Node.js Basics

- ***Node.js*** is a tool that is often thought of as a server framework. Node is more general than that, though: it is a tool that simply allows us to run JS outside the browser.

- We can use Node for many purposes, including:
    - Writing simple command-line tools.
    - Writing dynamic web servers.

- You can download and install Node.js from https://nodejs.org.

Node.js on the command line

- At its simplest, Node can be used to create basic command line tools using JS.

- Here's a simple hello world program:

```
console.log("Hello world.");
```

- If we assume our hello world program is stored in `hello.js`, we can run it with Node as follows:

```
node hello.js
```

- You can also just run `node` without any arguments to get a simple command prompt (similar to the console in the browser's dev tools).

- Command line arguments can be accessed via the array `process.argv`. For example, this program would print each of the command line arguments:

```
for (var i = 0; i < process.argv.length; i++) {
     console.log(i, process.argv[i]);
}
```

- The `process` object, which we see above when accessing `process.argv`, is a global object available within a Node process. It holds other important properties about the running process, such as:
    - `process.env` – the environment variables available to the running process
    - `process.stdout`, `process.stderr`, and `process.stdin` – objects representing the process's standard output, error, and input streams
    - `process.exit()` – a method to exit the running process
    - `process.cwd()` – a method to access the current working directory of a process
    - Lots of others: https://nodejs.org/api/process.html

- A couple other important values available in all Node process are:
    - `__dirname` – a string containing the path of the directory in which the currently executing code resides
    - `__filename` – a string containing the path of the file in which the currently executing code resides
    - Both `__dirname` and `__filename` are actually local to each individual module (more on modules below), not global like `process`.

Modules

- Beyond some essentials, like `console`, `process`, `__dirname`, and `__filename`, which we saw above, there isn't much functionality available in the global scope in a Node application.
    - You can see all the globals here: https://nodejs.org/api/globals.html.

- Node has many other pieces of functionality available as importable ***modules***, which we can access using the global `require()` function.

- For example, we can import `fs`, Node's built-in module for working with the file system, like this:

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

- Then, we can use the methods of `fs`, e.g.:

```
fs.readFile(...);
```

- There are many important built-in modules available in Node:
    - `fs` – file system operations: https://nodejs.org/api/fs.html
    - `path` – utilities for working with file and directory paths: https://nodejs.org/api/path.html
    - `url` – utilities for working with URLs: https://nodejs.org/api/url.html
    - `http` – HTTP perver and client: https://nodejs.org/api/http.html
    - Lots of others: https://nodejs.org/api/

- We can also write our own modules, which can be imported using their path.

- Module functionality is specified (exported) using the `module.exports` property.

- For example, if I wanted to write a simple module to compute the circumference of a circle given its radius, I could write a file `circumference.js`, like this:

```
module.exports = function (r) {
     return 2 * Math.PI * r;
};
```

- Then if my module is stored in the same directory as my application file, I could use it like this:

```
var circumference = require('./circumference');
console.log("The circumference of a circle with
     radius 4 is:", circumference(4));
```

    - The path specified in require() is usually relative to the path of the current file, e.g.:
        - `require('./circumference')`
        - `require('./lib/circumference')`

NPM and 3rd-party modules

- There is also a huge collection of third-party modules available via the Node package manager (NPM).

- You can find these 3rd-party modules at https://www.npmjs.com/.

- Once you find a module you want to install, you can use the `npm` terminal command. Here, I'm installing `figlet`, a module to create ASCII art from text:

  ```
  npm install figlet
  ```

- I can import and use my installed modules with `require()`:

  ```
  var figlet = require('figlet');
  figlet('Hello world!', function (err, data) {
       if (!err) {
            console.log(data);
       }
  });
  ```

The `fs` module

- The `fs` module is one of a handful of very important built in Node modules. It allows us to interact with the filesystem of the running process.

- The `fs` module has lots of methods available. A few key ones are:

  - `fs.readFile()` – reads a file *asynchronously*. A callback function is used to provide handling functionality when the file is read:

    ```
    fs.readFile(
         '/path/to/file.txt',
         function (err, data) {
              if (err) {
                   throw err;
              }
              console.log("File contents:", data);
         });
    ```

  - `fs.readFileSync()` – reads a file *synchronously* (i.e. blocks until the file is read). Returns the contents of the file:

    ```
    var data =
         fs.readFileSync('/path/to/file.txt');
    ```

- `fs.writeFile()` – writes a file *asynchronously*. A callback function is used to provide handling functionality when the file is written:

  ```
  fs.writeFile(
       '/path/to/file.txt',
       "This will be the contents of the file.",
       function (err) {
            if (err) {
                 throw err;
            }
            console.log("The file is written.");
       });
  ```

- `fs.writeFileSync()` – writes a file *synchronously*. Analogous to `fs.readFileSync()`.

- `fs.readdir()` – reads the contents of a directory *asynchronously*. A callback function is used to provide handling functionality when the directory contents are read. They are provided as a list of filenames:

  ```
  fs.readdir(
       '/path/to/directory',
       function (err, files) {
            if (err) {
                 throw err;
            }
            console.log("Directory contents:",
                 files);
       });
  ```

- `fs.readdir()` – reads the contents of a directory *synchronously*.

Blocking vs. non-blocking

- The `fs` examples above illustrate the important difference between blocking and non-blocking operations.
  - Here, the term “blocking” is used synonymously with ***synchronous***, and the term “non-blocking” is used synonymously with ***asynchronous***.

            - This is not always the case.

- A ***blocking*** operation is one that causes the running process to wait until the operation's completion.
    - Here, the term "blocking" is used synonymously with ***synchronous***.

- A ***non-blocking*** operation is one where the running process can continue to do work while the operation completes in the background.
    - Here, the term "non-blocking" is used synonymously with ***asynchronous***.
    - In Node, non-blocking/asynchronous operations typically use a callback function to handle completion.

- In the `fs` examples we saw above, the methods whose names ended in `Sync` were blocking/synchronous methods.
    - Code after a call to these methods would not be executed until the call finished.

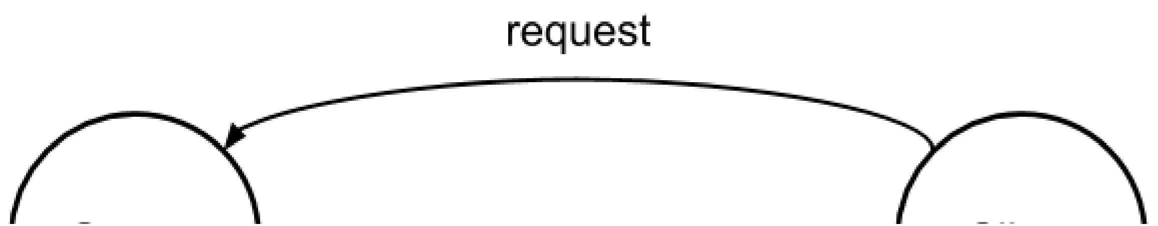
- The non-`Sync` methods above are non-blocking/asynchronous.
    - Code after a call to these methods would continue to be executed immediately while the non-blocking operation was performed in the background.
    - When the non-blocking operation completed, the execution of code would be interrupted, and the operation's callback would be executed.

- Execution of JS code in Node is ***single-threaded***, so using non-blocking operations allows us to increase throughput by allowing other code to execute while waiting for long-running non-JS operations, like file system or network I/O.

- This can be illustrated nicely with a kitchen analogy.

HTTP requests and responses

- HTTP is a request-response protocol. That means that a client (e.g. a web browser) submits a request message to a server, and the server, in turn, processes that request and returns a response message back to the client.



- Each HTTP request message contains of the following:
    - A ***request line*** specifying the location of the requested resource as a ***URI*** and an ***HTTP method*** indicating the desired action to take on the resource.
        - More on HTTP methods just below.
    - ***Request headers***, providing information about the request or the desired response.
    - An optional message body, containing info specifically related to the request (e.g. data to upload, form values, etc.).

- The URI (uniform resource identifier) that specifies the location of the requested resource and, when dealing with web servers, most commonly takes the form of an absolute path to that resource on a given server. This path is typically the last part of the URL from which the request is formed.

- For example, let's say a user requested the following URL:

  https://nodejs.org/api/http.html

  The absolute path corresponding to this URL, which would specified in the HTTP request to the nodejs.org server, would be this:

  ```
  /api/http.html
  ```

- There are several different HTTP methods that can be specified in a request. Some of the most important ones are:
    - ***GET*** – requests data from a specified resource.
    - ***POST*** – submits data to be processed to a specified resource.
    - ***PUT*** – uploads a representation of a specified resource.
    - ***DELETE*** – deletes a specified resource.
    - There are several others, too:
        - HEAD, TRACE, OPTIONS, CONNECT, PATCH.

- Each HTTP response message contains of the following:
    - A ***status line*** specifying an ***HTTP status code*** and a status message.
        - More on HTTP status codes in a bit.
    - Response headers, providing information about the response.

    - An optional message body, containing response data (e.g. the HTML of the requested web page, an image, etc.).

- There are lots of HTTP status codes. Some important ones are:
    - ***200*** – the standard response for a successful request.
    - ***3xx*** – a 3xx code indicates some kind of redirect.
    - ***4xx*** – a 4xx code indicates an error by the client, e.g.:
        - 401 – unauthorized
        - 403 – forbidden
        - 404 – not found
        - 405 – method not allowed
    - ***5xx*** – a 5xx code indicates an error by the server, e.g.:
        - 500 – internal server error

Node `http` module

- The Node `http` module is another very important module that allows us to build an HTTP server or to make HTTP requests.

- Let's look at how you might create a simple HTTP server using the `http` module.

- First, we'd need to import the `http` module to make use of its methods:

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

- We can create a basic server using the `http.createServer()` method, which takes a callback function describing how to process and respond to a single request:

```
var server = http.createServer(requestHandler);
```

    - We'll look at our `requestHandler()` function in just a bit.

- Once the server is created, we simply have to start it running on some port:

```
server.listen(8000);
```

- Let's look at what our `requestHandler()` function might look like. It is given two arguments: one representing an HTTP request and one representing our

HTTP response:

```
function requestHandler(request, response) {
      ...
}
```

- The `request` argument has several useful fields telling us about the HTTP request that was made:
    - `request.method` – a string indicating the HTTP method associated with the request (e.g. `'GET'`, `'POST'`, etc.)
    - `request.url` – a string indicating the URL of the requested resource, usually without the host or protocol (e.g. `'/index.html'` instead of `'http://www.host.com/index.html'`)
    - `request.headers` – an object containing the headers associated with the request, e.g.:

      ```
      {
            "Content-Type": "application/json",
            "Accept-Language": "en-US"
      }
      ```

    - Accessing the request body, if it exists, is more complicated. There are a few events we listen for (with callbacks) to handle body data:
        - `request.on('error', function(err) {})`
        - `request.on('data', function(data) {})`
            - You can use this function to concatenate data as it arrives into an appropriate data structure.
        - `request.on('end', function() {})`

- The `response` argument lets us respond to the request:
    - `response.statusCode` – set this value to an HTTP status code to indicate to the user what happened to their request
    - `response.setHeader(name, value)` – call this method to set the value of a single header field in the response, e.g.:

      ```
      response.setHeader("Content-Type",
           "application/json");
      ```

    - `response.writeHead(status, headers)` – you can use this method to immediately write an HTTP status and an entire set of headers into the response stream (instead of relying on Node to send them at the appropriate time when you use `statusCode` and `setHeader()`):

      ```
      response.writeHead(200, {
           "Content-Type": "application/json"
      });
      ```

    - `response.write(data)` – this method writes some raw data (e.g. HTML strings) to the response stream. You can call it multiple times to write multiple pieces of data, e.g.:

      ```
      response.write("<html>");
      response.write("<body>");
      response.write("<h1>Hello World!</h1>");
      response.write("</body>");
      response.write("</html>");
      ```

    - `response.end([data])` – this method ends the response stream and finishes sending it to the requesting client. You can send some last piece of data using this function.

    - `response.on('error', function (err) {})` – you can use a callback on the response's `'error'` event to handle response-related errors.
- So, our requestHandler() function might look something like this:

  ```
  function requestHandler(request, response) {
       response.statusCode = 200;
       response.setHeader("Content-Type", "text/html");
       response.write("<html>");
       response.write("<body>");
       response.write("<h1>Hello!</h1>");
       response.write("<p>");
       response.write("You requested: " + request.url);
       response.write("</p>");
       response.write("</body>");
       response.write("</html>");
  ```

```
     response.end();
}
```

- A whole Hello World server might look like this:

```
var http = require("http");
var server = http.createServer(function (req, resp) {
     response.statusCode = 200;
     response.setHeader("Content-Type", "text/html");
     response.write("<html>");
     response.write("<body>");
     response.write("<h1>Hello!</h1>");
     response.write("<p>");
     response.write("You requested: " + request.url);
     response.write("</p>");
     response.write("</body>");
     response.write("</html>");
     response.end();
});

server.listen(8000);
```