MyWebUniversity

python

Red Hat Enterprise Linux Manual Pages

FPDF Library PDF generator

Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'scripts.7' command

$ man scripts.7

SCRIPTS(7)   SCRIPTS(7)

NAME

scripts - How npm handles the "scripts" field

Description

The "scripts" property of your package.json file supports a number of

built-in scripts and their preset life cycle events as well as arbi?

trary scripts. These all can be executed by running npm run-script

<stage> or npm run <stage> for short. Pre and post commands with match?

ing 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>.

Pre & Post Scripts

To create "pre" or "post" scripts for any scripts defined in the

"scripts" section of the package.json, simply create another script

with a matching name and add "pre" or "post" to the beginning of them.

```
 {
   "scripts": {
     "precompress": "{{ executes BEFORE the `compress` script }}",
     "compress": "{{ run command to compress files }}",
     "postcompress": "{{ executes AFTER `compress` script }}"
   }
 }
```

In this example npm run compress would execute these scripts as de?

scribed.

Life Cycle Scripts

There are some special life cycle scripts that happen only in certain situations. These scripts happen in addition to the pre<event>, post<event>, and <event> scripts.

? prepare, prepublish, prepublishOnly, prepack, postpack, dependencies

prepare (since npm@4.0.0)

? Runs any time before the package is packed, i.e. during npm publish and npm pack

? Runs BEFORE the package is packed

? Runs BEFORE the package is published

? Runs on local npm install without any arguments

? Run AFTER prepublish, but BEFORE prepublishOnly

? NOTE: If a package being installed through git contains a prepare script, its dependencies and devDependencies will be installed, and the prepare script will be run, before the package is packaged and installed.

? As of npm@7 these scripts run in the background. To see the output, run with: --foreground-scripts.

prepublish (DEPRECATED)

? Does not run during npm publish, but does run during npm ci and npm install. See below for more info.

prepublishOnly

? Runs BEFORE the package is prepared and packed, ONLY on npm publish.

prepack

? Runs BEFORE a tarball is packed (on "npm pack", "npm publish", and when installing a git dependencies).

? NOTE: "npm run pack" is NOT the same as "npm pack". "npm run pack" is an arbitrary user defined script name, where as, "npm pack" is a CLI defined command.

postpack

? Runs AFTER the tarball has been generated but before it is moved to its final destination (if at all, publish does not save the tarball locally)

dependencies

? Runs AFTER any operations that modify the node_modules directory IF
  changes occurred.

? Does NOT run in global mode

Prepare and Prepublish

Deprecation Note: prepublish

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 be?
low). It has also turned out to be, in practice, very confusing
https://github.com/npm/npm/issues/10074. 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 justi?
fication, 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 variabil?
ity. 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.

Dependencies

The dependencies script is run any time an npm command causes changes
to the node_modules directory. It is run AFTER the changes have been
applied and the package.json and package-lock.json files have been up?
dated.

Life Cycle Operation Order

npm help cache add

? prepare

npm help ci

? preinstall

? install

? postinstall

? prepublish

? preprepare

? prepare

? postprepare These all run after the actual installation of modules
into node_modules, in order, with no internal actions happening in
between

npm help diff

? prepare

npm help install

These also run when you run npm install -g <pkg-name>

? preinstall

? install

? postinstall

? prepublish

? preprepare

? prepare

? postprepare

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 de?

fault the install command to compile using node-gyp via node-gyp re?

build

These are run from the scripts of <pkg-name>

npm help pack

? prepack

? prepare

? postpack

npm help publish

? prepublishOnly

? prepack

? prepare

? postpack

? publish

? postpublish

prepare will not run during --dry-run

npm help rebuild

? preinstall

? install

? postinstall

? prepare

prepare is only run if the current directory is a symlink (e.g. with

linked packages)

npm help restart

If there is a restart script defined, these events are run, otherwise

stop and start are both run if present, including their pre and post

iterations)

? prerestart

? restart

? postrestart

npm run <user defined> /commands/npm-run-script

? pre<user-defined>

? <user-defined>

? post<user-defined>

npm help start

? prestart

? start

? poststart

If there is a server.js file in the root of your package, then npm will default the start command to node server.js. prestart and poststart will still run in this case.

npm help stop

? prestop

? stop

? poststop

npm help test

? pretest

? test

? posttest

npm help version

? preversion

? version

? postversion

A Note on a lack of npm help uninstall scripts

While npm v6 had uninstall lifecycle scripts, npm v7 does not. Removal of a package can happen for a wide variety of reasons, and there's no clear way to currently give the script enough context to be useful.

Reasons for a package removal include:

? a user directly uninstalled this package

? a user uninstalled a dependant package and so this dependency is be?
ing uninstalled

? a user uninstalled a dependant package but another package also de?
pends on this version

? this version has been merged as a duplicate with another version

? etc.

Due to the lack of necessary context, uninstall lifecycle scripts are not implemented and will not function.

User

When npm is run as root, scripts are always run with the effective uid

and gid of the working directory owner.

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 ex?

ported 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_pack?

age_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.

See package.json /configuring-npm/package-json for more on package con?

figs.

current lifecycle event

Lastly, the npm_lifecycle_event environment variable is set to which? ever 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 Node.js or even Java? Script programs. They just have to be some kind of executable file.

Best Practices

? Don't exit with a non-zero error code unless you really mean it. Ex? cept 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 /configuring-npm/package-json to see all the things that you can specify and enable by simply describing your package appro? priately. 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 prepare for anything else. You should almost never have to explicitly set a pre? install 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 architec? ture.

? Scripts are run from the root of the package folder, regardless of what the current working directory is when npm is invoked. If you want your script to use different behavior based on what subdirectory you're in, you can use the INIT_CWD environment variable, which holds the full path you were in when you ran npm run.

See Also

? npm help run-script

? package.json /configuring-npm/package-json

? npm help developers

? npm help install