HyperionDev

Node.js

Task

Visit our website

Introduction

Welcome to the Node.js task! Node.js lets you run JavaScript on servers, making it possible to write server-side code. It uses an event-driven, non-blocking model, allowing it to handle many tasks simultaneously, which is ideal for apps that need to be fast and support many users.

Node.js includes built-in tools (modules) to simplify tasks like file management. The **File System** (fs) module, for instance, allows for the reading, writing, and managing of files. This is useful for storing or displaying content on websites.

Before Node.js, PHP was commonly used, but it struggled with real-time interactions. Node.js, by contrast, improved server speed and enabled real-time functionality. In this task, we'll install Node.js and write simple programs to explore its features.

Node.js

Installing Node.js

To get started, you'll need to refer to the "**Additional Reading – Installation, Sharing, and Collaboration**" guide for detailed instructions on installing Node.js. This guide will take you step-by-step through the installation process on your operating system.

Once you've completed the installation, you'll be ready to start writing and running your first Node.js programs.

Getting started with Node.js

Now that you have your Node.js installation set up, let's say hello!

1. **Create your file:** Make a new file called **helloWorld.js** and add the following line of code:

   ```
   console.log("Hello world!");
   ```

2. **Open your command-line interface (CLI):** To run your program, you'll need to use your operating system's command-line interface.

3. **Navigate to your file location:** Use the `cd` command to navigate to the directory where you saved `helloWorld.js`.

4. **Run your code:** Once you're in the correct directory, type the following command to execute your script:

```
node helloWorld.js
```

5. **View the output:** Your code will run, and you should see the message “Hello world!” printed in the command line.

Managing Node packages

Now that we know how to run a JavaScript file using Node.js, let’s explore some practical applications. When you downloaded and installed Node.js, you also installed the **Node package manager** (NPM). NPM is a tool that helps you manage packages, which are collections of reusable code that can add specific functionality to your Node.js applications.

A **package** can contain one or more **modules**, and it typically includes a **package.json** file that provides metadata about the package, such as its name, version, and **dependencies**. NPM allows you to easily install, update, and manage these packages, making it simple to incorporate external libraries and tools into your projects.

Understanding modules

A **module** is a reusable piece of code that has been organised into a separate file. Modules can help break applications into smaller, more manageable parts, making them easier to develop and maintain.

Key benefits of using modules:

- **Encapsulation:** Encapsulation is a core principle in software development that allows you to group related functions, variables, and data into a single unit, known as a module. By encapsulating specific functionalities within a module, you create a clear boundary between different sections of your code.
- **Reusability:** Once you create a module, you can use it throughout various parts of your application or even in different projects. This encourages code reuse and reduces duplication, ultimately saving you time and effort during development. By leveraging reusable modules, you can build applications more efficiently and maintain consistency within your codebase.

When using modules in your projects, there are two primary methods of importing them:

1. **Common JavaScript syntax (traditional way):** This method uses the `require()` function to import modules. For example, to import a module named `mathUtils.js`, you would write:

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

2. **ES6 syntax:** ES6, also known as ECMAScript 2015, introduced a more modern approach to importing modules. Using ES6 syntax, it's possible for us to use the `import` statement to bring in modules. For instance, to import the same `mathUtils.js` module, you would write:

```
import { add } from './mathUtils.js';
```

Node.js includes a few built-in modules, such as `fs` (file system), `http` (HTTP server), and `path` (file and directory paths), which can be used without additional installation.

To create your own modules, you can easily export functions, objects, or variables from a JavaScript file. For example, if you created a file named `mathUtils.js`, you might have exported a function called `add`:

```
function add(a, b) {
    return a + b;
}

module.exports = { add };
```

Creating your own package

Let's create a package called `my_first_package`. Follow these steps to learn how to create and manage your own package:

1. **Create your package directory:** Open your terminal or command line and type the following command to create a new directory:

```
mkdir my_first_package
```

2. **Navigate into your package directory:** Change into the new directory by typing:

```
cd my_first_package
```

3. **Initialise a new Node package:** To set up your new package, run the following command in the terminal or command line after navigating to your package directory:

```
npm init
```

Understanding npm init

When you run `npm init`, you're creating a new Node package. This command sets up a `package.json` file in your module's directory. The `package.json` file is essential because it contains important information about your module, such as its name, version, description, entry point (the main file), scripts, dependencies, and more.

Initialising a new Node module is required because it helps manage your project's configuration and dependencies. Without it, Node wouldn't know how to handle your module, making it difficult to install other packages or share your module with others.

Simply hit “Enter” at each prompt for all of the defaults unless you want to change specific values. After running this command, you will see a new file called **package.json** created in the directory.

Adding external packages with NPM

Congratulations! You have now created your first package, which is a huge step in building modular, reusable code. However, writing and creating everything from scratch can be time-consuming, especially when there is common functionality that you are attempting to implement. That’s where external packages come in, and this is where NPM starts to shine.

One of the major benefits of using Node.js and NPM is the vast collection of packages that are available to help you accomplish almost any task. External packages are bundles of reusable code that have been created by others, which you can easily add to your own package.

To start with, let’s include a common utility package called **Lodash**. This package contains helpful utility functions that simplify working with JavaScript data structures.

To add Lodash to your package, open the terminal and start by navigating into the folder where you created your package (unless you are already there). This is the same folder where you ran `npm init` and created the **package.json** file.

Once you are in the correct folder, install Lodash by typing:

```
npm install lodash
```

Running this command does several things:

1. **Downloads Lodash:** Using NPM it fetches the Lodash package and downloads it into a special folder called **node_modules** inside your package directory. This is where all external packages for your project will be stored.

2. **Updates package.json:** Once Lodash has been installed, NPM will automatically update your **package.json** file to include Lodash as a dependency for your project. If you open the **package.json** file you might notice a new section similar to the following:

   ```
   "dependencies": {
       "lodash": "^4.17.21"
     }
   ```

   This addition within the **package.json** file indicates that the package is dependent on Lodash and also specifies the installed version.

3. **Creates or updates package-lock.json:** Along with the above, you will notice a file called **package-lock.json**. This file contains detailed information about the exact version of Lodash that was installed along with its dependencies, their versions, and other metadata.

Using Lodash in your project

After installing Lodash, let's take a look at how it can be used in JavaScript. For this example, we'll use Lodash's `first()` function, which returns the first element in an array.

To get started, follow these steps:

1. **Create a new JavaScript file:** In the same directory where your **package.json** is located, create a new file named **index.js**. This file will be used to contain the code that utilises Lodash.

2. **Import Lodash:** At the top of your **index.js** file, you can import Lodash using either CommonJS or ES6 syntax. Here's how:

   A. **Using CommonJS:**

   ```
   const lodash = require('lodash'); // Importing Lodash using CommonJS
   ```

   With the above syntax, the `require()` function dynamically imports the Lodash module and assigns it to the variable `lodash`. This is the traditional way of importing modules in Node.js.

   B. **Using ES6:**

   ```
   import lodash from 'lodash'; // Importing Lodash using ES6 syntax
   ```

   The ES6 syntax uses the import keyword, which is part of the modern JavaScript standard. This approach allows for a more declarative style of importing modules. However, to use ES6 syntax in Node.js, ensure your environment is set up properly by including `"type": "module"` in your **package.json**. Alternatively, you can use third-party tools to facilitate ES6 module imports into your project.

3. **Use Lodash's first() function:** After importing Lodash, we can use one of its utility functions. For this example, we will be using `lodash.first()` to get the first element from an array:

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

   // An array of numbers
   const numbers = [10, 20, 30, 40, 50];

   // Using Lodash to get the first element of the array
   const firstElement = lodash.first(numbers);

   // Output: The first element is: 10
   console.log(`The first element is: ${firstElement}`);
   ```

4. **Run the code:** To see the result of the code, we can run the code by navigating back to the terminal and running the following command in the same folder as the **package.json** and **index.js** files:

   ```
   node index.js
   ```

Installed packages: CommonJS vs ES6

As demonstrated above, there are two main module systems used to import modules in Node.js, namely **CommonJS** and **ES6**.

CommonJS is the most commonly used module system in Node.js. It uses the `require()` method to import modules dynamically at runtime. This means that modules are loaded as the code runs, allowing for flexibility in determining which module to import. You can even use it within conditionals or loops.

For example:

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

ES6 is the module system supported by modern web browsers. It uses the `import` statement for static imports, meaning that the modules are loaded at compile-time, before the code runs. Because of this static nature, you cannot use variables or place `import` statements inside conditionals or loops.

Here's an example of ES6 syntax:

```
import lodash from 'lodash';
```

For the ES6 import syntax to work in a Node.js environment, you need to ensure that your **package.json** file includes `"type": "module"`. This configuration tells Node.js to treat your files as ES6 modules, enabling the use of the `import` statement.

Set up your own scripts in Node.js

Sometimes, navigating file names can be confusing, especially when needing to input different arguments. NPM provides a handy tool to navigate this: **scripting**.

Recall the **package.json** file that was set up when you initialised your module. There are a few impressive features included in the file, one of which is using scripts. In fact, it comes bundled with one default script:

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

This could be more helpful. However, if you have set up a testing framework, this can end up being handy. To better understand how this works, let's make an example ourselves.

To get started, follow these steps:

1. **Create your files:** You can start by creating two files, `foo.js` and `bar.js`. In `foo.js`, place the following code:

   ```
   console.log("I am working right now.")
   ```

   In `bar.js`, place the following code:

   ```
   console.log("Goodnight!")
   ```

2. **Modify package.json:**

   Now, let's make a modification to our **package.json** file to include the custom scripts:

   ```
   "scripts": {
     "test": "echo \"Error: no test specified\" && exit 1",
     "work": "node foo.js",
     "sleep": "node bar.js"
   },
   ```

3. **Run your scripts:**

   You can now run these scripts using your terminal. Make sure you are in the same directory as your two JavaScript files.

   To run the `work` script, type:

   ```
   npm run work
   ```

   You should see output similar to this:

   ```
   > my_first_package@1.0.0 work
   > node foo.js

   I am working right now.
   ```

   Similarly, to run the `sleep` script, type:

   ```
   npm run sleep
   ```

You will get an output similar to the one below:

```
> my_first_package@1.0.0 sleep
> node bar.js

Goodnight!
```

This approach of creating predefined scripts with simple names is an effective way to streamline your workflow.

In conclusion, managing Node packages like Lodash with NPM, and setting up custom scripts in Node.js, streamlines development, enhances code reusability, and boosts productivity.

Practical task

Follow these steps:

- Create a package called **my_first_task**.
- Initialise the package using NPM.
- Install **Lodash** within the package.
- Create a script called **remove_duplicates.js**.
- Within this script, you will need to import **Lodash**, and use the **uniq** function.
- Create the following array:
  `[1, 2, 10, 100, 10, 2, 5, 6, 10, 1000, 7, 2, 100, 1, 5, 7, 10]`
- Display the original array in the console.
- Using Lodash, display that same array, but with all duplicates removed.
- Finally, set up your package to run the script using:

```
npm run rdup
```

Once you are ready to have your code reviewed, **delete** the **node_modules** folder (this folder typically contains hundreds of files which has the potential to **slow down your computer**), compress your project folder, and add it to the relevant task folder.

Be sure to place files for submission inside your **task folder** and click "**Request review**" on your dashboard.

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Do you think we've done a good job or do you think the content of this task, or this course as a whole, can be improved?

Share your thoughts anonymously using this **form**.