

COM644 Full-Stack Web and App Development

Practical A1: Introducing Node.js

Aims

- To introduce Node.js and the Node Package Manager npm
- To check the Node and npm installation
- To demonstrate the Node console
- To introduce Node source files
- To demonstrate splitting code across separate source files
- To introduce the concept of blocking code
- To demonstrate the use of anonymous and named callback functions
- To illustrate the problem of computational blocking

Contents

A1.1 INSTALLING AND RUNNING NODE.JS	2
A1.1.1 CHECKING YOUR INSTALLATION	2
A1.1.2 RUNNING NODE.JS FROM THE COMMAND LINE	3
A1.1.3 CREATING AND RUNNING NODE.JS SOURCE FILES.....	4
A1.2 SPLITTING NODE APPLICATIONS INTO MULTIPLE FILES.....	5
A1.2.1 IMPORTING CODE TO RUN IMMEDIATELY	5
A1.2.2 IMPORTING FUNCTIONS	5
A1.2.3 IMPORTING FUNCTIONS THAT RETURN VALUES.....	8
A1.3 NON-BLOCKING CODE.....	10
A1.3.1 SYNCHRONOUS I/O OPERATIONS AND BLOCKING.....	11
A1.3.2 ASYNCHRONOUS I/O – NON-BLOCKING CODE AND CALLBACK FUNCTIONS.....	12
A1.3.2 COMPUTATIONAL BLOCKING	14

A1.1 Installing and Running Node.js

Node.js (often called simply “Node”) is a development platform built on top of Google’s V8 Javascript engine. Node uses an **event-driven, non-blocking** input/output model that runs as a single process, resulting in a very lightweight and efficient platform that is well-suited to the implementation of web server applications. Node is increasing rapidly in popularity with large-scale online organisations – including Netflix, PayPal, GoDaddy, LinkedIn and many others.

Node comes bundled with a package manager called **npm** (Node Package Manager). This makes it easier for developers to publish and share useful libraries.

Node and **npm** are installed for you on the lab systems, but they can be downloaded free from <https://nodejs.org>.

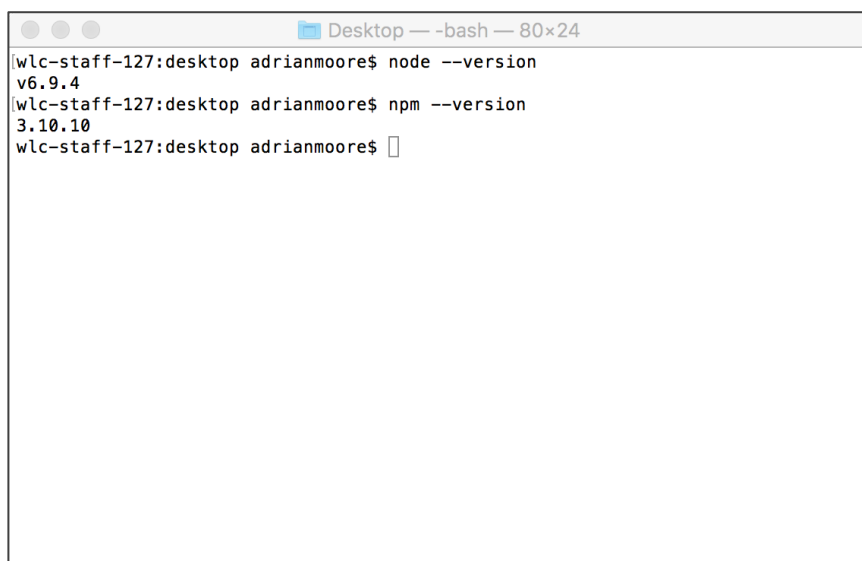
A1.1.1 Checking your installation

We can check for the availability of Node and npm by requesting the version number of the installation on our machine. Open a Command window on your PC and issue the following commands

```
U:\> node --version
```

```
U:\> npm --version
```

If both Node and npm are installed properly, you should see a result similar to that illustrated in Figure A1.1 below.

A screenshot of a terminal window titled "Desktop — -bash — 80x24". The terminal shows the following commands and their outputs:

```
wlc-staff-127:desktop adrianmoore$ node --version  
v6.9.4  
wlc-staff-127:desktop adrianmoore$ npm --version  
3.10.10  
wlc-staff-127:desktop adrianmoore$
```

Figure A1.1 Checking the Node.js and npm installation

A1.1.2 Running Node.js from the command line

Now that we are satisfied that Node and npm are available on the PC, we can issue our first Node commands by entering the Node console with the command


```
U:\> node
```

You should find a command line interface with the Node console prompt `>`, at which we can enter Javascript commands.

As a first example, we will use the Javascript **`console.log()`** command that allows us to output information to the console. Enter the commands below to print out a simple “Hello World” message, then another version that uses a Javascript variable

```
> console.log("Hello world!")  
  
> var name = "Adrian"  
  
> console.log("Hello " + name)
```

Verify that you receive output such as that shown in Figure A1.2 below.

A screenshot of a terminal window titled "Desktop — node — 80x24". The terminal shows the following interaction:

```
wlc-staff-127:Desktop adrianmoore$ node  
> console.log("Hello world!")  
Hello world!  
undefined  
> var name = "Adrian"  
undefined  
> console.log("Hello " + name)  
Hello Adrian  
undefined  
> 
```

Figure A1.2 The Node command prompt

A1.1.3 Creating and running Node.js source files

Using the Node console allows us to quickly verify that the platform is installed and working, but in order to do any serious development, we need to be able to organise our code into source files. Indeed, the way in which Node manages collections of files is one of its main strengths as a platform for rapid development of serious applications.

Exit the Node console by entering CTRL-C twice and create a folder called **A1** in which we will arrange our files in this practical. Now, navigate into the **A1** folder and create a new file called **app.js** containing the following code.

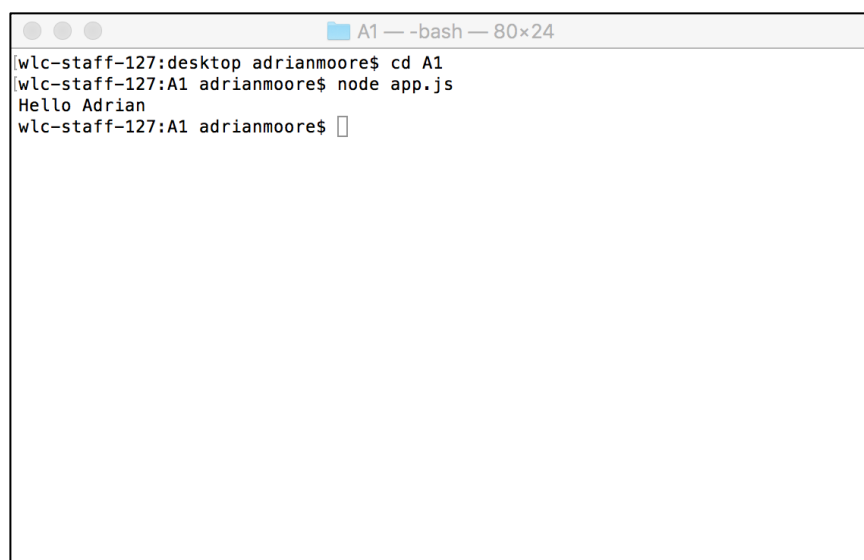
File: A1/app.js

```
var name="Adrian";  
console.log("Hello " + name);
```

Now, navigate into the A1 folder in the Command window and run the new file by the command

```
U:\A1> node app.js
```

If everything is in place, you should see output such as that illustrated in Figure A1.3 below.



```
A1 - bash - 80x24  
wlc-staff-127:desktop adrianmoore$ cd A1  
wlc-staff-127:A1 adrianmoore$ node app.js  
Hello Adrian  
wlc-staff-127:A1 adrianmoore$
```

Figure A1.3 Running a Node file from the command line

A1.2 Splitting Node applications into multiple files

One of the main principles in Node development is that we want our code to be modular – where no single file should contain a large volume of code. With this in mind, we will now examine three ways in which we can move code into external files and call it from elsewhere in the application.

A1.2.1 Importing code to run immediately

First, we will examine the simplest case, where we want to call code from an external file to be run immediately.

Copy the hello world code from **app.js** into a new file called **instantHello.js** and replace the **app.js** code with the **require()** statement below so that you have the following code structure.

File: A1/app.js

```
require('./instantHello')
```

File: A1/instantHello.js

```
var name="Adrian";  
console.log("Hello " + name);
```

In the **require()** statement, note the **./** path to the external file. This is required to tell Node that the file we want to use is located in the current folder. If we do not provide a path, Node will assume that we are trying to import one of the standard Node libraries that is managed by npm (see Practical A2 for more information). Also, note the lack of a **.js** file extension – actually we could include this if we wanted, but the convention is to leave it off, so that Node will look BOTH for a file called **instantHello.js** and for a folder called **instantHello**.

We can now run **app.js** and verify how the application still works exactly as before.

A1.2.2 Importing functions

We do not always want to include code that is to be run straight away. It is much more common that we might define a function that will be called in response to some event. To

achieve this, we need to use the Node **module.exports()** method to **expose** the function that we want to make available.

Create a new sub-folder within **A1** called **talk**. Inside the new folder, create a file **goodbye.js** with the following code

File: A1/talk/goodbye.js

```
module.exports = function() {  
    console.log("Goodbye!");  
};
```

This code creates an anonymous function (i.e. one with no name) which uses **console.log()** to output a message and makes the function available to other code files by assigning it to **module.exports()**. Now we need to **require** the **goodbye** file within **app.js** and call it as a regular function as follows.

File: A1/app.js

```
require('./instantHello');  
var goodbye = require('./talk/goodbye');  
  
goodbye();
```

Note how the function acquires its name by the variable to which we assign the exposed function. This is an example of Javascript **functions as first order objects** (i.e. functions can be used just as primitive data types and can be used in assignments, stored in arrays and passed as parameters just like objects of any other type). This is a very important concept in Node and one to which we will frequently return.

You can now verify the operation of this code by running **app.js** in the Command window and verifying that you receive output as illustrated in Figure A1.4 below.

A terminal window titled "A1 — -bash — 80x24" showing the execution of a Node.js script. The prompt is "wlc-staff-127:A1 adrianmoore\$". The command "node app.js" is entered, and the output is "Hello Adrian" followed by "Goodbye" on the next line. The prompt returns to "wlc-staff-127:A1 adrianmoore\$".

```
wlc-staff-127:A1 adrianmoore$ node app.js
Hello Adrian
Goodbye
wlc-staff-127:A1 adrianmoore$
```

Figure A1.4 Exposing functions from an external file

Sometimes we want to expose multiple functions from a single file. We will demonstrate this by re-factoring the previous example to replace the “Hello world” message by a second function that will be also exported from the code file. Create a new file inside the talk folder called **index.js** as follows

File: A1/talk/index.js

```
var filename = "index.js";

var hello = function(name) {
    console.log("Hello " + name);
};

var goodbye = function() {
    console.log("Goodbye from " + filename);
};

module.exports = {
    hello : hello,
    goodbye : goodbye
};
```

Here, the functions are exposed by defining them as a Javascript object of key/value pairs, where the key is name of the function to be exposed and the value is the name by which it will be known in code where it is **required**. Note that all that is exposed are the two

functions – the variable **filename** is private to this file and can be used by any of the functions, but is not exposed to the wider application. We might equally have implemented private functions that are used by those to be exposed, but are not themselves exposed.

Now we require the new functions within **app.js** and use them. Note that the **index.js** filename does not need to be specified, it will be assumed as long as we provide the path.

File: A1/app.js

```
var talk = require('./talk');  
  
talk.hello("Adrian");  
talk.goodbye();
```

Run **app.js** from the Command prompt and verify that you receive the output as illustrated in Fig A1.5

A terminal window titled 'A1 - bash - 80x24' showing the execution of 'node app.js'. The output is 'Hello Adrian' followed by 'Goodbye from index.js'. The prompt 'wlc-staff-127:A1 adrianmoore\$' is visible before and after the command.

```
wlc-staff-127:A1 adrianmoore$ node app.js  
Hello Adrian  
Goodbye from index.js  
wlc-staff-127:A1 adrianmoore$
```

Figure A1.5 Exposing multiple functions

A1.2.3 Importing functions that return values

Sometimes, our functions that are exposed through **module.exports()** will return values that we want to use within the main (calling) part of the code.

We will demonstrate this by creating a new file within the **talk** folder called **question.js** with the following code.

File: A1/talk/question.js

```
var answer = "Now that's a good question!";

module.exports.ask = function(question) {
  console.log(question);
  return answer;
};
```

Here, we define a function called **ask()** which will be exposed to the main application by chaining it to the **module.exports** method. The function will accept a parameter called **question**, which it outputs using **console.log()** and then returns an **answer**, defined here as a private string.

Now, we go back to **app.js** and require the new function. Then, we can create a new variable **answer** to hold the value returned from the exposed function and use **console.log()** to display it.

File: A1/app.js

```
var question = require('./talk/question');

var answer = question.ask("What is the meaning of
                           life?");

console.log(answer);
```

You can now run **app.js** in the Command window and verify that you receive output such as that shown in Figure A1.6 below.

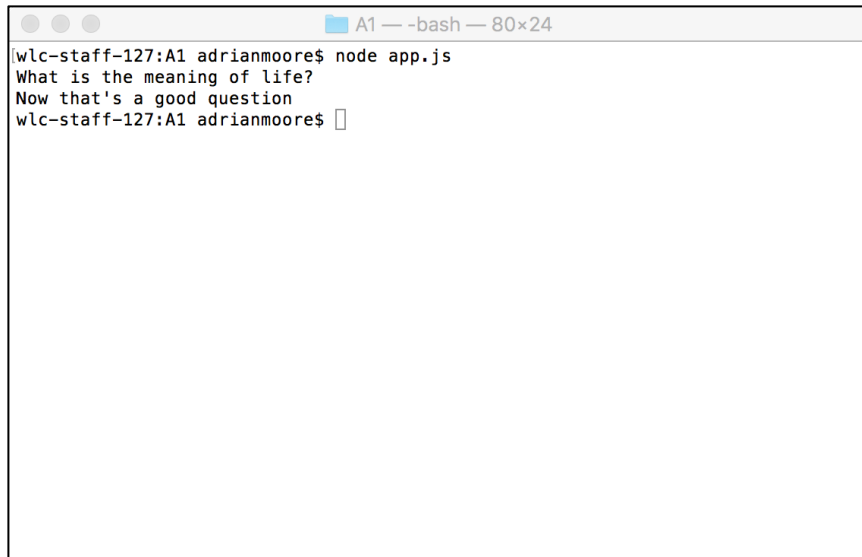
A terminal window titled 'A1 - -bash - 80x24' showing a Node.js application running. The prompt is 'wlc-staff-127:A1 adrianmoore\$'. The user enters 'node app.js'. The application outputs 'What is the meaning of life?' and 'Now that's a good question'. The prompt returns to 'wlc-staff-127:A1 adrianmoore\$'.

Figure A1.6 Accepting a return value from an exposed function

A1.3 Non-blocking code

Node.js is designed to provide **I/O scalability**. As Node is single threaded, it is very important that one user's (e.g.) database access does not slow down the server for other users, hence input/output operations in Node are asynchronous and non-blocking. The consequence of this is that later operations in the code sequence should not be forced to wait for earlier functions to complete.

As an example of how we can specify operations to run 'out of order', create a new file within your **A1** folder called **setTimeout.js**, with code as follows

File: A1/setTimeout.js

```
console.log("1. Start app");

var holdOn = setTimeout(function() {
    console.log("2. In the setTimeout");
}, 1000);

console.log("3. End app");
```

Here, we have 3 calls to **console.log()**, one at the beginning of the code, one at the end, and one within a function that is a parameter to a Javascript **setTimeout()** function. The purpose of **setTimeout()** is to introduce a delay before a specified action is executed,

with the delay expressed in milliseconds, hence we should have a delay of 1 second before the second `console.log()` is executed.

Run the file `setTimeout.js` in the Command window and observe how the first and third messages appear instantly, while the second does not appear until after the one second delay has elapsed. Also, consider how the delay in the second message has not prevented the third from being displayed – i.e. the `setTimeout()` function has been implemented as a **non-blocking** operation.

A terminal window titled 'A1 - bash - 80x24' showing the execution of a Node.js script. The prompt is 'wlc-staff-127:A1 adrianmoore\$'. The command 'node setTimeout.js' has been entered. The output shows three lines: '1. Start app', '3. End app', and '2. In the setTimeout'. The order of execution is 1, 3, then 2, demonstrating that the setTimeout function is non-blocking and does not prevent subsequent code from running.

```
wlc-staff-127:A1 adrianmoore$ node setTimeout.js
1. Start app
3. End app
2. In the setTimeout
wlc-staff-127:A1 adrianmoore$
```

Figure A1.7 Simulating a non-blocking operation

A1.3.1 Synchronous I/O operations and blocking

Now create a file called `readFileSync.js` which uses the native Node module `fs` that allows us to read files from the local file system

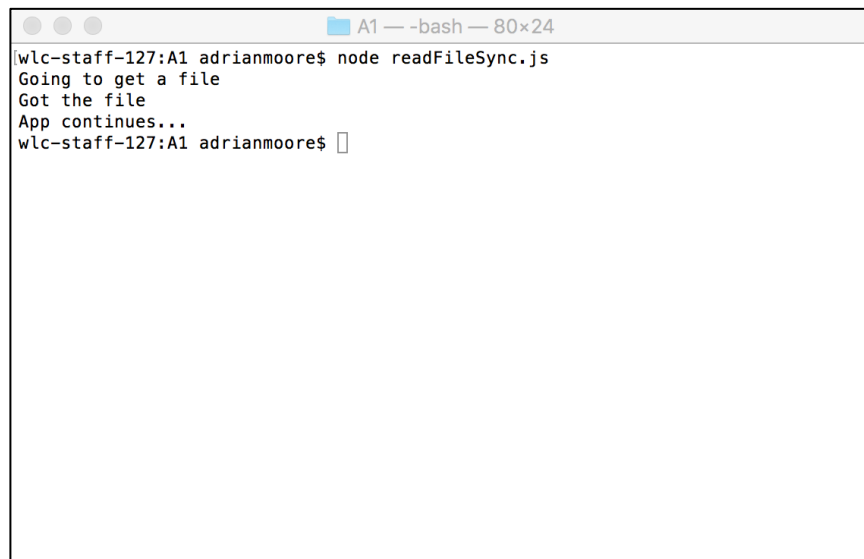
File: A1/readFileSync.js

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

console.log("Going to get a file");
var file = fs.readFileSync('readFileSync.js');
console.log("Got the file");

console.log("App continues...");
```

The **readFileSync()** method reads a file in a synchronous (i.e. blocking) manner, requiring that subsequent operations cannot be executed until the file read is complete. Run **readFileSync.js** in the Command window and verify that the **console.log()** messages are generated in the same order as they appear in the code.

A terminal window titled 'A1 - bash - 80x24' showing the execution of a Node.js script. The prompt is 'wlc-staff-127:A1 adrianmoore\$'. The command 'node readFileSync.js' has been entered. The output shows three lines of log messages: 'Going to get a file', 'Got the file', and 'App continues...'. The prompt returns to 'wlc-staff-127:A1 adrianmoore\$' after the execution.

```
wlc-staff-127:A1 adrianmoore$ node readFileSync.js
Going to get a file
Got the file
App continues...
wlc-staff-127:A1 adrianmoore$
```

Figure A1.8 Synchronous operation

Although this file input operation is a trivial one, we can prove its blocking nature by refactoring the code to use an asynchronous (i.e. non-blocking) file read instead and comparing the output.

A1.3.2 Asynchronous I/O – non-blocking code and callback functions

Create the new file **readFileAsync.js** in the A1 folder, with code as shown below.

File: A1/readFileAsync.js

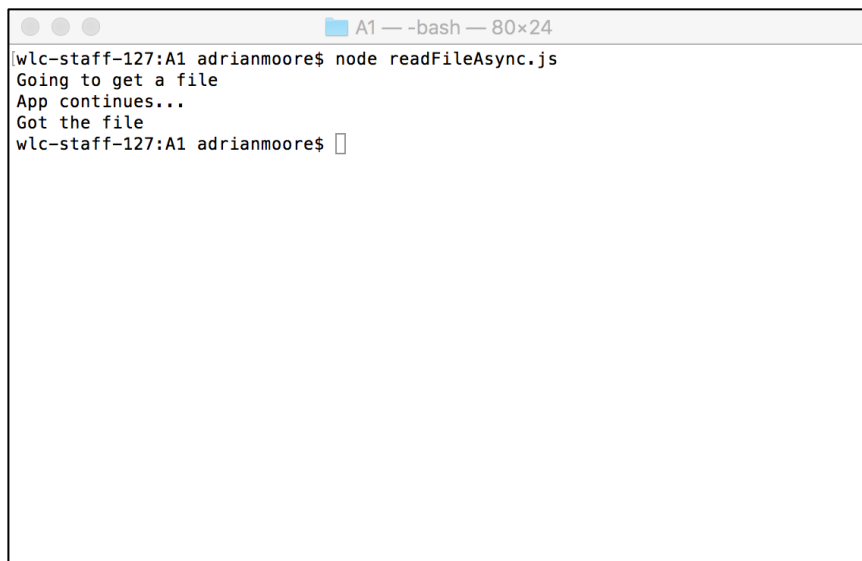
```
var fs = require('fs');

console.log("Going to get a file");
fs.readFile('readFileSync.js', function(err, file) {
    console.log("Got the file");
});

console.log("App continues...");
```

Here, we use the asynchronous method **readFile()**, which accepts as a second parameter an **anonymous callback function** that is executed when the read operation is complete. The callback function takes 2 parameters, an object which is populated when a read error occurs, and an object that will contain the contents of the file that has been read. We will see later (Practical A3) how these parameters can be used, but for now we want to verify that the time taken by the read operation is sufficient for the third **console.log()** message to be displayed before the file read is complete.

Run **readFileAsync.js** in the Command window and verify that the **console.log()** messages appear in the order shown in Figure A1.9 – i.e. the file read operation is proven as **non-blocking**.

A terminal window titled 'A1 - bash - 80x24' showing the execution of a Node.js script. The prompt is 'wlc-staff-127:A1 adrianmoore\$'. The command 'node readFileAsync.js' has been entered. The output consists of four lines: 'Going to get a file', 'App continues...', 'Got the file', and the prompt 'wlc-staff-127:A1 adrianmoore\$' again. The output messages are displayed sequentially, demonstrating that the asynchronous file read operation does not block the subsequent console.log statements.

```
wlc-staff-127:A1 adrianmoore$ node readFileAsync.js
Going to get a file
App continues...
Got the file
wlc-staff-127:A1 adrianmoore$
```

Figure A1.9 Asynchronous operation

Anonymous callback functions are a powerful and widely used element of Node.js, but to improve code readability we could just as easily re-write the callback as a named function as shown below.

File: A1/readFileAsync.js

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

var onFileLoad = function(err, file) {
    console.log("Got the file");
}

console.log("Going to get a file");
fs.readFile('readFileSync.js', onFileLoad);

console.log("App continues...");
```

Here, we define a function assigned to the variable **onFileLoad**, which is then specified as the callback from the **readFile()** method.

Make these changes to **readFileAsync.js** and verify that the operation is identical to the previous version with the anonymous callback.

A1.3.2 Computational blocking

Not all delays in code execution are caused by I/O operations, and although Node is not designed for **computational scalability**, there is a means by which we can implement non-blocking execution for computationally-intensive operations.

The generation of the Fibonacci sequence is a classic example of an algorithm that is computationally intensive for high values. Create the file **fibonacci.js** that calculates and displays the 42nd value in the Fibonacci sequence and the file **computationalBlocking.js** that **requires** the Fibonacci code and sandwiches it between a pair of **console.log()** statements.

Note: There is nothing special about the selection of the 42nd value in the Fibonacci sequence. It is simply one that generates a decent delay in execution to illustrate the point in this example. Try changing the value to see the effect of the delay time.


File: A1/fibonacci.js

```
var fibonacci = function(n) {  
  if (n <= 2) {  
    return 1;  
  } else {  
    return fibonacci(n-1) + fibonacci(n-2);  
  }  
};  
  
console.log( fibonacci(42) );
```

File: A1/computationalBlocking.js

```
console.log(1)  
require('./fibonacci');  
console.log(2);
```

Run the file **computationalBlocking.js** in the Command window and observe the delay while the Fibonacci value is calculated.

A terminal window titled "A1 — -bash — 80x24" showing the execution of the file computationalBlocking.js. The output is: 1, 267914296, 2. The prompt is wlc-staff-127:A1 adrianmoore\$.

```
wlc-staff-127:A1 adrianmoore$ node computationalBlocking.js  
1  
267914296  
2  
wlc-staff-127:A1 adrianmoore$
```

Figure A1.10 Computational blocking

To preserve the non-blocking nature of Node, we need to find a way to allow execution to continue while the Fibonacci value is calculated. Create a new file **computationalNonBlocking.js** with code as shown below.

File: A1/computationalNonBlocking.js

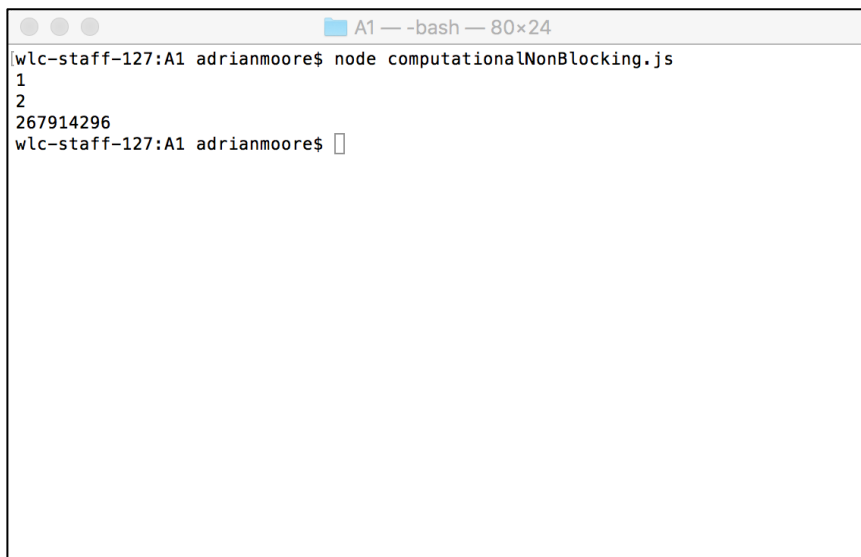
```
var child_process = require('child_process');

console.log(1)
var newProcess = child_process.spawn(
    'node', ['fibonacci.js'], {
    stdio : 'inherit'
});

console.log(2);
```

Here, we use the standard Node module **child_process** that provides a **spawn()** method that launches a separate process in which to perform the calculation – hence providing a non-blocking environment for the original code sequence. The Javascript object **{ stdio : inherit }** passed as the third parameter specifies that the I/O environment on the child should be inherited from the parent – hence allowing the parent and child to share the same console window.

Run **computationalNonBlocking.js** in the Command window and verify that the Fibonacci calculation is now non-blocking – the second **console.log()** message is not delayed by the calculation.



```
A1 — -bash — 80x24
wlc-staff-127:A1 adrianmoore$ node computationalNonBlocking.js
1
2
267914296
wlc-staff-127:A1 adrianmoore$
```

Figure A1.11 Spawning a child process to avoid blocking

Note: Obviously, any operations that depend on the calculation must still wait on the result to be made available. In such circumstances, we would create a callback function containing the dependent code, so that code waiting on the calculation result does not fire prematurely, while other code is not blocked by the calculation. We will see frequent example of this in later Practicals.