Let’s create a small example:

cd /tmp

mkdir nodeexample

cd nodeexample

npm init -y

Wrote to /private/tmp/nodeexample/package.json:

{

"name": "nodeexample",

"version": "1.0.0",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC",

"dependencies": {

"chalk": "^4.0.0"

},

"devDependencies": {},

"description": ""

}

touch whatever.js

npm install chalk

npm WARN nodeexample@1.0.0 No description

npm WARN nodeexample@1.0.0 No repository field.

+ chalk@4.0.0

updated 1 package and audited 7 packages in 6.686s

1 package is looking for funding

run `npm fund` for details

found 0 vulnerabilities

echo "const chalk = require('chalk');" >> whatever.js

echo "console.log(chalk.blue('Hello world'));" >> whatever.js

cat whatever.js

const chalk = require('chalk');

console.log(chalk.blue('Hello world'));

Now this can be run with Node:

node /tmp/nodeexample/whatever.js

Hello world

Here is our current file structure:

fs::dir\_tree("/tmp/nodeexample")

/tmp/nodeexample

├── node\_modules

│ ├── @types

│ │ └── color-name

│ │ ├── LICENSE

│ │ ├── README.md

│ │ ├── index.d.ts

│ │ └── package.json

│ ├── ansi-styles

│ │ ├── index.d.ts

│ │ ├── index.js

│ │ ├── license

│ │ ├── package.json

│ │ └── readme.md

│ ├── chalk

│ │ ├── index.d.ts

│ │ ├── license

│ │ ├── package.json

│ │ ├── readme.md

│ │ └── source

│ │ ├── index.js

│ │ ├── templates.js

│ │ └── util.js

│ ├── color-convert

│ │ ├── CHANGELOG.md

│ │ ├── LICENSE

│ │ ├── README.md

│ │ ├── conversions.js

│ │ ├── index.js

│ │ ├── package.json

│ │ └── route.js

│ ├── color-name

│ │ ├── LICENSE

│ │ ├── README.md

│ │ ├── index.js

│ │ └── package.json

│ ├── has-flag

│ │ ├── index.d.ts

│ │ ├── index.js

│ │ ├── license

│ │ ├── package.json

│ │ └── readme.md

│ └── supports-color

│ ├── browser.js

│ ├── index.js

│ ├── license

│ ├── package.json

│ └── readme.md

├── package-lock.json

├── package.json

└── whatever.js

As you can see, you have a node\_modules folder that contains all the  
modules, installed with your machine specific requirements.

Let’s now move this file to another folder (imagine it’s a git clone,  
or you’ve received a zip), where we won’t be sharing the node\_modules  
folder: the users will have to install it to their machine.

mkdir /tmp/nodeexample2

cp /tmp/nodeexample/package-lock.json /tmp/nodeexample2/package-lock.json

cp /tmp/nodeexample/package.json /tmp/nodeexample2/package.json

cp /tmp/nodeexample/whatever.js /tmp/nodeexample2/whatever.js

But if we try to run this script:

node /tmp/nodeexample2/whatever.js

node /tmp/nodeexample2/whatever.js

internal/modules/cjs/loader.js:979

throw err;

^

Error: Cannot find module 'chalk'

Require stack:

- /private/tmp/nodeexample2/whatever.js

at Function.Module.\_resolveFilename (internal/modules/cjs/loader.js:976:15)

at Function.Module.\_load (internal/modules/cjs/loader.js:859:27)

at Module.require (internal/modules/cjs/loader.js:1036:19)

at require (internal/modules/cjs/helpers.js:72:18)

at Object. (/private/tmp/nodeexample2/whatever.js:1:15)

at Module.\_compile (internal/modules/cjs/loader.js:1147:30)

at Object.Module.\_extensions..js (internal/modules/cjs/loader.js:1167:10)

at Module.load (internal/modules/cjs/loader.js:996:32)

at Function.Module.\_load (internal/modules/cjs/loader.js:896:14)

at Function.executeUserEntryPoint [as runMain] (internal/modules/run\_main.js:71:12) {

code: 'MODULE\_NOT\_FOUND',

requireStack: [ '/private/tmp/nodeexample2/whatever.js' ]

}

We have a “Module not found” error: that’s because we haven’t installed  
the dependencies yet. Let’s do that:

cd /tmp/nodeexample2 && npm install

npm WARN nodeexample@1.0.0 No description

npm WARN nodeexample@1.0.0 No repository field.

added 7 packages from 4 contributors and audited 7 packages in 2.132s

2 packages are looking for funding

run `npm fund` for details

found 0 vulnerabilities

fs::dir\_tree("/tmp/nodeexample2", recurse= 1)

/tmp/nodeexample2

├── node\_modules

│ ├── @types

│ ├── ansi-styles

│ ├── chalk

│ ├── color-convert

│ ├── color-name

│ ├── has-flag

│ └── supports-color

├── package-lock.json

├── package.json

└── whatever.js

cd /tmp/nodeexample2 && node whatever.js

Hello world

Tada !

Ok, but how can we bundle this into an R package? Here is how it will  
work:

* On our machine, we will create the full, working script into the  
  inst/ folder of the package, and share everything but our  
  node\_modules folder
* After the users have installed our package on their machines, they  
  will have inside their package installation folder something that  
  will look like the version of our /tmp/nodeexample2 just after our  
  cp: script.js, package.json and package-lock.json (so no  
  node\_modules folder, hence no dependencies).
* Then, from R, they will run an installation wrapper, that will call  
  npm install inside the package installation folder, *i.e* inside  
  system.file(package = "mypak"). That will add all the required  
  node\_modules.
* Once the installation is completed, we will call the Node script  
  inside the working directory where we just installed everything.  
  This script will take command line arguments passed from R

**node-minify**

While I’m at it, let’s try to use something that I might use in the  
future: node-minify, a node library which can minify CSS, notably  
through the clean-css extension:  
<https://www.npmjs.com/package/@node-minify/clean-css>.

If you don’t know what the minification is and what it’s used for, it’s  
the process of removing every unnecessary characters from a file so that  
it’s lighter. Because you know, on the web every byte counts.

See <https://en.wikipedia.org/wiki/Minification_(programming)> for more  
info.

**Step 1, create the package**

I won’t expand on that, please refer to online documentation.

**Step 2, initiate npm infrastructure**

Once in the package, here is the script to initiate everything:

mkdir -p inst/node

cd inst/node

npm init -y

npm install @node-minify/core @node-minify/clean-css

touch app.js

This app.js will do one thing: take the path to an input and an output  
file, and then run the node-minify with these two arguments.

**Step 3, creating the NodeJS script**

Here is app.js:

const minify = require('@node-minify/core');

const cleanCSS = require('@node-minify/clean-css');

minify({

compressor: cleanCSS,

input: process.argv[2],

output: process.argv[3],

callback: (e, res) => {}

});

Let’s now create a dummy css file:

echo "body {" >> test.css

echo " color:white;" >> test.css

echo "}" >> test.css

And try to process it:

node app.js test.css test2.css

cat test2.css

body{color:#fff}

Nice, we now have a script in inst/ that can be run with Node! How to  
make it available in R?

**Step 4, building functions**

Let’s start by ignoring the node\_modules folder.

usethis::use\_build\_ignore("inst/node/node\_modules/")

Then, create a function that will install the Node app on the users’  
machines, *i.e* where the package is installed.

minifyr\_npm\_install <- function(

force = FALSE

){

# Prompt the users unless they bypass (we're installing stuff on their machine)

if (!force) {

ok <- yesno::yesno("This will install our app on your local library.

Are you ok with that? ")

} else {

ok <- TRUE

}

# If user is ok, run npm install in the node folder in the package folder

# We should also check that the infra is not already there

if (ok){

processx::run(

command = "npm",

args = c("install"),

wd = system.file("node", package = "minifyr")

)

}

}

Let’s now build a function to run the minifyer:

minifyr\_run <- function(

input,

output

){

# We're taking the absolute path as we will move to another folder to

# execute the Node Script

input <- fs::path\_abs(input)

output <- fs::path\_abs(output)

processx::run(

command = "node",

args = c(

"app.js",

input,

output

),

wd = system.file("node", package = "minifyr")

)

return(output)

}

And here it is!

And with some extra package infrastructure, we’ve got everything we need  


**Step 5, try on our machine**

Let’s run the built package on our machine:

# To do once

minifyr::minifyr\_npm\_install()

Then, if we have a look at our local package lib:

fs::dir\_tree(

system.file(

"node",

package = "minifyr"

),

recurse = FALSE

)

/Library/Frameworks/R.framework/Versions/3.6/Resources/library/minifyr/node

├── app.js

├── node\_modules

├── package-lock.json

└── package.json

Let’s try our function:

# Dummy CSS creation

echo "body {" > test.css

echo " color:white;" >> test.css

echo "}" >> test.css

cat test.css

body {

color:white;

}

minifyr::minifyr\_run(

"test.css",

"test2.css"

)

cat test2.css

body{color:#fff}

Tada !

**Step 6, one last thing**

Of course, one cool thing would be to test that npm and Node are  
installed on the user’s machine. We can do that by running the version  
commands fornpm and node, and check if the results of system() are  
either 0 or 127, 127 meaning that the command failed to run.

node\_available <- function(){

test <- suppressWarnings(

system(

"npm -v",

ignore.stdout = TRUE,

ignore.stderr = TRUE

)

)

attempt::warn\_if(

test,

~ .x != 0,

"Error launching npm"

)

test <- suppressWarnings(

system(

"node -v",

ignore.stdout = TRUE,

ignore.stderr = TRUE

)

)

attempt::message\_if(

test,

~ .x != 0,

"Error launching Node"

)

}