npm-package.json

Specifics of npm's package.json handling

DESCRIPTION

This document is all you need to know about what's required in your package.json file. It must be actual JSON, not just a JavaScript object literal.

A lot of the behavior described in this document is affected by the config settings described in `npm-config`.

name

If you plan to publish your package, the *most* important things in your package.json are the name and version fields as they will be required. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version. If you don't plan to publish your package, the name and version fields are optional.

The name is what your thing is called.

Some rules:

- The name must be less than or equal to 214 characters. This includes the scope for scoped packages.
- The name can't start with a dot or an underscore.
- New packages must not have uppercase letters in the name.
- The name ends up being part of a URL, an argument on the command line, and a folder name. Therefore, the name can't contain any non-URL-safe characters.

Some tips:

- Don't use the same name as a core Node module.
- Don't put "js" or "node" in the name. It's assumed that it's js, since you're writing a package.json file, and you can specify the engine using the "engines" field. (See below.)
- The name will probably be passed as an argument to require(), so it should be something short, but also reasonably descriptive.

- You may want to check the npm registry to see if there's something by that name already, before you get too attached to it. https://www.npmjs.com/

A name can be optionally prefixed by a scope, e.g. `@myorg/mypackage` . See `npm-scope` for more detail.

version

If you plan to publish your package, the *most* important things in your package.json are the name and version fields as they will be required. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version. If you don't plan to publish your package, the name and version fields are optional.

Version must be parseable by node-semver, which is bundled with npm as a dependency. (`npm install semver` to use it yourself.)

More on version numbers and ranges at semver.

description

Put a description in it. It's a string. This helps people discover your package, as it's listed in `npm search` .

keywords

Put keywords in it. It's an array of strings. This helps people discover your package as it's listed in `npm search` .

homepage

The url to the project homepage.

Example:

```
"homepage": "https://github.com/owner/project#readme"
```

bugs

The url to your project's issue tracker and / or the email address to which issues should be reported. These are helpful for people who encounter issues with your package.

It should look like this:

```
{ "url" : "https://github.com/owner/project/issues"
, "email" : "project@hostname.com"
}
```

You can specify either one or both values. If you want to provide only a url, you can specify the value for “bugs” as a simple string instead of an object.

If a url is provided, it will be used by the `npm bugs` command.

license

You should specify a license for your package so that people know how they are permitted to use it, and any restrictions you’re placing on it.

If you’re using a common license such as BSD-2-Clause or MIT, add a current SPDX license identifier for the license you’re using, like this:

```
{ "license" : "BSD-3-Clause" }
```

You can check the full list of SPDX license IDs. Ideally you should pick one that is OSI approved.

If your package is licensed under multiple common licenses, use an SPDX license expression syntax version 2.0 string, like this:

```
{ "license" : "(ISC OR GPL-3.0)" }
```

If you are using a license that hasn’t been assigned an SPDX identifier, or if you are using a custom license, use a string value like this one:

```
{ "license" : "SEE LICENSE IN <filename>" }
```

Then include a file named `<filename>` at the top level of the package.

Some old packages used license objects or a “licenses” property containing an array of license objects:

```
// Not valid metadata
{ "license" :
  { "type" : "ISC"
  , "url" : "https://opensource.org/licenses/ISC"
  }
```

```
}

// Not valid metadata
{ "licenses" :
  [
    { "type": "MIT"
    , "url": "https://www.opensource.org/licenses/mit-license.php"
    }
  , { "type": "Apache-2.0"
    , "url": "https://opensource.org/licenses/apache2.0.php"
    }
  ]
}
```

Those styles are now deprecated. Instead, use SPDX expressions, like this:

```
{ "license": "ISC" }

{ "license": "(MIT OR Apache-2.0)" }
```

Finally, if you do not wish to grant others the right to use a private or unpublished package under any terms:

```
{ "license": "UNLICENSED" }
```

Consider also setting **`"private": true`** to prevent accidental publication.

people fields: author, contributors

The “author” is one person. “contributors” is an array of people. A “person” is an object with a “name” field and optionally “url” and “email”, like this:

```
{ "name" : "Barney Rubble"
, "email" : "b@rubble.com"
, "url" : "http://barnyrubble.tumblr.com/"
}
```

Or you can shorten that all into a single string, and npm will parse it for you:

```
"Barney Rubble <b@rubble.com> (http://barnyrubble.tumblr.com/)"
```

Both email and url are optional either way.

npm also sets a top-level “maintainers” field with your npm user info.

files

The optional `files` field is an array of file patterns that describes the entries to be included when your package is installed as a dependency. File patterns follow a similar syntax to `.gitignore`, but reversed: including a file, directory, or glob pattern (`*`, `**/*`, and such) will make it so that file is included in the tarball when it’s packed. Omitting the field will make it default to `["*"]`, which means it will include all files.

Some special files and directories are also included or excluded regardless of whether they exist in the `files` array (see below).

You can also provide a `.npmignore` file in the root of your package or in subdirectories, which will keep files from being included. At the root of your package it will not override the “files” field, but in subdirectories it will. The `.npmignore` file works just like a `.gitignore`. If there is a `.gitignore` file, and `.npmignore` is missing, `.gitignore`’s contents will be used instead.

Files included with the “package.json#files” field *cannot* be excluded through `.npmignore` or `.gitignore`.

Certain files are always included, regardless of settings:

- `package.json`
- `README`
- `CHANGES` / `CHANGELOG` / `HISTORY`
- `LICENSE` / `LICENCE`
- `NOTICE`
- The file in the “main” field

`README`, `CHANGES`, `LICENSE` & `NOTICE` can have any case and extension.

Conversely, some files are always ignored:

- `.git`
- `CVS`
- `.svn`
- `.hg`
- `.lock-wscript`
- `.wafpickle-N`
- `.*.swp`

- `.DS_Store`
- `._*`
- `npm-debug.log`
- `.npmrc`
- `node_modules`
- `config.gypi`
- `*.orig`
- `package-lock.json` (use shrinkwrap instead)

main

The main field is a module ID that is the primary entry point to your program. That is, if your package is named `foo`, and a user installs it, and then does `require("foo")`, then your main module's exports object will be returned.

This should be a module ID relative to the root of your package folder.

For most modules, it makes the most sense to have a main script and often not much else.

browser

If your module is meant to be used client-side the browser field should be used instead of the main field. This is helpful to hint users that it might rely on primitives that aren't available in Node.js modules. (e.g. `window`)

bin

A lot of packages have one or more executable files that they'd like to install into the PATH. npm makes this pretty easy (in fact, it uses this feature to install the "npm" executable.)

To use this, supply a `bin` field in your package.json which is a map of command name to local file name. On install, npm will symlink that file into `prefix/bin` for global installs, or `./node_modules/.bin/` for local installs.

For example, myapp could have this:

```
{ "bin" : { "myapp" : "./cli.js" } }
```

So, when you install myapp, it'll create a symlink from the `cli.js` script to `/usr/local/bin/myapp`.

If you have a single executable, and its name should be the name of the package, then you can just supply it as a string. For example:

```
{ "name": "my-program"
, "version": "1.2.5"
, "bin": "./path/to/program" }
```

would be the same as this:

```
{ "name": "my-program"
, "version": "1.2.5"
, "bin" : { "my-program" : "./path/to/program" } }
```

Please make sure that your file(s) referenced in `bin` starts with `#!/usr/bin/env node`, otherwise the scripts are started without the node executable!

man

Specify either a single file or an array of filenames to put in place for the `man` program to find.

If only a single file is provided, then it's installed such that it is the result from `man <pkgname>`, regardless of its actual filename. For example:

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
, "man" : "./man/doc.1"
}
```

would link the `./man/doc.1` file in such that it is the target for `man foo`

If the filename doesn't start with the package name, then it's prefixed. So, this:

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
, "man" : [ "./man/foo.1", "./man/bar.1" ]
}
```

will create files to do `man foo` and `man foo-bar`.

Man files must end with a number, and optionally a `.gz` suffix if they are compressed. The number dictates which man section the file is installed into.

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
, "man" : [ "./man/foo.1", "./man/foo.2" ]
}
```

will create entries for `man foo` and `man 2 foo`

directories

The CommonJS Packages spec details a few ways that you can indicate the structure of your package using a `directories` object. If you look at npm's package.json, you'll see that it has directories for doc, lib, and man.

In the future, this information may be used in other creative ways.

directories.lib

Tell people where the bulk of your library is. Nothing special is done with the lib folder in any way, but it's useful meta info.

directories.bin

If you specify a `bin` directory in `directories.bin` , all the files in that folder will be added.

Because of the way the `bin` directive works, specifying both a `bin` path and setting `directories.bin` is an error. If you want to specify individual files, use `bin` , and for all the files in an existing `bin` directory, use `directories.bin` .

directories.man

A folder that is full of man pages. Sugar to generate a "man" array by walking the folder.

directories.doc

Put markdown files in here. Eventually, these will be displayed nicely, maybe, someday.

directories.example

Put example scripts in here. Someday, it might be exposed in some clever way.

directories.test

Put your tests in here. It is currently not exposed, but it might be in the future.

repository

Specify the place where your code lives. This is helpful for people who want to contribute. If the git repo is on GitHub, then the `npm docs` command will be able to find you.

Do it like this:

```
"repository": {
  "type" : "git",
  "url" : "https://github.com/npm/cli.git"
}

"repository": {
  "type" : "svn",
  "url" : "https://v8.googlecode.com/svn/trunk/"
}
```

The URL should be a publicly available (perhaps read-only) url that can be handed directly to a VCS program without any modification. It should not be a url to an html project page that you put in your browser. It's for computers.

For GitHub, GitHub gist, Bitbucket, or GitLab repositories you can use the same shortcut syntax you use for `npm install` :

```
"repository": "npm/npm"

"repository": "github:user/repo"

"repository": "gist:11081aaa281"

"repository": "bitbucket:user/repo"

"repository": "gitlab:user/repo"
```

If the `package.json` for your package is not in the root directory (for example if it is part of a monorepo), you can specify the directory in which it lives:

```
"repository": {
  "type" : "git",
  "url" : "https://github.com/facebook/react.git",
  "directory": "packages/react-dom"
}
```

scripts

The “scripts” property is a dictionary containing script commands that are run at various times in the lifecycle of your package. The key is the lifecycle event, and the value is the command to run at that point.

See `npm-scripts` to find out more about writing package scripts.

config

A “config” object can be used to set configuration parameters used in package scripts that persist across upgrades. For instance, if a package had the following:

```
{ "name" : "foo"
, "config" : { "port" : "8080" } }
```

and then had a “start” command that then referenced the `npm_package_config_port` environment variable, then the user could override that by doing `npm config set foo:port 8001` .

See `npm-config` and `npm-scripts` for more on package configs.

dependencies

Dependencies are specified in a simple object that maps a package name to a version range. The version range is a string which has one or more space-separated descriptors. Dependencies can also be identified with a tarball or git URL.

Please do not put test harnesses or transpilers in your `dependencies` object. See `devDependencies` , below.

See semver for more details about specifying version ranges.

- `version` Must match `version` exactly
- `>version` Must be greater than `version`

- `>=version` etc
- `<version`
- `<=version`
- `~version` “Approximately equivalent to version” See semver
- `^version` “Compatible with version” See semver
- `1.2.x` 1.2.0, 1.2.1, etc., but not 1.3.0
- `http://...` See ‘URLs as Dependencies’ below
- `*` Matches any version
- `""` (just an empty string) Same as `*`
- `version1 - version2` Same as `>=version1 <=version2`.
- `range1 || range2` Passes if either range1 or range2 are satisfied.
- `git...` See ‘Git URLs as Dependencies’ below
- `user/repo` See ‘GitHub URLs’ below
- `tag` A specific version tagged and published as `tag` See `npm-dist-tag`
- `path/path/path` See Local Paths below

For example, these are all valid:

```
{ "dependencies" :
  { "foo" : "1.0.0 - 2.9999.9999"
  , "bar" : ">=1.0.2 <2.1.2"
  , "baz" : ">1.0.2 <=2.3.4"
  , "boo" : "2.0.1"
  , "qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
  , "asd" : "http://asdf.com/asdf.tar.gz"
  , "til" : "~1.2"
  , "elf" : "~1.2.3"
  , "two" : "2.x"
  , "thr" : "3.3.x"
  , "lat" : "latest"
  , "dyl" : "file:../dyl"
  }
}
```

URLs as Dependencies

You may specify a tarball URL in place of a version range.

This tarball will be downloaded and installed locally to your package at install time.

Git URLs as Dependencies

Git urls are of the form:

```
<protocol>://[<user>[:<password>]@]<hostname>[:<port>][:][/]<path>[#<commit-
```

`<protocol>` is one of `git` , `git+ssh` , `git+http` , `git+https` , or `git+file` .

If `#<commit-ish>` is provided, it will be used to clone exactly that commit. If the commit-ish has the format `#semver:<semver>` , `<semver>` can be any valid semver range or exact version, and npm will look for any tags or refs matching that range in the remote repository, much as it would for a registry dependency. If neither `#<commit-ish>` or `#semver:<semver>` is specified, then `master` is used.

Examples:

```
git+ssh://git@github.com:npm/cli.git#v1.0.27
git+ssh://git@github.com:npm/cli#semver:^5.0
git+https://isaacs@github.com/npm/cli.git
git://github.com/npm/cli.git#v1.0.27
```

GitHub URLs

As of version 1.1.65, you can refer to GitHub urls as just “foo”: “user/foo-project”. Just as with git URLs, a `commit-ish` suffix can be included. For example:

```
{
  "name": "foo",
  "version": "0.0.0",
  "dependencies": {
    "express": "expressjs/express",
    "mocha": "mochajs/mocha#4727d357ea",
    "module": "user/repo#feature\/branch"
  }
}
```

Local Paths

As of version 2.0.0 you can provide a path to a local directory that contains a package. Local paths can be saved using `npm install -S` or `npm install --save` , using any of these forms:

```
../foo/bar
~/foo/bar
./foo/bar
/foo/bar
```

in which case they will be normalized to a relative path and added to your `package.json`. For example:

```
{
  "name": "baz",
  "dependencies": {
    "bar": "file:../foo/bar"
  }
}
```

This feature is helpful for local offline development and creating tests that require npm installing where you don't want to hit an external server, but should not be used when publishing packages to the public registry.

devDependencies

If someone is planning on downloading and using your module in their program, then they probably don't want or need to download and build the external test or documentation framework that you use.

In this case, it's best to map these additional items in a `devDependencies` object.

These things will be installed when doing `npm link` or `npm install` from the root of a package, and can be managed like any other npm configuration param. See `npm-config` for more on the topic.

For build steps that are not platform-specific, such as compiling CoffeeScript or other languages to JavaScript, use the `prepare` script to do this, and make the required package a devDependency.

For example:

```
{ "name": "ethopia-waza",
  "description": "a delightfully fruity coffee varietal",
  "version": "1.2.3",
  "devDependencies": {
    "coffee-script": "~1.6.3"
```

```
  },
  "scripts": {
    "prepare": "coffee -o lib/ -c src/waza.coffee"
  },
  "main": "lib/waza.js"
}
```

The `prepare` script will be run before publishing, so that users can consume the functionality without requiring them to compile it themselves. In dev mode (ie, locally running `npm install`), it'll run this script as well, so that you can test it easily.

peerDependencies

In some cases, you want to express the compatibility of your package with a host tool or library, while not necessarily doing a `require` of this host. This is usually referred to as a *plugin*. Notably, your module may be exposing a specific interface, expected and specified by the host documentation.

For example:

```
{
  "name": "tea-latte",
  "version": "1.3.5",
  "peerDependencies": {
    "tea": "2.x"
  }
}
```

This ensures your package `tea-latte` can be installed *along* with the second major version of the host package `tea` only. `npm install tea-latte` could possibly yield the following dependency graph:

```
├── tea-latte@1.3.5
└── tea@2.2.0
```

NOTE: npm versions 1 and 2 will automatically install `peerDependencies` if they are not explicitly depended upon higher in the dependency tree. In the next major version of npm (npm@3), this will no longer be the case. You will receive a warning that the peerDependency is not installed instead. The behavior in npms 1 & 2 was frequently confusing and could easily put you into dependency hell, a situation that npm is designed to avoid as much as possible.

Trying to install another plugin with a conflicting requirement will cause an error. For this reason, make sure your plugin requirement is as broad as possible, and not to lock it down to specific patch versions.

Assuming the host complies with semver, only changes in the host package's major version will break your plugin. Thus, if you've worked with every 1.x version of the host package, use `"^1.0"` or `"1.x"` to express this. If you depend on features introduced in 1.5.2, use `">= 1.5.2 < 2"` .

bundledDependencies

This defines an array of package names that will be bundled when publishing the package.

In cases where you need to preserve npm packages locally or have them available through a single file download, you can bundle the packages in a tarball file by specifying the package names in the `bundledDependencies` array and executing `npm pack` .

For example:

If we define a package.json like this:

```
{
  "name": "awesome-web-framework",
  "version": "1.0.0",
  "bundledDependencies": [
    "renderized", "super-streams"
  ]
}
```

we can obtain `awesome-web-framework-1.0.0.tgz` file by running `npm pack` . This file contains the dependencies `renderized` and `super-streams` which can be installed in a new project by executing `npm install awesome-web-framework-1.0.0.tgz` . Note that the package names do not include any versions, as that information is specified in `dependencies` .

If this is spelled `"bundleDependencies"` , then that is also honored.

optionalDependencies

If a dependency can be used, but you would like npm to proceed if it cannot be found or fails to install, then you may put it in the `optionalDependencies` object. This is a map of package name to version or url, just like the `dependencies` object. The difference is that build failures do not cause installation to fail.

It is still your program's responsibility to handle the lack of the dependency. For example, something like this:

```
try {
  var foo = require('foo')
  var fooVersion = require('foo/package.json').version
} catch (er) {
  foo = null
}
if ( notGoodFooVersion(fooVersion) ) {
  foo = null
}

// .. then later in your program ..

if (foo) {
  foo.doFooThings()
}
```

Entries in **`optionalDependencies`** will override entries of the same name in **`dependencies`**, so it's usually best to only put in one place.

engines

You can specify the version of node that your stuff works on:

```
{ "engines" : { "node" : ">=0.10.3 <0.12" } }
```

And, like with dependencies, if you don't specify the version (or if you specify "*" as the version), then any version of node will do.

If you specify an "engines" field, then npm will require that "node" be somewhere on that list. If "engines" is omitted, then npm will just assume that it works on node.

You can also use the "engines" field to specify which versions of npm are capable of properly installing your program. For example:

```
{ "engines" : { "npm" : "~1.0.20" } }
```

Unless the user has set the **`engine-strict`** config flag, this field is advisory only and will only produce warnings when your package is installed as a dependency.

engineStrict

This feature was removed in npm 3.0.0

Prior to npm 3.0.0, this feature was used to treat this package as if the user had set `engine-strict`. It is no longer used.

os

You can specify which operating systems your module will run on:

```
"os" : [ "darwin", "linux" ]
```

You can also blacklist instead of whitelist operating systems, just prepend the blacklisted os with a '!':

```
"os" : [ "!win32" ]
```

The host operating system is determined by `process.platform`

It is allowed to both blacklist, and whitelist, although there isn't any good reason to do this.

cpu

If your code only runs on certain cpu architectures, you can specify which ones.

```
"cpu" : [ "x64", "ia32" ]
```

Like the `os` option, you can also blacklist architectures:

```
"cpu" : [ "!arm", "!mips" ]
```

The host architecture is determined by `process.arch`

preferGlobal

DEPRECATED

This option used to trigger an npm warning, but it will no longer warn. It is purely there for informational purposes. It is now recommended that you install any binaries as local devDependencies wherever possible.

private

If you set `"private": true` in your package.json, then npm will refuse to publish it.

This is a way to prevent accidental publication of private repositories. If you would like to ensure that a given package is only ever published to a specific registry (for example, an internal registry), then use the `publishConfig` dictionary described below to override the `registry` config param at publish-time.

publishConfig

This is a set of config values that will be used at publish-time. It's especially handy if you want to set the tag, registry or access, so that you can ensure that a given package is not tagged with "latest", published to the global public registry or that a scoped module is private by default.

Any config values can be overridden, but only "tag", "registry" and "access" probably matter for the purposes of publishing.

See `npm-config` to see the list of config options that can be overridden.

DEFAULT VALUES

npm will default some values based on package contents.

- `"scripts": {"start": "node server.js"}`

  If there is a `server.js` file in the root of your package, then npm will default the `start` command to `node server.js` .

- `"scripts":{"install": "node-gyp rebuild"}`

  If there is a `binding.gyp` file in the root of your package and you have not defined an `install` or `preinstall` script, npm will default the `install` command to compile using node-gyp.

- `"contributors": [...]`

  If there is an `AUTHORS` file in the root of your package, npm will treat each line as a `Name <email> (url)` format, where email and url are optional. Lines which start with a `#` or are blank, will be ignored.