Node.js Introduction

What is Node.js?

Node.js, usually called Node, is an open source platform designed for building fast, scalable network applications using the Chrome JavaScript engine. You can use it to build I/O intensive applications that talk to networks, databases, file systems or other sources.

Node does I/O in a way that is asynchronous which lets it handle lots of different things simultaneously.

Node is not a framework nor a programming language. Programs running on Node are written in JavaScript, the same JavaScript you can use on webpages except that it doesn't run in your browser but on a JavaScript engine. Therefore, you could call Node.js a platform.

You can run Node on your Mac, PC or server, in the cloud at for example Nodejitsu and Heroku but also on a Raspberry Pi[1].

The obligatory Hello World

After you downloaded Node from nodejs.org and installed it on your computer your good to go. Create a new file `helloworld.js` (JavaScript files use the `.js` extension) with this content:

```
// just saying hi!
console.log("Hello World!");
```

Now run it on Node from a command prompt or terminal: `node helloworld.js`

Core Modules

Node has small group of modules commonly called the 'Node core' that you can use to write your programs with. Here are some but not all of them:

- **fs** for filesystem access
- **http** webserver and client
- **net** TCP

[1] for example to control a quadcopter

- **dgram** UDP
- **dns** DNS lookups
- **os** Operating system specific information
- **buffer** Allocating binary chunks of memory
- **url** Parsing URL's
- **path** Handling and transforming file paths

You can use these modules with the 'require' clause like so:

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

Modules and NPM

A lot of functionality you might need is already written by other developers. If you are lucky, someone has made a module that does the thing you need. Node comes with its own module manager: NPM.

NPM can be used to search, install, remove, update packages and lots more. NPM avoids dependency conflicts between modules by nesting dependencies.

If you are looking for a twitter module you can search for twitter:

```
npm search twitter
```

NPM will search the repository and returns a list of modules (if you search for 'twitter' you get more then 200 results). If you want to install a module you can use the 'install' command:

```
npm install ntwitter
```

If you write programs in Node that depend on modules, it is a good idea to include a `package.json` file that lists the dependencies. Below is an example, but you can add lots more information:

```
{
  "name": "CoolnessApp",
  "version": "0.0.1",
  "dependencies": {
    "node-static": "0.6.x",
    "socket.io": "0.9.x"
  }
}
```

With a package.json fill you can install all dependencies by running:

```
npm install
```

and npm will install all required modules and their dependencies.

Some details on Callbacks

Callbacks are an important concept in Node. Since Node is a asynchronous platform, it won't wait for asynchronous operations to complete before continuing the execution of the program. So if, for example, you want to query a

really slow database and it would take seconds to complete, you could write something like this:[2]

```
var result = db.query("SELECT * FROM largeTable");
doSomethingWithTheResult(result);
console.log("We are done!");
```

It would take seconds before Node continues with the 'console.log' code. Since Node runs only one proces, this becomes a global problem. If you would do this in a webserver and there would be multiple clients connected to the server, everyone would have to wait before the program continues.

This is different from a language like PHP which runs on a webserver that starts a new process for each request.

You can avoid this by using callbacks. Callbacks are references to functions that can be executed at a later time, for example when the database query is complete. With a callback, you know what **will** be executed, but not **when**.

This way, the program continues while the query is running. When the query finishes, the callback is executed, as the following code shows:

```
db.query("SELECT * FROM largeTable", function(result) {
  doSomethingWithTheResult(result);
});
console.log("query is running, but we don't have to wait...");
```

The callback is the anonymous function which is passed as the second argument to the 'query' function. When 'db.query' is done and has retrieved the results from the database, it will call this anonymous function and pass in the results.

It doesn't have to be an anonymous function, it can also be a function reference as shown in the next two listings below.

```
function myCallback(result) {
  doSomethingWithTheResult(result);
}

db.query("SELECT * FROM largeTable", myCallback);
console.log("query is running, but we don't have to wait...");
```

```
var myCallback = function(result) {
  doSomethingWithTheResult(result);
}

db.query("SELECT * FROM largeTable", myCallback);
console.log("query is running, but we don't have to wait...");
```

Note that the following is *not* correct:

```
var myCallback = function(result) {
  doSomethingWithTheResult(result);
}

db.query("SELECT * FROM largeTable", myCallback(result));
console.log("query is running, but we don't have to wait...");
```

Here, instead of providing a function reference (that is, only the name of the function *without* parantheses), we would already call the function 'myCallback'

[2]This code is just an example to understand node's behavior, not a example on how to work with databases.

before executing 'db.query' and then pass the result that 'myCallback' returns as a parameter 'into db.query'. This is not what we want.
Real working example of a callback:

```
var callback = function() {
  console.log("nothing to see here...");
}

setTimeout(callback, 1000)

console.log("doing something usefull while waiting...");
```

If you run this, the output will be:

```
$> node callback.js
doing something usefull while waiting...
nothing to see here...
```

Events

One of the reason Node has such a good performance is because a lot of code is based on events. Events work based on the publish/subscribe design pattern. If you are interested in an event, for example if someone connects to your server, you subscribe to that event and each time the event occurs, your code is called.

```
var http = require('http');
var app = http.createServer(handler);
app.listen(1337);
function handler(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World!\n');
}


// subscribe to a certain event
app.on('connection', function (stream) {
  console.log('someone connected to our server!');
});
```

It is also possible to generate custom events using event emitters. If you want your object to emit events you'll have to inherit from EventEmitter. The following code gives a basic example.

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();

// subscribe to an event
emitter.on('someImportantEvent', function() {
  // callback executed when event is emitted
  console.log("event fired");
});

// emitter is emitting the event...
emitter.emit('someImportantEvent');
```

Hosting a Node.js app

Of course, once you have written something cool you want the world to see it. There are several options for doing this.

Hosting can be done in the cloud at Nodejitsu, free for the thirty days, and at Heroku, free for one node. Both are easy to work with once you get the hang of it.

A much more complete list can by found at Ryan Dahl's list of Node hosting options.

Examples and code snippets

Webserver

Node doesn't run on a webserver, instead you have to write your own!

The next example does exactly that. It creates a webserver using the `http` library, response to every request with `hello world` with a http 200 header.

```
var http = require('http');
var app = http.createServer(handler);
app.listen(1337);
function handler(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World!\n');
}
```

If you want to handle requests to different URL's in different ways, you'll need to route those requests to different parts of code. Now, that is out of scope for this workshop, but the node beginners book [3] has a nice example of how you can do this.

Serving static web content

Most often you want to serve some static content like HTML, CSS, JavaScript files and images to the visitor of your website. It would be painful to handle all those request hardcoded in your program. It is much better to use a static server for those requests. This can be done with a module called 'Node-static':

Install node static using npm:

```
npm install node-static
```

The webserver itself is programmed in the file `server.js` like this:

```
var http = require('http');
var static = require('node-static');

// http server
var app = http.createServer(handler);
app.listen(1337);

// create static content server with cache disabled
var file = new(static.Server)('./public', {
  cache: 0
```

[3] `http://www.nodebeginner.org/`

```
});

function handler(req, res) {
  file.serve(req, res);
}
```

Now you can put HTML files like `Index.html` in the public folder:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Node.js static webserver</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>It works</h1>
</body>
</html>
```

And of course you can do the same with CSS files like `style.css`:

```
h1 {
  color: blue;
}
```

Socket I/O Events

Creating realtime connectivity between your program and a browser can be done using a module called socket.io. There are others, but I don't have enough time right now to check them all out. Socket.io automatically tries to use the best way to communicate with your browser. If your browser supports websockets, it will use websockets. Otherwise it will use Flash sockets, Ajax long polling, Ajax multipart streaming, a forever Iframe or JSONP polling.

As programmers, we don't have to worry about that very much. Listening for and sending events between browser and server is for both the server and the client done in JavaScript so sometimes, when you are really concentrated and you are switching back and forth between server code and client code you'll forget what you are doing (in my experience). But the nice thing about it is that you don't have to switch between programming languages all the time.

First listing is the node server (`code/socketIO/server.js`):

```
var http = require('http');
var path = require('path');
var static = require('node-static');
var sio = require('socket.io');

// http server
var app = http.createServer(handler);
app.listen(1337);
var file = new static.Server('./public');
function handler(req, res) {
  file.serve(req, res);
}

// socket I/O
var io = sio.listen(app);
io.sockets.on('connection', function (socket) {
  // user connected, send him the time
```

```
    var now = new Date().toDateString();
    io.sockets.emit('news', {hello: 'world', time: now});

    // handle incoming event
    socket.on('press', function (data) {
      console.log('user pressed a button:', data);
    });
  });
```

Second one is the client (`code/socketIO/public/client.js`):

```
var socket = io.connect();

socket.on('connect', function() {
  document.getElementById('connected').innerHTML = 'connected';
});

socket.on('disconnect', function() {
  document.getElementById('connected').innerHTML = 'offline';
});

socket.on('news', function(data) {
  console.log('hello', data.hello);
  console.log('it is now', data.time);
});

window.onload = function(e) {
  console.log('loaded');
  document.getElementById('button').onclick = function() {
    console.log('pressed...');
    socket.emit('press', 'yes I pressed the button');
  }
}
```

And the HTML to tie them together (`code/socketIO/public/index.html`) (notice the script tag for the socket.io library):

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Socket I/O example</title>
  <link href='/style.css' rel='stylesheet'/>
  <script src='/socket.io/socket.io.js'></script>
  <script src='/client.js'></script>
</head>
<body>
  <div id="connected">not connected</div>
  <form>
    <input type="button" id="button" value="press me" />
  </form>
</body>
</html>
```

Weather API

Forecast.io is a weather site that provides (global) weather, but they also have a developers API. To use this API you need to register at developer.forecast.io and then you'll get an API key and a thousand free requests to their API a day.

The API can be queried using a REST interface and returns results in JSON format. There is a Node module that provides a wrapper around this interface.

In the code folder there is a working example that uses that module. Don't forget to run `npm install` before you run it.

To avoid hardcoding the API key in the code we use the environment to set it, in Mac/OS and Linux this can be done in the terminal like so:[4]

```
export FORECAST_API_KEY=dd987b047af59deaed4a2e2f55d21a0
```

This is a good practice for all API keys you might need, especially if you want to use different keys in development and production or if you want to share your code online.
Then we can start using the forecast.io API for example by asking the weather for a specific location. In the next example the weather is requested and the summary is retrieved from the response.

```
var util = require('util');
var Forecast = require('forecast.io');

var options = {
  APIKey: process.env.FORECAST_API_KEY,
  units: 'si'
};

forecast = new Forecast(options);

forecast.get(51.513417, 7.456795, function (err, res, data) {
  if (err) throw err;
  console.log(util.inspect(data.hourly.summary));
});
```

If you run this, it will show the weather forecast summary for the specified location (coordinates).

```
node weather.js
```

Realtime goodness

I find it fun to connect API's together through Node. Especially with API's that push data realtime. There are a lot of API's and most of them work with REST interfaces that return JSON or XML. Each API differs in the way they work. Twitter has a nice one where you can subscribe on hashtags or coordinates. Instagram gives nice results too but I find their API a little bit cumbersome.

There are many more and I encourage you to try them out! Here are some (obvious) examples:

- twitter hashtag subscription (example in code folder)
- lastfm search
- instagram integration (example in code folder, only works online)
- foursquare https://developer.foursquare.com/overview/realtime
- Collection of (realtime) API's: programmableweb.com

[4] In Windows you'll have to do complicated things with system settings (right click on 'My Computer', 'select properties', open the 'advanced' tab, select 'Environmental variables', choose new, enter a name 'FORECAST_API_KEY' and a value: your API key).

More information

- Node.js homepage: nodejs.org
- Node Packaged Modules: npmjs.org
- Socket.IO: socket.io
- The art of node: github.com/maxogden/art-of-node
- Streams: github.com/substack/stream-handbook
- Node and Express: ericleads.com/2013/05/getting-started-with-node-and-express
- Heroku: heroku.com
- Nodejitsu: nodejitsu.com
- List of hosting options: github.com/joyent/node/wiki/Node-Hosting
- The Node beginner Book: nodebeginner.org
- Uit in Enschede REST API: uitinenschede.nl/api/json
- Organisations that use node: github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node