**NPM**: Node Package Manager

Largest software registry in the world.

babel-cli package

react package

**Lecture 312**:

If we want to create public package or different people work on the same project, then we need to create package.json in the root of our project.

If we don’t plan to distribute our software and want to work on it just on our local computer, then we can go on without package.json file.

JSON: Javascript object notation

JSON is just text that is used for data exchange.

Format for data exchange

JSON object contains key-value pair.

It is very easy to convert JSON into js object and back.

We can either create package.json manually or with help of npm or yarn.

“npm init” or “yarn init”

Initialize project and create package.json.

Set name, version, author of the project.

Each project and each dependency has version.

**Lecture 313**: Semantic Versioning

Each public package must have unique name and version.

Each time we update a package, we must update its version.

Universal agreement for software versioning.

(semantic versioning or SemVer)

**Semver.org**

Version consists of 3 parts:

5.21.17

Major version, Minor version, Patch

If we perform some bugfixes, then we need to change patch version.

If we introduce some new features and our package remains compatible with previous versions, then we just update minor version.

If we introduce some non-compatible features or change our project completely then we need to change major version.

If other packages depend on our package and if we update major version, then we don’t guarantee that new updated version will work as before.

If our package depends on an external package, then we can specify which version of the external package we need.

Exact version 5.21.17

Greater than >5.21.1

Compatible changes ^5.21.8 (only minor and patch versions may change)

Minor-level changes ~5.21.8

**Pre-Release versions**:

Alpha, beta and release candidate

1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-beta <

1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 <

1.0.0

Added after the version.

Alpha version is the oldest

1.0.0 (finally release package with no suffixes)

**Lecture 314: Packages installation**

Install packages using npm or yarn

“npm install”: Install packages that are listed in the package.json

If package.json file is absent, then no packages will be installed.

Dependencies are installed under node\_modules directory.

If this folder is absent, then it is created automatically.

All of this happens inside the project folder.

npm install <package>

Install package in the project folder and add it as dependency in package.json

npm install <package> --save-dev

Install package in the project folder and add it as development dependency in package.json

**Lecture 315**:

Moment.js has its own package.json that describes this package.

Using require, we can import external modules or packages.

Node.js understands ‘require’ syntax but Google chrome does not understand it.

Run the script using node.js

moment does not have its own dependencies.

**Lecture 316**: Install Semver package

semver package allows us to perform different kinds of checks with versions.

This semver package is used in babel.

semver package does not have its own dependencies.

.bin folder

contains semver file

const semver = require(“semver”)

semver.helperMethods

If version is valid then we get its value back, else we get null.

semver.coerce(..)

This method coerces the passed in string to an object which has valid version as one of its property.

^2.22.2

>=2.22.2 <3.0.0

Only minor and patch versions can be changed.

~2.22.2

>=2.22.2 < 2.23.0

Semver package is used in Babel when we compare versions of different browsers.

If a certain package has its own dependencies, all of those dependencies will be installed along with this package.

**Lecture 319**:

Dependencies vs Development Dependencies

Dependency is a package installed using “npm install <package>”

If project contains package.json file and it has list of dependencies and does not have node\_modules folder and we enter ‘npm install’ command then all dependencies listed in package.json file will be installed.

If package A is installed using “npm install <packageA>” and it has dependency on package B, then package B will be also installed along with its dependencies.

Installation of dependencies is transitive.

In package.json file, we will just see only 1 dependency, package A.

In node\_modules folder, we will see package B and package C. We will also see these packages in package-lock.json file.

Development dependencies are also installed using “npm install”

* Not installed using “npm install --production”
* If package A is installed using “npm install <packageA>” and it has a devDependency on package B, then package B will not be installed.

Development dependencies are not installed in production environment.

Installation of development dependencies is

non-transitive.

We can also have node.js environment production flag.

In this case also, development dependencies will not be installed.

**Lecture 320**: Examine Development Dependencies

We can also uninstall a dependency:

“npm uninstall browserslist”

Its dependencies will also be deleted.

Now, lets add browserslist as a development dependency.

“npm install --save-dev browserslist” or

“npm install -D browserslist”

browserslist has 2 own dependencies.

Those packages will be installed as well.

It does not matter whether we add development dependency or a dependency, if we install this dependency in our project, dependencies of this dependency will be added as well.

Now let’s delete node\_modules folder.

If we do “npm i” then we will get the same set of packages as before.

node\_modules folder can be easily deleted when we move project from one computer to another or when we use distributed package control system like git.

If our project has package.json file and if node\_modules folder is absent and we execute “npm i” command or just yarn command, then all packages will be install.

Size of node\_modules folder can grow.

node\_modules folder does not need to be committed to any source control system.

It must be created on each computer that pulls our project.

Now let’s do installation using production flag.

“npm i --production”

node\_modules package is created again.

Packages listed as development dependencies were not listed.

That is the meaning of the production flag.

If a specified dependency has its own development dependency, then they are never installed in the project.

Should we add a package as a dependency or a dev dependency ?

**Lecture 321**:

Browser App vs Server Package

First we need to answer:

Are we building browser application or server package ?

Browser application is intended to be run in the browser.

These applications must have atleast 1 HTML file and it is usually called index.html.

Browser loads this file, loads some scripts and runs those scripts.

If we develop server package we do not have any html files. All we need to have is set of Javascript files.

Packages such as browserslist, semver are server packages.

‘semver’ package does not contain any html file. It has semver.js file.

Previously, we had executed this file using node.js and node.js is a server side technology.

We cannot execute semver.js file in the browser.

Each browser application must have atleast 1 HTML file.

In the browser we will get:

Uncaught ReferenceError: require is not defined

Browsers don’t understand this syntax but node.js understands it.

Browser does not support modules, common js modules.

It also does not support common ES6 modules.

If we want to make browsers work with modules, we need to use bundler such as webpack.

**Lecture 322**:

Create a browser application that uses functionality of moment.js package

* Create new package “clock”
* Initialize it and install ‘moment’ dependency
* Create index.html, src/style.css, src/index.js
* Place formatted time in the <h1>
* Update clock each second.

We are not allowed to use moment.js as a module.

Use the function located in moment.js file.

npm init -y

-y flag answers yes to all prompts.

‘npm install moment’

typeof exports and typeof define return false here.

We attach ‘moment’ property to the global variable.

And it is the result of factory() function execution.

global is the first argument and factory is the second argument of this function.

New property will be added to the ‘window’ object of the browser.

Use this function in the index.js file.

Use setTimeout()

We have taken part of moment.js and have inserted it into our application.

**NOTE**:

Dependencies and Development dependencies have Zero relation to the browser applications.

Browser does not know anything about the node\_modules folder.

It just downloads index.html file for example and then downloads scripts that are referenced in this index.html file.

Browser does not know anything about the backend, about package.json, node\_modules.

It just needs files required for application execution.

**Lecture 325**:

Add package as dependency or development dependency ?

It depends on the kind of our application.

When we just build browser applications, it does not matter how we will add packages because we will use those packages just to build final browser application.

In the last example, we built this application manually.

Some people say following:

If we use jquery or react in our application, these packages are front end packages so we need to add them as package dependencies.

If we just babel, mocha, etc then we need to add those packages as development dependencies.

Because these are not used for Front-End.

This is completely wrong.

Browser does not understand development dependencies or dependencies.

It just uses javascript files that we serve to the browser.

If we use functionality of jquery package or react package then we need to include this functionality in our js files.

We can add all packages that we use in our application as development dependencies.

Or we can add all of them as dependencies.

It doesn’t matter.

If we use our application just as a standalone application and no packages or libraries use our application, that is we just build the application for the browser.

In this case, it does not matter how we will add our dependencies.

Don’t install some packages as development dependencies and some as dependencies.

It does not matter for standalone application.

When it matters is when we create public package.

* Package is public
* Compiled version of our package uses features from dependent packages
* Other packages depend on our package

browserslist package depends on caniuse-lite package. It uses functions from caniuse-lite package inside of its own function.

That is why browserlist package has caniuse-lite package as a dependency.

My package depends on browserslist functionality.

We need to add dependencies and dev dependencies when we create public package.

Most packages in the world are used on the server only during development of other packages or applications.

That is why external packages have long list of development dependencies but only short list of dependencies.

It is not necessary that when we use external package with dependencies, then we will need those dependencies in our browser application if we build a browser app.

‘moment’ package has a long list of development dependencies. And these are not used in the final build.

Final build is located in moment.js file.

And we have used this build in our clock application.

That is why those dependencies are included as development dependencies.

They are not required for the compiled code of the ‘moment’ package.

If some packages are required for the final compiled code, then they will be added as a dependency.

If we just build standalone applications for the browser, we should add all packages as development dependencies.

We can add them as dependencies as well. But choose one way.

Dev dependencies are modules which are only required during development whereas dependencies are required at runtime. If you are deploying your application, dependencies has to be installed, or else your app simply will not work. Libraries that you call from your code that enables the program to run can be considered as dependencies.

Eg- React , React - dom

Dev dependency modules need not be installed in the production server since you are not gonna develop in that machine .compilers that covert your code to javascript , test frameworks and document generators can be considered as dev-dependencies since they are only required during development .

Eg- ESLint , Babel , webpack

**Lecture 326**:

“npm view <package>” prints information about the package.

versions is the number of versions of this package that have been published in the npm software registry.

To view list of all package versions,

“npm view <package> versions”

We can install a specific package version.

Current stable release is installed if we use

“npm install <package>”

“npm install <package@version>”

Some of the versions of moment.js could be vulnerable to a low severity regular expression denial of service when parsing dates as strings.

That is why it is recommended to update the version to some specified version or above.

**Lecture 328**:

Why package-lock.json file is needed ?

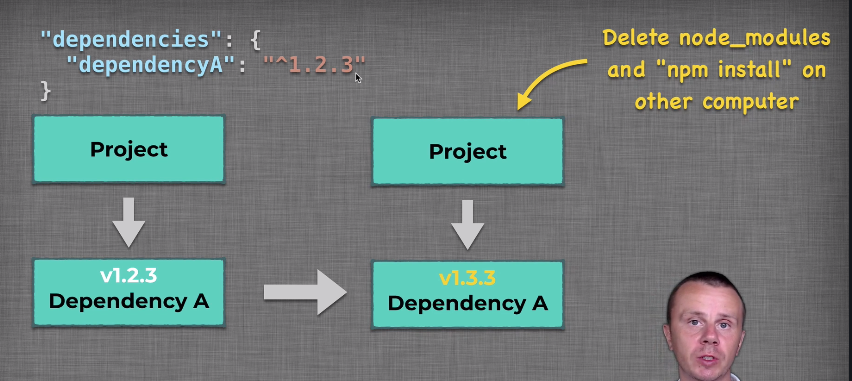
Alternative: yarn.lock

Lock file keeps versions tree of the project dependencies including child dependencies.

This file also includes all versions of each package.

Suppose different computers have different versions of an external package.

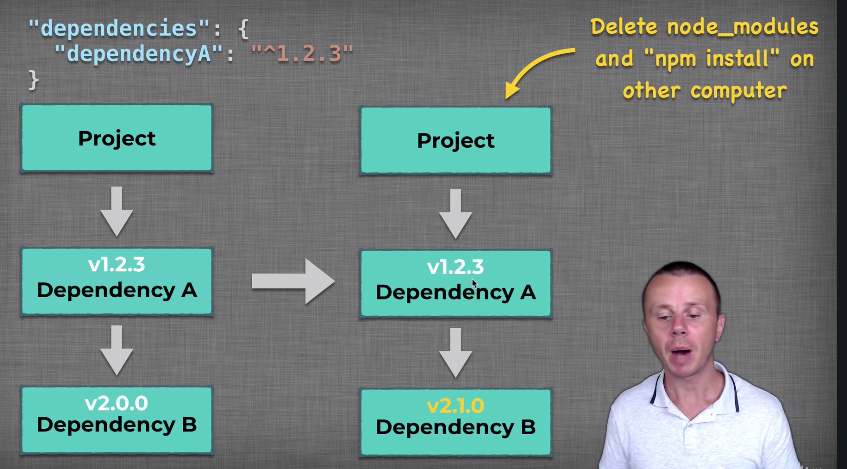
Minor version update means that some new features are added.



One user is able to use new features.

If we try to pull the updates and launch our project, we may get errors because we don’t have those new features of dependency A because our version is older than the version used to implement new updates in our project.

Here, version of direct dependency is changed.



Here dep A depends on dep B and version of dep B is changed.

In the package.json file we don’t have dep B listed.

It is listed in package.json file of depA.

Here also, we use different versions of dep B and problem may arise.

This is the problem that the lock file solves.

Lock file lists exact versions of all packages that are used in our project.

Now every user will pull same software versions of all packages.

“color.js”

Suppose we install ^1.1.0 version of ‘color’ as a dev dependency.

Now we delete node\_modules folder and package-lock.json file.

Now we again do “npm install”.

Now in the package-lock.json file, the specified version of color will be latest version compatible with 1.1.0

**Lecture 331**: How lock file is handled

* Lock file is created automatically starting from NPM v5.0.0
* It is updated automatically each time when we install a new package or update existing packages.

Lock file should be committed to source control systems like git.

node\_modules should not be committed.

Users who pull our code will install the same set of dependencies on their machine as we have on ours.

Version of dependencies will be consistent everywhere.

Lock file is not published to NPM software registry.

Lock file is just 1 for our project and npm packages are installed as dependencies of other projects.

When we install external packages, then we update lock file in our project and will list versions of dependencies and also versions of their dependencies if present.

So, lock file is just 1 for each project.

Inside the lock file,

“dev”: true means that it is a development dependency.

“requires”: {  
….

}

// list of dependencies of this package.

Suppose we have ‘mocha’ package as a development dependency.

And this package has its own dependencies. And they were installed as well.

Because we have installed ‘mocha’ package as a development dependency of our project, all its dependencies will be added as development dependencies into our project.

Large size of node\_modules: that is why it is never committed to source control systems.

**NOTE**:

If we commit package-lock.json file then people pulling our code will install the exact version of the dependencies as listed in the lock file even if newer versions are released.

If we don’t commit the lock file and just commit package.json file, then users performing “npm install” will install latest compatible versions of all dependencies.

Lock file

* Guarantees consistency of the dependencies versions
* Generated and updated automatically
* Committed to source control

package.json lists the versions of the dependencies which should be installed in order to make this project work.

package-lock.json specifies the exact version which is installed by “npm install” command.

**Lecture 333**:

npm update

Updates all listed packages to the latest release version within SemVer

^1.2.5 🡪 ^1.3.0

Newer version will be installed in the node\_modules folder.

If newer version is 2.0.0 then this package won’t be updated because of ^

npm update <package>

**Lecture 335**: Update project dependencies

* Create new project and perform npm init
* Install exact version of any package. Version must be older than latest version. Use --save-exact npm option.
* Install specific version of other package. Perform npm update and analyze result

If we specify exact version in package.json file, then no update will be performed.

If we specify ^ then latest compatible version will be installed.

**Lecture 336**: Introduction to NPM Scripts

Execute files and perform operations using npm commands.

Each npm script has its own name and all scripts are defined in package.json file in “scripts” section.

Different scripts such as “build”, “lint”, “test”, “flow”, “version-check”, “prettier”, etc.

When we do ‘npm init’ initially then there is only 1 default script called as “test”

To run this script we write: ‘npm test’

exit 1

npm ERR! Test failed. See above for more details.

“test” is a built in NPM script.

“npm start”

By default, this script executes a file called as server.js

Built in commands for scripts:

“npm start”

“npm stop”

“npm restart”

“npm test”

“npm prestart”

“npm poststart”

“prestart” script is automatically executed when we try to execute “start” script.

“prestart” is executed before “start”

“poststart” is executed after “start”

‘colors’ package modifies String.prototype and adds certain properties to String.

Each string will have access to properties such as green, inverse, rainbow and so on.

If we require ‘colors/safe’ package, then in this case we do not modify String.prototype.

require(“colors”)

We do not need to assign it to any variable as this module modifies global scope.

It changes prototype of the String.

In package.json, “main”: “lib/index.js”

This file is executed when we require this module.

Inside index.js there is

require(‘./extendStringPrototype’)()

This is the file which performs extension of the String prototype.

Different properties are added to String prototype.

See its working.

Safe option does not do modification of the String prototype.

We can require ‘colors/safe’ module.

In safe mode we need a reference variable.

const colors = require(“colors/safe”)

**Lecture 340**: Custom NPM Scripts

npm run <script>

Executes certain script.

It is an alias of npm run-script <script>

Run scripts in parallel.

Use external npm package, npm-run-all.

“all”: “npm-run-all --parallel custom1 custom2 …”

“npm-run-all” package

Results of the scripts could be mixed up.

--serial flag. One by one execution of scripts.

“npm run all”

npm looks in node\_modules, under .bin and finds npm-run-all file.

**Lecture 343**: NPM .bin folder

If we use external command such as npm-run-all to execute a script, then npm looks in node\_modules folder, under .bin for an executable file.

When we install any npm package, npm looks for “bin” field in the package.json file.

And if this field is present and contains certain key-value pairs then npm creates new files in the .bin folder.

In this case, “run-p”, “run-s”, “npm-run-all”

3 new files will be created.

Contents of following files will be copied to those newly created files.

The files that would be created will be executable files.

In npm-run-all we can see a special command,

#!/usr/bin/env node

This line is called shebang line.

Shebang determines which interpreter should be used for file execution on the Unix-like Operating systems.

The files in the .bin folder have no extension.

The machine does not know how to execute those files.

Shebang line gives instructions to the machine on how to launch a particular file.

Each file in the .bin folder is plain text file and to make it executable on unix like systems, we add the shebang line.

Shebang line must be the first line in the executable plain-text files.

**Mac**:

Let’s remove the shebang line.

“npm run all” gives many errors.

/usr/bin/env node launches node.js console.

This command uses env executable to launch certain interpreter which in our case is ‘node’.

node must be in our path.

Inside node interpreter, we launch contents of the file.

How are the commands parsed ?

**Windows**:

On windows, after executing “npm install”, under .bin folder we can see npm-run-all, run-p, etc files with .cmd extension.

These are executables in windows.

There is also a file without an extension.

(in .cmd file)

We use node … to execute corresponding js files.

package.json file is same as before.

Other npm packages with built in scripts.