The Basics of Package.json

In this chapter, we'll 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.

This is a series, based on one of the most featured whitepapers we have done by developers in the Node.js ecosystem. If you are interested in the complete guide, you can get it through this link.

The 2022 guide will include this, which we will be releasing by units of knowledge every Thursday in the following weeks. Today you are in part 1 of the guide:

1. The Essential npm Commands

    Using `npm init` to initialize a project

    Using `npm init --yes` to instantly initialize a project

    Install modules with `npm install`

    Install modules and save them to your `package.json` as a dependency

    Install modules and save them to your `package.json` as a developer dependency

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 namespacing are the 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 `license` 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. It's the module developer's chance to let users know what precisely a module does quickly. Search tools frequently index the `description` property 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 `require` 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, you'd need to execute 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 `dependencies` 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"
},
```

Remember that you can now monitor your applications and take your Node.js journey to a professional level with N|Solid.

> To get the best out of Node.js and low-cost observability, start a free trial of N|Solid.
>
> If you have any questions, please feel free to contact us at info@nodesource.com or through this form.
>
> And if you want to find out about our latest content and product releases, these are the channels to keep up to date with NodeSource:
>
> ↳ ↳ Nodesource's Twitter