Edinburgh Napier University

SET08101 Web Tech

Lab 10 - Datastores & Persistence

Dr Simon Wells

1 Aims

At the end of the practical portion of this topic you will be able to:

- Persist data to the filesystem
- Use SQLite3 to persist data using a relational database
- Use mLab to persist data in a cloud version of MongoDB

2 Activities

If we want a website to be dynamic, to let user's interact with it beyond just reading static data, then there is a good chance that we'll want to save some of that data. We have a number of strategies to achieve this. We'll take two main approaches to persisting our data, firstly on the client side, and then on the server side.

For the client side we'll look at ways to store data using the browser; cookies and web-storage

For the Server-side we also have three approaches, firstly using the filesystem, a simple, straightforward, and robust solution. Secondly we'll try a traditional relational database, then lastly we'll look at a cloud-based version of a NoSQL datastore.

Data storage is a huge topic so this lab only really scratches the surface. However, you should, by the end, feel comfortable with persisting data on the server or client, and should be building some skills in deciding which approach to take depending upon the problems that you are presented with.

2.1 Cookies

We can create cookies from JavaScript. It's not the only way, but it is a good method to get started. You can do the following exercises from the Javascript console in your browser, first ensuring that you have cookies enabled. Cookies have a really simple JavaScript API. For the most part just using a call to document.cookie is sufficient to create, retrieve, update, and delete a cookie.

Enter the following into the console, replacing "your name" with your actual name.

```
1    document.cookie = "username=your name";
```

This will add the double-quoted content to the "cookie string" for the current page.

To retrieve our cookie we need to create a variable and then call document.cookie which returns the cookie string. In the next example we will do so then print out the variable using console.log()

```
1    var x = document.cookie;
2    console.log(x);
```

We can also set an expiry date for a cookie, for example:

```
1 document.cookie = "password=secret; expires=Thu, 18 Dec 2019 12:00:00 UTC";
```

After the expiry date, that particular key & value pair will be deleted.

If we repeat the retrieval of our cooke from above, we'll notice that our password key has been appended to hte previous string so both username and password are stored in the same cookie. This is because there is just one cookie string for each page. So to retrieve a particular key we need to pull apart the cookie using JavaScript to retrieve the specific key that we are interested in.

For example, to retrieve just the value for the username we could create a function that searches the cookie string for the supplied cookie_name, e.g.

```
function getCookie(cookie_name)
{
    var name = cookie_name + "=";
    var decodedCookie = decodeURIComponent(document.cookie);
    var ca = decodedCookie.split(';');
    for(var i = 0; i <ca.length; i++)
    {
        var c = ca[i];
        while (c.charAt(0) == ' ')
        {
            c = c.substring(1);
        }
        if (c.indexOf(name) == 0)
        {
            return c.substring(name.length, c.length);
        }
    }
    return "";
}
```

We can then use our function to retrieve a specific value from the cookie, for example, the username:

```
console.log(getCookie("username"));
```

Deleting a cookie is achieved by setting an expiry date that is in the past, for example:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC;";
```

To update a value stored in a cookie you merely replace the existing key with the new kvalue, for example, here we just set a new username with an updated value which will replace the existing value:

```
document.cookie = "username=Carol Danvers;";
```

Your cookies should probably not live forever, it is good practise to set an expiry date which is a reasonable length of time into the future. To make it easier to program, a nice way to do this is to use a function, for example, this one will create a cookie using the supplied key and value and will set an expiry date based upon the supplied value for the number of days.

```
function setCookie(name, value, expiry)
{
    var d = new Date();
    d.setTime(d.getTime() + (expiry*24*60*60*1000));
    var expires = "expires="+ d.toUTCString();
    document.cookie = name + "=" + value + ";" + expires;
}
```

Now create a simple site using HTML, CSS, & JS that uses cookies to persist some data. A good place to start is to create a simple form that asks some questions about the user. stores the responses in a cookie, then uses the data in the cookie to personalise a message to the user. For example, asking a user what their name is, then whether they are happy or sad and respond with something appropriate that is personalised to them.

2.2 Web Storage

This option provides more space for your site to store data on the client. The amount of space is browser specific, and the API is only supported by newer browsers, so you'll have to check first, in code, whether web storage is supported.

The basic idea with web-storage is that you can store keys and values. Both keys and values are always strings, so you may have to do some type conversion for some of your variables. If you have JSON objects, then you can store these by converting them to a string using the JSON.stringify function. Thera are two types of web-storage available, localStorage and sessionStorage. The main difference is that sessionStorage is only available whilst the current window is open and is deleted afterwards, i.e. once your current session is over. However data stored in local storage will be available for ever unless explicitly deleted, for example, by your webapp, or by the user clearing all local data, reformatting their drive, etc.

```
function storageAvailable(type)
{
    try
    {
        var storage = window[type],
            x = '__storage_test__';
        storage.setItem(x, x);
        storage.removeItem(x);
        return true;
    }
    catch(e)
    {
        return e instanceof DOMException && (
            e.code === 22 ||
            e.code === 1014 ||
            e.name === 'QuotaExceededError' ||
            e.name === 'NS_ERROR_DOM_QUOTA_REACHED') &&
            storage.length !== 0;
    }
}
```

In this we attempt to use the storage API and then catch any exceptions that are thrown. Even if storage is available, there are reason why it might not be usable, either due to local settings, such as privacy settings, the way that private browsing works (if you are in a private browsing window then you might have local storage available but the amount of space to store anything in is set to zero), or because you've already filled it up.

We can call our storageAvailable function with the value "localStorage" or "sessionstorage" to see which is available to use.

```
if (storageAvailable('localStorage'))
{
    // We can do something with the localStorage...
}
else
{
    // Have to come up with an alternative way to store data
}
```

We can then actually add data to the storage using the setItem function, for example,

```
localStorage.setItem('name','simon');
```

Checking whether something has been set can be quite useful. We can achieve that like this using the getItem function (we assume that we, as programmers, will know which keys our web pages will have set):

```
if(localStorage.getItem('name')) {console.log("set");}
```

We can also retrieve that data using the getItem function, for example, let's retrieve what we just stored and display it:

```
var myname = localStorage.getItem('name');
console.log(myname);
```

2.3 Filesystem

Whilst JS usually has little support for input and output which is all handled by the browser environment in which the JS runs, Node gives us other opportunities. Saving files to the file system is a useful and simple technique that can lead to robust, reliable sites and good performance. We can use the *fs* Node module. We'll need to install fs using NPM first, e.g.

```
npm install fs
```

Now we can use it from code, for example, put the following code into a file called *filesystem.js*:

```
var http = require("http"), fs = require("fs");

http.createServer(function (request, response) {
  request.on("end", function () {
    fs.readFile("test.txt", 'utf-8', function (error, data) {
      response.writeHead(200, {
        'Content-Type': 'text/plain'
      });
      data = parseInt(data) + 1;
      fs.writeFile('test.txt', data);
      response.end('This page was refreshed ' + data + ' times!');
    });
  });
}).listen(5000);
```

We read and write files by using calls to the fs.readFile() and fs.writeFile() methods respectively. Note that you will need to create the file *test.txt* in the same folder as filesystem.js before running it. When you run the file you should be able to access it at `http://localhost:5000`

2.4 SQLite3

A drawback of the file system as a way to persist your data is that performing *ad hoc* queries of your data is more difficult. Unless you have developed a plan for searching all your files, or have a strategy for finding data across multiple files, it can be difficult to find specific things that you need. Databases generally provide ways to manage searching through your data and can be useful when the size of your data becomes large. However, they also add great complexity to your design compared with a site that merely reads and writes files on the local filesystem. For small amounts of data they can be overkill and you should seldom reach for a database as your solution to data storage without a positive reason why you are doing so.

Rather than installing a large SQL-based datastore manager we will use SQLite3. This is an efficient, embedded datastore, that is available on most platforms, and roundly regarded as a reliable, well-engineered. It also happens to perhaps be the most widely deployed datastore on the planet embedded within many operating systems, email clients, and web browsers as a way to reliably persist data.

Create a folder called counter then navigate into it. We can install SQLite3 using NPM:

```
npm install sqlite3
```

Once the SQLite3 package is installed you can use it from Node. First however you will need to create a database file. Create a file called count.db in counter folder. This will be the file that SQLite3 uses to store data from our counter node app. Note that the node app doesn't create this file for you and will throw an error if it can't find the nominated database file. You can write some setup code if you like that will create this file for you but that is left as an exercise...

```
var path = require('path');
var dbPath = path.resolve(__dirname, 'count.db')

var sqlite3 = require('sqlite3').verbose();
var db = new sqlite3.Database(dbPath);
```

```
6
7  db.serialize(function() {
8      db.run("CREATE TABLE IF NOT EXISTS counts (key TEXT, value INTEGER)");
9      db.run("INSERT INTO counts (key, value) VALUES (?, ?)", "counter", 0);
10 });
11
12
13
14 var express = require('express');
15 var counterapp = express();
16
17 counterapp.get('/data', function(req, res){
18     db.get("SELECT value FROM counts", function(err, row){
19         res.json({ "count" : row.value });
20     });
21 });
22
23 counterapp.post('/data', function(req, res){
24     db.run("UPDATE counts SET value = value + 1 WHERE key = ?", "counter", function
           (err, row){
25         if (err){
26             console.err(err);
27             res.status(500);
28         }
29         else {
30             res.status(202);
31         }
32         res.end();
33     });
34 });
35
36
37 counterapp.listen(5000);
38
39 console.log("Submit GET or POST to http://localhost:5000/data");
```

Use curl to POST to the api:

```
curl -X POST localhost:5000/data
```

Then you can read the count of hits by visiting: `http://localhost:5000/data` in your browser.

This app doesn't do much. It merely stores a small amount of data, basically a count of calls to the API endpoint, then tells you how many times the API has been called. The serialize function initialises a table in our count.db file. The post function demonstrates a simple way to insert data into the table and the get function performs a simple query on the database to retrieve the stored data. Note that the retrieved data in the get function returns JSON. Perhaps try to format this data as HTML instead as a useful exercise?

2.5 MongoDB

A drawback of many traditional relational databases is the need to develop schema, normalise data, and construct tables. As a project develops this can be a straightforward task, however early in a project, it can be difficult to know what needs to be stored, how the data is interelated, and how you will need to store or process it. Prematurely developing a schema could even influence your design away from the optimum. NoSQL and schemaless databases can be useful in the early prototyping stage of a project. They provide a way to have a datastore that is easy to set up, but which doesn't require you to prematurely decide on data structure. You can develop a prototype with a NoSQL datastore early in the project, then, when you have a need to optimise your data storage you can make measurements of performance and scalability then decide whether to swith out the NoSQL datastore for something a little more optimised for your problem (which may well be an SQL datastore such as PostgreSQL). The key is to use a lightweight solution initially, until you know enough about the potential solutions that you can make a good choice for the final stages of the project.

Rather than installing our own version of MongoDB locally[1] we will use an online web-service called mLab[2]. Visit mLab and set up an account. When you log in you should see a control panel similar to the following:

Figure 1

Click the "Create new" button. Now choose the single-node tab uder *Plan* to reveal the free *sandbox* option.

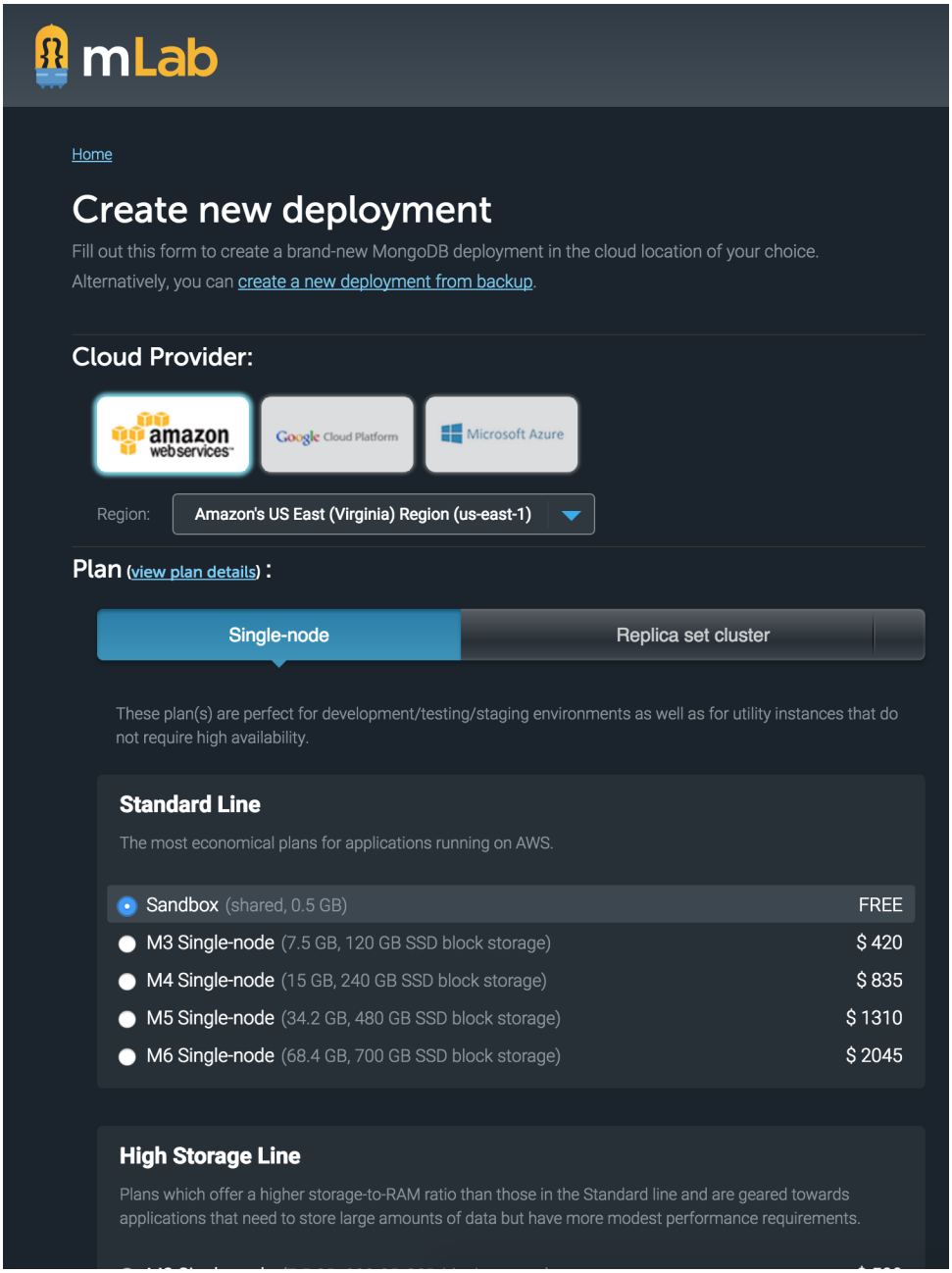

Figure 2

[1] You can install MongoDB if you like but it's a little outside the scope of this lab
[2] `http://mlab.com/`

Give your new database a name then click “Create new MongoDB deployment”

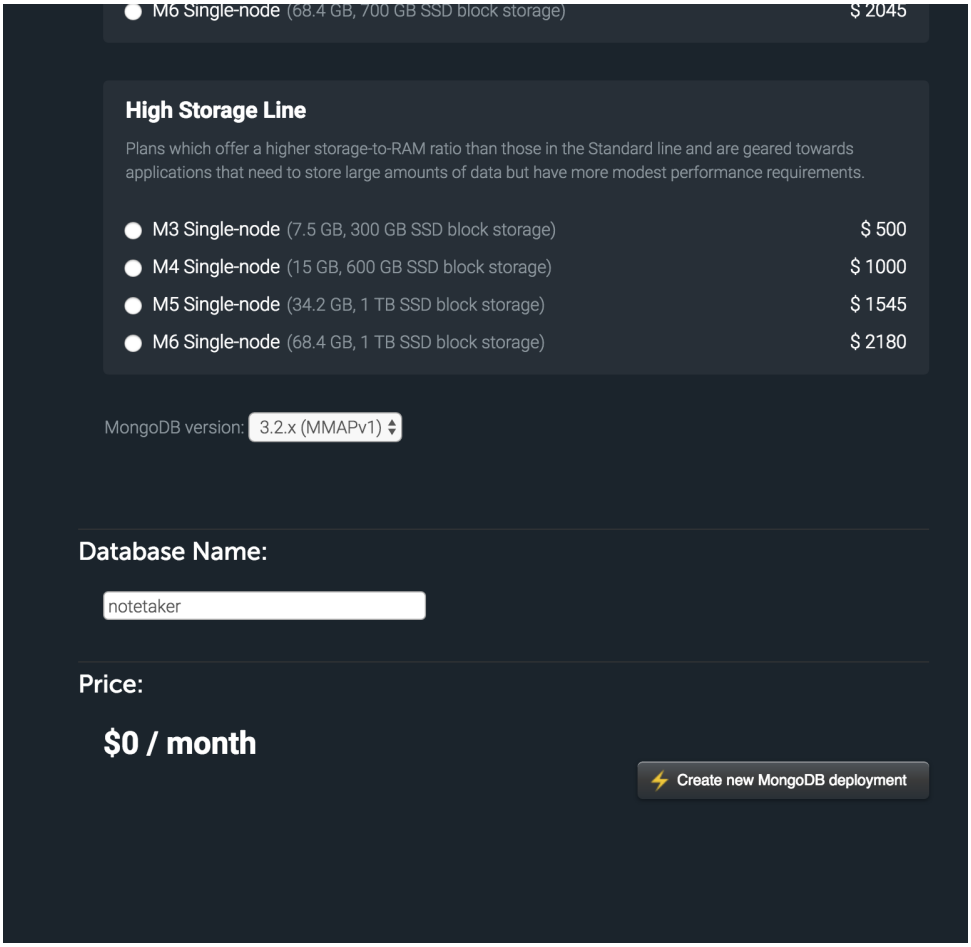

Figure 3

Think of a MongoDB, for now, as a collection of collections. So we need to name a default collection to insert data into. Click the “+ Add collection” button, then type in a name for the collection, e.g. “’notes”.

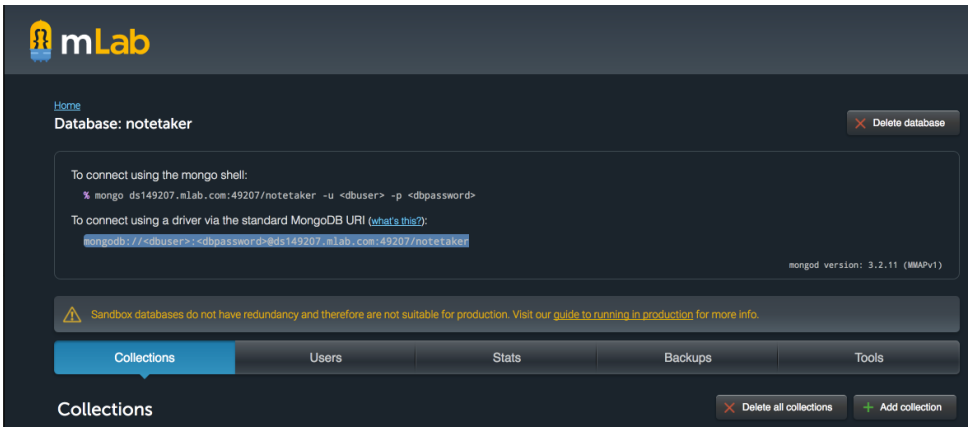

Figure 4

On the database page, mLab gives you connection details for your collection on the database page, e.g.

```
mongodb://¡dbuser¿:¡dbpassword¿@dsxxxxxx.mlab.com:49207/notetaker
```

We still need a username and password however, so we need to create a user by clicking the "'Add database user" button and entering a username and password.

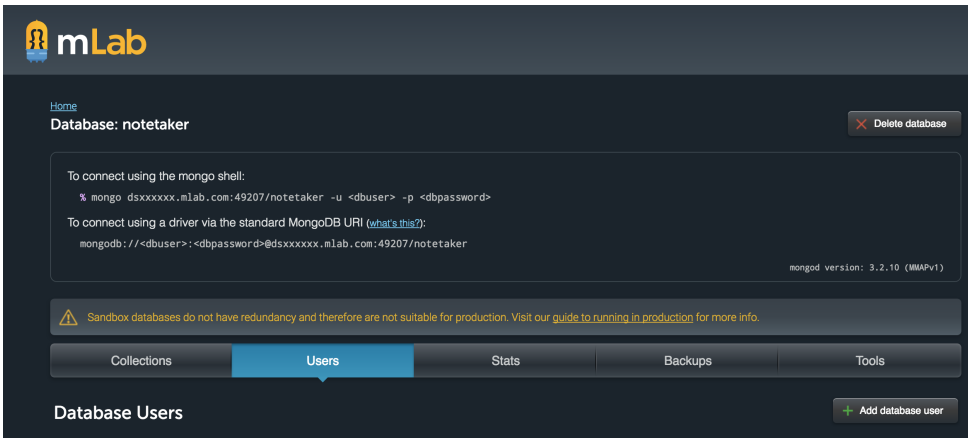

Figure 5

You should now be able to connect to your database from node. First install a MongoDB package using npm, e.g.

```
npm install mongodb
```

We can now access our mLab instance of a Mongo database using JavaScript within a node app as indicated in the following fragment:

```
const MongoClient = require('mongodb').MongoClient;

const MONGO_URL = 'YOUR_URL_HERE';

MongoClient.connect(MONGO_URL, (err, db) => {
  if (err) {
    return console.log(err);
  }

  // Do something with db here, like inserting a record
  db.collection('notes').insertOne(
    {
      title: 'Hello MongoDB',
      text: 'Hopefully this works!'
    },
    function (err, res) {
      if (err) {
        db.close();
        return console.log(err);
      }
      // Success
      db.close();
    }
  )
});
```

Obviously there's a whole lot more that MongoDB and mLab can do, but that is what documentation is for. Perhaps a first exercise might be to reimplement the hit counter app that we used in filesystem and sql persistence to use a Mongo instance instead?

2.6 Challenges

You should consider how to incorporate appropriate data storage into the sites that we've already developed during previous challenges. For example, if you've been working on the dungeon-crawler game then cookies are a really good, light-weight method of storing a small amount of data about the current player. For example, the players health, what they are carrying, which locations they've visited. Obviously, you have a small amount of space to store your data in a cookie, but this should be more than sufficient for simple player specific data. If you are generating a larger dungeon, then

you might want to consider using one of the other types of client-side storage. The nice thing about this approach is that each time your player revisits the site they should be able to pick up at exactly the place that they left off. This is a nice way to persist a game whilst maintaining a very lightweight, casula approach for your player.