The New York Code + Design Academy

MODULES AND PACKAGES

Agenda

- What are Node Modules
  - How do we use them?
- Node Packages
  - How do we get them?

Node Modules

What Are Node Modules?

- In order to offer code to other developers, and use other developers code, we package code in to modules
- Modules are a way of exposing well defined functionality to other parts of our code for reuse and organization
- Modules can be as small as a single variable, or as large as a multi-function library
    - jQuery is, in fact, a module!

Why do we use them?

- We can reuse the code without having to copy and past ie
- It's easier to maintain small files with simple code, than multi-hundred line huge JS files
- We avoid overwriting the names of other pieces of code on accident
- We explicitly define our dependencies (what code we need to function)

Core Modules

Node comes with certain modules built in for our use, which are sometimes referred to as **Core Modules**.

- `fs` - Stands for File System. Includes events, classes and functions needed to work with file input/output.
- `path` - Includes functions for working with file and directory paths (Commonly used with `fs`.)
- `util` - Includes functions that assist the developer with debugging, inspecting and deprecating code.
- `http, url, querystring` - Includes events, classes and functions needed to create an HTTP server and parse the input.

External Modules

- External modules are sometimes referred to as **Third Party Modules**
- These are modules created by other developers which they make publicly available
- These modules can save us a lot of time by not having to reinvent the wheel

Custom Modules

- Custom modules are the ones we write for ourselves
- They are usually specific to a developers app
- This can as simple as a greeting module

```
// module.js
    console.log('Hello Class!');

// app.js
    require('./module.js');
```

Using Modules

`require()`

- `require()` is a function available in node that points to the path of a module that will be used
- If a path is specified, `require` will traverse to it and look for the module there
- `require` supports relative and absolute paths by default

Importing a Module

- Here, the first `require` looks for a file called `relative.js`, located in the same directory as the current file
- The second `require` looks for a file on an absolute path
- Neither need to specify a `.js` extension, since that's assumed in Node

```
var relative = require('./relative');
var absolute = require('/some/absolute/path/absolute');
```

Exporting a Module

- Whatever `module.exports` is set to define, is what is available when including a module is your app
- `module.exports` is returned to the requiring file
- `exports` collects properties, ultimately attaching them to `module.exports`

```
//module_two.js, exports a simple string 'hello'
module.exports = 'hello';

//app_two.js, prints out 'hello'
var b = require('./module_two')
console.log(b)
```

Exporting a Module (cont.)

- Exports can be *any* named thing or value, including functions or classes

```
// hello.js
function helloWorld() {
    return "Hello World!";
};

module.exports = helloWorld;

// world.js
class World {
  hello() {
    return "Hello World!";
  }
}

module.exports = World;
```

Multiple Functions Example

- When you want to export multiple things, you can export an object with everything accessible by specific named keys

```
// hello.js - Export a "namespace" that
// has helloWorld and helloPerson functions
module.exports = {
    helloWorld: function() {
    return "Hello World!";
  },
    helloPerson: function(name) {
    return "Hello " + name + "!";
  },
};

// app.js, use hello.js
var Hello = require("./hello.js");
Hello.helloWorld();
Hello.helloPerson("Will");
```

Initialize Function Module

- Exports that require parameters for initialization can take in arguments
- Just export a function that returns the real export, and specify arguments

```
// teller.js, exports a function that must be called to initialize
var info;
function infoTeller() {
  console.log("You initialized me with " + info);
};

module.exports = function(args) {
    info = args;
    return infoTeller;
};

// app.js, call the initializer
var teller = require('./teller')("It's a beautiful day.");
teller();
```

Exercise: Modularize Your String Descender

Take your string descender from the Introduction to Node assignment, and turn it in to a module. It should export a `function` that takes in a `starCount` number argument, and prints that many stars in that many seconds. For example:

```
// Import your starPrinter function module
var starPrinter = require("./star-printer.js");

// Have it print 10 stars over 10 seconds, identical
// to the output of the previous assignment. Experiment
// with different numbers to make sure it works!
starPrinter(10);
```

Be sure to re-push your assignment in this new module style.