Node.js

Node.js (or just *Node*) is a stand-alone engine that executes JavaScript applications on the server. This technology is quite fast (built with Google's V8 Engine), but is quite low-level and somewhat verbose. Typically, you will layer high-level abstractions and frameworks on top of it (like Express).

Advantages of applications written in Node.js include:

- Fast
- Scalable
- Light-weight and efficient

These advantages are due to its event-driven, non-blocking I/O model. This approach may take a while to get used to, however.

Background

Node.js was created by Ryan Dahl starting in 2009. While quite young, its popularity has grown quickly. Its reliability is due mostly to its underlying simplicity instead of maturing over many years (see this Wikipedia entry).

- nodejs Mailing List is the primary support channel for Node.js questions.
- NodeConf is a regular, migrating conference for Node.js developers and interested parties. Howard Abrams will be attending one in January 2012.

Example Code

The following is a typical example that creates a web server that always says `Hello World` to any HTTP request on port `1337`.

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

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337/');
```

If you have your Node environment installed then you can run this program by creating a file *example.js* that contains the code above. Then issue the command:

```
$ node example.js
Server running at http://127.0.0.1:1337/
```

Now point your browser at http://127.0.0.1:1337 and voila! "Hello World" is displayed in the web page.

Node embraces JavaScript *inner asynchronous nature* (for better or worse). So everything is handled in a *lambda*... er, *callback function* in typical event driven fashion. As you can see in the code above, we begin with an `http` object. We then call two methods on that object:

```
http.createServer();
http.listen();
```

While the `listen()` method takes a couple of scalar parameters, the `createServer()` method takes a function. This function is called each time a request comes in... *asynchronously*. **How do you organize your code in order to build something bigger?** I suggest reading the The Node Beginner Book, a free, online book that shows a good approach at structuring a large amount of code.

How asynchronous are we talking about here? A lot. For instance, let's suppose you had a filename, and wanted to get the full path to it. This is a synchronous operation in most languages, like Java. But in Node, it is asynchronous. Keep in mind, the *following code is wrong, wrong, **wrong**!*

```
var path = "js/somecode.js";
var realpath;

fs.realpath(path, function(err, resolvedPath){
   if (!err) {
      realpath = resolvedPath;
   }
});

console.log(realpath);
```

Running this code will most likely return `null`. Why? By the time the `console.log()` method is called, the inner function may not have finished ... remember, it is asynchronous. Moving the `console.log()` routine *inside* the asynchronous call will work.

How do I execute code sequentially? The short answer is, you don't. Node forces you to use JavaScript more correctly. Stop using global variables (like the `realpath` in the example above) and pass everything into functions. In other words, become a *functional programmer*.

Scared you or interested you?

Installation

To install on Windows or Macs, download and use the supplied installation programs. For our Ubuntu development environment, use the following command:

```
sudo apt-get install nodejs
```

Node.js may already be installed

The CloudPortal Developer Script can be use to install both Node.js and NPM, the Node Package Manager. Before you install these by hand, you may want to download and run that script.

You can't really use Node without installing support libraries. Similar to Ruby's `gem`, Node has a NPM (also on Github) that is used to grab accepted libraries and install them locally. Installation is simple:

```
$ curl http://npmjs.org/install.sh | clean=no sudo bash
```

At this point, you can install a library simply looking up the package name and entering the following on the command line:

```
$ npm install redis  # or mongodb or ... you get the picture
```

Now, in your JavaScript code, you can simply put a `require` at the top:

```
var redis = require("redis");
// ...
```

CloudPortal

We could possibly use Node as the server component for our CloudPortal. The following section get into some low-level details, however, we wouldn't use it. Instead we'd use the following modules for a higher-level abstraction:

- express for serving up static pages (similar to Sinatra)
- mongrue for handling REST calls and storing data
- jsdom for making REST calls to CloudBase by integrating jQuery methods). See these instructions.

However, if you want to know some low-level details, proceed!

Serving the Client Code

The Client Code is dynamic only on the user's browser. From the server's point of view, this is static content. Node is quite low-level, and serving static content is something you would have to code for. We can write it directly, as on this code from the mongrue project:

```
var type = "text/html";
var extname = path.extname(file);
switch (extname) {
        case '.js':
            type = 'text/javascript';
            break;
        case '.css':
            type = 'text/css';
            break;
}

path.exists(file, function(exists) {
        if (exists) {
            fs.readFile(file, 'utf8', function (err, data) {
                if (err) {
                    response.writeHead(500);
                    response.end();
                }
                sendFile(response, type, data);
            });
        }
        else {
            response.writeHead(404);
            response.end();
        }
});

/**
 * This function sends some arbitrary HTML gunk.
 */
function sendFile(response, type, data) {
    var body;
    if (data) {
        body = new Buffer(data);
    }
    else {
        body = new Buffer();
    }

    response.writeHead(200, {
        'Content-Length': body.length,
        'Content-Type': type });
    response.end(data);
}
```

We will not use the above code, for instead of writing our own, we will use the Express module that will handle all that work for us.

Serving JSON via REST

JavaScript and JSON were obviously made for each other, so accepting JSON requests through a REST interface is trivial. We'd begin by incorporating the mongrue project and extending it to support the REST *resources* we expect.

For instance, the following function can be used to send a collection of entries (based on a `GET` request without an ID):

```
/**
 * This function takes an "object" or "array" and sends the
 * results to the client in a standard way.
 */
function sendItems(response, items) {
    if (items) {
        var body = JSON.stringify(items);
        response.writeHead(200, {
            'Content-Length': body.length,
            'Content-Type': 'application/json' });
        response.end(body);
    }
    else {
        sendError(response, 404, "Nothing to return");
    }
}
```

Caching changes in a Local Database

If we used the **mongrue** (mentioned above), then it could already store the REST requests in a local MongoDB database. For instance, the following function could be called when a `POST` is referenced:

```
conn.collection('team', function(err, collection) {
  create(response, collection, body);
});

// ...

function create(response, collection, body) {
    var options = {safe:true};
    collection.insert(body, options, function(err, objects) {
        if (err) {
            responses.sendDbError(response, err);
        }
        else {
            responses.sendItem(response, objects);
        }
 collection.db.close();
    });
}
```

We would probably choose an abstraction layer, for instance, using the mongrue project.

Accessing back-end REST Servers

Making remote API calls, is straight-forward, using the library support already built into Node, ala http.request:

```
var options = {
  host: url,
  port: 80,
  path: '/resource?id=foo&bar=baz',
  method: 'POST'
};

http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
}).end();
```

Another tempting option is to use jsdom and load the (normally client-side) jQuery library, and use its `jQuery.ajax` utility functions to grab and parse the remote REST calls. That would be very nice.

Debugging Applications

Node applications can be started with an open debug port. Our IDE, Springsource Tool Suite (STS), has a plugin that allows you to connect to this port and debug your application.

Since STS is based on Eclipse, you can follow these online instructions.

Some other references to debuggers for Node:

1. node-inspector - https://github.com/dannycoates/node-inspector
2. ndb - https://github.com/smtlaissezfaire/ndb (written in node)

References

The following are books and articles on Node.js that may be of interest:

- Wait, What's Node.js Good for Again?
- Why Everyone Is Talking About Node
- The Node Beginner's Book is a must read for learning how to structure a Node project. (in Chinese)