and then the article IDs and titles are injected into the template by using the EJS <%= value %> syntax (iii).

The res.format method can be used for displaying specific articles as well. This is where things start to get interesting, because for this application to make sense, articles should look clean and easy to read.

# Using npm for client-side dependencies

With the templates in place, the next step is to add some style. Rather than creating a style sheet, it's easier to reuse existing styles, and you can even do this with npm! The popular Bootstrap (http://getbootstrap.com/) client-side framework is available on npm (www.npmjs.com/package/bootstrap), so add it to this project:

```
npm install bootstrap --save
```

If you look at node_modules/bootstrap/, you'll see the source for the Bootstrap project. Then, in the dist/css folder, you'll find the CSS files that come with Bootstrap. To use this in your project, you need to be able to serve static files.

# SERVING STATIC FILES

When you need to send client-side JavaScript, images, and CSS back to the browser, Express has some built-in middleware called express.static. To use it, you point it at a directory that contains static files, and those files will then be available to the Browser. Near the top of the main Express app file (index.js), there are some lines that load the middleware required by the project:

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));:
```

To load Bootstrap's CSS, use express.static to register the file at the right URL:

```
app.use('/css/bootstrap.css',
express.static('node_modules/bootstrap/dist/css/bootstrap.css') );
```

Now you can add /css/bootstrap.css to your templates to get some cool Bootstrap styles. Here's what views/head.ejs should look like:

```
1  html>
2  <head>
3      <title>Later</title>
4      <link rel="stylesheet" href="/css/bootstrap.css">
5  </head>
6  <body>
7      <div class="container">
```

This is only Bootstrap's CSS; Bootstrap also comes with other files, including icons, fonts, and jQuery plugins. You could add more of these files to your project, or use a tool to bundle them all up so loading them is easier.

# DOING MORE WITH NPM AND CLIENT-SIDE DEVELOPMENT

The previous example is a simple use of a library intended for browsers through npm. Web developers typically download Bootstrap's files and then add them to their project
manually, particularly web designers who work on simpler static sites.

But modern front-end developers use npm for both downloading libraries and loading them in client-side JavaScript. With tools such as Browserify (http://browserify.org/) and webpack (http://webpack.github.io/), you get all the power of npm installation and require for loading dependencies. Imagine being able to type const React = require('react') in not just Node code, but code for front-end development as well! This is beyond the scope of this chapter, but it gives you a hint of the power you can unlock by combining techniques from Node programming with front-end development.

# Summary

- You can quickly build a Node web application from scratch with npm init and Express.
- The command to install a dependency is npm install.
- Express allows you to make web applications with RESTful APIs.
- Selecting the right database and database module requires some up-front investigation and depends on your requirements.
- SQLite is handy for small projects.
- EJS is an easy way to render templates in Express.
- Express supports lots of template engines, including Pug and Mustache.

# References

All content taken from Chapter 3 of Node.js in Action 2nd Edition