SWEN303

My First Node.js App

Introduction

This document will cover making a basic Node.js application themed around a video store. We are going to use and look at the following technologies (documentation linked):

- Node.js
- Express.js
- Jade (Pug) template language
- Bootstrap
- Jade-bootstrap
- BaseX
- XQuery
- XPath

Note that this document is quite long, so will take multiple lectures to cover, will be modified as the course progresses as I try to add content to address issues that students are having, and is full of mistakes (so please tell me so I can fix them).

Getting Started

Required Software

In order to do this we are going to need the following software:

1. A terminal running bash or tcsh. If you are Linux or Mac this is just your usual terminal. If you are on Windows you will require Cygwin. The Cygwin bash shell is necessary to run all the commands that are used in this handout (but not for actually getting your website to work).
2. A web browser
3. Git
4. Node.js (v 5.7.1)
5. An editor to write JavaScript. This can be any text editor. Popular ones I am aware of include:
    - Lighttable
    - Sublime
    - Atom
    - Webstorm (via their student license)

Basic Heads Up About Node.js

Node.js can be run using the command

```
node
```

This will run an interactive shell for Node.js where you can enter commands and see them executed (like the Chrome / Firefox JavaScript Console). You can run a file by giving it the filename.

It is basically JavaScript (but not exactly). It has its own package manager `npm`. You can install a package in a given directory using

```
npm install <Package Name>
```

Or globally on the computer using

```
npm install -g <Package Name>
```

Though the global install won't work on the ECS computers.

Setting Up Express

We aren't going to build everything ourselves. Instead we are going to use the express generator to create the skeleton of our application. The express generator is a Node.js package which we can use to help us.

Installing the Express Generator

If you are doing this on your own laptop you can just run:

```
npm install -g express-generator
```

to install the generator to your computer (the -g option installs the package globally on the computer, rather than just putting it in the current directory).

However, on the ECS systems we aren't allowed to install stuff globally, so instead we need to install things just to the local directory.

So we start by creating a new directory for our project.

```
mkdir movieWebsite
```

Then we go into the directory and install the express generator.

```
cd movieWebsite
npm install express-generator
```

While this installs the software - it doesn't actually put it on the PATH - so we can't use it normally from the command line. We will then run the following command to add the installed programs to the PATH (not needed if you installed it globally).

```
foreach dir (`find . -type d -iname "bin" -exec readlink -f {} \;`)
  setenv PATH "${dir}:${PATH}"
end
```

Note: This is assuming you are running `tcsh` and none of your directories have spaces in the filenames. Tcsh is the default on the ECS computers, but if you are using your own machine, you probably have `bash`. The `bash` equivalent is (still assuming no spaces in filenames):

```
for dir in $(find . -type d -iname "bin" -exec readlink -f {} \;)
do
  export PATH="${dir}:${PATH}"
done
```

Looking at these two commands you might decide to put them into a shell script so that they can be easily run, since their effects only last as long as the tab you typed the code into is open. Note that if you do this and then run it in the usual fashion

```
./myScript.sh
```

it won't work. Instead you need to run it as

```
source myScript.sh
```

Also note that this only makes visible the node libraries that were installed when you ran the command. If you install a new library (or close the tab) you will need to run that command again.

We are now ready to start our first project.

The App Skeleton

In order to create the basic skeleton of our application we go the directory above the one where the want the project to be and run.

```
express --git -f movieWebsite
```

This has created a new default webpage. We now need to go into the directory and install all of the dependencies it now requires. Note that `-f` was required since we were creating it in an existing directory where we installed express generator. If you had it installed globally you wouldn't need the `-f`.

```
cd test
npm install
```

We can now test our website by running

```
bin/www
```

And go to it at the address http://localhost:3000/. There is also a second page at http://localhost:3000/users.

The Web Architecture

Before you can really understanding what is going on with the simple webpage, we first need to go over a few basics of how websites work.

The first thing to keep in mind is that there are two 'places' where stuff has to happen: server and client. I am not going to talk about the server and client as different 'places'. However, you will also notice, that most likely for the assignments in this course, in most cases the server and client will be on the same actual computer.

Server

This is the bit of the computation that happens away from the web browser. Anything can happen up there, but there is a standard way that it always looks. The client will try to access some sort of URL, and they will get back a response (some lump of data that hopefully the client has some way of dealing with).

For example, if I trying to access `www.example.com/index.html` (a fake address), then the server on the other side will send me back an HTML page. This page can have JavaScript and CSS and other things embedded in it (or referenced from it - in which case the browser will need to go and get those too, and everything they reference, until it has everything).

Note that there is notionally a single server side. So all the computation done on the server end is likely to be reliable and reproducable (if for no reason other than you tend to have control over it). Of course the internet itself is not reliable which can cause its own set of problems.

Client

This is essentially what goes on in the web browser. It will generally render HTML and run any JavaScript embedded in the webpage. Note that it is the web browser that is making the decisions here - so each browser can choose to do things in its own way (this can become a very real source of difficulty in web programming).

Division Of Labour

Traditionally it was very obvious what was client and what was server side. Languages that were meant to run in the browser (HTML, JavaScript, etc) would be client side code. While languages that were meant to run on a server (C#, Ruby on Rail, PHP, and friends) were server side. Node.js blurs this line. Now both the server and client sides and can run JavaScript.

This provides the advantage that there is one less language to learn (since you would need to learn JavaScript for web site coding anyway). However, this advantage is a bit of a trap. Yes. It is true that now you can use JavaScript on both sides, so you don't need to learn another language. But you still need to learn a new set of libraries. Often it isn't learning the syntax of a new language that is the problem - it's learning all the new libraries that come with it, and Node.js doesn't do much to address that problem.

Sometimes you may not be sure if something should be done on the server of client side. In general there are a few points that can help decide where it should go.

- Accessing persistent information should be done on the server
- Anything that you need to trust should be done on the server
- Rendering is always done client side
- It is considered in many places healthy to put any work that doesn't require the persistent datastore, or high levels of trust onto the client. This has advantage of taking load of the server, so that it can deal with more load. Note, that this puts load onto the client (which can be slower) and people can get pretty annoyed waiting for stuff to run.

Understanding The Skeleton

With the server and client as context let's look at the files that are in our default application.

Files

At the root level we can see a number of files and directories. Let's start by looking at each file.

.gitignore

This is a file contiaining a list of files for git to ignore when you turn this into a git repository. Ignoring files is a really important part of maintaining a healthy git repository as including certain files may cause problems for other people.

You will probably need to add more things to the gitignore file as you go. However you are extremely unlikely to need to remove anything from it. Exactly what should and should not go into the `.gitignore` file I may try to talk about in future, but for now will ignore.

app.js

This is the main entry point of your application. This file is like in Java:

```
public static void main(String[] args){
    ...
}
```

It contains the Node.js JavaScript code that will control the server. This does not specify any of the client behaviour.

We will look at what is in it in more detail later.

package.json

This is a JSON file that contains details about the project. It contains some information that is used to build the package (particularly what Node.js modules it depends on), but a lot of it is just metadata about the project. Metadata is important, but this course will not look at in any level of detail.

Directories

Now lets consider each directory

bin

This directory contains a short executable script which can be used to start your server. It sets up some connection stuff - like which port to use and the like - and launches your website.

node_modules

This is where `npm install` puts any files that it downloads. This directory should not be committed to git (and the default `.gitignore` will exclude it).

Basically you are not likely to spend much time looking in here. The script which adds installed node modules to the path looks in here to find everything that you've installed.

routes

This is where you specify what different URLs to your domain should do.

Basically the idea here is that every request to your website goes to the same place. However, what it does depends on the later part of the URL. So consider going to your own webpage: `http://localhost:3000/index`.

Then we can break the address up into to parts: where it's going, and what you want to do there.

The where it's going is the first bit: `http://localhost:3000`. This says that this website is hosted on this computer, and that you have to access it via port 3000. Note that sometimes later parts of the URL can be used for this too (we will see an example of this later).

The what to do there part is basically everything that is left over after you've specified where. So in this example it is: `/index`.

The routes directory holds JS files where you specify what you want different 'what to do parts' of the URL to do (or what pages you want them to bring up). This way of looking at the world comes from Express seeing the world from a REST point of view. It doesn't matter what exactly this means - but it is important to realise that this isn't the only way of looking at the world.

public

This is where stuff that doesn't need executing lives. Files which live here can just be given to the client directly. This is things like images, CSS, and (client-side) JavaScript for the webpages.

views

Here live the Jade (Pug) template files. These are the webpages that are given to the clients. However, unlike the files in the `public` directory which are served unmodified, these are first processed by the server, then the resulting HTML page is served to the client.

.git (not there yet)

You won't have one of these yet. This directory is the thing that makes a directory a git repository. It stores all the information that git requires (including the entire history of your project).

Some Final Additions

Before we continue, we will first change the setup of the project a little bit to support all the additional things that we want to do.

1. Make this a git repository

   ```
   git init
   git remote add origin <github url>
   ```

2. Add the other libraries we need to package.json

   - jade-bootstrap
   - cheerio
   - eslint
   - basex

3. Run

   ```
   npm install
   ```

   in order to install the new packages we added.

4. We can now also rerun the script to add the installed files to the PATH, so we have everything we just installed.

Understanding The Example

We can start by looking through a number of key pages in the system.

app.js

Before we can modify much we should first have some idea of what is going on in our main file. While there is a whole bunch of stuff there we are only going to pull out a few key bits and look at those.

At the top of the file you can see a number of lines of the form:

```
var x = require('x')
```

These lines are importing libraries. Well, loading what is in those files and putting it into a variable.

The express library is the library that is in charge of all the routing. The app variable is the function we are using to control the app. You can see this being created in:

```
var app = express();
```

The line:

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

is setting up the public directory to be used to serve files which do not need to be processed by Node.js. Files stored in there can be accessed just by their directory and file name.

So suppose you have your logo stored in `public/images/MyLogo.png`. Then that will be accessible from `http://localhost:3000/images/MyLogo.png`. Note that it does not care what kind of file is there, or what directory it is stored in. The names are just there for your (the programmers) convenience.

The last part which we need to focus on (but not the last lines in the file) are the following.

```
app.use('/', routes);
app.use('/users', users);
```

These lines tell node where to look for the code to deal with URLs going to those locations. If we look earlier in the document we'll notice that both routes and users refer to files in the routes directory (`index.js` and `users.js` respectively).

This means that any URLS that are `http://localhost:3000/users/something` will use the code in the `users.js` file, and they will be given the `something` part of the URL. Anything that doesn't match that but is `http:localhost:3000/something`, will use the code in the `index.js` file and will be passed the `something` part of the URL.

Jade

Jade is a template language for HTML. Basically it's used to describe what HTML should appear, but with a few extra quirks. 1. You don't actually write in HTML. It uses a short hand notation. 1. You can pass in data from the server. 1. You can make decisions about what to draw 1. It supports mixins - which are like macros that insert a whole bunch of Jade in one go (and take arguments).

The basics of Jade are pretty straight forward. You type the name of tag and that will create it. Nesting is done using indentation. Mixins are writting using `+<name>(<args>)`. `.<class>` creates a `div` with that class (the assumption is that if you are going to need a random element with a given class, then it's most likely going to be a `div`).

We are going to use Jade-Bootstap components to create our simple website. For now though we are going to start by making some trivial modifications to the webpage. We will make bigger changes later.

Web Page Structure

HTML pages have a generally fairly simple structure. The first line of the file is a doctype. This isn't really part of the HTML, it just tells the browser what version of HTML to expect.

The rest of the document in inside of an `html` tag. The webpage is then broken up into two parts - and tags - the head and body. The head contains general metadata and stuff which applied to the whole page. Things like the page title (the text that gets displayed in the title bar), what CSS stylesheets to use and so on. The body contains the actual content of the page that you expect to see on the screen.

The Existing Jade Pages

All the Jade pages live in the `views` directory. There are three pages in here. * layout.jade * index.jade * error.jade

The `index.jade` page is the one that is served to the user. `error.jade` is the page it shows on an error, and `layout.jade` is a base template it uses to structure both the other pages.

The Basic Template

We can start by looking at `layout.jade`.

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

Note how this is pretty short, and doesn't look like HTML. Which is all basically good things. Now let's take a closer look.

```
doctype html
```

Every HTML page starts with a doctype. This is the Jade equivant, and tells the browser that it is written using HTML 5. Note that in normal HTML the doctype does not have a closing bit. In Jade nothing has a closing tag so we don't really notice.

```
html
```

The `html` tag is the tag that everything else in the document is inside. In Jade you are inside another tag if it is above you, and you are more indented than it.

So with

```
tree
  apple
```

you get

```
<tree>
  <apple />
</tree>
```

where `apple` is inside of `tree`. But with

```
tree
apple
```

you get

```
<tree />
<apple />
```

`apple` is after and **not** inside `tree`.

Keeping this in mind, the `html` contains the head which contains:

```
    title=title
```

This makes the title of the page (which is shown in the title bar) the title that is passed in to the page by node (we'll see where node does this later). The `=` tells Jade that you want the title text to be what is in the variable title, rather than making the title actually be 'title'. Note that the `=` is only useful where you want to variable to be **all** of the text inside the tag. If you want a mixture of normal text and variables you need something else (which we will see later).

The title then also contains a:

```
    link(rel='stylesheet', href='/stylesheets/style.css')
```

This is a link tag that links to a CSS stylesheet. The `stylesheets` directory lives in the `public` directory of your node project. Therefore, like anything in the `public` directory, it can be accessed just by it's path inside the `public` directory. Note also how tag attributes are assigned similarly to function arguments in Java.

So a jade expression:

```
tag(attr="value")
```

becomes

```
<tag attr="value">
```

After this we have the `body`. This is where everything that is visible in the document will go. This is followed by a single line

```
    block content
```

The `block` keyword in Jade doesn't actually put anything there. Instead `block` is the keyword Jade uses to set up places where templates can override content. Annoyingly Jade uses the same word to override a block as to define a block for overriding.

This is because a common idea in Jade (and other template languages) is to reuse common bits. So you can define a common template (note that this is also how the jade-bootstrap pages work), and then fill it in with the bits you need for your specific page.

In this case the use of `block content` is setting up a place for pages based on this template to put there content. The fact there is is a single block keyword means that there is only a single such point. `content` is it's name. When overriding (or extending) a block you use it's name to specify which block you are replacing the content for.

The Basic Content Page

The `index.jade` now makes uses of the common template defined in the layout page to further build on it.

It starts with:

```
extends layout
```

This means that it gets everything that is inside the `layout.jade` file. However, we know that the layout file doesn't really contain any interesting content, only a space for us to put it in.

The next line

```
block content
```

says to replace everything in the content block in layout, with what is provided in this file. As there is nothing in the layout file, it simply fills in it. Sometimes you don't want to replace everything that is already in a block - but rather want to append (add to the end) or prepend (add to the start) to it. For that purpose the you can write `block append <name>` or `block prepend <name>`. Though in this cases the word `block` is optional, and you can simply write `append` or `prepend`.

So the content block is replaced with

```
h1= title
p Welcome to #{title}
```

The first line is just like what we saw before with the title. The second line is more interesting.

If you just put down a tag name, any text that follows it is just the text that you want to write in it. However, since we want to put the title into that text we need to escape it, so the system knows that it is a variable and not the word title.

More Jade

There are a couple of extra bits about the Jade language that you should be aware of before we move on.

Multiple lines: Sometimes you want to have a tag followed by more text than can fit onto a single line. In that case you need to use a `|` to indicate that the text all belongs to the one tag like in the following example.

```
p | This is a
  | really long line
```

Note that this can be broken up with other things which get nested inside.

```
p | This is a really long line
  span (some more text)
  | the rest of the line.
```

Escaping: So far we have seen two different ways of escaping text (tell jade that it's a variable rather than just text). However, there are in fact three.

The first two are using `=` and `#{}`. The third one uses `!{}`. This works the same ways as `#{}` but is unsafe. If you pass in a bit of HTML into `#{}` it will actually write the `<>` on the page. So any embedded code is escaped and rendered visible on screen. In constract `!{}` drops the text straight into the page. This is good becasue you can have tags in your text like `<b>text</b>` and they will display as bold rather than just showing up. However, it also hides the tags (if you want them to be visible), and if you don't trust the source of the text can be a security risk.

Mixins: Mixins are basically functions in Jade. Their basic purpose is to prevent repetitive typing. Mixins are similar to tags but they start with a `+` and take arguments (just like a Java function).

For example Jade-Bootstrap has a mixin to include a glyphicon which takes a string for the name as an input:

```
+icon("film")
```

This is really just a shorthand for:

```
span.glyphicon(class="glyphicon-film",aria-hidden="true")
```

And if we look at it's definition we can see that it's defined in a way that is similar to what we would expect, but with no `{}`s and starts with the word `mixin` to show that it's a mixin.

```
mixin icon(name)
        span.glyphicon(class="glyphicon-#{name}",aria-hidden="true")
```

Class and ID Shorthand: Because assigning classes and ids is so common Jade provides shorthands for both. You can add a class by adding `.classname` to the tag, or and id with `#id`. The following two lines would both be a `div` with class `apple`.

```
div.apple
.apple
```

Note that because divs are so common if you just write a class (or id) with no tag attached (as in the second example) it will automatically assume that the tag is meant to be a div and put one in.

BaseX

At the moment, the default website doesn't have a basex server of any sort attached. BaseX is actually an entirely separate program from Node.js and we will need to download it from their website. When you download it, be sure to download the **ZIP Archive** version.

You should unzip it to the directory above your website. And in that same directory place your XML files. So your directory should contain (at least) these three directories.

- basex
- movieWebsite
- Colenso

Starting BaseX

We can then start BaseX with a database called Colenso containing all the documents in the Colenso directory with the following command. We are going to use the Colenso database for this section rather than the movie database, because the Colenso database is a bit more complicated and there are a few extra things we need to learn about to use it.

```
basex/bin/basexserver -d -c"CREATE DATABASE Colenso ./Colenso"
```

In this the `-d` starts it in debugging mode, and the `-c` gives it a command to run on startup.

We can then run

```
cd movieWebsite
node
```

to get an interactive node shell. We can use this to explore our database a bit using the basex library we installed. Note how we needed to go into our website directory in order to do this. This is because the node libraries we installed are only visible to node when it's inside your project, rather than everywhere.

Connecting To The Server

The basex node library doesn't run a BaseX server. It just talks to one which already exists and is running. So for this section make sure that you have a running BaseX server while we run these commands.

The first thing we need to do is load the BaseX node library. This is a lot like a Java `import` statement, but we need to save the result to a variable.

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

We then then need to connect to the running BaseX server. We know that the default username and password are both *admin*, the default port is 1984, and it's running on the computer that we are on now. Note that the IP address for *this computer* is always 127.0.0.1.

```
var client = new basex.Session("127.0.0.1", 1984, "admin", "admin");
```

Running Commands

Once we have a client we can call the `execute` function in order to run some kind of query or command against the database. The `execute` function takes two arguments: a command to run (a String), and (optionally) a function to run after the command is finished. Note that every command String starts with what type of command it is. You can see the list here.

Note that this is pretty different from how you would expect to do it in Java. There you would expect to do something like:

```
try{
  Result res = client.execute("My Command");
  doSomething(res);
} catch (IOException e){
  ...
}
```

It does the computation, waits for the result, then does the next time. If there is a problem, it throws and exception which is dealt with in the catch block.

If we want to run a simple command that we don't care about the result of we can just run it in the expected way.

Let's connect to the Colenso database so we can access the XML files we loaded into it.

```
client.execute("OPEN Colenso");
```

We don't really care what the output of this command is. Of course, is we were being careful we would actually check that the command succeeded and that the Colenso database exists. However, we are just going to ignore that problem and assume that everything works fine without problems.

How about running a command that is supposed to return a result we care about? Let's try running some xquery commands.

We can try search for every name tag in the database. We then want to print out the results.

```
client.execute("XQUERY //name",
  function(error, result) {
    if(error) {
      console.error(error);
    } else{
      console.log(result);
    }
  });
```

When we run this we get zero results. Before we try to understand what went wrong with the query, let's first look at how we structured our call.

It starts as we would expect, with a String query. Note how the String start with `XQUERY` indicating that this is an XQuery instruction. It then has an XQuery query `//name` which says find every `name` tag in the database. However, it is the function that is more interesting. We can read the code as do the query, then when it is complete, give the result to the function (the second argument).

Note how we don't have a try-catch block here. Instead the function has two arguments - an error and a result. If an error didn't occur then the error argument will have a falsey value in it (one that is false when you do a logical test). The result argument contains the result of the computation if no error occurs.

This function will always be called after the query is executed. Such functions are often referred to as callbacks. The term callback doesn't refer to something special about the function, but rather to the fact that it is called after a function is finished running. This is similar in many ways to how you can have listeners waiting for events (in this case a function finishing), though you only have a single callback where you may have any number of listeners.

The other thing to pay attention to is the structure of the result argument. Notice how it is an object with three fields - result, info and ok. If we were being very careful we could use all of these fields to get a better understanding of how our queries are happening. But for this tutorial we are just going to use the `result` field, because if everything goes well that is the one that actually stores the result we need to use.

XQuery With The Colenso Database

You probably still remember that our perfectly resonable query didn't return any results, even though there are a lot of `name` tags in the database that we are using.

The reason is that the files in the database are all TEI files. They are XML files with a specified schema. So the name of the tag isn't just `name` it's name from the tei schema which has an address of `http://www.tei-c.org/ns/1.0`.

There are two ways to tell our query about the tei schema.

1. We can tell the query that unless told otherwise, it should assume that all of the tags in the query are from the tei schema. We do this by adding the following to the start of the XQuery.

   ```
   declare default element namespace 'http://www.tei-c.org/ns/1.0';
   ```

2. We can declare a short hand (`tei` in our example) to refer to the full address and then refer to tags as `tei:tag`. The following line does this.

   ```
   declare namespace tei='http://www.tei-c.org/ns/1.0';
   ```

Using either method, we would now repeat the above query and get the results we would expect. However, since there are a lot of `name` elements in the database, we are going to run a query with a lot more filtering (so there is only one result, and your terminal doesn't overflow).

Note that from now on we will use a much simplified printing method which just prints the result if it's there, and otherwise prints nothing.

Using the first method we can do:

```
client.execute("XQUERY declare default element namespace 'http://www.tei-c.org/ns/1.0'; " +
"//name[@type = 'place' and position() = 1 and . = 'Manawarakau']",
function(err,res) { if(!err) console.log(res.result)} )
```

And using the second:

```
client.execute("XQUERY declare namespace tei= 'http://www.tei-c.org/ns/1.0'; " +
"//tei:name[@type = 'place' and position() = 1 and . = 'Manawarakau']",
function(err,res) { if(!err) console.log(res.result)} )
```

Both of which function correctly and find us the one instance where this tag occurs.

Filtering

In the above queries we used a whole bunch of filters to narrow down what we were searching for. Let's have a look at that in a little bit more detail. We can see what we apply the filtering rules in `[]` attached to the element of interest. Different rules can be combined with logical operators like `and` and `or` as you would expect.

So we have the following filters:

- `@type = 'place'` - This says that the tag must have the attribute type with a the value place i.e. `<name type='place' ...>`.
- `position() = 1` - This means that if the parent node has more than one of these nodes, give us the first one. Note that this doesn't mean you'll only get one result in total. It only means that you will get the first from each parent, but you can have as many different parents, and therefore results, as you like.
- `. = 'Manawarakau'` - This checks to make sure that the text in the tag is equal to 'Manawarakau' i.e. `<name ... >Manawarakau</name>`.

This does of course raise the question of if I have many results how do I get the first one? Or the 5th to the 10th?

Fortunately, the queries are treated as an array, so each element has a position (index) that can be used to restrict it.

So to get the first element of a query that returns a **lot** of results:

```
client.execute("XQUERY declare namespace tei= 'http://www.tei-c.org/ns/1.0'; " +
"(//tei:p)[1]",
function(err,res) { if(!err) console.log(res.result)} )
```

Or the 5th to 10th:

```
client.execute("XQUERY declare namespace tei= 'http://www.tei-c.org/ns/1.0'; " +
"(//tei:p)[position() = (5 to 10) ]",
function(err,res) { if(!err) console.log(res.result)} )
```

Other Interesting Commands

We can use commands to look at what databases are available:

```
client.execute("LIST .", function(err,res) { if(!err) console.log(res.result)} )
```

Or to look at what files are in any given database:

```
client.execute("LIST Colenso", function(err,res) { if(!err) console.log(res.result)} )
```

We can also retreive the raw text of a given file:

```
client.execute("XQUERY doc('Colenso/McLean/private_letters/PrLMcL-0024.xml')",
function(err,res) { if(!err) console.log(res.result)})
```

Note how the path start with the database, then the location within the database.

We can use this single document lookup as the start of an XQuery expression to limit what is searched though:

```
client.execute("XQUERY declare namespace tei='http://www.tei-c.org/ns/1.0'; " +
"(doc('Colenso/McLean/private_letters/PrLMcL-0024.xml')//tei:p)[1]",
function(err,res) { if(!err) console.log(res.result)} )
```

We can also do a similar thing to only look in a given directory. Note again how this path starts with the database name.

```
client.execute("XQUERY declare namespace tei='http://www.tei-c.org/ns/1.0'; " +
"(collection('Colenso/Hooker/')//tei:p[position() = 1])",
function(err,res) { if(!err) console.log(res.result)} )
```

We can also use the `db:path()` function to find out what document a match came from.

```
client.execute("XQUERY declare namespace tei='http://www.tei-c.org/ns/1.0'; " +
"for $n in (collection('Colenso/Hooker/')//tei:p[position() = 1])\n" +
"return db:path($n)",
function(err,res) { if(!err) console.log(res.result)} )
```

Notice that this is using a more complex FLWOR expression.

FLWOR Expressions

FLWOR expressions in XQuery allow you to express more complicated queries. I will not explain them fully here, if you want to use them you should go and read up on them (they aren't hugely complicated).

The basic idea is that you can loop over every match to your query and apply some additional filters and rules to it, as well as reformat the output. They contain [up to] 5 parts as is in the name: for, let, where order, return.

ESLint

ESLint is a program that checks if a JavaScript file follows a set of good practice rules. ESLint does not have a single set of rules that it believes are correct. Instead, it relies on having a configuration file to tell it what sort of style it expects. Therefore, before we use ESLint we need to give it a set of rules. These live in the `.eslintrc` file. While we could write these ourselves - it would be a lot of work. We can instead use `eslint.js --init` to create one for us.

It doesn't actually matter what set of rules you follow. The main reason for having a set of rules is to make your code easier to debug and read by making it more consistent. This is especially important when you may have more than one developer working on a single project, as they have significantly different prefered ways of writing things.

We can then check our code by running:

```
eslint.js .
```

Making it More Interesting

Now lets have a look at how we can make our website a little bit more interesting by integrating Jade-Bootstrap and some BaseX queries.

Jade-Bootstrap

I have previously recommended the Jade-Bootstrap library, but so far I have only explained what Jade is. So what is Jade-Bootstrap?

Jade-Bootstrap is basically just a Jade template that includes the Bootstrap CSS files, and provides helper mixins to make it easy to create Bootstrap components as you see on the bootstrap components page.

The main entry point is the `_bootstrap.jade` file, which you can extend from within your file (in the views directory) with:

```
extends ../node_modules/jade-bootstrap/_bootstrap

block body
```

There is only one extension point for content you want in the page - `body` - so all the code goes in there. There is also an extension point for linking additional CSS sheets - `styles` - and a third for additional JavaScript - `scripts`.

It contains a large collection of mixins for basically as the examples shown in the bootstrap components page, which you can find demonstrated on their website

Modifying Our Page

We are now going to have a simple navigation bar (that collapses when the screen is small), a title, some text, and the output of a query from the database. Basically everything you could want for a small website.

The current `index.jade` is essentially replaces with what is below.

```
extends ../node_modules/jade-bootstrap/_bootstrap

block body
  +navbar("ECS Movies", "dropdown_menu")
    +nav_item("/", "active") Home
    +nav_item("/list") Stock
  .container
    h1= title
    .container
    p| Welcome to #{title}. We have an excellent collection of videos in stock for you.
     | We are based in #{place}.
```

Note how we are now using everything we discussed earlier about Jade and Jade-Bootstrap. We can also see that there are two variables that need to be passed into this page for it to work: `title` and `place`.

Adding A BaseX Query

However, this hasn't actually changed any of the backend code, and so there will not be any `place` variable provided. To change this we need to modify `routes/index.js`.

We need to add the following to the variable declarations at the top of the file. These are just to use the node BaseX library, to connect to the database and to tell it to use the Colenso database. Note that you **must** have a BaseX server running in order for this code to work.

```
...
var basex = require('basex');
var client = new basex.Session("127.0.0.1", 1984, "admin", "admin");
client.execute("OPEN Colenso");
```

We need to also change the routing rule. Routing rules have a very simple structure. They have a `get` function which takes two arguments:

1. The page the user tried to get (`/` is if they just go to the website and don't type anything more.
2. A function (that takes two arguments).
    1. The first argument is used to keep track of the request, query strings and the like.
    2. The second argument lets you send a reponse. You can use `res.render` (assuming that you named the second argument res), to render a Jade page (the first argument is the Jade template to render, the second is an object containing the feilds the Jade page needs). You can also use `res.send` to just send some data.

In the following function we use all this to render a Jade page and give it the result of a database query.

```
router.get("/",function(req,res){
client.execute("XQUERY declare default element namespace 'http://www.tei-c.org/ns/1.0';" +
" (//name[@type='place'])[1] ",
function (error, result) {
  if(error){ console.error(error);}
  else {
     res.render('index', { title: 'ECS Video Rental', place: result.result });
  }
      }
      );
});
```

Note how on the resulting page the `<name ...>` part of the XML result is visible. This is because we used the `#{place}` rather than `!{place}` to place the data. But even if we used `!{place}`, `<name>` is not a standard HTML tag, so the browser won't be able to do anything interesting with it unless we provide some custom CSS to tell it what to do.