

# npm-scripts

## How npm handles the "scripts" field

### DESCRIPTION

npm supports the "scripts" property of the package.json file, for the following scripts:

- **prepublish**: Run BEFORE the package is packed and published, as well as on local **npm install** without any arguments. (See below)
- **prepare**: Run both BEFORE the package is packed and published, on local **npm install** without any arguments, and when installing git dependencies (See below). This is run AFTER **prepublish**, but BEFORE **prepublishOnly**.
- **prepublishOnly**: Run BEFORE the package is prepared and packed, ONLY on **npm publish**. (See below.)
- **prepack**: run BEFORE a tarball is packed (on **npm pack**, **npm publish**, and when installing git dependencies)
- **postpack**: Run AFTER the tarball has been generated and moved to its final destination.
- **publish**, **postpublish**: Run AFTER the package is published.
- **preinstall**: Run BEFORE the package is installed
- **install**, **postinstall**: Run AFTER the package is installed.
- **preuninstall**, **uninstall**: Run BEFORE the package is uninstalled.
- **postuninstall**: Run AFTER the package is uninstalled.
- **preversion**: Run BEFORE bumping the package version.
- **version**: Run AFTER bumping the package version, but BEFORE commit.
- **postversion**: Run AFTER bumping the package version, and AFTER commit.
- **pretest**, **test**, **posttest**: Run by the **npm test** command.
- **prestop**, **stop**, **poststop**: Run by the **npm stop** command.
- **prestart**, **start**, **poststart**: Run by the **npm start** command.
- **prerestart**, **restart**, **postrestart**: Run by the **npm restart** command. Note: **npm restart** will run the stop and start scripts if no **restart** script is provided.
- **preshrinkwrap**, **shrinkwrap**, **postshrinkwrap**: Run by the **npm shrinkwrap** command.

Additionally, arbitrary scripts can be executed by running **npm run-script**

**<stage>**. *Pre* and *post* commands with matching names will be run for those as well (e.g. **premyscript**, **myscript**, **postmyscript**). Scripts from dependencies can be run with **npm explore <pkg> -- npm run <stage>**.

# PREPUBLISH AND PREPARE

## DEPRECATION NOTE

Since `npm@1.1.71`, the npm CLI has run the `prepublish` script for both `npm publish` and `npm install`, because it's a convenient way to prepare a package for use (some common use cases are described in the section below). It has also turned out to be, in practice, **very confusing**. As of `npm@4.0.0`, a new event has been introduced, `prepare`, that preserves this existing behavior. A *new* event, `prepublishOnly` has been added as a transitional strategy to allow users to avoid the confusing behavior of existing npm versions and only run on `npm publish` (for instance, running the tests one last time to ensure they're in good shape).

See <https://github.com/npm/npm/issues/10074> for a much lengthier justification, with further reading, for this change.

## USE CASES

If you need to perform operations on your package before it is used, in a way that is not dependent on the operating system or architecture of the target system, use a `prepublish` script. This includes tasks such as:

- Compiling CoffeeScript source code into JavaScript.
- Creating minified versions of JavaScript source code.
- Fetching remote resources that your package will use.

The advantage of doing these things at `prepublish` time is that they can be done once, in a single place, thus reducing complexity and variability. Additionally, this means that:

- You can depend on `coffee-script` as a `devDependency`, and thus your users don't need to have it installed.
- You don't need to include minifiers in your package, reducing the size for your users.
- You don't need to rely on your users having `curl` or `wget` or other system tools on the target machines.

## DEFAULT VALUES

npm will default some script values based on package contents.

- `"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`.

- `"install": "node-gyp rebuild"` :

If there is a `binding.gyp` file in the root of your package and you haven't defined your own `install` or `preinstall` scripts, npm will default the `install` command to compile using node-gyp.

## USER

If npm was invoked with root privileges, then it will change the uid to the user account or uid specified by the `user` config, which defaults to `nobody`. Set the `unsafe-perm` flag to run scripts with root privileges.

## ENVIRONMENT

Package scripts run in an environment where many pieces of information are made available regarding the setup of npm and the current state of the process.

### path

If you depend on modules that define executable scripts, like test suites, then those executables will be added to the `PATH` for executing the scripts. So, if your `package.json` has this:

```
{ "name" : "foo"
, "dependencies" : { "bar" : "0.1.x" }
, "scripts": { "start" : "bar ./test" } }
```

then you could run `npm start` to execute the `bar` script, which is exported into the `node_modules/.bin` directory on `npm install`.

### package.json vars

The `package.json` fields are tacked onto the `npm_package_` prefix. So, for instance, if you had `{"name":"foo", "version":"1.2.5"}` in your `package.json` file, then your package scripts would have the `npm_package_name` environment variable set to "foo", and the `npm_package_version` set to "1.2.5". You can access these variables in your code with `process.env.npm_package_name` and `process.env.npm_package_version`, and so on for other fields.

### configuration

Configuration parameters are put in the environment with the `npm_config_` prefix. For instance, you can view the effective `root` config by checking the `npm_config_root` environment variable.

## Special: package.json “config” object

The package.json “config” keys are overwritten in the environment if there is a config param of `<name>[@<version>]:<key>` . For example, if the package.json has this:

```
{ "name" : "foo"
, "config" : { "port" : "8080" }
, "scripts" : { "start" : "node server.js" } }
```

and the server.js is this:

```
http.createServer(...).listen(process.env.npm_package_config_port)
```

then the user could change the behavior by doing:

```
npm config set foo:port 80
```

## current lifecycle event

Lastly, the `npm_lifecycle_event` environment variable is set to whichever stage of the cycle is being executed. So, you could have a single script used for different parts of the process which switches based on what’s currently happening.

Objects are flattened following this format, so if you had `{"scripts": {"install": "foo.js"}}` in your package.json, then you’d see this in the script:

```
process.env.npm_package_scripts_install === "foo.js"
```

## EXAMPLES

For example, if your package.json contains this:

```
{ "scripts" :
  { "install" : "scripts/install.js"
  , "postinstall" : "scripts/install.js"
  , "uninstall" : "scripts/uninstall.js"
  }
}
```

---

then `scripts/install.js` will be called for the install and post-install stages of the lifecycle, and `scripts/uninstall.js` will be called when the package is uninstalled. Since `scripts/install.js` is running for two different phases, it would be wise in this case to look at the `npm_lifecycle_event` environment variable.

If you want to run a make command, you can do so. This works just fine:

```
{ "scripts" :  
  { "preinstall" : "./configure"  
    , "install" : "make && make install"  
    , "test" : "make test"  
  }  
}
```

## EXITING

Scripts are run by passing the line as a script argument to `sh`.

If the script exits with a code other than 0, then this will abort the process.

Note that these script files don't have to be nodejs or even javascript programs. They just have to be some kind of executable file.

## HOOK SCRIPTS

If you want to run a specific script at a specific lifecycle event for ALL packages, then you can use a hook script.

Place an executable file at `node_modules/.hooks/{eventname}`, and it'll get run for all packages when they are going through that point in the package lifecycle for any packages installed in that root.

Hook scripts are run exactly the same way as package.json scripts. That is, they are in a separate child process, with the env described above.

## BEST PRACTICES

- Don't exit with a non-zero error code unless you *really* mean it. Except for uninstall scripts, this will cause the npm action to fail, and potentially be rolled back. If the failure is minor or only will prevent some optional features, then it's better to just print a warning and exit successfully.

- Try not to use scripts to do what npm can do for you. Read through **package.json** to see all the things that you can specify and enable by simply describing your package appropriately. In general, this will lead to a more robust and consistent state.
- Inspect the env to determine where to put things. For instance, if the **npm\_config\_binroot** environment variable is set to **/home/user/bin** , then don't try to install executables into **/usr/local/bin** . The user probably set it up that way for a reason.
- Don't prefix your script commands with "sudo". If root permissions are required for some reason, then it'll fail with that error, and the user will sudo the npm command in question.
- Don't use **install** . Use a **.gyp** file for compilation, and **prepublish** for anything else. You should almost never have to explicitly set a preinstall or install script. If you are doing this, please consider if there is another option. The only valid use of **install** or **preinstall** scripts is for compilation which must be done on the target architecture.