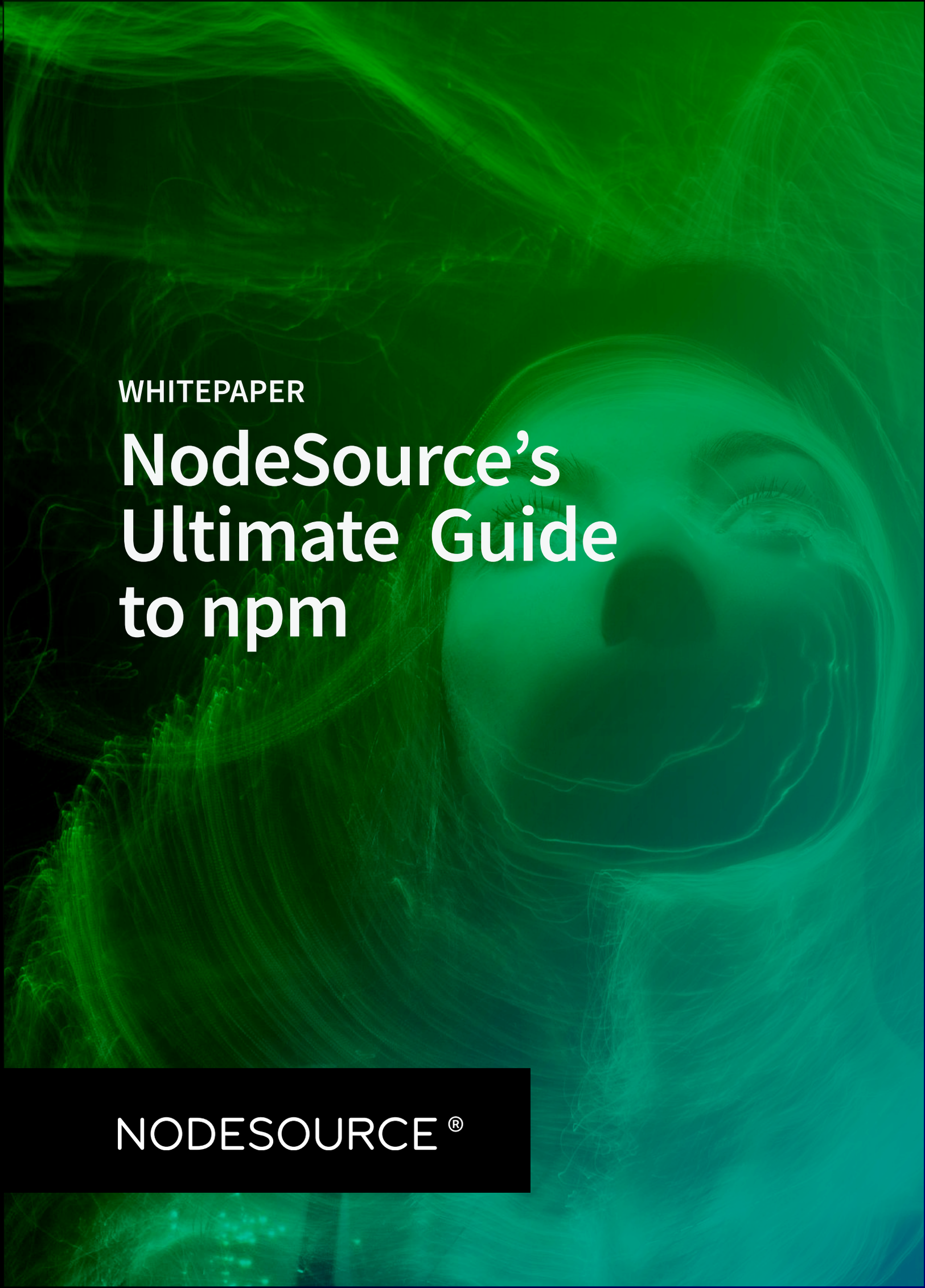

WHITEPAPER

# NodeSource's Ultimate Guide to npm

NODESOURCE®

**About Nodesource**

We empower organizations of all sizes to successfully adopt and integrate Node.js by providing products and services that enable teams to build, manage, and analyze mission-critical applications. To learn more about NodeSource, visit https://www.nodesource.com

**About N|Solid**

N|Solid is an augmented version of the Node.js runtime. Solve immediate issues and identify problems you don't know exist, today and tomorrow. Know your Node with N|Solid SaaS or On-Prem. Get all the metrics, security and insights regardless of your subscription plan.

# NodeSource's Ultimate Guide to npm

Knowing your tools is essential; if you can't use the tools at your disposal, you're going to end up spending more time struggling and less time creating, building, and deploying. The Node.js ecosystem has done a fantastic job in providing tools that help speed development workflows and streamline the process of writing and shipping code.

One of the most essential tools in the Node.js developer ecosystem is the npm CLI — it's one tool that enabled Node.js to become what it is today. It still comes bundled with the core Node.js project and is an instrumental part of developer workflows.

Knowing this, understanding the npm CLI is critically important to any team working with Node.js today — and is why we've written this NodeSource's Ultimate Guide to npm.

This series will cover what you need to know to use the npm CLI as a Node.js developer. We'll split it into three significant chapters (or sections) for better consumption, namely:

- **Chapter 1:** The Basics: Getting started with npm.
- **Chapter 2:** The Basics of Package.json
- **Chapter 3:** Understanding dependencies inside your Package.json

CHAPTER 1:

# The Basics: Getting started with *npm*

# The Basics: Getting started with npm

## Up and running with the primary tool for the world's largest module ecosystem

Today, `npm` is a cornerstone of modern web development, whether used exclusively with Node.js as a package manager or as a build tool for the front end.

Understanding npm as a tool —particularly the core concepts— can be difficult for beginners. We've written up this guide for getting a grasp on npm, especially for those who are entirely new to Node.js, npm, and the surrounding ecosystem.

## The Essential npm Commands

When using npm, you're most likely using the command-line tool for most of your interactions. Here's a detailed rundown of the commands you'll encounter and need to use most frequently.

### USING `npm init` TO INITIALIZE A PROJECT

The `npm init` command is a step-by-step tool to build out the scaffolding for your project. It will prompt for input on a few aspects in the following order:

- **The project's name:** Defaults to the containing directory name.
- **The project's initial version:** 1.0.0 by default.
- **The project's description:**
- **The project's entry point:** Meaning the main file is to be executed when run.
- **The project's test command:** To trigger testing with something like Standard.
- **The project's git repository:** Where the source code can be found.
- **The project's keywords:** Tags related to the project.
- **The project's license:** This defaults to ISC. Most open-source Node.js projects are MIT.

If you're content with the suggestion that the `npm init` command provides next to the prompt; you can simply hit <Return> or <Enter> keys to accept it and move on to the next prompt.

Once you run through the `npm init` steps above, a `package.json` file will be generated and placed in the current directory. If you run it inside a directory that's not exclusively for your project, don't worry! It won't do anything other than create a `package.json` file.

You can move it to a directory dedicated to your project or create an entirely new one in such directory.

```
$ npm init # This will trigger the initialization
```

### USING `npm init --yes` TO INSTANTLY INITIALIZE A PROJECT

If you want to get on to building your project and don't want to spend the (albeit brief) time answering the prompts that come from `npm init`, you can use the `--yes` flag on the `npm init` command to automatically populate all options with the default values.

> **Note:** You can configure what these default values are with the npm configuration commands, which we'll cover in the section "Automating `npm init` Just a Bit More."

```
$ npm init --yes # This will trigger automatically populated
initialization.
```

### INSTALL MODULES WITH `npm install`

Installing modules from npm is one of the most basic things you should learn to do when getting started with npm. As you dive deeper, you'll begin to learn some variations on installing modules, but here's the very core of what you need to know to install a standalone module into the current directory:

```
$ npm install <module>
```

In the above command, you'd replace `<module>` with the name of the module you want to install. For example, if you're going to install Express (the most used and most well known Node.js web framework), you could run the following command:

```
$ npm install express
```

The above instruction will install the `express` module into `/node_modules` in the current directory and add it as a dependency inside the `package.json` file. Whenever you install a module from npm, it will be installed into the `node_modules` folder.

In addition to triggering an install of a single module, you can install all modules listed as `dependencies` and `devDependencies` in the `package.json` in the current directory. To do so, you'll simply need to run the command itself:

```
$ npm install
```

Once you run this, npm will begin installing all of the current project's dependencies.

As an aside, one thing to note is an alias for `npm install` that you may see in the wild when working with modules from the ecosystem. The alias is `npm i`, where I take the place of `install`.

This seemingly minor alias is a small gotcha for beginners to the Node.js and npm ecosystems. There's no standardized, single way module creators and maintainers will instruct how to install their module.

**Usage:**

```
$ npm install <module> # Where <module> is the name of the
module you want to install

$ npm i <module> # Where <module> is the name of the module
you want to install - using the i alias for installation
```

### INSTALL MODULES AND SAVE THEM TO YOUR `package.json` AS A DEPENDENCY

As with `npm init`, the `npm install` command has a flag or two that you'll find helpful in your workflow — it'll save you time and effort concerning your project's `package.json` file.

Before npm 5, when you ran `npm install` to install a module, it was only added to the `node_modules` directory. Thus, if you want to add it to the project's dependencies in the `package.json`, you must add the optional flag `--save` (or `-S`) to the command. Nowadays, since this is the default behavior, no flag is needed (although it's kept for compatibility purposes); however, if for some reason you want to go back to the old usage (i.e., install only to the `node_modules` folder but not add it to the `package.json` `dependencies` section) the `--no save` flag is what you're looking for.

**Usage:**

```
$ npm install <module> --save # Where <module> is the name of
the  module you want to install - Kept for compatibility
$ npm install <module> --no-save # Where <module> is the name
of the  module you want to install - To avoid adding it as a
dependency
```

### INSTALL MODULES AND SAVE THEM TO YOUR `package.json` AS A DEVELOPER DEPENDENCY

There's a flag that is nearly an exact duplicate, in terms of functionality, of the old `--save` flag when installing a module: `--save-dev` (or `-D`). There are a few key differences between the two: instead of installing and adding the module to `package.json` as an entry in `dependencies`, it will save it as an entry in the `devDependencies`.

The semantic difference here is that `dependencies` are used in production — whatever your project would entail. On the other hand, `devDependencies` are a collection of the dependencies used during the development of your application: the modules that you need to use to build it but don't need when it's running. This could include testing tools, a local server to speed up your development, etc.

**Usage:**

```
$ npm install <module> --save-dev # Where <module> is the
name of the  module you want to install
```

### INSTALL MODULES GLOBALLY ON YOUR SYSTEM

The final and most common flags for `npm install` that you should know are those used to install a module globally on your system.

Global modules can be beneficial. There are numerous tools, utilities, and more for development and general usage that you can install and set available for all the projects inside your environment.

To install a module from npm in such a way, you'll simply need to use the `--global` flag when running the install command to have it installed globally rather than locally (restricted to the current directory).

> **Note:** One caveat with global modules is that npm will install them to a system directory, not a local one. With this as the default, you'll typically need to authenticate as a privileged user on your system to install global modules. As a best practice, you should change the default installation location from a system directory to a user directory.

Usage:

```
$ npm install <module> --global    # Where <module> is the
name of  the module you want to install globally

$ npm install <module> -g  # Where <module> is the name of the
module you want to install globally, using the -g alias
```

CHAPTER 2:

# The basics of *Package.json*

# The Basics of Package.json

This chapter will give you a kickstart introduction to effectively using `package.json` with `Node.js` and `npm.`

The `package.json` file is core to the Node.js ecosystem and is a fundamental part of understanding and working with Node.js, npm, and even modern JavaScript. This file is used as a manifest, storing information about applications, modules, packages, and more.

Because understanding it is essential to working with Node.js, it's a good idea to grasp the commonly found and most crucial properties of a  package.json file to use it effectively.

## IDENTIFYING METADATA INSIDE `package.json`

### THE `name` PROPERTY

The `name` property in a `package.json` file is one of the fundamental components of the `package.json` structure. At its core, the `name` is a string that is exactly what you would expect: the name of the module that the `package.json` is describing.
Inside your package.json, the name property as a string would look something like this:

```
"name": "metaverse"
```

There are only a few material restrictions on the `name` property:
• Maximum length of 214 URL-friendly characters
• No uppercase letters
• No leading periods (.) or underscores (_) (Except for scoped packages)

However, some software ecosystems have developed standard naming conventions that enable discoverability.  A few examples of this kind of namespacing are babel-plugin- for Babel and the webpack-loader tooling.

### THE `version` PROPERTY

The `version` property is a crucial part of a `package.json`, as it denotes the current version of the module that the `package.json` file is describing.

While the version property isn't required to follow semver (semantic versioning) standards, which is the model used by the vast majority of modules and projects in the Node.js ecosystem, it's what you'll typically find in the `version` property of a `package.json` file.

Inside your `package.json`, the `version` property as a string using semver could look like this:

```
"version": "5.12.4"
```

### THE `license` PROPERTY

The license property of a `package.json` file is used to note the module that the `package.json` file describes. While there are some complex ways to use the licensed property of a `package.json` file (to do things like dual-licensing or defining your own license), the most typical usage is to use an SPDX License identifier. Some examples that you may recognize are MIT, ISC, and GPL-3.0.

Inside your `package.json`, the `licensed` property with an MIT license looks like this:

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

### THE `description` PROPERTY

The `description` property of a `package.json` file is a string that contains a human-readable description of the module. Basically, it's the module developer's chance to quickly let users know what exactly a module does. The `description` property is frequently indexed by search tools like npm search and the npm CLI search tool to help find relevant packages based on a search query.

Inside your `package.json`, the `description` property would look like this:

```
  "description": "The Metaverse virtual reality. The final outcome of all virtual worlds, augmented reality, and the Internet."
```

### THE `Keywords` PROPERTY

The `keywords` property inside a `package.json` file is, as you may have guessed, a collection of keywords that describe a module. Keywords can help identify a package, related modules and software, and concepts.

The `keywords` property is always an array, with one or more strings as the array's values; each one of these strings will, in turn, be one of the project's keywords.

Inside your `package.json`, the `keywords` array would look something like this:

```
  "keywords": [
  "metaverse",
    "virtual reality",
    "augmented reality",
    "snow crash"
  ]
```

## FUNCTIONAL METADATA INSIDE `package.json`

### THE `main` PROPERTY

The `main` property of a `package.json` is a direction to the entry point to the module that the `package.json` is describing. In a Node.js application, when the module is called via a required statement, the module's exports from the file named in the `main` property will be returned to the Node.js application.

Inside your `package.json`, the `main` property, with an entry point of `app.js`, would look like this:

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

### THE `repository` PROPERTY

The `repository` property of a `package.json` is an array that defines where the source code for the module lives. Typically, this would be a public GitHub repo for open source projects, with the repository array noting that the type of version control is git and the URL of the repo itself. One thing to note about this is that it's not just a URL where the repo can be accessed from, but the full URL the version control can be accessed from.

Inside your `package.json`, the `repository` property would look like this:

```
  "repository": {
    "type": "git",
    "url": "https://github.com/bnb/metaverse.git"
  }
```

## THE `scripts` PROPERTY

The `scripts` property of a `package.json` file is simple conceptually but complex functionally, to the point that it's used as a build tool by many.

At its simplest, the `scripts` property contains a set of entries; the key for each entry is a `script` name, and the corresponding value is a user-defined command to be executed. Scripts are frequently used to test, build, and streamline the needed commands to work with a module.

Inside your `package.json`, the `scripts` property with a build command to execute `tsc` (presumably to transpile your application using TypeScript) and a test command using Standard would look like this:

```
 "scripts": {
  "build": "tsc",
  "test": "standard"
}
```

To run scripts in the `scripts` property of a `package.json`, you'll need to use the default `npm run` command. So, to run the above example's build, you'd need to run this:

**Usage:**

```
$ npm run build
```

That said, to run the test suite with Standard, you'd need to run this:

**Usage:**

```
$ npm test
```

Notice that npm does not require the `run` keyword as part of the given `script` command for some tasks like the `test`, `start`, and `stop` by default.

### THE `dependecies` PROPERTY

The `dependencies` property of a module's `package.json` is defined by the other modules that this module uses. Each entry in the `dependencies` property includes the name and version of other packages required to run this package.

> **Note:** You'll frequently find carets (^) and tildes (~) included with package versions. These are the notations for version range — taking a deep dive into these is outside the scope of this guide, but you can learn more in our primer on semver. In addition, you may specify URLs or local paths in place of a version range.

Inside your `package.json`, the `dependencies` property of your module may look something like this:

```
"dependencies": {
 "async": "^0.2.10",
   "npm2es": "~0.4.2",
   "optimist": "~0.6.0",
   "request": "~2.30.0",
   "skateboard": "^1.5.1",
   "split": "^0.3.0",
   "weld": "^0.2.2"
}
```

### THE `devDependencies` PROPERTY

The `devDependencies` property of `package.json` is almost identical to the `dependencies` property in terms of structure. The main difference: while the `dependencies` property is used to define the dependencies that a module needs to run in production, `devDependencies` property is commonly used to define the dependencies the module needs to run in development.

Inside your `package.json`, the `devDependencies` property would look something like this

```
"devDependencies": {
   "escape-html": "^1.0.3",
   "lucene-query-parser": "^1.0.1"
}
```

CHAPTER 3:

# Understanding dependencies inside your *Package.json*

## UNDERSTANDING THE DIFFERENT TYPES OF DEPENDENCIES AND OTHER HOST SPECS INSIDE PACKAGE.JSON

Having dependencies in your project's `package.json` allows the project to install the versions of the modules it depends on. By running an install command inside a project, you can install all of the dependencies listed in the project's `package.json`, meaning they don't have to be (and rarely should be) bundled with the project itself.

The separation of dependencies needed for production and dependencies needed for development is one of the majorly important aspects of `package.json`. You're likely not to need a tool to watch your CSS files for changes in production and refresh the app when they change. But in both production and development, you'll want to have the modules that enable what you're trying to accomplish with your project – things like your web framework,  API tools, and code utilities.

Furthermore, there are other lesser-known types of dependencies and specifications that help you to customize your package for specific host environments, namely:

- **peerDependencies:** They are use to express compatibility with a host tool or library while not requiring them inside the project. As of npm v7, they are installed by default.

- **peerDependenciesMeta:** Allows peer dependencies to be marked as optional so that integration and interaction with other packages don't warn you about requiring all of them to be installed.

- **optionalDependencies:** As its name suggests, it is used to avoid build failures when the dependency cannot be found or fails to install. However, it would be best to handle the lack of dependency inside your code.

- **bundledDependencies:** Useful for cases when some special packages need to be preserved locally by including them inside the tarball file generated after publishing your project.

- **Engines:** It can be used for specifying the node and/or npm versions your stuff works on.

- **OS:** Your module will run on an array of allowed and/or blocked (if prepended with a bang "!" sign) operating systems.

- **CPU:** Similar to the previous one. An array of allowed or blocked CPU architectures the code was designed for.

What would a project's `package.json` look like with `dependencies` and `devDependencies`?

Let's expand on the previous example of a `package.json` to include some.

```
{
 "name": "metaverse",
 "version": "0.92.12",
 "description": "The Metaverse virtual reality. The final
 outcome of all virtual worlds, augmented reality, and the
 Internet.",
 "main": "index.js",
 "license": "MIT",
 "devDependencies": {
  "mocha": "~3.1",
  "native-hello-world": "^1.0.0",
  "should": "~3.3",
  "sinon": "~1.9"
 },
 "dependencies": {
  "fill-keys": "^1.0.2",
  "module-not-found-error": "^1.0.0",
  "resolve": "~1.1.7"
 }
}
```

One key difference between the dependencies and the other common parts of `package.json` is that they're both objects with multiple key/value pairs. Every key in `dependencies`, `devDependencies`, and peerDependencies is is a package's name, and every value is the version range that's acceptable to install (according to semver).

*Semver is a specification outlining a method of encoding the nature of change between releases of a "public interface". You can read more about "Semver" here You also may find useful "ABC's of JavaScript and Node.js".

# What next?

The ecosystem surrounding npm is continuously evolving. There are always going to be hard problems to solve – things like security, licensing, management, module optimization, and more.

Tools like NCM - NodeSource Certified Modules are aiming to directly help alleviate these problems in a secure and reliable way, directly in N|Solid. That said, there will never be one right answer to any individual's or organization's direct needs, but rather many answers to solve the problems – that we face both now and in the future – on a case-by- case basis.

We have amazing features to help you on your journey as a developer, install N|Solid and start enjoying advanced knowledge and control over your application. process in Node.js #KnowyourNode

If you've got any specific questions you'd like to ask, feel free to reach out to us at @NodeSource on Twitter – we'd love to hear your thoughts and feedback on NodeSource's Ultimate Guide to npm.