Agenda

- Pre-defined modules (buffer , etc) — Discussion basis
- Modules managed by npm (Nodemon, Chalk, etc.)
- User Defined modules
- Debugging Node.js application

Let's take a look at a few more Core modules of NodeJS.

Buffer

`Buffer` objects are used to represent a fixed-length sequence of bytes. Many Node.js APIs support `Buffer` s.

The `Buffer` class is a subclass of JavaScript's `Uint8Array` class and extends it with methods that cover additional use cases. Node.js APIs acce plain `Uint8Array` s wherever `Buffer` s are supported as well.

While the `Buffer` class is available within the global scope, it is still recommended to explicitly reference it via an import or require statement.

Example :

```
const { Buffer } = require('node:buffer');

// Creates a zero-filled Buffer of length 10.
const buf1 = Buffer.alloc(10);

// Creates a Buffer of length 10,
// filled with bytes which all have the value `1`.
const buf2 = Buffer.alloc(10, 1);

// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using fill(), write(), or other functions that fill the Buffer's
// contents.
const buf3 = Buffer.allocUnsafe(10);

// Creates a Buffer containing the bytes [1, 2, 3].
const buf4 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing the bytes [1, 1, 1, 1] – the entries
// are all truncated using `(value & 255)` to fit into the range 0–255.
const buf5 = Buffer.from([257, 257.5, -255, '1']);

// Creates a Buffer containing the UTF-8-encoded bytes for the string 'tést':
// [0x74, 0xc3, 0xa9, 0x73, 0x74] (in hexadecimal notation)
// [116, 195, 169, 115, 116] (in decimal notation)
const buf6 = Buffer.from('tést');

// Creates a Buffer containing the Latin-1 bytes [0x74, 0xe9, 0x73, 0x74].
const buf7 = Buffer.from('tést', 'latin1');
```

URL

The `node:url` module provides utilities for URL resolution and parsing. It can be accessed using:

```
const url = require('node:url');
```

URL strings and URL objects

A URL string is a structured string containing multiple meaningful components. When parsed, a URL object is returned containing properties for each these components.

The `node:url` module provides two APIs for working with URLs: a legacy API that is Node.js specific, and a newer API that implements th same WHATWG URL Standard used by web browsers.

Parsing the URL string using the WHATWG API:

```
const myURL =
  new URL('https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash');
```

Parsing the URL string using the Legacy API:

```
const url = require('node:url');
const myURL =
  url.parse('https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash');
```

`new URL(input[, base])`

- `input` The absolute or relative input URL to parse. If `input` is relative, then `base` is required. If `input` is absolute, the `base` is ignored. If `input` is not a string, it is converted to a string first.
- `base` The base URL to resolve against if the `input` is not absolute. If `base` is not a string, it is converted to a string first.

Creates a new `URL` object by parsing the `input` relative to the `base`. If `base` is passed as a string, it will be parsed equivalent to `new URL(base)`.

```
const myURL = new URL('/foo', 'https://example.org/');
// https://example.org/foo
```

A `TypeError` will be thrown if the `input` or `base` are not valid URLs.

Unicode characters appearing within the host name of `input` will be automatically converted to ASCII using the Punycode algorithm.

```
const myURL = new URL('https://測試');
// https://xn--g6w251d/
```

In cases where it is not known in advance if `input` is an absolute URL and a `base` is provided, it is advised to validate that the `origin` the `URL` object is what is expected.

```
let myURL = new URL('http://Example.com/', 'https://example.org/');
// http://example.com/

myURL = new URL('https://Example.com/', 'https://example.org/');
// https://example.com/

myURL = new URL('foo://Example.com/', 'https://example.org/');
// foo://Example.com/

myURL = new URL('http:Example.com/', 'https://example.org/');
// http://example.com/

myURL = new URL('https:Example.com/', 'https://example.org/');
// https://example.org/Example.com/

myURL = new URL('foo:Example.com/', 'https://example.org/');
// foo:Example.com/
```

`url.host`

-

Gets and sets the host portion of the URL.

```
const myURL = new URL('https://example.org:81/foo');
console.log(myURL.host);
// Prints example.org:81

myURL.host = 'example.com:82';
```

```
console.log(myURL.href);
// Prints https://example.com:82/foo
```

Invalid host values assigned to the `host` property are ignored.

`url.hostname`

-

Gets and sets the host name portion of the URL. The key difference between `url.host` and `url.hostname` that `url.hostname` does *not* include the port.

```
const myURL = new URL('https://example.org:81/foo');
console.log(myURL.hostname);
// Prints example.org

// Setting the hostname does not change the port
myURL.hostname = 'example.com:82';
console.log(myURL.href);
// Prints https://example.com:81/foo

// Use myURL.host to change the hostname and port
myURL.host = 'example.org:82';
console.log(myURL.href);
// Prints https://example.org:82/foo
```

Invalid host name values assigned to the `hostname` property are ignored.

`url.href`

-

Gets and sets the serialized URL.

```
const myURL = new URL('https://example.org/foo');
console.log(myURL.href);
// Prints https://example.org/foo

myURL.href = 'https://example.com/bar';
console.log(myURL.href);
// Prints https://example.com/bar
```

Getting the value of the `href` property is equivalent to calling `url.toString()` .

Setting the value of this property to a new value is equivalent to creating a new `URL` object using `new URL(value)` . Each of the `URL` object properties will be modified.

If the value assigned to the `href` property is not a valid URL, a `TypeError` will be thrown.

`url.pathname`

-

Gets and sets the path portion of the URL.

```
const myURL = new URL('https://example.org/abc/xyz?123');
console.log(myURL.pathname);
// Prints /abc/xyz

myURL.pathname = '/abcdef';
console.log(myURL.href);
// Prints https://example.org/abcdef?123
```

Invalid URL characters included in the value assigned to the `pathname` property are percent-encoded.

Managing Modules with NPM

The Node.js Package Manager (npm) is the default and most popular package manager in the Node.js ecosystem, and is primarily used to install ar manage external modules in a Node.js project. It is also commonly used to install a wide range of CLI tools and run project scripts. npm tracks th modules installed in a project with the `package.json` file, which resides in a project's directory and contains:

- All the modules needed for a project and their installed versions
- All the metadata for a project, such as the author, the license, etc.
- Scripts that can be run to automate tasks within the project

As you create more complex Node.js projects, managing your metadata and dependencies with the `package.json` file will provide you with mo predictable builds, since all external dependencies are kept the same. The file will keep track of this information automatically; while you may change th file directly to update your project's metadata, you will seldom need to interact with it directly to manage modules.

Installing Modules

It is common in software development to use external libraries to perform ancillary tasks in projects. This allows the developer to focus on the busines logic and create the application more quickly and efficiently by utilizing tools and code that others have written that accomplish tasks one needs.

For example, if our module has to make an external API request to get some data, we could use an HTTP library to make that task easier. Since o main goal is to return pertinent geographical data to the user, we could install a package that makes HTTP requests easier for us instead of rewriting th code for ourselves, a task that is beyond the scope of our project.

Let's run through this example. Create a sample application, where you will use the axios library, which will help you make HTTP requests. Install it b entering the following in your shell:

```
npm install axios --save
```

You begin this command with `npm install`, which will install the package (for brevity you can also use `npm i`). You then list the packages that yc want installed, separated by a space. In this case, this is `axios`. Finally, you end the command with the optional `--save` parameter, which specifie that `axios` will be saved as a project dependency.

When the library is installed, you will see output similar to the following:

```
// OUTPUT
...
+ axios@0.27.2
added 5 packages from 8 contributors and audited 5 packages in 0.764s
found 0 vulnerabilities
```

Now, open the `package.json` file, using a text editor of your choice. You'll see a new property, as highlighted in the following:

```
{
  "name": "AlmaBetter",
  "version": "1.0.0",
  "description": "Finds the country of origin of the incoming request",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ip",
    "geo",
    "country"
  ],
  "author": "AlmaBetter",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.27.2"
  }
}
```

The `--save` option told `npm` to update the `package.json` with the module and version that was just installed. This is great, as other develope working on your projects can easily see what external dependencies are needed.

> **Note**: You may have noticed the `^` before the version number for the `axios` dependency. Recall that semantic versioning consists of three digits: **MAJOR**, **MINOR**, and **PATCH**. The `^` symbol signifies that any higher MINOR or PATCH version would satisfy this version constraint. If you see `~` at the beginning of a version number, then only higher PATCH versions satisfy the constraint.

When you are finished reviewing `package.json`, close the file.

Development Dependencies

Packages that are used for the development of a project but not for building or running it in production are called *development dependencies*. They a not necessary for your module or application to work in production, but may be helpful while writing the code.

For example, it's common for developers to use *code linters* to ensure their code follows best practices and to keep the style consistent. While this useful for development, this only adds to the size of the distributable without providing a tangible benefit when deployed in production.

Install a linter as a development dependency for your project. Try this out in your shell:

```
npm i eslint@8.0.0 --save-dev
```

In this command, you used the `--save-dev` flag. This will save `eslint` as a dependency that is only needed for development. Notice also that yc added `@8.0.0` to your dependency name. When modules are updated, they are tagged with a version. The `@` tells npm to look for a specific tag the module you are installing. Without a specified tag, npm installs the latest tagged version. Open `package.json` again, it will show the following :

```
{
  "name": "AlmaBetter",
  "version": "1.0.0",
  "description": "Finds the country of origin of the incoming request",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ip",
    "geo",
    "country"
  ],
  "author": "AlmaBetter",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.27.2"
  },
    "devDependencies": {
    "eslint": "^8.0.0"
    }
}
```

`eslint` has been saved as a `devDependencies`, along with the version number you specified earlier. Exit `package.json`.

Automatically Generated Files: `node_modules` and `package-lock.json`

When you first install a package to a Node.js project, `npm` automatically creates the `node_modules` folder to store the modules needed for yo project and the `package-lock.json` file that you examined earlier.

Confirm these are in your working directory. In your shell, type `ls` and press `ENTER`. You will observe the following output:

```
// OUTPUT
node_modules    package.json    package-lock.json
```

The `node_modules` folder contains every installed dependency for your project. In most cases, you should **not** commit this folder into your versic controlled repository. As you install more dependencies, the size of this folder will quickly grow. Furthermore, the `package-lock.json` file keeps record of the exact versions installed in a more succinct way, so including `node_modules` is not necessary.

While the `package.json` file lists dependencies that tell us the suitable versions that should be installed for the project, the `package-lock.json` fi keeps track of all changes in `package.json` or `node_modules` and tells us the exact version of the package installed. You usually commit this your version controlled repository instead of `node_modules`, as it's a cleaner representation of all your dependencies.

Installing from package.json

With your `package.json` and `package-lock.json` files, you can quickly set up the same project dependencies before you start development on new project. To demonstrate this, move up a level in your directory tree and create a new folder named in the same directory level as the previou sample folder:

```
cd ..
mkdir cloned_sampleproject
```

Move into your new directory:

```
cd cloned_sampleproject
```

Now copy the `package.json` and `package-lock.json` files from `locator` to `cloned_locator`. To install the required modules for this projec type:

```
npm i
```

npm will check for a `package-lock.json` file to install the modules. If no lock file is available, it would read from the `package.json` file to determir the installations. It is usually quicker to install from `package-lock.json`, since the lock file contains the exact version of modules and the dependencies, meaning npm does not have to spend time figuring out a suitable version to install.

When deploying to production, you may want to skip the development dependencies. Recall that development dependencies are stored the `devDependencies` section of `package.json`, and have no impact on the running of your app. When installing modules as part of the deployme process to deploy your application, omit the dev dependencies by running:

```
npm i --production
```

The `--production` flag ignores the `devDependencies` section during installation. For now, stick with your development build.

Global Installations

So far, you have been installing npm modules for the `locator` project. npm also allows you to install packages *globally*. This means that the packag is available to your user in the wider system, like any other shell command. This ability is useful for the many Node.js modules that are CLI tools.

To install a package globally, you append the `-g` flag to the command :

```
npm i nodemon -g
```

Managing Modules

A complete package manager can do a lot more than install modules. npm has over 20 commands relating to dependency management available. In th step, you will:

- List modules you have installed.
- Update modules to a more recent version.
- Uninstall modules you no longer need.
- Perform a security audit on your modules to find and fix security flaws.

While these examples will be done in the example folder, all of these commands can be run globally by appending the `-g` flag at the end of ther exactly like you do when installing globally.

Listing Modules

If you would like to know which modules are installed in a project, it would be easier to use the `list` or `ls` command instead of readir the `package.json` directly. To do this, enter:

```
npm ls
// OUTPUT
├── axios@0.27.2
└── eslint@8.0.0
```

The `--depth` option allows you to specify what level of the dependency tree you want to see. When it's `0`, you only see your top level dependencie If you want to see the entire dependency tree, use the `--all` argument:

```
npm ls --all

//OUTPUT
```

```
├─┬ axios@0.27.2
│ ├── follow-redirects@1.15.1
│ └─┬ form-data@4.0.0
│   ├── asynckit@0.4.0
│   ├─┬ combined-stream@1.0.8
│   │ └── delayed-stream@1.0.0
│   └─┬ mime-types@2.1.35
│     └── mime-db@1.52.0
└─┬ eslint@8.0.0
  ├─┬ @eslint/eslintrc@1.3.0
  │ ├── ajv@6.12.6 deduped
  │ ├── debug@4.3.4 deduped
  │ ├── espree@9.3.2 deduped
  │ ├── globals@13.15.0 deduped
  │ ├── ignore@5.2.0
  │ ├── import-fresh@3.3.0 deduped
  │ ├── js-yaml@4.1.0 deduped
  │ ├── minimatch@3.1.2 deduped
  │ └── strip-json-comments@3.1.1 deduped
```

Updating Modules

It is a good practice to keep your npm modules up to date. This improves your likelihood of getting the latest security fixes for a module. Us the `outdated` command to check if any modules can be updated:

```
npm outdated
// OUTPUT
Package  Current  Wanted  Latest  Location             Depended by
eslint     8.0.0  8.17.0  8.17.0  node_modules/eslint  sampleproject
```

This command first lists the `Package` that's installed and the `Current` version. The `Wanted` column shows which version satisfies your versio requirement in `package.json` . The `Latest` column shows the most recent version of the module that was published.

The `Location` column states where in the dependency tree the package is located. The `outdated` command has the `--depth` flag like `ls` . E default, the depth is 0.

It seems that you can update `eslint` to a more recent version. Use the `update` or `up` command like this:

```
npm up eslint

// OUTPUT
removed 7 packages, changed 4 packages, and audited 91 packages in 1s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

To see which version of `eslint` that you are using now, you can use `npm ls` using the package name as an argument:

```
npm ls eslint
```

The output will resemble the `npm ls` command you used before, but include only the `eslint` package's versions:

```
Output
└─┬ eslint@8.17.0
  └─┬ eslint-utils@3.0.0
    └── eslint@8.17.0 deduped
```

If you wanted to update all modules at once, then you would enter:

```
npm up
```

Uninstalling Modules

The npm `uninstall` command can remove modules from your projects. This means the module will no longer be installed the `node_modules` folder, nor will it be seen in your `package.json` and `package-lock.json` files.

Removing dependencies from a project is a normal activity in the software development lifecycle. A dependency may not solve the problem a advertised, or may not provide a satisfactory development experience. In these cases, it may better to uninstall the dependency and build your ow module.

Imagine that `axios` does not provide the development experience you would have liked for making HTTP requests. Uninstall `axios` wi the `uninstall` or `un` command by entering:

```
npm un axios
// Output
removed 8 packages, and audited 83 packages in 542ms

13 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

It doesn't explicitly say that `axios` was removed. To verify that it was uninstalled, list the dependencies once again.

Auditing Modules

npm provides an `audit` command to highlight potential security risks in your dependencies. To see the audit in action, install an outdated version the request module by running the following:

```
npm i request@2.60.0

// When you install this outdated version of request, you'll notice output similar to the following:
npm WARN deprecated cryptiles@2.0.5: This version has been deprecated in accordance with the hapi support policy (hapi.i
npm WARN deprecated sntp@1.0.9: This module moved to @hapi/sntp. Please make sure to switch over as this distribution is
npm WARN deprecated boom@2.10.1: This version has been deprecated in accordance with the hapi support policy (hapi.im/su
npm WARN deprecated node-uuid@1.4.8: Use uuid module instead
npm WARN deprecated har-validator@1.8.0: this library is no longer supported
npm WARN deprecated hoek@2.16.3: This version has been deprecated in accordance with the hapi support policy (hapi.im/su
npm WARN deprecated request@2.60.0: request has been deprecated, see https://github.com/request/request/issues/3142
npm WARN deprecated hawk@3.1.3: This module moved to @hapi/hawk. Please make sure to switch over as this distribution is

added 56 packages, and audited 139 packages in 4s

13 packages are looking for funding
  run `npm fund` for details

9 vulnerabilities (5 moderate, 2 high, 2 critical)

To address all issues, run:
  npm audit fix --force

Run `npm audit` for details.
```

npm is telling you that you have deprecated packages and vulnerabilities in your dependencies. To get more details, audit your entire project with:

```
npm audit

// OUTPUT
# npm audit report

bl  <1.2.3
Severity: moderate
Remote Memory Exposure in bl - https://github.com/advisories/GHSA-pp7h-53gx-mx7r
fix available via `npm audit fix`
node_modules/bl
  request  2.16.0 - 2.86.0
```

```
  Depends on vulnerable versions of bl
  Depends on vulnerable versions of hawk
  Depends on vulnerable versions of qs
  Depends on vulnerable versions of tunnel-agent
  node_modules/request

cryptiles  <=4.1.1
Severity: critical
Insufficient Entropy in cryptiles - https://github.com/advisories/GHSA-rq8g-5pc5-wrhr
Depends on vulnerable versions of boom
fix available via `npm audit fix`
node_modules/cryptiles
  hawk  <=9.0.0
  Depends on vulnerable versions of boom
  Depends on vulnerable versions of cryptiles
  Depends on vulnerable versions of hoek
  Depends on vulnerable versions of sntp
  node_modules/hawk

. . .

9 vulnerabilities (5 moderate, 2 high, 2 critical)

To address all issues, run:
  npm audit fix
```

You can see the path of the vulnerability, and sometimes npm offers ways for you to fix it. You can run the update command as suggested, or you can ru the `fix` subcommand of `audit`. In your shell, enter:

```
npm audit fix

// OUTPUT
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in certain
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142

added 19 packages, removed 34 packages, changed 13 packages, and audited 124 packages in 3s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

npm was able to safely update two of the packages, decreasing your vulnerabilities by the same amount. However, you still have three deprecate packages in your dependencies. The `audit fix` command does not always fix every problem. Although a version of a module may have a securi vulnerability, if you update it to a version with a different API then it could break code higher up in the dependency tree.

You can use the `--force` parameter to ensure the vulnerabilities are gone, like this:

```
npm audit fix --force
```

As mentioned before, this is not recommended unless you are sure that it won't break functionality.

Let's install and take a look at some npm packages that we'll be using frequently with Node.

Nodemon

In Node.js, you need to restart the process to make changes take effect. This adds an extra step to your workflow. You can eliminate this extra step b using `[nodemon](https://nodemon.io/)` to restart the process automatically.

`nodemon` is a command-line interface (CLI) utility that wraps your Node app, watches the file system, and automatically restarts the process.

Let's learn about installing, setting up, and configuring `nodemon`.

Installing `nodemon`

First, you will need to install `nodemon` on your machine. Install the utility either globally or locally on your project using `npm`:

Global Installation

You can install `nodemon` globally with `npm`:

```
npm install nodemon --global
```

Local Installation

You can also install `nodemon` locally. When performing a local installation, you can install `nodemon` as a dev dependency with `--save-dev` (or `-dev`).

Install `nodemon` locally with `npm`:

```
npm install nodemon --save-dev
```

One thing to be aware of with a local install is that you will only be able to use the `nodemon` in that particular project.

After you've installed `nodemon` you can use it to start a Node script. For example, if you have a nodejs server in a `server.js` file, you ca start `nodemon` and watch for changes like this:

```
nodemon server.js
```

You can pass in arguments the same way as if you were running the script with Node:

```
nodemonserver.js 3006
```

Every time you make a change to a file with one of the default watched extensions (`.js`, `.mjs`, `.json`, `.coffee`, or `.litcoffee`) in th current directory or a subdirectory, the process will restart.

You can restart the process at any time by typing `rs` and hitting `ENTER`.

Alternatively, `nodemon` will also look for a `main` file specified in your project's `package.json` file:

```
{
  // ...
"main": "server.js",
  // ...
}
```

If a `main` file is not specified, `nodemon` will search for a `start` script:

```
{
  // ...
"scripts": {
"start": "node server.js"
  },
  // ...
}
```

Once you make the changes to `package.json`, you can then call `nodemon` to start the example app in watch mode without having to pas in `server.js`.

Using Options

You can modify the configuration settings available to `nodemon`.

Let's go over some of the main options:

- `-exec`: Use the `-exec` switch to specify a binary to execute the file with. For example, when combined with the `[ts-node](https://github.com/TypeStrong/ts-node)` binary, `-exec` can become useful to watch for changes and run TypeScript files.
- `-ext`: Specify different file extensions to watch. For this switch, provide a comma-separated list of file extensions (e.g., `-ext js,ts`).
- `-delay`: By default, `nodemon` waits for one second to restart the process when a file changes, but with the `-delay` switch, you can specify a different delay. For example, `nodemon --delay 3.2` for a 3.2-second delay.
- `-watch`: Use the `-watch` switch to specify multiple directories or files to watch. Add one `-watch` switch for each directory you want to watch. By default, the current directory and its subdirectories are watched, so with `-watch` you can narrow that to only specific

subdirectories or files.

- `-ignore` : Use the `-ignore` switch to ignore certain files, file patterns, or directories.
- `-verbose` : A more verbose output with information about what file(s) changed to trigger a restart.

You can view all the available options with the following command:

```
nodemon --help
```

Using these options, let's create the command to satisfy the following scenario:

- watching the `server` directory
- specifying files with a `.ts` extension
- ignoring files with a `.test.ts` suffix
- executing the file (`server/server.ts`) with `ts-node`
- waiting for three seconds to restart after a file changes

```
nodemon --watch server --ext ts --exec ts-node --ignore '*.test.ts' --delay 3 server/server.ts
```

The terminal output will display:

```
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): server
[nodemon] watching extensions: ts
[nodemon] starting `ts-node server/server.ts`
```

This command combines `--watch` , `--ext` , `--exec` , `--ignore` , and `--delay` options to satisfy the conditions for our scenario.

Using Configurations

In the previous example, adding configuration switches when running `nodemon` can get tedious. A better solution for projects that require complicate configurations is to define these options in a `nodemon.json` file.

For example, here are the same configurations as the previous command line example, but placed in a `nodemon.json` file:

```
{
  "watch": [
    "server"
  ],
  "ext": "ts",
  "ignore": [
    "*.test.ts"
  ],
  "delay": "3",
  "execMap": {
    "ts": "ts-node"
  }
}
```

> Note the use of `execMap` instead of the `--exec` switch. `execMap` allows you to specify binaries for certain file extensions.

Alternatively, if you would rather not add a `nodemon.json` config file to your project, you can add these configurations to the `package.json` fi under a `nodemonConfig` key:

```
{
  "name": "nodemon-example",
  "version": "1.0.0",
  "description": "",
  "nodemonConfig": {
    "watch": [
```

```
      "server"
    ],
    "ext": "ts",
    "ignore": [
      "*.test.ts"
    ],
    "delay": "3",
    "execMap": {
      "ts": "ts-node"
    }
  },
  // ...
```

Once you make the changes to either `nodemon.json` or `package.json`, you can then start `nodemon` with the desired script `nodemon` will pick u the configurations and use them. This way, your configurations can be saved, shared, and repeated to avoid copy-and-pasting or typing errors in th command line.

Chalk

The **chalk** module is a third-party library that can be used for styling of texts. It allows the users to create their own themes in a Node.js project.

- This module helps the users to customize the response messages with different colors as per the preferences.
- It also improves the readability by providing colors and makes it easier to detect warnings and errors.

Installation :

```
npm i chalk
```

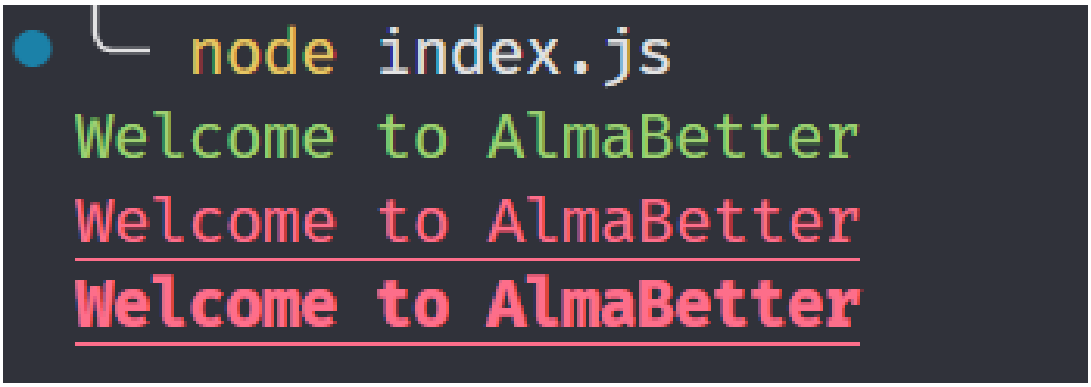
Example 1

Create a file with the name "**chalk.js**" and copy the following code. After creating the file, use the command "**node chalk.js**" to run this code as shown the example below :

```
// Importing the chalk module
const chalk=require("chalk");

// Coloring different text messages
console.log(chalk.green("Welcome to AlmaBetter"))
console.log(chalk.red.underline("Welcome to AlmaBetter"))
console.log(chalk.red.underline.bold("Welcome to AlmaBetter"))
```

Output :



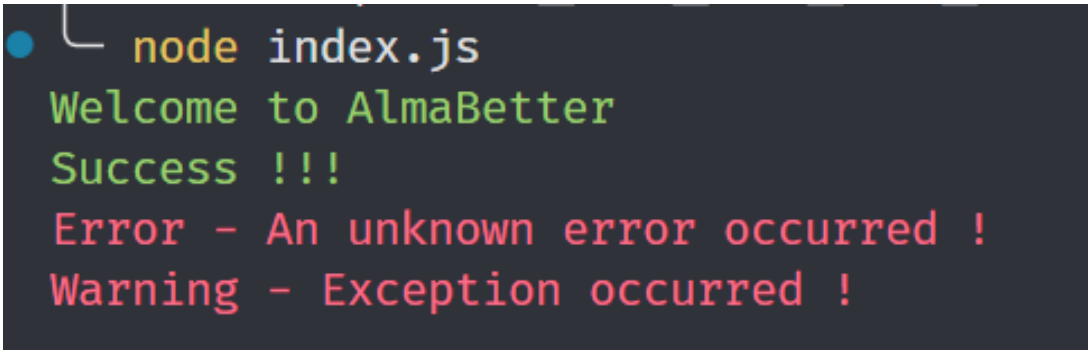
Let's take another example :

```
// Importing the chalk module
const chalk=require("chalk");

// Coloring different text messages
const welcome=chalk.green;
const warning=chalk.red;
```

```
console.log(welcome("Welcome to AlmaBetter"))
console.log(welcome("Success !!!"))
console.log(warning("Error - An unknown error occurred !"))
console.log(warning("Warning - Exception occurred !"))
```

Output :



User-Defined Modules

Creating a Module

This step will guide you through creating your first Node.js module. Your module will contain a collection of colors in an array and provide a function to g one at random. You will use the Node.js built-in `exports` property to make the function and array available to external programs.

First, you'll begin by deciding what data about colors you will store in your module. Every color will be an object that contains a `name` property th humans can easily identify, and a `code` property that is a string containing an HTML color code. HTML color codes are six-digit hexadecimal numbe that allow you to change the color of elements on a web page.

You will then decide what colors you want to support in your module. Your module will contain an array called `allColors` that will contain six color Your module will also include a function called `getRandomColor()` that will randomly select a color from your array and return it.

In your terminal, make a new folder called `colors` and move into it:

```
mkdir colors
cd colors
// INITIALIZE NPM
npm init -y
```

You used the `-y` flag to skip the usual prompts to customize your `package.json` . If this were a module you wished to publish to npm, you wou answer all these prompts with relevant data.

Now, open the folder in a text-editor. First, you'll define a `Color` class. Your `Color` class will be instantiated with its name and HTML code. Add th following lines to create the class:

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}
```

Now that you have your data structure for `Color` , add some instances into your module. Write the following array to the file:

```
const allColors = [
  new Color('brightred', '#E74C3C'),
  new Color('soothingpurple', '#9B59B6'),
  new Color('skyblue', '#5DADE2'),
  new Color('leafygreen', '#48C9B0'),
  new Color('sunkissedyellow', '#F4D03F'),
  new Color('groovygray', '#D7DBDD'),
];
```

Finally, enter a function that randomly selects an item from the `allColors` array you just created:

```
exports.getRandomColor = () => {
  return allColors[Math.floor(Math.random() * allColors.length)];
}

exports.allColors = allColors;
```

The `exports` keyword references a global object available in every Node.js module. All functions and objects stored in a module's `exports` obje are exposed when other Node.js modules import it. The `getRandomColor()` function was created directly on the `exports` object, for example. Yo then added an `allColors` property to the `exports` object that references the local constant `allColors` array created earlier in the script.

When other modules import this module, both `allColors` and `getRandomColor()` will be exposed and available for usage.

Testing the Module

Before you build a complete application, take a moment to confirm that your module is working. In this step, you will use the REPL to loa the `colors` module. While in the REPL, you will call the `getRandomColor()` function to see if it behaves as you expect it to.

Start the Node.js REPL in the same folder as the `index.js` file:

```
node
```

When the REPL has started, you will see the `>` prompt. This means you can enter JavaScript code that will be immediately evaluated. First, enter th following:

```
colors = require('./index');
```

In this command, `require()` loads the `colors` module at its entry point. When you press `ENTER` you will get:

```
{
  getRandomColor: [Function],
  allColors: [
    Color { name: 'brightred', code: '#E74C3C' },
    Color { name: 'soothingpurple', code: '#9B59B6' },
    Color { name: 'skyblue', code: '#5DADE2' },
    Color { name: 'leafygreen', code: '#48C9B0' },
    Color { name: 'sunkissedyellow', code: '#F4D03F' },
    Color { name: 'groovygray', code: '#D7DBDD' }
  ]
}
```

The REPL shows us the value of `colors`, which are all the functions and objects imported from the `index.js` file. When you us the `require` keyword, Node.js returns all the contents within the `exports` object of a module.

Recall that you added `getRandomColor()` and `allColors` to `exports` in the `colors` module. For that reason, you see them both in the REF when they are imported. Call the function from command line to see if it works.

Saving Local Module as a Dependency

While testing your module in the REPL, you imported it with a *relative path*. This means you used the location of the `index.js` file in relation to th working directory to get its contents. While this works, it is usually a better programming experience to import modules by their names so that the impo is not broken when the context is changed. In this step, you will install the `colors` module with npm's local module `install` feature.

Set up a new Node.js module outside the `colors` folder. First, go to the previous directory and create a new folder:

```
cd ..
mkdir really-large-application
cd really-large-application
// INITIALIZE NPM
npm init -y
```

Now, install your `colors` module and use the `--save` flag so it will be recorded in your `package.json` file:

```
npm install --save ../colors
```

You just installed your `colors` module in the new project. Open the `package.json` file to see the new local dependency:

```
{
  "name": "really-large-application",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "colors": "file:../colors"
  }
}
```

The `colors` module was copied to your `node_modules` directory. Verify it's there with the following command:

```
ls node_modules
// OUTPUT
colors
```

Use your installed local module in this new program. Re-open your text editor and create another JavaScript file `index.js` .

The program will first import the `colors` module. It will then choose a color at random using the `getRandomColor()` function provided by th module. Finally, it will print a message to the console that tells the user what color to use.

Enter the following code in `index.js` :

```
const colors = require('colors');

const chosenColor = colors.getRandomColor();
console.log(`You should use ${chosenColor.name} on your website. It's HTML code is ${chosenColor.code}`);
```

Save and exit this file.

Run the file :

```
node index.js
// OUTPUT
You should use leafygreen on your website. It's HTML code is #48C9B0
```

You've now successfully installed the `colors` module and can manage it like any other npm package used in your project. However, if you added mo colors and functions to your local `colors` module, you would have to run `npm update` in your applications to be able to use the new options.

Debugging Nodejs Applications

In Node.js development, tracing a coding error back to its source can save a lot of time over the course of a project. But as a program grows complexity, it becomes harder and harder to do this efficiently. To solve this problem, developers use tools like a *debugger*, a program that allov developers to inspect their program as it runs. By replaying the code line-by-line and observing how it changes the program's state, debuggers ca provide insight into how a program is running, making it easier to find bugs.

A common practice programmers use to track bugs in their code is to print statements as the program runs. In Node.js, that involves addir extra `console.log()` or `console.debug()` statements in their modules. While this technique can be used quickly, it is also manual, making it les scalable and more prone to errors. Using this method, it is possible to mistakenly log sensitive information to the console, which could provide maliciou agents with private information about customers or your application. On the other hand, debuggers provide a systematic way to observe what happening in a program, without exposing your program to security threats.

The key features of debuggers are *watching* objects and adding *breakpoints*. By watching objects, a debugger can help track the changes of a variab as the programmer steps through a program. Breakpoints are markers that a programmer can place in their code to stop the code from continuir beyond points that the developer is investigating.

We will first debug code using the built-in Node.js debugger tool, setting up watchers and breakpoints so you can find the root cause of a bug. Then we use Google Chrome DevTools as a Graphical User Interface (GUI) alternative to the command line Node.js debugger.

Using Watchers with the Node.js Debugger

Debuggers are primarily useful for two features: their ability to *watch* variables and observe how they change when a program is run and their ability stop and start code execution at different locations called *breakpoints*. In this step, we will run through how to watch variables to identify errors in code.

Watching variables as we step through code gives us insight into how the values of variables change as the program runs. Let's practice watchir variables to help us find and fix logical errors in our code with an example.

We begin by setting up our coding environment. In your terminal, create a new node project called `debugging`:

```
mkdir debugging
cd debugging
npm init -y
```

Open a new file called `badLoop.js`. Our code will iterate over an array and add numbers into a total sum, which in our example will be used to add u the number of daily orders over the course of a week at a store. The program will return the sum of all the numbers in the array. In the editor, enter th following code:

```
let orders = [341, 454, 198, 264, 307];

let totalOrders = 0;

for (let i = 0; i <= orders.length; i++) {
  totalOrders += orders[i];
}

console.log(totalOrders);
```

We start by creating the `orders` array, which stores five numbers. We then initialize `totalOrders` to `0`, as it will store the total of the five number In the `[for](https://www.digitalocean.com/community/tutorials/how-to-construct-for-loops-in-javascript)` loop, we iteratively ac each value in `orders` to `totalOrders`. Finally, we print the total amount of orders at the end of the program.

Save and exit from the editor. Now run this program with `node`.

```
// OUTPUT
NaN
```

`NaN` in JavaScript means **Not a Number**. Given that all the input are valid numbers, this is unexpected behavior. To find the error, let's use the Node. debugger to see what happens to the two variables that are changed in the `for` loop: `totalOrders` and `i`.

When we want to use the built-in Node.js debugger on a program, we include `inspect` before the file name. In your terminal, run the `node` commar with this debugger option as follows:

```
node inspect badLoop.js
```

When you start the debugger, you will find output like this:

```
< Debugger listening on ws://127.0.0.1:9229/e1ebba25-04b8-410b-811e-8a0c0902717a
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in badLoop.js:1
> 1 let orders = [341, 454, 198, 264, 307];
  2
  3 let totalOrders = 0;
```

The first line shows us the URL of our debug server. That's used when we want to debug with external clients, like a web browser as we'll see later o Note that this server listens on port `:9229` of the `localhost` (`127.0.0.1`) by default. For security reasons, it is recommended to avoid exposir this port to the public.

After the debugger is attached, the debugger outputs `Break on start in badLoop.js:1`.

Breakpoints are places in our code where we'd like execution to stop. By default, Node.js's debugger stops execution at the beginning of the file.

The debugger then shows us a snippet of code, followed by a special `debug` prompt:

```
...
> 1 let orders = [341, 454, 198, 264, 307];
  2
```

```
  3 let totalOrders = 0;
debug>
```

The `>` next to `1` indicates which line we've reached in our execution, and the prompt is where we will type in our commends to the debugger. Whe this output appears, the debugger is ready to accept commands.

When using a debugger, we step through code by telling the debugger to go to the next line that the program will execute. Node.js allows the followir commands to use a debugger:

- `c` or `cont` : Continue execution to the next breakpoint or to the end of the program.
- `n` or `next` : Move to the next line of code.
- `s` or `step` : Step into a function. By default, we only step through code in the block or scope we're debugging. By stepping into a function, we can inspect the code of the function our code calls and observe how it reacts to our data.
- `o` : Step out of a function. After stepping into a function, the debugger goes back to the main file when the function returns. We can use this command to go back to the original function we were debugging before the function has finished execution.
- `pause` : Pause the running code.

We'll be stepping through this code line-by-line. Press `n` to go to the next line.

Our debugger will now be stuck on the third line of code:

```
break in badLoop.js:3
  1 let orders = [341, 454, 198, 264, 307];
  2
> 3 let totalOrders = 0;
  4
  5 for (let i = 0; i <= orders.length; i++) {
```

Empty lines are skipped for convenience. If we press `n` once more in the debug console, our debugger will be situated on the fifth line of code:

```
break in badLoop.js:5
  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

We are now beginning our loop. If the terminal supports color, the `0` in `let i = 0` will be highlighted. The debugger highlights the part of the coc the program is about to execute, and in a `for` loop, the counter initialization is executed first. From here, we can watch to see why `totalOrders` returning `NaN` instead of a number. In this loop, two variables are changed every iteration— `totalOrders` and `i` . Let's set up watchers for both those variables.

We'll first add a watcher for the `totalOrders` variable. In the interactive shell, enter this:

```
debug> watch('totalOrders')
```

To watch a variable, we use the built-in `watch()` function with a string argument that contains the variable name. As we press `ENTER` c the `watch()` function, the prompt will move to the next line without providing feedback, but the watch word will be visible when we move the debugg to the next line.

Now let's add a watcher for the variable `i` :

```
debug> watch('i')
```

Now we can see our watchers in action. Press `n` to go to the next step. The debug console will show this:

```
break in badLoop.js:5
Watchers:
  0: totalOrders = 0
  1: i = 0

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
```

```
  6   totalOrders += orders[i];
  7 }
```

The debugger now displays the values of `totalOrders` and `i` before showing the line of code, as shown in the output. These values are update every time a line of code changes them.

At this point, the debugger is highlighting `length` in `orders.length`. This means the program is about to check the condition before it executes th code within its block. After the code is executed, the final expression `i++` will be executed.

Enter `n` in the console to enter the `for` loop's body:

```
break in badLoop.js:6
Watchers:
  0: totalOrders = 0
  1: i = 0

  4
  5 for (let i = 0; i <= orders.length; i++) {
> 6   totalOrders += orders[i];
  7 }
  8
```

This step updates the `totalOrders` variable. Therefore, after this step is complete our variable and watcher will be updated.

Press `n` to confirm. You will see this:

```
Watchers:
  0: totalOrders = 341
  1: i = 0

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

As highlighted, `totalOrders` now has the value of the first order: `341`.

Our debugger is just about to process the final condition of the loop. Enter `n` so we execute this line and update `i`:

```
break in badLoop.js:5
Watchers:
  0: totalOrders = 341
  1: i = 1

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

After initialization, we had to step through the code four times to see the variables updated. Stepping through the code like this can be tedious. But f now, by setting up our watchers, we are ready to observe their values and find our problem.

Step through the program by entering `n` twelve more times, observing the output. Your console will display this:

```
break in badLoop.js:5
Watchers:
  0: totalOrders = 1564
  1: i = 5

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

Recall that our `orders` array has five items, and `i` is now at position `5`. But since `i` is used as the index of an array, there is no valu at `orders[5]`; the last value of the `orders` array is at index `4`. This means that `orders[5]` will have a value of `undefined`.

Type `n` in the console and you'll observe that the code in the loop is executed:

```
break in badLoop.js:6
Watchers:
  0: totalOrders = 1564
  1: i = 5

  4
  5 for (let i = 0; i <= orders.length; i++) {
> 6   totalOrders += orders[i];
  7 }
  8
```

Typing `n` once more shows the value of `totalOrders` after that iteration:

```
break in badLoop.js:5
Watchers:
  0: totalOrders = NaN
  1: i = 5

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

Through debugging and watching `totalOrders` and `i`, we can see that our loop is iterating six times instead of fiv When `i` is `5`, `orders[5]` is added to `totalOrders`. Since `orders[5]` is `undefined`, adding this to a number will yield `NaN`. The proble with our code therefore lies within our `for` loop's condition. Instead of checking if `i` is less than or equal to the length of the `orders` array, w should only check that it's less than the length.

Let's exit our debugger, make the changes and run the code again. In the debug prompt, type the exit command and press `ENTER`:

```
debug> .exit
```

Now that you've exited the debugger, open `badLoop.js` in your text editor and change the for loop's condition.

If you run the program now, correct output will be displayed.

Using Breakpoints With the Node.js Debugger

It's common for Node.js projects to consist of many interconnected modules. Debugging each module line-by-line would be time consuming, especial as an app scales in complexity. To solve this problem, breakpoints allow us to jump to a line of code where we'd like to pause execution and inspect th program.

When debugging in Node.js, we add a breakpoint by adding the `debugger` keyword directly to our code. We can then go from one breakpoint to th next by pressing `c` in the debugger console instead of `n`. At each breakpoint, we can set up watchers for expressions of interest.

Let's see this with an example. In this step, we'll set up a program that reads a list of sentences and determines the most common word used througho all the text. Our sample code will return the first word with the highest number of occurrences.

For this exercise, we will create `sentence.txt` which will contain the raw data that our program will process. Enter the following te to `sentence.txt`:

```
Whale shark Rhincodon typus gigantic but harmless shark family Rhincodontidae that is the largest living fish
Whale sharks are found in marine environments worldwide but mainly in tropical oceans
They make up the only species of the genus Rhincodon and are classified within the order Orectolobiformes a group contai
The whale shark is enormous and reportedly capable of reaching a maximum length of about 18 metres 59 feet
Most specimens that have been studied however weighed about 15 tons about 14 metric tons and averaged about 12 metres 39
The body coloration is distinctive
Light vertical and horizontal stripes form a checkerboard pattern on a dark background and light spots mark the fins and
```

Save and exit the file.

Now let's add our code to `textHelper.js`. This module will contain some handy functions we'll use to process the text file, making it easier determine the most popular word. Open `textHelper.js` in your text editor.

We'll create three functions to process the data in `sentences.txt`. The first will be to read the file. Type the following into `textHelper.js`:

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

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  return sentences;
};
```

First, we import the [fs](https://nodejs.org/api/fs.html#fs_file_system) Node.js library so we can read files. We then crea the `readFile()` function that uses `readFileSync()` to load the data from `sentences.txt` as a `Buffer` object and the `toString()` metho to return it as a string.

The next function we'll add processes a string of text and flattens it to an array with its words. Add the following code into the editor:

```
...

const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
  let words = flatSentence.split(' ');
  words = words.map((word) => word.trim().toLowerCase());
  return words;
};
```

In this code, we are using the methods `split()`, `join()`, and `map()` to manipulate the string into an array of individual words. The function als lowercases each word to make counting easier.

The last function needed returns the counts of different words in a string array. Add the last function like this:

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (word in map) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
  });

  return map;
};

// Export these functions, to make them available to other modules
module.exports = { readFile, getWords, countWords };
```

Save and exit.

Our third and final file we'll use for this exercise will use the `textHelper.js` module to find the most popular word in our text. Open `index.js` wi your text editor. We begin our code by importing the `textHelpers.js` module and continue by creating a new array containing stop words.

Stop words are commonly used words in a language that we filter out before processing a text. We can use this to find more meaningful data than th result that the most popular word in English text is `the` or `a`.

Continue by using the `textHelper.js` module functions to get a JavaScript object with words and their counts. Here's what the code will be at th point:

```
const textHelper = require('./textHelper');

const stopwords = ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours', 'yourself', 'you

let sentences = textHelper.readFile();
```

```
let words = textHelper.getWords(sentences);
let wordCounts = textHelper.countWords(words);
```

We can then complete this module by determining the words with the highest frequency. To do this, we'll loop through each key of the object with th word counts and compare its count to the previously stored maximum. If the word's count is higher, it becomes the new maximum.

Add the following lines of code to compute the most popular word:

```
...

let max = -Infinity;
let mostPopular = '';

Object.entries(wordCounts).forEach(([word, count]) => {
  if (stopwords.indexOf(word) === -1) {
    if (count > max) {
      max = count;
      mostPopular = word;
    }
  }
});

console.log(`The most popular word in the text is "${mostPopular}" with ${max} occurrences`);
```

In this code, we are using `Object.entries()` to transform the key-value pairs in the `wordCounts` object into individual arrays, all of which a nested within a larger array. We then use the `forEach()` method and some conditional statements to test the count of each word and store the highe number.

Save and exit the file.

Let's now run this file to see it in action

```
node index.js
// OUTPUT
The most popular word in the text is "whale" with 1 occurrences
```

From reading the text, we can see that the answer is incorrect. A quick search in `sentences.txt` would highlight that the word `whale` appears mo than once.

We have quite a few functions that can cause this error: We may not be reading the entire file, or we may not be processing the text into the array ar JavaScript object correctly. Our algorithm for finding the maximum word could also be incorrect. The best way to figure out what's wrong is to use th debugger.

Even without a large codebase, we don't want to spend time stepping through each line of code to observe when things change. Instead, we can us breakpoints to go to those key moments before the function returns and observe the output.

Let's add breakpoints in each function in the `textHelper.js` module. To do so, we need to add the keyword `debugger` into our code.

Open the `textHelper.js` file in the text editor. First, we'll add the breakpoint to the `readFile()` function like this:

```
...

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  debugger;
  return sentences;
};

...
```

Next, we'll add another breakpoint to the `getWords()` function:

```
...

const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
```

```
  let words = flatSentence.split(' ');
  words = words.map((word) => word.trim().toLowerCase());
  debugger;
  return words;
};

...
```

Finally, we'll add a breakpoint to the `countWords()` function:

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (word in map) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
  });

  debugger;
  return map;
};

...
```

Save and exit `textHelper.js`.

Let's begin the debugging process. Although the breakpoints are in `textHelpers.js`, we are debugging the main point of entry of o application: `index.js`. Start a debugging session by entering the following command in your shell:

```
node inspect index.js
// OUTPUT
< Debugger listening on ws://127.0.0.1:9229/b2d3ce0e-3a64-4836-bdbf-84b6083d6d30
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in index.js:1
> 1 const textHelper = require('./textHelper');
  2
  3 const stopwords = ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours', 'yourself',
```

This time, enter `c` into the interactive debugger. As a reminder, `c` is short for continue. This jumps the debugger to the next breakpoint in the cod After pressing `c` and typing `ENTER`, you will see this in your console:

```
break in textHelper.js:6
  4   let data = fs.readFileSync('sentences.txt');
  5   let sentences = data.toString();
> 6   debugger;
  7   return sentences;
  8 };
```

We've now saved some debugging time by going directly to our breakpoint.

In this function, we want to be sure that all the text in the file is being returned. Add a watcher for the `sentences` variable so we can see what's beir returned. Press `n` to move to the next line of code so we can observe what's in `sentences`. You will see the following output:

```
break in textHelper.js:7
Watchers:
  0: sentences =
    'Whale shark Rhincodon typus gigantic but harmless shark family Rhincodontidae that is the largest living fish\n' +
      'Whale sharks are found in marine environments worldwide but mainly in tropical oceans\n' +
      'They make up the only species of the genus Rhincodon and are classified within the order Orectolobiformes a group
```

```
      'The whale shark is enormous and reportedly capable of reaching a maximum length of about 18 metres 59 feet\n' +
      'Most specimens that have been studied however weighed about 15 tons about 14 metric tons and averaged about 12 me
      'The body coloration is distinctive\n' +
      'Light vertical and horizontal stripes form a checkerboard pattern on a dark background and light spots mark the f

  5   let sentences = data.toString();
  6   debugger;
> 7   return sentences;
  8 };
  9
```

It seems that we aren't having any problems reading the file; the problem must lie elsewhere in our code. Let's move to the next breakpoint b pressing `c` once again. When you do, you'll see this output:

```
break in textHelper.js:15
Watchers:
  0: sentences =
    ReferenceError: sentences is not defined
        at eval (eval at getWords (your_file_path/debugger/textHelper.js:15:3), <anonymous>:1:1)
        at Object.getWords (your_file_path/debugger/textHelper.js:15:3)
        at Object.<anonymous> (your_file_path/debugger/index.js:7:24)
        at Module._compile (internal/modules/cjs/loader.js:1125:14)
        at Object.Module._extensions..js (internal/modules/cjs/loader.js:1167:10)
        at Module.load (internal/modules/cjs/loader.js:983:32)
        at Function.Module._load (internal/modules/cjs/loader.js:891:14)
        at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:71:12)
        at internal/main/run_main_module.js:17:47

 13   let words = flatSentence.split(' ');
 14   words = words.map((word) => word.trim().toLowerCase());
>15   debugger;
 16   return words;
 17 };
```

We get this error message because we set up a watcher for the `sentences` variable, but that variable does not exist in our current function scope. watcher lasts for the entire debugging session, so as long as we keep watching `sentences` where it's not defined, we'll continue to see this error.

We can stop watching variables with the `unwatch()` command. Let's unwatch `sentences` so we no longer have to see this error message eve time the debugger prints its output. In the interactive prompt, enter this command:

```
degub> unwatch('sentences')
```

The debugger does not output anything when you unwatch a variable.

Back in the `getWords()` function, we want to be sure that we are returning a list of words that are taken from the text we loaded earlier. Let's watch th value of the `words` variable. Then, enter `n` to go to the next line of the debugger, so we can see what's being stored in `words`. The debugger w show the following:

```
break in textHelper.js:16
Watchers:
  0: words =
    [ 'whale',
      'shark',
      'rhincodon',
      'typus',
      'gigantic',
      'but',
      'harmless',
      ...
      'metres',
      '39',
      'feet',
      'in',
      'length',
      '',
```

```
      'the',
      'body',
      'coloration',
      ... ]

 14   words = words.map((word) => word.trim().toLowerCase());
 15   debugger;
>16   return words;
 17 };
 18
```

The debugger does not print out the entire array as it's quite long and would make the output harder to read. However, the output meets our expectatior of what should be stored: the text from `sentences` split into lowercase strings. It seems that `getWords()` is functioning correctly.

Let's move on to observe the `countWords()` function. First, unwatch the `words` array so we don't cause any debugger errors when we are at th next breakpoint. Next, enter `c` in the prompt. At our last breakpoint, we will see this in the shell:

```
break in textHelper.js:29
 27   });
 28
>29   debugger;
 30   return map;
 31 };
```

In this function, we want to be sure that the `map` variable correctly contains the count of each word from our sentences. First, let's tell the debugger watch the `map` variable. Press `n` to move to the next line. The debugger will then display this:

```
break in textHelper.js:30
Watchers:
  0: map =
    { 12: NaN,
      14: NaN,
      15: NaN,
      18: NaN,
      39: NaN,
      59: NaN,
      whale: 1,
      shark: 1,
      rhincodon: 1,
      typus: NaN,
      gigantic: NaN,
      ... }

 28
 29   debugger;
>30   return map;
 31 };
 32
```

That does not look correct. It seems as though the method for counting words is producing erroneous results. We don't know why those values are beir entered, so our next step is to debug what's happening in the loop used on the `words` array. To do this, we need to make some changes to where w place our breakpoint.

First, exit the debug console and open `textHelper.js` so we can edit the breakpoints. Knowing that `readFile()` and `getWords()` are workin we will remove their breakpoints. We then want to remove the breakpoint in `countWords()` from the end of the function, and add two new breakpoin to the beginning and end of the `forEach()` block.

Edit `textHelper.js` so it looks like this:

```
...

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  return sentences;
};
```

```
const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
  let words = flatSentence.split(' ');
  words = words.map((word) => word.trim().toLowerCase());
  return words;
};

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    debugger;
    if (word in map) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
    debugger;
  });

  return map;
};

...
```

Save and Exit and start the debugger again. To get insight into what's happening, we want to debug a few things in the loop. First, let's set up a watch for `word`, the argument used in the `forEach()` loop containing the string that the loop is currently looking at.

So far, we have only watched variables. But watches are not limited to variables. We can watch any valid JavaScript expression that's used in our code.

In practical terms, we can add a watcher for the condition `word in map`, which determines how we count numbers. In the debug prompt, create th watcher:

```
debug> watch('word in map')
//Let's also add a watcher for the value that's being modified in the map variable:
debug> watch('map[word]')
```

Watchers can even be expressions that aren't used in our code but could be evaluated with the code we have. Let's see how this works by adding watcher for the length of the `word` variable:

```
debug> watch('word.length')
```

Now that we've set up all our watchers, let's enter `c` into the debugger prompt so we can see how the first element in the loop of `countWords()` evaluated. The debugger will print this output:

```
break in textHelper.js:20
Watchers:
  0: word = 'whale'
  1: word in map = false
  2: map[word] = undefined
  3: word.length = 5

 18   let map = {};
 19   words.forEach((word) => {
>20     debugger;
 21     if (word in map) {
 22       map[word] = 1;
```

The first word in the loop is `whale`. At this point, the `map` object has no key with `whale` as its empty. Following from that, when lookir up `whale` in `map`, we get `undefined`. Lastly, the length of `whale` is `5`. That does not help us debug the problem, but it does validate that w can watch any expression that could be evaluated with the code while debugging.

Press `c` once more to see what's changed by the end of the loop. The debugger will show this:

```
break in textHelper.js:26
Watchers:
  0: word = 'whale'
  1: word in map = true
  2: map[word] = NaN
  3: word.length = 5

 24       map[word] += 1;
 25     }
>26     debugger;
 27   });
 28
```

At the end of the loop, `word in map` is now true as the `map` variable contains a `whale` key. The value of `map` for the `whale` key is `NaN`, whic highlights our problem. The `if` statement in `countWords()` is meant to set a word's count to one if it's new, and add one if it existed already.

The culprit is the `if` statement's condition. We should set `map[word]` to `1` if the `word` is not found in `map`. Right now, we are adding or if `word` is found. At the beginning of the loop, `map["whale"]` is `undefined`. In JavaScript, `undefined + 1` evaluates to `NaN` —not a number.

The fix for this would be to change the condition of the `if` statement from `(word in map)` to `(!(word in map))`, using the `!` operator to te if `word` is not in `map`. Let's make that change in the `countWords()` function to see what happens.

Exit the debugger and edit `textHelper.js` and modify the `countWords()` function as following:

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (!(word in map)) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
  });

  return map;
};

...
```

Running this code will now will produce the correct output.

Conclusion

In this session we've learned :

- NodeJS core modules like Buffer
- Management of modules using npm
- Creation and management of user-defined modules
- Debugging of NodeJS applications

Interview Questions

> How can you manage the packages in your Node.js project?

We can manage the packages in our Node.js project by using several package installers and their configuration file accordingly. Most of them use npm yarn. The npm and yarn both provide almost all libraries of JavaScript with extended features of controlling environment-specific configurations. We ca use package.json and package-lock.json to maintain versions of libs being installed in a project. So, there is no issue in porting that app to a differe environment.

> What are the modules in Node.js?

In Node.js applications, modules are like JavaScript libraries and include a set of functions. To include a module in a Node.js application, we must us the require() function with the parentheses containing the module's name.

> What are buffers in Node.js?

In general, a buffer is a temporary memory mainly used by the stream to hold on to some data until it is consumed. Buffers are used to represent a fixe size chunk of memory allocated outside of the V8 JavaScript engine. It can't be resized. It is like an array of integers, which each represents a byte data. It is implemented by the Node. js Buffer class. Buffers also support legacy encodings like ASCII, utf-8, etc.

> What is the package.json file?

The package.json file is the heart of a Node.js system. This file holds the metadata for a particular project. The package.json file is found in the ro directory of any Node application or module.

Thank You !