What's what? - Package.json cheatsheet!

Recently I started a **series about Node.js** and its built-in API. There, as I said, we'll be exploring every single Node.js API in-depth (or at least that's the plan). But, talking about Node.js without mentioning the famous **package.json** 🗄 would be a sign of big ignorance of the importance of this file.

I think we all know what package.json is. Probably nearly every new JS-related project is started by setting up this particular file. And still, developers' knowledge about such important, seemingly simple JSON file is often limited to just a few fields. Of course, it's nothing wrong - you don't have to know the whole specification, but it's good to at least have some clue about what's what. This is especially important when e.g. creating **your first public NPM package**. 🚀 And, because of inconsistent, incomplete or hard-to-read documentations about it around the web, many newcomers often publish their first packages... with **errors**. 🚫

With all above said, in this article, we'll finally fix this flaw. We're going to explore package.json as deeply as possible! So, consider bookmarking 🔖 this page for later use as the full-fledged **package.json cheatsheet**! 📋 Let's get started!

Basics

name

I think the `name` property needs no explanation. Your package won't function correctly without this **compulsory field**. There are only some rules that you must obey when naming your new package:

- Your name should be **unique** (when publishing to NPM) ☝;
- If you have proper NPM organization register, you can use your package with so-called **scope** e.g. @organization/package;
- The character length of the name should be no bigger than **214**

characters including scoping;
- **No capped letters** and **underscore** (_) or **dot** (.) **on the beginning**;
- You can use only **URL-safe chars** - your name will most likely be typed by others in terminals and used as URL at NPM page;

version

The second required and no-brainer field. `version`, together with name form a **unique ID** for **every release** of your package. Because, guess what! - you should/must change the version with every new release of your package! Also, your version string should be parsable by **node-semver**, meaning that it should have a certain, predictable **structure**. 🏗 You should probably have seen all possible version codes by now, browsing the NPM. Usually, it comes in a form of 3 numbers separated by dots (.), e.g. **0.0.0**. After that, an optional tag (e.g. **next**, **beta**, **alpha**) preceded by a dash and optionally followed by yet another dot and number, e.g. **0.0.0-next.0**. Of course, you shouldn't just drop any new version you think is suitable (especially when your package is used by other people). That's why **versioning guidelines** like **Semantic Versioning** have been created.

Information

description

It's good to provide your users with at least some **info about your package**. ℹ A short `description` string in your main JSON file can serve that purpose great! It will then be displayed in **NPM listings** and e.g. in **VS Code pop-ups**. Of course, it won't be enough and so **README.md** file at your project's root can be a good idea. Mentioned file can later be used for your package's **NPM page**.

keywords

`keywords` give you an option to improve the *"SEO"* of your package. 😄 This array of strings will make your package **rank higher** when your possible users will search through NPM listings by keywords that much the ones you provided.

license

`license` is a simple and short, but very important field. It's this string that lets your users know at what terms you share your code. The string must be one of **SPDX identifiers** (short forms), like **MIT**, **GPL-3.0** and etc. If you yet don't know what kind of license suits you best, consider checking out **this page**, where you can quickly understand and pick the best one (SPDX identifiers included!). It's a good practice (if not a requirement) to later put the actual text of your license-of-choice at your project's root in **LICENSE.md** file. 📝

homepage

If you have a nice **landing page** for your package, you can freely put its **URL** here. It will later be displayed at NPM page and in various other places.

repository

If you're publishing your package publicly on NPM, high chances are it's some kind of open source software. Thus, information about the location of the actual **source-code** 📦 may come in handy. The `repository` field can do just that! There are two ways in which you can set it up - an **object** with 2 or 3 properties, i.e. `type` (your repository type, like **"git"** or **"svn"** in a string), `url` (the URL of your repository) and optional `directory` within your repo (if it's e.g. **monorepo**). The other possible form is a **string** (that can also be used to install packages from repos directly) in a format of "**provider**:**user/repo**" (where the provider can be **"github"**, **"gitlab"** or **"bitbucket"**) or "**gist**:**id**" (for Gists only).

bugs

Yeah, `bugs` happen so often that they deserve their own, separate field. 😄 It usually should point out to the **issues page** of your repository or to any other place where these issues can **be reported**.

author

Proper first creator of the package deserves **proper credits**. 👏 This field can have two possible types of value: an object or a string. The **object** can have 3 properties - `name`, `email` and `url` (for author's website). The different, shorter format is a **string** version, with predefined formatting (when compared to object) - **"*name* <*email*> (*url*)"**. These different brackets are required, but not all of them needs to be provided. You can omit e.g. (url) and provide only name and email. NPM will be happy with any variation.

contributors

`contributors` are just as important as the author himself. 👬 And, as each should be noted, there's a special property for that too! `contributors` is an **array of objects or strings**, where same rules as with singular author field apply. Another interesting feature is that you can provide proper **AUTHORS.md** file at the root of your project where, line by line, contributors of your project will be provided (in string format mentioned earlier). This will later be used as contributors **default value**.

Files

files

There's a high chance that you may want your final package to include only certain files. To do this you have two options. You can provide a `files` property in your package.json, in form of an **array of strings** (with support for **separate files**, **directories**, and **wildcards** 🌟), to **include only certain files** in package send to NPM. Another option is to provide the **.npmignore** file (like popular **.gitignore**) which will later be used to **exclude certain files**. Of course, there are some files that will never respect those rules and will always be included (e.g. **README.md** or **LICENSE.md**) or excluded (e.g. **node_modules** or **.git**).

main

Probably everybody knows the `main` field. It should point to default, the most important file (**entry point**) of the whole package. It will be included in the final bundle no matter `files` configuration.

browser

With `browser` property, we're getting to different variations of main files for your package. It can be used when e.g. you use some kind of module bundler which outputs different formats (like **IIFE** or **UMD**). Browser field should point to file, that may be **used in browsers** 🖥 and be dependent on **global variables** of this environment (e.g. `window`).

unpkg

Maybe it's not 100% official, but **UNPKG** as **NPM-based CDN** have gained so much popularity and user base, that this property might be worth a closer look. `unpkg` should point to a file that will later be exclusively used by UNPKG to provide its CDN support. It's usually the same as the earlier-mentioned `browser`.

module

If you have one, `module` property should point out the file that is an entry point for your modular (not-bundled) code base. It's targeted towards more **modern environments**. 🚃

typings

The `typings` or `types` (shorter alternative) field shows really how popular **TypeScript** together with **great development tooling** it provides has become. This property should point to the entry file of your **TypeScript declaration files** (if you have one). It will later be sent to NPM and available to download and provide good **IDE support** for your users. This is a bit more convenient than uploading your typings separately to something like **DefinitelyTyped**, at least IMHO.

bin

If your package is some kind of executable file, it must include this field. It should point out to the **main file** of your **Node.js executable** or have a form of an **object** with keys corresponding to **different executables** and values to **their files**. Just remember that you should begin your executable files with this magic line ✨ - `#!/usr/bin/env node`.

man

If you have any documentation in the form of **man pages** for your package, feel free to provide it here. It can be a string pointing to a **single file or an array** of such.

directories

`directories` may be one of these mysterious fields that not many know what it exactly does. I'll tell you - it's mostly just **meta-info**. Exactly two fields provide some functionality and nothing else. Of course, all of them should have a form of string pointing to **different directories**.

- `lib` - meta info about where your library is exactly located in your package;
- `bin` - directory where all your **executable files** are located. Can be used instead of providing different files, one by one, with `bin` property. Know that you cannot use these two properties together - **only one of them**;
- `man` - directory where all your **man pages** are located. You can use this instead of providing an array of files through `man` property. It's certainly less tiresome;
- `doc` - meta info about directory where **markdown documentation** for a given package is located;
- `example` - meta info about directory where you have **example code**;
- `test` - meta info about directory where your **test files** are located;

Keep in mind that, as NPM official documentation mentions, this data can be used in the future to provide some additional features e.g. nice documentation or whatever...

Tasks

scripts

I think you know well what `scripts` field does. It's a simple object with keys corresponding to commands and their values to what they should do. You most likely use script names like **"build"** or **"start"**, but did you know that there are some scripts that are executed automatically when predefined event occurs? There are quite a few of them and you can find **the complete list here**. ☜

config

`config` property has a form of a special object, where you can specify some **configuration** that you can later **use in your scripts**. 🕴 💻. For example, a config property named `port` can be later referenced using `npm_package_config_port`, where the preceding part is always required. These options can also be altered using e.g. `npm config set [package]:[prop] [value]`.

Dependencies

dependencies

Everybody knows what **NPM dependencies** are and... memes about how **deep** they can get. 😄 And `dependencies` is a field responsible for all of that. Here all your dependencies that are required to be installed (and are most likely used) by your package **must be listed**. `npm install` or `yarn add` will automatically **manage these for you**. Just remember that you can also list & install dependencies from **URLs**, **Git URLs**, **GitHub URLs**, **linked packages** and **local paths**.

devDependencies

Development dependencies (those installed with `--save-dev` or `--dev`) are meant to contain all dependencies that are required during the **development process** 🏗 of the given package. It can be e.g. **testing**

framework, **module bundler** or **transpiler**. All of these won't be installed for standard use of the package.

peerDependencies

Peer dependencies (this time **not configured automatically**) allow you to specify compatibility of your package with some other ones. This should have a form of an **object** with **compatible packages names** as keys and **their respective versions** (following node-semver, e.g. 0.x.x) as values. Since NPM v3 these dependencies aren't installed by default.

optionalDependencies

If any of your packages are **optional**, i.e. don't really have to be installed but can be helpful, you can add them here. These will be installed only if possible (e.g. if a platform is compatible). The often seen example of this is an NPM module called **fsevents**, available only on Mac OS.

bundledDependencies

Bundled dependencies should have a form of an array with names of dependencies that would be bundled with your final package. This can be useful when preserving your project with **tarball files**, which, bundled using `npm pack`, will include files here specified. 🎁

Platform

engines

`engines` is a very useful property (just like any other), allowing you to specify an object of **libraries** and **runtimes** (like **Node.js**, **NPM** or **React Native**). Object keys correspond to specific names (just like the ones on NPM - lower case, dashed) and values in the form of compatible **versions strings** (node-semver-compatible). This is especially useful when your package depends on **modern features** (available only in latest Node.js releases) or on other, usually globally-installed libraries and runtimes.

os

If your package can run only on **specific operating systems**, you can specify this fact with `os` property, in the form of an array of **OS code names**. You can specify only the ones that are allowed (e.g. `["linux"]`) or those that aren't with preceding **exclamation mark** (**!**) (e.g. `["!win32"]`).

cpu

Just like with `os`, by using `cpu` property, you can specify on which kinds of CPUs your code can run. Same rules (include & exclude) apply (e.g. `["x64"]` or `["!arm"]`).

Publishing

private

If you want your package to remain **private** (or an **entry to a multirepo**) you can set the `private` property to **true**. It will ensure that your package **won't be published**, even if you'd make a mistake and try to do it. It's **false by default**. 🔒

publishConfig

If you want (or have a real reason behind this), you can override any of **numerous NPM config values** before publishing your package, with this particular property. This should have a form of an object. Most likely you'll want to change only values like `"tag"`, `"registry"` or `"access"`.

Custom fields

Beyond all the fields above, package.json has become a place for various different tools to place their own, **custom fields** and **configs** there. Just like mentioned **UNPKG**, **Babel**, **Prettier** 💉 and tons of **different tools** allow you to do so. It's almost always written in their **documentation**. But, with such a big number of tools, package.json has become a bit... **crowded**. 👎 Thus, IMHO, it's better to use **separate config files** whenever possible and leave package.json only for all, still numerous, stuff listed above.

That's all!

I really, really hope this article helped some of you publishing **your first public packages** or allowed you to **learn something new**. 🎇 This is the knowledge I wish I had a few weeks back. 😄 With this post, even I as a writer have improved my understanding of all that package.json stuff.