CIRCUIT STREAM

Software Development Bootcamp

Unit 3: Backend Development

Lesson 2: Modules, NPM & GitHub Workflow

Randy Gulak

Agenda

- Recap (Lesson 1: Node.js Intro, NPM Install, First GitHub Push)
- Section 1: Node.js Built-in Modules (`path`, `fs`)
- Section 2: Creating Your Own Modules (`module.exports`, `require`)
- Section 3: NPM & `package.json` Deep Dive
- Section 4: GitHub Workflow (Pushing Updates)
- Summary & Key Takeaways
- Next Steps & Preview

Learning Objectives

By the end of this class, you will be able to:

- **Utilize** common Node.js built-in modules like `path` and `fs`.
- **Create** custom JavaScript modules using `module.exports`.
- **Import** custom modules into other files using `require()`.
- **Explain** the key parts of `package.json` (dependencies, scripts).
- **Differentiate** between dependencies and devDependencies.
- **Use** `npm run` to execute scripts defined in `package.json`.
- **Push** subsequent code changes to a remote GitHub repository.

Recap: Lesson 1 Key Points

- **Node.js:** Runs JavaScript outside the browser, on the server.
- **CLI Basics:** Navigating (`cd`), listing (`ls`), running scripts (`node app.js`).
- **NPM:** Node Package Manager.
  - `npm init -y`: Created `package.json`.
  - `npm install <package>`: Installed external modules (like `lodash`) into `node_modules`.
- **Modules Intro:** Used `require('lodash')` to import an external module.
- **GitHub:** Initialized a Git repo and published it to GitHub using VS Code.

Section 1: Node.js Built-in Modules (1/4)

Objective:

- Learn how to use modules that come standard with Node.js.

What are Built-in Modules?

- Functionality included directly with Node.js - no `npm install` needed!
- Provide core capabilities for server-side development (networking, file system, OS info, etc.).
- You still need to `require()` them to use them in your code.

Common Examples:

- `path`: Utilities for working with file and directory paths.
- `fs` (File System): Interact with the file system (read/write files).
- `http` / `https`: Create web servers and clients.
- `os`: Get operating system information.

Section 1: Node.js Built-in Modules (2/4)

Example: The `path` Module

- Handles file paths correctly across different operating systems (Windows `\` vs macOS/Linux `/`).

```javascript
// Import the built-in 'path' module
const path = require('path');

const myFilePath = '/users/test/documents/file.txt'; // Example path

// Get the directory name
console.log('Dirname:', path.dirname(myFilePath));
// Output: /users/test/documents

// Get the base filename
console.log('Basename:', path.basename(myFilePath));
// Output: file.txt

// Get the file extension
console.log('Extension:', path.extname(myFilePath));
// Output: .txt

// Join path segments together (OS-specific separator)
const fullPath = path.join('users', 'test', 'notes', 'note.md');
console.log('Joined Path:', fullPath);
// Output: users/test/notes/note.md (or users\test\notes\note.md on Windows)
```

Activity: Add the `path` module examples to your `app.js` and run it (`node app.js`).

Section 1: Node.js Built-in Modules (3/4)

Example: The `fs` (File System) Module

- Allows your Node.js program to read from and write to files.
- **Synchronous vs. Asynchronous:** Most `fs` methods have both sync (`...Sync`) and async versions. Async is generally preferred for servers, but sync is simpler for basic scripts/learning.

```
// Import the built-in 'fs' module
const fs = require('fs');
const path = require('path'); // We often use 'path' with 'fs'

// --- Writing to a file (Synchronous) ---
const filePath = path.join(__dirname, 'hello.txt'); // __dirname = current folder
const fileContent = 'Hello from Node.js File System!';

try {
  fs.writeFileSync(filePath, fileContent, 'utf8');
  console.log(`Successfully wrote to ${filePath}`);
} catch (err) {
  console.error('Error writing file:', err);
}

// --- Reading from a file (Synchronous) ---
try {
  const data = fs.readFileSync(filePath, 'utf8');
  console.log(`Read from file: ${data}`);
} catch (err) {
  console.error('Error reading file:', err);
}
```

Activity: Add the `fs` examples. Run `node app.js`. Check if `hello.txt` was created and read!

Section 1: Node.js Built-in Modules (4/4)

Reading Node.js Documentation

- The official Node.js API documentation is your best resource: https://nodejs.org/api/
- **How to Read It:**
    1. **Find the Module:** Navigate to the module you need (e.g., `fs`, `path`, `http`).
    2. **Table of Contents:** Use the ToC on the right to find specific methods/properties.
    3. **Method Signature:** Understand the parameters (required/optional), their types, and what the method returns.
    4. **Description:** Read the explanation of what the method does.
    5. **Code Examples:** Look for usage examples – often the quickest way to understand.
- **Practice:** Look up `fs.existsSync()` or `path.resolve()` in the docs.

Getting comfortable reading documentation is a crucial skill for any developer!

Section 2: Creating Your Own Modules (1/3)

Objective:

- Organize code into separate files (modules) and share functionality between them.

Why Create Modules? (Recap)

- **Organization:** Keeps related code together.
- **Reusability:** Write a function once, use it in multiple places.
- **Maintainability:** Easier to update or fix bugs in focused modules.
- **Collaboration:** Different team members can work on different modules.

The Core Idea: Exporting & Requiring

1. **Export:** Make functions, objects, or variables available *from* a module file.
2. **Require:** Import that exported functionality *into* another file where you need it.

Section 2: Creating Your Own Modules (2/3)

Exporting with `module.exports`

- Every Node.js file is implicitly a module.
- `module.exports` is a special object. Whatever you assign to it is what gets exported.

```
// --- logger.js ---

function logInfo(message) {
  console.log(`[INFO] ${new Date().toISOString()}: ${message}`);
}

function logError(message) {
  console.error(`[ERROR] ${new Date().toISOString()}: ${message}`);
}

// Option 1: Export an object containing the functions
module.exports = {
  info: logInfo,
  error: logError
};

// Option 2 (Alternative): Export a single function (if module only does one thing)
// module.exports = logInfo;

// Option 3 (Less Common): Add properties directly
// module.exports.info = logInfo;
// module.exports.error = logError;
```

Section 2: Creating Your Own Modules (3/3)

Importing with `require()`

- Use `require()` with a *relative path* (starting with `./` or `../`) to import your own modules.

```
// --- app.js ---

// Import the entire object exported from logger.js
const logger = require('./logger.js'); // Note the ./ path

// Use the imported functions
logger.info('Application started.');

const user = 'Alice';
if (!user) {
  logger.error('No user found!');
} else {
  logger.info(`User ${user} logged in.`);
}

// If logger.js only exported one function (e.g., module.exports = logInfo)
// const log = require('./logger.js');
// log('Something happened.');
```

Activity:

1. Create a `logger.js` file with the logging functions and `module.exports`.
2. Modify `app.js` to `require('./logger.js')` and use the `logger.info()` and `logger.error()` functions.
3. Run `node app.js`.

Section 3: NPM & `package.json` Deep Dive (1/4)

Objective:

- Understand key sections of `package.json` and manage different types of dependencies.

`package.json` Recap:

- The manifest file for your Node.js project.
- Created by `npm init`.
- Tracks metadata, dependencies, and scripts.

Focus Areas Today:

1. `dependencies` vs. `devDependencies`
2. `scripts` section
3. Semantic Versioning (SemVer)

Section 3: NPM & `package.json` Deep Dive (2/4)

`dependencies` vs. `devDependencies`

- **`dependencies`**: Packages required for your application to *run* in production.
  - Examples: `lodash`, `express`, database drivers.
  - Installed via: `npm install <package_name>` (or `npm i <package_name>`)
  - Saved automatically to `dependencies` section in `package.json`.
- **`devDependencies`**: Packages needed only for *development* and *testing*.
  - Examples: Testing libraries (`jest`), code linters (`eslint`), utility tools (`nodemon`).
  - Installed via: `npm install --save-dev <package_name>` (or `npm i -D <package_name>`)
  - Saved to `devDependencies` section in `package.json`.

Why the difference? When deploying your application, you often only need the runtime `dependencies`, not the development tools, saving space and installation time.

Section 3: NPM & `package.json` Deep Dive (3/4)

Example: Using `nodemon` (Dev Dependency)

- `nodemon` automatically restarts your Node.js application when file changes are detected – great for development!

1. **Install as Dev Dependency:**

   ```
   npm install --save-dev nodemon
   # or: npm i -D nodemon
   ```

   - Check your `package.json` - `nodemon` should be under `devDependencies`.

2. **Add a `scripts` entry in `package.json`:**

   ```
   {
     "name": "lesson1-backend",
     // ... other properties
     "scripts": {
       "start": "node app.js", // Standard command to run normally
       "dev": "nodemon app.js"   // Command using nodemon for development
     },
     // ... dependencies/devDependencies
   }
   ```

3. **Run the script:**

   ```
   npm run dev
   ```

   - Now, make a change to `app.js` and save it. `nodemon` should automatically restart the script!
   - (Use `Ctrl+C` to stop `nodemon`).

Activity: Install `nodemon` as a dev dependency, add the `start` and `dev` scripts, and run `npm run dev`. Test the auto-restart.

Section 3: NPM & `package.json` Deep Dive (4/4)

Semantic Versioning (SemVer)

- How package versions are numbered: `MAJOR.MINOR.PATCH` (e.g., `16.4.1`)
  - **MAJOR:** Incompatible API changes.
  - **MINOR:** Added functionality (backwards-compatible).
  - **PATCH:** Bug fixes (backwards-compatible).
- **Symbols in `package.json`:**
  - `^` (Caret): Allows updates to MINOR and PATCH versions (e.g., `^16.4.1` allows `16.5.0`, `16.4.2`, but NOT `17.0.0`). **Most common.**
  - `~` (Tilde): Allows updates to PATCH version only (e.g., `~16.4.1` allows `16.4.2`, but NOT `16.5.0`).
  - `*` or `x`: Allows any version (use with caution).
  - Exact version (`16.4.1`): Only allows this specific version.
- **`package-lock.json`:** Records the *exact* versions of all installed packages (including dependencies of dependencies) to ensure consistent installs across different machines/times.

Read More: The Basics of Package.json

Section 4: GitHub Workflow (Pushing Updates)

Objective:

- Practice the common workflow of committing and pushing code changes to GitHub.

1. **Make Code Changes:** Modify your files (e.g., add a new feature using a module, fix a bug).
2. **Stage Changes:** Go to the Source Control tab (Git icon).
   - Changed files appear under "Changes".
   - Click the "+" icon next to the files you want to include in the next commit.
3. **Commit Changes:**
   - Write a clear, concise commit message describing *what you changed* (e.g., "Add logger module for info/error messages", "Fix calculation bug").
   - Click the Checkmark ✓ icon.
4. **Push Changes:**
   - Click the "Sync Changes" button (usually has arrows and might show number of commits to push/pull) in the status bar or the `...` menu -> Push.
   - This uploads your local commit(s) to the `origin` remote (your GitHub repository).

Activity:

1. Make a small change to your `app.js` or `logger.js` file.
2. Go to VS Code Source Control.
3. Stage the change.
4. Write a commit message (e.g., "Update logger message format").
5. Commit.
6. Click "Sync Changes" / Push.
7. Verify the change on your GitHub repository online!

Summary / Key Takeaways

- **Built-in Modules:** Use `require()` for core Node.js features (`path`, `fs`, etc.). No install needed.
- **Custom Modules:** Organize code using `module.exports = ...` and `require('./relative/path')`.
- **`package.json`**:
  - `dependencies`: Needed to run the app.
  - `devDependencies`: Only for development (`npm i -D`).
  - `scripts`: Define command shortcuts (`npm run <script>`).
- **NPM Workflow:** `npm install`, `npm run dev`.
- **GitHub Workflow:** Make Changes -> Stage -> Commit -> Push (Sync).

Questions?

Thank you!