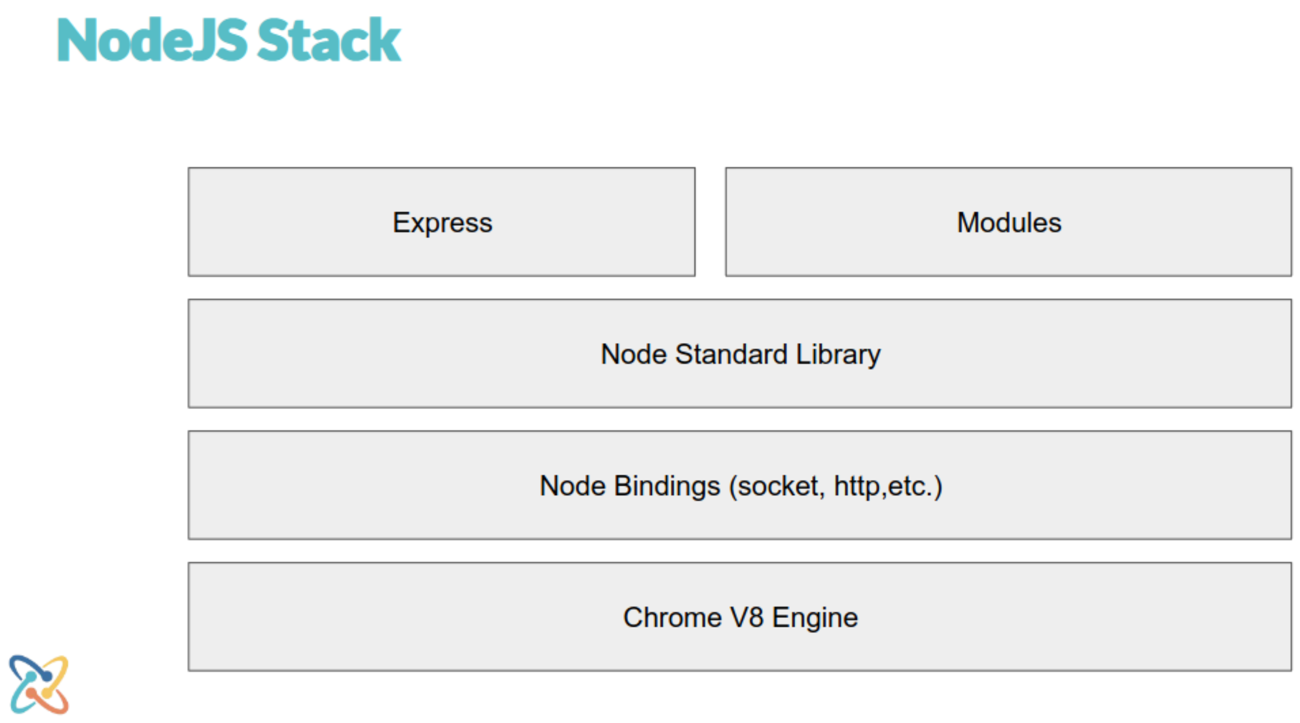
Intro to NodeJS | JavaScript on the Server

What is NodeJS?

- JavaScript on the server
  - Linux, Windows, Mac
- Utilizes the Chrome V8 Engine
  - Chrome V8 Engine Wikipedia
- Node executes JavaScript directly without a browser
  - No access to the DOM!
- Sits on top of the operating system and provides a common set of modules that run anywhere (the dream finally realized!!)

NodeJS Stack



Starting Node

```
$ node file.js
```

```
$ node
> const test = "Hello World!";
> console.log(test);
```

Core Modules

- Prepackaged code that performs certain functions/tasks
- Common Modules
  - **path** - manipulate paths to files
  - **url** - manipulate and parse URLs
  - **http** - make HTTP requests and listen on ports
  - **fs** - read and write files on the disk
  - Several more - Node.js Modules

Using Modules (path)

```
const path = require("path");
path.extname("README.md"); // Returns '.md'
path.dirname("images/sky.png"); // Returns 'images'
```

- `require` - imports packaged code into your variable name
    - Node looks in node_modules or core for the package name
- `import` - can be used instead of `require` when using babel or TypeScript
    - `import * as path from 'path'`

File Paths in NodeJS

- `__dirname`
    - Global variable in node that points to the currently executing script's directory
    - If you ran node from /Users/you/Source, `__dirname` would be equal to the absolute path
- Node expects absolute paths
    - The path module helps construct paths
- `path.join`
    - Constructs absolute paths out of parts

```
path.join(__dirname, "README.md");
path.dirname(__dirname, "../images/sky.png");
```

Core Module: fs

- Used to access the file system

```
const fs = require("fs");
fs.readFile(path.join(__dirname, "data.json"), function (err, data) {
    if (err) return console.log(err);
    console.log(data.toString());
});
```

Core Module: http

- Used to make HTTP requests and host HTTP servers
    - We're going to use ExpressJS which uses http

```
const http = require("http");
const server = http.createServer(function (req, res) {
    // This function gets called when the server gets a request
    // req holds request data | res is for composing a response
    // req.method shows request method (GET, POST, PUT, DELETE)
});
server.listen(process.env.port);
```

Core Module: url

```
const url = require("url"); // Example for http://covalence.io/full-stack?p=1
http.createServer(function (req, res) {
    req.url; // Full URL of the request
    const uData = url.parse(req.url, true);
    // uData -> Convenient object for accessing URL info
    uData.query.p; // '1'
    uData.pathname; // '/full-stack'
});
```

Node Package Manager (npm)

- Pre-installed with NodeJS
- Command Line: **npm**
    - Used to maintain packages for your project
    - Downloads from public repository at npmjs.com
- `npm init`
    - Initializes a Node project and creates a `package.json`

package.json

- `npm init` - creates a `package.json` in the current directory
- `package.json` should exist in the root of your project

- Contains information about your project structure
  - Node modules required for production and development
    - dependencies and devDependencies
  - Scripts and main entry point
  - Name, version, description, author, license

Installing Modules

```
npm install request --save
```

- Downloads the module from npmjs.com to `node_modules`
- Adds the module to your `package.json`
- Can be used just like core modules once installed
  - `const request = require('request')`
- Some modules can/must be installed globally with the `-g` flag
  - Not added to `package.json`
  - CLI commands can be accessed from any path

Readline

```
 const readline = require("readline");
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
});
rl.question("What do you think of Node.js? ", (answer) => {
    console.log(`Thank you for your valuable feedback: ${answer}`);
    rl.close();
});
```

Node Project Setup

What does `npm init` do for us?

As the `init` command hints at, it *initializes* a new project for us.

The first thing is prompts us for is a *package name*. This field is something that identifies your package on npmjs.com (in case you were to upload it to NPM, it would have to be unique across *all packages* there, but otherwise you're free to name it whatever on your own computer). By default, it will take your folder name as the *package name* field. *Entry point* describes your starting file, like `index.js` or `server/server.js` . Some of the other fields there are also more relevant to a public package that you would upload, so don't fret about them too much! As far as software licenses go, if you wanted to look more in depth there's a website ChooseALicense.com that allows you to get a TL;DR of what some of the more common software licenses mean.

All those steps can be a good bit of work, so if you don't want to change any of the default options, you can also run `npm init -y` , where the `-y` flag answers *yes* to all the default options automatically.

After all those steps, node will have generated the `package.json` for you which has the same structure and function as your previous labs, although it's a little more basic currently.

<Lightning exercise>

The `scripts` section in your `package.json` is where node will look whenever you say `npm run XYZ` . Try adding a line in there (ensuring that the previous line has a comma after it since JSON is a very rigid standard) saying `"donde-estoy": "pwd"` . `pwd` is the **p**rint **w**orking **d**irectory command, AKA logging out where that command is currently executing from. After that, your scripts section should look something like this. Try running it now with `npm run donde-estoy` !

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "donde-estoy": "pwd"
},
```

</Lightning exercise>

Making a real project with Node

We're going to be using the Studio Ghibli API again and interacting with the data - but unlike changes we made in our browser code that are lost whenever we refresh the data - we'll be saving our manipulated data to your filesystem using `fs` and `path`, so our changes are now *persistent*!

In years past, Node didn't have a convenient way of fetching data from an API, and you'd either have to use the built-in `https` module to make some slightly more verbose code, or you'd have to download a 3rd party library like `Axios`, `request`, `node-fetch`, or a handful of others to get some easier and shorter code. Thankfully as of Node v19 or higher, they have since implemented `fetch` in a way standard to how it's used in the browser, so you can write some familiar code to grab the data *and* now have the ability to do cool stuff with it via Node (like saving it to a file).