Day 3: NPM and Package Management

- Introduction to NPM (Node Package Manager)
- Installing and managing packages
- Understanding package.json
- Creating and publishing a package

Introduction to NPM (Node Package Manager)

NPM (Node Package Manager) is a powerful tool for managing dependencies and libraries in Node.js projects. It allows developers to share, reuse, and manage code in the form of packages. These packages can be libraries, tools, frameworks, or modules that can be easily installed and integrated into a Node.js project.

NPM has two main components:

1. **Website (npmjs.com)**: A platform where developers can search for and publish packages.
2. **Command-Line Interface (CLI)**: A tool that comes with Node.js, which helps in installing and managing packages.

Installing NPM:

When you install Node.js, NPM is automatically installed along with it. You can verify the installation by running the following command:

npm -v

This command will display the version of NPM installed on your system.

Why NPM is Important:

- **Simplifies Dependency Management**: NPM makes it easy to install and manage the third-party libraries your project depends on.
- **Community and Ecosystem**: NPM hosts over a million packages that can be used to build robust applications.
- **Code Reusability**: With NPM, you can reuse packages written by others, which speeds up development.

2. The package.json File

When managing packages, NPM uses a file called package.json to store the metadata about your project and its dependencies. It's automatically created when you initialize a Node.js project using:

```
npm init
```

Here's what the package.json typically contains:

- **Name**: The name of your project.

- **Version**: The version of your project.
- **Dependencies**: Lists the packages your project depends on.
- **DevDependencies**: Lists packages needed only for development (like testing tools).
- **Scripts**: Commands that can be run using npm run.

Key Sections of package.json

1. **name**
   - The name of your project or package. This should be unique if you plan to publish it to the NPM registry.

```
"name": "my-awesome-project"
```

version

- The version of your project, usually in the format major.minor.patch (e.g., 1.0.0). This follows the semantic versioning convention, which ensures that you communicate changes in the software effectively.

```
"version": "1.0.0"
```

description

- A brief description of your project. This is useful for others to understand what your project does, especially if you publish it.

```
"description": "This is an awesome Node.js project"
```

main

- This defines the entry point of your application. It points to the main file of your project, which will be executed when your package is required by another module.

```
"main": "index.js"
```

scripts

- This section allows you to define custom commands that you can run using npm run <script-name>. You can automate tasks such as testing, building, or running your app.

```
"scripts": {
   "start": "node index.js",
   "test": "mocha"
 }
```

dependencies

- This section lists all the packages your project needs to function. When a user runs npm install, these dependencies will be installed.

```
"dependencies": {
   "express": "^4.17.1",
   "lodash": "^4.17.21"
 }
```

The caret (^) before a version number indicates that NPM can install updates that don't change the major version (i.e., non-breaking changes).

devDependencies

- These are packages required only during the development of your project (e.g., testing frameworks, build tools). They are not installed when the project is in production.

```
"devDependencies": {
   "mocha": "^8.3.2",
   "eslint": "^7.25.0"
 }
```

These packages can be installed with npm install --save-dev.

repository

- This section provides the URL of the repository where your project is hosted (e.g., GitHub). It helps others find the source code of your project.

```
"repository": {
   "type": "git",
   "url": "https://github.com/username/my-awesome-project.git"
 }
```

keywords

- A list of keywords that describe your project. These are useful for making your project discoverable in search engines or the NPM registry.

```
"keywords": [
   "node",
   "express",
   "web",
   "server"
 ]
```

author

- Information about the author of the project.

```
"author": "mahesh <mahesh@example.com>"
```

license

- Specifies the license under which your project is released (e.g., MIT, GPL). This is important for open-source projects.

```
"license": "MIT"
```

engines

- Defines the versions of Node.js or NPM that your project is compatible with.

```

"engines": {
    "node": ">=14.0.0",
    "npm": ">=6.0.0"
  }
```

bugs

- Information on where users can report issues or bugs for your project.

```
"bugs": {
    "url": "https://github.com/username/my-awesome-project/issues"
  }
```

homepage

- The URL of the homepage for your project (e.g., a GitHub repository or documentation page).

```
"homepage": "https://github.com/username/my-awesome-project"
```

```
{
    "name": "my-awesome-project",
    "version": "1.0.0",
    "description": "This is an awesome Node.js project",
    "main": "index.js",
    "scripts": {
      "start": "node index.js",
      "test": "mocha"
    },
  "dependencies": {
    "express": "^4.17.1",
    "lodash": "^4.17.21"
  },
  "devDependencies": {
    "mocha": "^8.3.2",
    "eslint": "^7.25.0"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/username/my-awesome-project.git"
  },
  "keywords": [
    "node",
    "express",
    "web",
    "server"
  ],
  "author": "mahesh <mahesh@example.com>",
  "license": "MIT",
  "engines": {
    "node": ">=14.0.0",
    "npm": ">=6.0.0"
  },
  "bugs": {
    "url": "https://github.com/username/my-awesome-project/issues"
  },
  "homepage": "https://github.com/username/my-awesome-project"
}
```

Creating and Publishing a Package

Understanding:

- **Creating a package** means writing a piece of code that solves a specific problem or adds certain functionality, and then making it reusable for other developers.
- **Publishing a package** is when you share your code with the developer community by uploading it to the NPM registry, so other developers can install and use it in their projects.

Practical Example: Setting Up a Simple Node.js Project with NPM

Step 1: Create a Simple Node.js Project

Let’s say you want to build a **simple web server** using Node.js. You’ll use NPM to install a package that simplifies this process.

1. Create a new directory for your project:

```
mkdir my-npm-example
cd my-npm-example
```

Initialize the project with npm init:

Run the following command to create a **package.json** file for your project. This file will keep track of your project's metadata (name, version, etc.) and its dependencies (installed packages).

```
npm init
```

NPM will ask you a few questions like:

- **name**: random-quote-generator
- **version**: (default is 1.0.0)
- **entry point**: (default is index.js)

After answering the questions, you'll see a package.json file created in your project folder.

Check the package.json:

```
{
  "name": "random-quote-generator",
  "version": "1.0.0",
  "description": "An npm package to random-quote-generator",
  "main": "index.js",
  "scripts": {
    "test": "npm run test"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/HemaMaruthi/Front-End-Development.git"
  },
  "keywords": [
    "npm"
  ],
  "author": "Mahesh H",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/HemaMaruthi/Front-End-Development/issues"
  },
  "homepage": "https://github.com/HemaMaruthi/Front-End-Development#readme"
}
```

Create the index.js file:

```
const quotes = [
   "The only limit to our realization of tomorrow is our doubts of today. – Franklin D.
Roosevelt",
   "The purpose of our lives is to be happy. – Dalai Lama",
   "Life is what happens when you're busy making other plans. – John Lennon",
   "Get busy living or get busy dying. – Stephen King",
   "You only live once, but if you do it right, once is enough. – Mae West",
   "In the end, we only regret the chances we didn’t take. – Lewis Carroll"
 ];

 // Function to get a random quote
 function getRandomQuote() {
   const randomIndex = Math.floor(Math.random() * quotes.length);
   console.log(quotes[randomIndex])
   return quotes[randomIndex];
 }
 getRandomQuote()
 // Export the function
 module.exports = getRandomQuote;
```

Test the Package Locally

```
 npm link
```

Create script.js

```
mkdir test-quote-app
cd test-quote-app
npm init -y
```

```
npm link random-quote-generator
```

```
import pkg from 'random-quote-generator';
const {getRandomQuote} = pkg
```

or

```
import {getRandomQuote} from 'random-quote-generator';
getRandomQuote()
```

Run the code

```
node index.js
```

Prepare for Publishing:

Before publishing, ensure that you have logged into NPM using:

```
npm login
```

You'll need to provide your username, password, and email for your NPM account.

Publish the Package

```
  npm publish
```

Global test:

Creating new folder ➔ test.js

```
  npm install random-quote-generator
```

```
console.log("Hello")

import random from 'node-random-quote-generator';
random("Mahesh")
```

Updating the Package

If you make changes to your package, you will need to increment the version in the package.json file before publishing it again. You can do this manually or use the following command to bump the version automatically:

For a patch (bug fix):

npm version patch

For a minor update (new features, no breaking changes):

npm version minor

For a major update (breaking changes):

npm version major

Then, publish the updated version:

npm publish

3. Installing and Managing Packages with NPM

NPM makes it easy to install, update, and remove packages in a Node.js project. Packages can be libraries, tools, frameworks, or modules that you need to use in your application. NPM allows you to manage these packages both globally and locally within your project.

a. Installing Packages

There are two main types of installations in NPM:

- **Local Installation** (Default): Installs the package only for your project and adds it to the node_modules directory within your project folder.
- **Global Installation**: Installs the package globally on your system so that it can be accessed anywhere via the command line.

Local Installation (Default)

When you install a package locally, it's available only within that specific project. To install a package locally, run the following command:

```
npm install <package-name>
```

Global Installation

To install a package globally, use the -g flag. Global packages are installed in a system directory, making them accessible from anywhere on your system.

```
npm install -g <package-name>
```

b. Managing Packages

NPM also provides several commands to manage installed packages, including updating, uninstalling, and checking the installed version.

Listing Installed Packages

To view all the installed packages in your project, use:

```
npm list
npm list -g
```

Updating Packages

To update a package to its latest version, use the npm update command. This updates the specified package within your project:

```
npm update <package-name>
```

Uninstalling Packages

To remove a package from your project, use the npm uninstall command:

```
npm uninstall <package-name>
```

Interview Questions:

1. Introduction to NPM (Node Package Manager)

Q1. What is NPM, and why is it important in Node.js development?
Answer:
NPM (Node Package Manager) is the default package manager for Node.js. It allows developers to install, manage, and share packages (modules) of JavaScript code. NPM is crucial because it provides access to a vast ecosystem of reusable packages, helps manage dependencies in a project, and simplifies code sharing.

Q2. How do you check the version of NPM installed on your system?
Answer:
You can check the installed version of NPM using the following command in the terminal:

```
npm -v
```

Q3. What are the two main components of NPM?
Answer:
NPM has two main components:

1. **The NPM Registry**: A public repository (npmjs.com) where developers can publish and download packages.
2. **The NPM CLI**: A command-line tool used for interacting with the NPM registry, installing, and managing packages.

Q4. What's the difference between a local and a global installation of an NPM package?
Answer:

- **Local installation** installs the package within the current project, under the node_modules directory. It is available only to that project. Example:

```
npm install express
```

Global installation installs the package globally on the system, making it available to all projects. Example:

```
npm install -g nodemon
```

Installing and Managing Packages

Q1. How do you install a package locally in a Node.js project? Provide an example.
Answer:
You can install a package locally by running the following command in the terminal:

```
npm install <package-name>
```

Example:

```
npm install express
```

This installs express in the node_modules folder of the current project and adds it as a dependency in package.json

Q2. What is the purpose of the -g flag when installing an NPM package?
Answer:
The -g flag is used to install a package globally, meaning the package will be available system-wide and can be used in any Node.js project without needing to install it locally. This is typically used for command-line tools like nodemon or http-server.

Q3. How can you install a specific version of a package using NPM?
Answer:
You can install a specific version of a package by specifying the version number after the package name:

```
npm install <package-name>@<version>
```

Q4. What is the difference between dependencies and devDependencies?
Answer:

- **dependencies**: These are the packages required for the project to run in production. They are installed when you run npm install without any flags.
- **devDependencies**: These are packages only needed during development, such as testing frameworks or build tools. They are installed with the --save-dev flag and are not required in the production environment.

```
npm install mocha --save-dev
```

Q5. How do you uninstall a package, and what impact does it have on the package.json file?
Answer:
You can uninstall a package using the following command:

```
npm uninstall <package-name>
```

This will remove the package from the node_modules directory and also delete it from the dependencies or devDependencies section in package.json.

3. Understanding package.json

Q1. What is package.json, and why is it essential in a Node.js project?
Answer:
package.json is a file that contains metadata about the project, including its name, version, description, author, and the list of dependencies. It is essential because it defines the project's

dependencies and scripts, making it easy to share, version, and replicate the project environment on different machines.

Q2. What does the main field in package.json specify?
Answer:
The main field in package.json specifies the entry point of the application (the file that Node.js should load first when the package is required). By default, it is index.js, but it can be set to any file.

```
"main": "app.js"
```

Q3. What is the purpose of the scripts field in package.json? Provide an example.
Answer:
The scripts field defines command-line scripts that can be run using npm run <script-name>. It is used to automate common tasks such as testing, building, or starting the server.

```
"scripts": {
  "start": "node index.js",
  "test": "mocha"
}
```

Running npm run start will execute node index.js, and npm run test will run mocha.

Q4. Explain semantic versioning and how NPM uses it to manage package versions (e.g., ^, ~).
Answer:
Semantic versioning follows the MAJOR.MINOR.PATCH format:

- **MAJOR**: Introduces breaking changes.
- **MINOR**: Adds backward-compatible features.
- **PATCH**: Fixes bugs without adding new features.
- ^: Allows updating to newer minor or patch versions (non-breaking changes).
- ~: Allows updating to newer patch versions only.

For example, "lodash": "^4.17.0" will accept any version of lodash that is >=4.17.0 <5.0.0, while "lodash": "~4.17.0" will accept any patch version 4.17.x.

Q5. What are peerDependencies, and when would you use them?
Answer:
peerDependencies are used when a package expects its host application to provide a dependency. This is often used for plugins or libraries that extend functionality and rely on the host project to have a specific version of a dependency installed.

Example: If you create a plugin for React, you'd declare react as a peerDependency, so the user must install a compatible version of React.

4. Creating and Publishing a Package

Q1. Explain the steps required to create a simple NPM package from scratch.
Answer:

- Create a new directory for the project

```
mkdir my-package

cd my-package
```

- Initialize the project with npm init to create a package.json file

```
npm init
```

- Write the package code in a file (e.g., index.js).
- Test the package locally (using npm link).
- Publish the package with npm publish once it's ready.

Q2. What is the purpose of the npm login command, and when do you use it?
Answer:
npm login allows you to log into your NPM account via the command line. It is required before publishing a package to the NPM registry. You need to provide your username, password, and email to authenticate.

Q3. How do you publish a package to the NPM registry?
Answer:
After ensuring that your package is ready for publishing and you've logged into your NPM account, you can publish the package using the following command:

```
npm publish
```

If the package name is unique and valid, it will be published to the NPM registry and available for others to install.

Q4. How can you increment the version of your package before publishing an update? Explain the process using semantic versioning.
Answer:
You can update the version in package.json using the npm version command:

- **Patch version**: For bug fixes

```
npm version patch
```

Minor version: For backward-compatible new features

```
npm version major
```

After updating the version, you can publish the updated package using:

```
npm publish
```

Q5. What is the .npmignore file, and how does it differ from .gitignore?

Answer:
The .npmignore file is used to specify files or directories that should be excluded when publishing a package to the NPM registry. It works similarly to .gitignore, but it is specifically for NPM. If both .gitignore and .npmignore are present, .npmignore takes precedence when publishing a package.