Web Frame Work Using JavaScript

Vivek Sharma

The Course Break Down:

- Class 1: Mean Stack Fundamentals.
- Class 2: Learning Node.js
- Class 3: Implementing HTTP Services in Node.js
- Class 4: Scaling the Applications Using Node.js
- Class 5: Understanding NoSql and MongoDB
- Class 6: Understanding Express and Implementation.
- Class 7: Understanding Angular
- Class 8: Understanding the Angular Directives and Angular Web Application.
- Class 9: Creating a Shopping Cart.
- Class 10: Creating another project.

WELCOME

Some Points before we start off:

- Introduction.
- Grading/ Assignments.
- About the Class. [Basic to Professional]
- Pace.
- Interview Preparation.
- Labs.

Today's Overview

1. Core Node JS
2. Adding Workflow items to Node.js
3. Considerations for the Mobile Development.
4. Types of Mobile Apps

Learning Node

Node.js File-Based Module System

- Kevin Dongaoor created CommonJS in 2009 with the goal to specify an ecosystem for JavaScript modules on the server. Node.js follows the CommonJS module specification. Following are a few salient points of the module system:
  1. Each file is its own module.
  2. Each file has access to the current module definition using the module variable.
  3. The export of the current module is determined by the module.exports variable.
  4. To import a module, use the globally available require function.

  Lets look at the example. Open the intro folder.

Node.js require Function

- The Node.js require function is the main way of importing a module into the current file. There are three kinds of modules in Node.js: core modules, file modules, and external node_modules, all of which use the require function. We are discussing file modules at the moment.
- When we make a require call with a relative path—for example, something like require('./filename') or require('../foldername/filename')—Node.js runs the destination JavaScript file in a new scope and returns whatever was the final value for module.exports in that file. This is the basis of file modules.
- The Ramficiations of such a design approach are the following :
  - Node.js Is Safe:
  - Conditionally Load a Module
  - Shared State
  - Object Factories

Node.js is Secure

- Modules in many programming environments are not safe and pollute the global scope. A simple example of this is PHP. Say you have a file foo.php that simply defines a function foo, as shown:

```
function foo($something){
        return $something;
}
```

- If you want to reuse this function in a file bar.php, you can simply include foo.php using the include function, and then everything from the file foo.php becomes a part of the (global) scope of bar.php. This allows you to use the function foo, as shown :

```
include('foo.php');
foo();
```

- This design has quite a few negative implications. For example, what a variable foo means in a current file may change based on what you import. As a result, you cannot safely include two files, foo1 and foo2, if there is a chance that they have some variable with the same name. Additionally, everything gets imported, so you cannot have local only variables in a module. You can overcome this in PHP using namespaces, but Node.js avoids the potential of namespace pollution altogether.
- Using the require function only gives you the module.exports variable, and you need to assign the result to a variable locally in order to use it in scope, as shown:

```
var yourChoiceOfLocalName = require('./foo');
```

- There is no accidental global scope—there are explicit names and files with similar internal local variable names that can coexist peacefully.

You can conditionally Load a Module

- **require** behaves just like any other function in JavaScript. It has no special properties. This means that you can choose to call it based on some condition and therefore load the module only if you need it, as :

```
if(iReallyNeedThisModule){
    var foo = require('./foo');
}
```

- This allows you to lazy load a module only on first use, based on your requirements.

Blocking

- The require function blocks further code execution until the module has been loaded. This means that the code following the require call is not executed until the module has been loaded and executed. This allows you to avoid providing an unnecessary callback like you need to do for all async I/O in Node.js.
- The code snippet below Demonstrate That Modules Are Loaded Synchronously

```
// Blocks execution till module is loaded
var foo = require('./foo');

// Continue execution after it is loaded
console.log('loaded foo');
foo();
```

Cached:

- As you may recall, reading something from the file system is an order of magnitude slower than reading it from RAM.
- Hence, after the first time a require call is made to a particular file, the module.exports is cached. The next time a call is made to require that resolves to the same file (in other words, it does not matter what the original relative file path passed to the require call is as long as the destination file is the same), the module.exports variable of the destination file is returned from memory, keeping things fast.
- Let us open : intro/cached/bar.js

```
var t1 = new Date().getTime();
var foo1 = require('./foo');
console.log(new Date().getTime() - t1); // > 0

var t2 = new Date().getTime();
var foo2 = require('./foo');
console.log(new Date().getTime() - t2); // approx 0
```

Shared State

- Having some mechanism to share state between modules is useful in various contexts.
- Since modules are cached, every module that require's foo.js will get the same (mutable) object if we return an object foo from a module foo.js. The following example demonstrates this process with a simple example:
  1. We export an object: [intro/shared/foo.js]

```
module.exports = {
   something: 123
};
```

  2. This object is modified in app.js : [intro/shared/app.js]

```
var foo = require('./foo');
console.log('initial something:', foo.something); // 123
// Now modify something:
foo.something = 456;
// Now load bar:
var bas = require('./bar');
```

  3. This modification affects what is returned by require in bar.js : [intro/shared/bar.js]

```
var foo = require('./foo');
console.log('in another module:', foo.something); // 456
```

  4. This allows you to share in-memory objects between modules that are useful for things like using modules for configuration.

Object Factories

- As we have shown, the same object is returned each time a require call resolves to the same file in a Node.js process. If you want some form of new object creation mechanism for each require function call, you can export a function from the source module that returns a new object. Then require the module at your destination and call this imported function to create a new object.
- In the following example we export a function and then use this function to create a new object [intro/factory/foo.js]

```
module.exports = function () {
    return {
        something: 123
    };
};
```

- Then we import the module in the app.js [intro/factory/app.js]

```
var foo = require('./foo');
// create a new object
var obj = foo();
// use it
console.log(obj.something); // 123
```

- Note that you can even do this in one step (in other words, require('./foo')();)

Node.js Exports

module.exports

- As stated earlier, each file in Node.js is a module.
- The items that we intend to export from a module should be attached to the module.exports variable. It is important to note that module.exports is already defined to be a new empty object in every file.
- That is, module.exports = {} is implicitly present. By default, every module exports an empty object, in other words, {}. [intro/module.exports/app.js]

  console.log(module.exports); // {}

Exports Alias:

- If your use case demands to export more than one variable from the module, there are more than one way to do that as explained in the points below:
  1. Create a new object literal and assign that to module.exports. [intro/exports/foo1.js].
  2. Create and attach the objects that we want to export inline to the modules.export. [intro/exports/foo2.js]
  3. Creating an alias for module.exports called exports so instead of typing module.exports.something every time, you can simply use exports.something. [intro/exports/foo3.js].
  4. It is important to note that exports is just like any other JavaScript variable; Node.js simply does exports = module.exports for us. If we add something for example, foo to exports, that is exports.foo = 123, we are effectively doing module.exports.foo = 123 since JavaScript variables are references, as discussed in Chapter 2.
  5. However, if you do exports = 123, you break the reference to module.exports; that is, exports no longer points to module.exports. Also, it does not make module.exports = 123. Therefore, it is very important to know that you should only use the exports alias to attach stuff and not assign stuff to it directly. If you want to assign a single export, use module.exports = as we have been doing until this section.

- ***Exports Alias:***
- If your use case demands to export more than one variable from the module, there are more than one way to do that as explained in the points below:
- Create a new object literal and assign that to module.exports. [intro/exports/foo1.js].
- Create and attach the objects that we want to export inline to the modules.export. [intro/exports/foo2.js]
- Creating an alias for module.exports called exports so instead of typing module.exports.something every time, you can simply use exports.something. [intro/exports/foo3.js].
    1. It is important to note that exports is just like any other JavaScript variable; Node.js simply does exports = module.exports for us. If we add something for example, foo to exports, that is exports.foo = 123, we are effectively doing module.exports.foo = 123 since JavaScript variables are references, as discussed repeatedly.
    2. However, if you do exports = 123, you break the reference to module.exports; that is, exports no longer points to module.exports. Also, it does not make module.exports = 123. Therefore, it is very important to know that you should only use the exports alias to attach stuff and not assign stuff to it directly. If you want to assign a single export, use module.exports = as we have been doing until this section.
- All of these methods are equivalent from consumption (import) point of view.[intro/exports/app.js]

Modules Best Practices

- Following are the best practices rpescribed by the active Node.js community:
  1. Do Not Use the .js Extension.

     It is better to do require('./foo') instead of require('./foo.js') even though both work fine for Node.js. For browser-based module systems (such as RequireJS, which we look at later in this chapter), it is assumed that you do not provide the .js extension since we cannot look at the server filesystem to see what you meant.
  2. Relative Paths

     When using file-based modules, you need to use relative paths (in other words, do require('./foo') instead of require('foo')).

     Reason: Non-relative paths are reserved for core modules and node_modules. We discuss core modules in this chapter and node_modules in the next chapter.
  3. Utilize exports

     Try and use the exports alias when you want to export more than one thing.

     Reason: It keeps what is exported close to its definition. It is also conventional to have a local variable for each thing you export so that you can easily use it locally.

     var foo = exports.foo = / whatever you want to export as `foo` from this module */ ;*
  4. Export an Entire Folder

     If you have too many modules that go together that you keep importing into other files, try to avoid repeating the import. See what I meant in the notes section.

Important Globals

- Node.js provides a fair number of globally available utility variables. Following are the rypes of these:
  - Some of these variables are true globals (shared between all modules) (For example the require funciton)
  - Some are local globals (variables specific to the current module) (For example module (used by module.exports) and exports.
- Let us examine a few more important globals.
- Now we would be looking at the following:
  1. console.
  2. Timers.
  3. __filename and __dirname.
  4. Process
  5. Buffer
  6. global

- console: Arguable one of the most important of the global variables.
  - It is all the more significant since it is so easy to start and restart a Node.js application from the command line, the console plays an important part in quickly showing what is happening in your application when you need to debug it.

- Timers:
  1. setTimeOut is Used to delay the processing of a function.
  2. This delay is the minimum duration after which the function is called but you cannot be sure that it would called exactly after the duration specified.
  3. The actual delay depends upon the the availability of the JS thread.
  4. It also depends upon when the operating system schedules the Node.js process to execute (normally not an issue). A quick example of setTimeout, which calls a function after 1,000 milliseconds (in other words, one second). [*globals/timers/setTimeout.js*]
  5. Also we have the setInterval function which is used to repeatedly call a function after the specified period of time. Just like the setTimeOut the actual duration may vary.[globals/timers/setInterval.js].
  6. Both setTimeout and setInterval return an object that can be used to clear the timeout/interval using the clearTimeout/clearInterval functions. [*globals/timers/clearInterval.js*].

- __filename and __dirname:
  - Available in each file , these variables give you the full path to the file and directory for the current module.
  - This means that they include everything right up to the root of the current drive this file resides on. [globals/fileAndDir/app.js]

process:

1. One of the most important globals.
2. Has a lot of important and useful attributes and methods.
3. it is a source of a few critical events.

1. Command Line Arguments

1. Since Node.js does not have a main function in the traditional C/C++/JAVA/C# sense, you use the process object to access the command line arguments.
2. The arguments are available as the process.argv member property, which is an array. The first element is node (that is, the node executable), the second element is the name of the JavaScript file passed into Node.js to start the process, and the remaining elements are the command line arguments. [globals/process/argv.js]
3. It also has an access to the standard I/O pipes for the process stdin, stdout, and stderr. stdin is the standard input pipe for the process, which is typically the console. [globals/process/stdin.js].
4. The stdout and stderr attributes of the process module are Writable streams that can be treated accordingly.

2. process.nextTick:

process.nextTick is a simple function that takes a callback function. It is used to put the callback into the next cycle of the Node.js event loop. It is designed to be highly efficient, and it is used by a number of Node.js core libraries. [globals/process/nexttick.js].

3. Buffers:

JS is great for the UNICODE strings, but what about the TCP streams and the file system. Well to accommodate those the NODE JS developers added the support using the native and fast support to handle binary data using by adding the BUFFER class which also designed to be a global.

Our main interaction with buffer will most likely be in the form of converting Buffer instances to string or strings to Buffer instances.

In order to do either of these conversions, you need to need to tell the Buffer class about what each character means in terms of bytes.

This information is called character encoding. Node.js supports all the popular encoding formats like ASCII, UTF-8, and UTF-16.

Converting strings to buffers is really simple. You just call the Buffer class constructor passing in a string and an encoding. Converting a Buffer instance to a string is just as simple. You call the Buffer instance's toString method passing in an encoding scheme. [globals/buffer/buffer.js].

2. global:

The variable global is our handle to the global namespace in Node.js. If you are familiar with front-end JavaScript development, this is somewhat similar to the window object. All the true globals we have seen (console, setTimeout, and process) are members of the global variable.

You can even add members to the global variable to make it available everywhere. [globals/global/addToGlobal.js].

Even though adding a member to global is something that you can do, it is strongly discouraged. The reason is that it makes it extremely difficult to know where a particular variable is coming from. The module system is designed to make it easy to analyze and maintain large codebases. Having globals all over the place is not maintainable, scalable, or reusable without risk. It is, however, useful to know the fact that it can be done and, more importantly, as a library developer you can extend Node.js any way you like.

Core Modules

- The Node.js design philosophy is to ship with a few battle-tested core modules and let the community build on these to provide advanced functionality.

Consuming Core Modules

- Very similar to consuming file-based modules that you write yourself. You still use the require function. The only difference is that instead of a relative path to the file, you simply specify the name of the module to the require function.
- For example, to consume the core path module, you write a require statement like var path = require('path'). As with file-based modules, there is no implicit global namespace pollution and what you get is a local variable that you name yourself to access the contents of the module.
- For example, in var path = require('path') we are storing it in a local variable called path.

- In the next few slides let us look into some of the Core Modules.

Path Module

- Use require('path') to load this module.
- The path module exports functions that provide useful string transformations common when working with the file system. The key motivation for using the path module is to remove inconsistencies in handling file system paths. For example, path.join uses the forward slash `/` on UNIX-based systems like Mac OS X vs. backward slash `\` on Windows systems. Here is a quick discussion and sample of a few of the more useful functions.

path.normalize(str)

- This function fixes up slashes to be OS specific, takes care of . and .. in the path, and also removes duplicate slashes.[core/path/normalize.js].

path.join([str1], [str2], …)

- This function joins any number of paths together, taking into account the operating system.[core/path/join.js].

dirname, basename, and extname:

1. path.dirname gives you the directory portion of a specific path string (OS independent),
2. path.basename gives you the name of the file.
3. path.extname gives you the file extension. [core/path/dir_base_ext.js]

fs Module

- The fs module provides access to the filesystem. Use require('fs') to load this module. The fs module has functions for renaming files, deleting files, reading files, and writing to files. A simple example to write to the file system and read from the file system.[core/fs/create.js].
- One of the great things about the fs module is that it has asynchronous as well as synchronous functions (using the -Sync postfix) for dealing with the file system. As an example, to delete a file you can use unlink or unlinkSync.
- A synchronous version [core/fs/deleteSync.js].
- An asynchronus version [core/fs/delete.js]
- The main difference is that the async version takes a callback and is passed the error object if there is one.
- We also saw that accessing the file system is an order of magnitude slower than accessing RAM. Accessing the filesystem synchronously blocks the JavaScript thread until the request is complete. It is better to use the asynchronous functions whenever possible in busy processes such as in a web server scenario.
- More information about the fs module can be found online in the official Node.js documentation (http://nodejs.org/api/fs.html).

os Module

- The os module provides a few basic (but vital) operating-system related utility functions and properties. You can access it using a require('os') call.
- For example, if we want to know the current system memory usage, we can use os.totalmem() and os.freemem() functions. [core/os/memory.js]

```
var os = require('os');
var gigaByte = 1 / (Math.pow(1024, 3));
console.log('Total Memory', os.totalmem() * gigaByte, 'GBs');
console.log('Available Memory', os.freemem() * gigaByte, 'GBs');
console.log('Percent consumed', 100 * (1 - os.freemem() / os.totalmem()));
```

- A vital facility provided by the os module is information about the number of CPUs available [core/os/cpus.js]

```
var os = require('os');
console.log('This machine has', os.cpus().length, 'CPUs');
```

- We will learn how to take advantage of this fact when we discuss scalability.

util Module

- The util module contains a number of useful functions that are general purpose. You can access the util module using a require('util') call. To log out something to the console with a timestamp, you can use the util.log function, [/util/log.js]

```
var util = require('util');
util.log('sample message'); // 27 Apr 18:00:35 - sample message
```

- Another extremely useful feature is string formatting using the util.format function. This function is similar to the C/C++ printf function. The first argument is a string that contains zero or more placeholders. Each placeholder is then replaced using the remaining arguments based on the meaning of the placeholder. Popular placeholders are %s (used for strings) and %d (used for numbers) [core/util/format.js]

```
var util = require('util');
var name = 'Vivek';
var money = 33;
//prints: Vivek has 33 dollars
console.log(util.format('%s has %d dollars', name, money));
```

- Additionally, util has a few functions to check if something is of a particular type (isArray, isDate, isError). [core/util/isType.js]

Node.js in the Browser.

- Since we now know that Nodejs is so modular we can take that modular approach to develop the browser code too using the NODE.js.
- But before we do that we need to understand the need for AMD and what differentiates it from CommonJS.

Introducing AMD

- As we discussed in the beginning of this chapter, Node.js follows the CommonJS module specification. This module system is great for the server environment when we have immediate access to the file system. We discussed that loading a module from the file system in Node.js is a blocking call for the first time.
- Consider the simple case of loading two modules that Show Loading Two Modules Using CommonJS

```
var foo = require('./foo');
var bar = require('./bar');
// continue code here
```

- In this example bar.js is not parsed until all of foo.js has been loaded. In fact, Node.js doesn't even know that you will need bar.js until foo.js is loaded and the line require('./bar') is parsed. This behavior is acceptable in a server environment where it is considered a part of the bootstrap process for your application. You mostly require things when starting your server and afterward these are returned from memory.
- However, if the same module system is used in the browser, each require statement would need to trigger an HTTP request to the server. This is an order of magnitude slower and less reliable than a file system access call. Loading a large number of modules can quickly degrade the user experience in the browser. The solution is async, in-parallel, and upfront loading of modules. To support this async loading, we need a way to declare that this file will depend upon ./foo and ./bar upfront and continue code execution using a callback. There is already a specification for exactly this called async module definition (AMD).

```
define(['./foo', './bar'], function(foo, bar){
    // continue code here
});
```

- The define function is not native to the browser. These must be provided by a third-party library. The most popular of these for the browser is RequireJS (http://requirejs.org/).

Setting Up RequireJS

- Since we need to serve HTML and JavaScript to a web browser, we need to create a basic web server. We will be using Chrome as our browser of choice as it is available on all platforms and has excellent developer tools support. The source code for this sample is available in the [/amd/base folder].

Starting the Web Server

- We will be using server.js, which is a very basic HTTP web server that we will write ourselves in coming seesions .
- Start the server using Node.js (node server.js). The server will start listening for incoming requests from the browser on port 3000.
- If you visit http://localhost:3000, the server will try to serve index.html from the same folder as server.js if it is available.

Download RequireJS

- http://requirejs.org/docs/download.html.
- It is a simple JavaScript file that you can include in your project. It is already present in [/amd/base folder].

Bootstrapping RequireJS

- Create a simple index.html in the same folder as [amd/base/index.html]

```
<html>
<script
  src="./require.js"
  data-main="./client/app">
</script>
<body>
  <p>Press Ctrl + Shift + J (Windows) or Cmd + Opt + J (MacOSX) to open up the console</p>
</body>
</html>
```

- We have a simple script tag to load require.js. When RequireJS loads, it looks at the data-main attribute on the script tag that loaded RequireJS and considers that as the application entry point. In our example, we set the data-main attribute to ./client/app and therefore RequireJS will try and load http://localhost:3000/client/app.js.

Client-Side Application Entry Point

- As we set up RequireJS to load /client/app.js, let's create a client folder and an app.js inside that folder that simply logs out something to the [amd/base/client/app.js]

```
console.log('Hello requirejs!');
```

- Now if you open up the browser http://localhost:3000 and open the dev tools (press F12), you should see the message logged to the console.

Working with AMD

- Now that we know how to start a RequireJS browser application, let's see how we can import/export variables in modules. We will create three modules: app.js, foo.js, and bar.js. We will use foo.js and bar.js from app.js using AMD. This demo is available in chapter3/amd/play folder.
- To export something from a module, you can simply return it from the define callback. For example, let's create a file foo.js that exports a simple function [amd/play/client/foo.js]
- To be upfront about all the modules we need in a file, the root of the file contains a call to define. To load modules ./foo and ./bar in app.js in the same folder [amd/play/client/app.js].
- *define* can take a special argument called exports, which behaves similar to the exports variable in Node.js. Let's create the module bar.js using this syntax [amd/play/client/bar.js].
- Now, let's complete app.js and consume both of these modules [amd/play/client/app.js].

Interesting Facts:

- Modules are cached. This is similar to how modules are cached in Node.js—that is, the same object is returned every time.
- Many of these arguments to define are optional and there are various ways to configure how modules are scanned in RequireJS.
- You can still do conditional loading of specific modules using a require call, which is another function provided by RequireJS as shown in the snippet below:

```
define(['./foo', './bar'], function(foo, bar){
    if(iReallyNeedThisModule){
        require(['./bas'], function(bas){
            // continue code here.
        });
    }
});
```

The objective here was to give a quick overview of how you can use RequireJS and understand that the browser is different from Node.js.

Porting Node.js code to the Browser

- As you can see, there are significant differences between the browser module systems (AMD) and the Node.js module system (CommonJS).
- However, the good news is that the Node.js community has developed a number of tools to take your CommonJS / Node.js code and transform it to be AMD / RequireJS compatible.
- The most commonly used one (and the one on which other tools rely) is Browserify (http://browserify.org/).
- Browserify is a command line tool that is available as an NPM module.
- To install Browserify as on the command line tool, simply execute the command *sudo npm install –g browserify. [This installs Browserify globally (a concept that will become clear in the next chapter) and makes it further available on the command line.]*

Browserify Demo:

For the demo we will create a few simple Node.js modules and then use Browserify to convert them to AMD syntax and run them in the browser. All of the code for this example is present in the [/amd/browserify folder].

We create three files:

1. foo.js [amd/browserify/node/foo.js]
2. bar.js [amd/browserify/node/bar.js]
3. app.js [amd/browserify/node/app.js]

All these files are coded in the Node.js/CommonJS specification and you can run app.js using the node.js to verify that it runs perfectly.

Now using bowserify we would convert this code so that it is an AMD compatible module. Command:

browserify app.js -o amdmodule.js

This takes app.js and all its dependencies (foo.js and bar.js) and converts them into a single AMD compatible module amdmodule.js in the same folder

As a final step, we simply load this module from our client app.js to show that it works in the browser. [amd/browserify/client/app.js].

One thing to note is that it is **not** possible to convert every Node.js module into a browser module. Specifically, Node.js modules that depend on features only available on the server (such as the file system) will not work in the browser.

Browserify has a lot of options and is also able to navigate NPM packages (node_modules). You can learn more about Browserify online at http://browserify.org/.

Summary

- Define your challenges
  - Technological as well as personal
- Set realistic expectation
  - Mastery is not achieved overnight
- Keep your eye on the goal
  - Mentorship programs

Resources

- <Intranet site text here>
  <hyperlink here>
- <Additional reading material text here>
  <hyperlink here>
- This slide deck and related resources:
  <hyperlink here>

QUESTIONS?

APPENDIX