npm-install

Install a package

SYNOPSIS

```
npm install (with no args, in package dir)
npm install [<@scope>/]<name>
npm install [<@scope>/]<name>@<tag>
npm install [<@scope>/]<name>@<version>
npm install [<@scope>/]<name>@<version range>
npm install <git-host>:<git-user>/<repo-name>
npm install <git repo url>
npm install <tarball file>
npm install <tarball url>
npm install <folder>

aliases: npm i, npm add
common options: [-P|--save-prod|-D|--save-dev|-O|--save-optional] [-E|--save
```

DESCRIPTION

This command installs a package, and any packages that it depends on. If the package has a package-lock or shrinkwrap file, the installation of dependencies will be driven by that, with an `npm-shrinkwrap.json` taking precedence if both files exist. See package-lock.json and npm-shrinkwrap.

A `package` is:

- a) a folder containing a program described by a `package.json` file
- b) a gzipped tarball containing (a)
- c) a url that resolves to (b)
- d) a `<name>@<version>` that is published on the registry (see `npm-registry`) with (c)
- e) a `<name>@<tag>` (see `npm-dist-tag`) that points to (d)
- f) a `<name>` that has a “latest” tag satisfying (e)
- g) a `<git remote url>` that resolves to (a)

Even if you never publish your package, you can still get a lot of benefits of using npm if you just want to write a node program (a), and perhaps if you also want to be able to easily install it elsewhere after packing it up into a tarball (b).

- `npm install` (in package directory, no arguments):

  Install the dependencies in the local node_modules folder.

  In global mode (ie, with `-g` or `--global` appended to the command), it installs the current package context (ie, the current working directory) as a global package.

  By default, `npm install` will install all modules listed as dependencies in `package.json` .

  With the `--production` flag (or when the `NODE_ENV` environment variable is set to `production`), npm will not install modules listed in `devDependencies` .

  > NOTE: The `--production` flag has no particular meaning when adding a dependency to a project.

- `npm install <folder>` :

  Install the package in the directory as a symlink in the current project. Its dependencies will be installed before it's linked. If `<folder>` sits inside the root of your project, its dependencies may be hoisted to the toplevel `node_modules` as they would for other types of dependencies.

- `npm install <tarball file>` :

  Install a package that is sitting on the filesystem. Note: if you just want to link a dev directory into your npm root, you can do this more easily by using `npm link` .

  Tarball requirements:

  - The filename *must* use `.tar` , `.tar.gz` , or `.tgz` as the extension.
  - The package contents should reside in a subfolder inside the tarball (usually it is called `package/`). npm strips one directory layer when installing the package (an equivalent of `tar x --strip-components=1` is run).
  - The package must contain a `package.json` file with `name` and `version` properties.

  Example:

  ```
  npm install ./package.tgz
  ```

- **`npm install <tarball url>`** :

  Fetch the tarball url, and then install it. In order to distinguish between this and other options, the argument must start with “http://” or “https://”

  Example:

  ```
      npm install https://github.com/indexzero/forever/tarball/v0.5.6
  ```

- **`npm install [<@scope>/]<name>`** :

  Do a **`<name>@<tag>`** install, where **`<tag>`** is the “tag” config. (See **`npm-config`** . The config’s default value is **`latest`** .)

  In most cases, this will install the version of the modules tagged as **`latest`** on the npm registry.

  Example:

  ```
      npm install sax
  ```

  `npm install` saves any specified packages into **`dependencies`** by default. Additionally, you can control where and how they get saved with some additional flags:

  - **`-P, --save-prod`** : Package will appear in your **`dependencies`** . This is the default unless **`-D`** or **`-O`** are present.
  - **`-D, --save-dev`** : Package will appear in your **`devDependencies`** .
  - **`-O, --save-optional`** : Package will appear in your **`optionalDependencies`** .
  - **`--no-save`** : Prevents saving to **`dependencies`** .

  When using any of the above options to save dependencies to your package.json, there are two additional, optional flags:

  - **`-E, --save-exact`** : Saved dependencies will be configured with an exact version rather than using npm’s default semver range operator.
  - **`-B, --save-bundle`** : Saved dependencies will also be added to your **`bundleDependencies`** list.

  Further, if you have an **`npm-shrinkwrap.json`** or **`package-lock.json`** then it will be updated as well.

`<scope>` is optional. The package will be downloaded from the registry associated with the specified scope. If no registry is associated with the given scope the default registry is assumed. See `npm-scope`.

Note: if you do not include the @-symbol on your scope name, npm will interpret this as a GitHub repository instead, see below. Scopes names must also be followed by a slash.

Examples:

```
      npm install sax
      npm install githubname/reponame
      npm install @myorg/privatepackage
      npm install node-tap --save-dev
      npm install dtrace-provider --save-optional
      npm install readable-stream --save-exact
      npm install ansi-regex --save-bundle
```

Note: If there is a file or folder named `<name>` in the current working directory, then it will try to install that, and only try to fetch the package by name if it is not valid.

- **`npm install [<@scope>/]<name>@<tag>`** :

  Install the version of the package that is referenced by the specified tag. If the tag does not exist in the registry data for that package, then this will fail.

  Example:

  ```
        npm install sax@latest
        npm install @myorg/mypackage@latest
  ```

- **`npm install [<@scope>/]<name>@<version>`** :

  Install the specified version of the package. This will fail if the version has not been published to the registry.

  Example:

  ```
        npm install sax@0.1.1
        npm install @myorg/privatepackage@1.5.0
  ```

- **`npm install [<@scope>/]<name>@<version range>`** :

  Install a version of the package matching the specified version range. This will follow the same rules for resolving dependencies described in `package.json`.

Note that most version ranges must be put in quotes so that your shell will treat it as a single argument.

Example:

```
    npm install sax@">=0.1.0 <0.2.0"
    npm install @myorg/privatepackage@">=0.1.0 <0.2.0"
```

- `npm install <git remote url>` :

  Installs the package from the hosted git provider, cloning it with `git` . For a full git remote url, only that URL will be attempted.

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

  `<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 the default branch of the repository is used.

  If the repository makes use of submodules, those submodules will be cloned as well.

  If the package being installed 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.

  The following git environment variables are recognized by npm and will be added to the environment when running git:

  - `GIT_ASKPASS`
  - `GIT_EXEC_PATH`
  - `GIT_PROXY_COMMAND`
  - `GIT_SSH`
  - `GIT_SSH_COMMAND`
  - `GIT_SSL_CAINFO`
  - `GIT_SSL_NO_VERIFY`

  See the git man page for details.

Examples:

```
    npm install git+ssh://git@github.com:npm/cli.git#v1.0.27
    npm install git+ssh://git@github.com:npm/cli#semver:^5.0
    npm install git+https://isaacs@github.com/npm/cli.git
    npm install git://github.com/npm/cli.git#v1.0.27
    GIT_SSH_COMMAND='ssh -i ~/.ssh/custom_ident' npm install git+ssh://g
```

- **`npm install <githubname>/<githubrepo>[#<commit-ish>]`** :
- **`npm install github:<githubname>/<githubrepo>[#<commit-ish>]`** :

  Install the package at `https://github.com/githubname/githubrepo` by attempting to clone it using `git` .

  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.

  As with regular git dependencies, `dependencies` and `devDependencies` will be installed if the package has a `prepare` script, before the package is done installing.

  Examples:

```
    npm install mygithubuser/myproject
    npm install github:mygithubuser/myproject
```

- **`npm install gist:[<githubname>/]<gistID>[#<commit-ish>|#semver:<semver>]`** :

  Install the package at `https://gist.github.com/gistID` by attempting to clone it using `git` . The GitHub username associated with the gist is optional and will not be saved in `package.json` .

  As with regular git dependencies, `dependencies` and `devDependencies` will be installed if the package has a `prepare` script, before the package is done installing.

  Example:

```
    npm install gist:101a11beef
```

- `npm install bitbucket:<bitbucketname>/<bitbucketrepo>[#<commit-ish>]` :

  Install the package at `https://bitbucket.org/bitbucketname/bitbucketrepo` by attempting to clone it using `git` .

  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.

  As with regular git dependencies, `dependencies` and `devDependencies` will be installed if the package has a `prepare` script, before the package is done installing.

  Example:

  ```
      npm install bitbucket:mybitbucketuser/myproject
  ```

- `npm install gitlab:<gitlabname>/<gitlabrepo>[#<commit-ish>]` :

  Install the package at `https://gitlab.com/gitlabname/gitlabrepo` by attempting to clone it using `git` .

  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.

  As with regular git dependencies, `dependencies` and `devDependencies` will be installed if the package has a `prepare` script, before the package is done installing.

  Example:

  ```
      npm install gitlab:mygitlabuser/myproject
      npm install gitlab:myusr/myproj#semver:^5.0
  ```

You may combine multiple arguments, and even multiple types of arguments. For example:

```
npm install sax@">=0.1.0 <0.2.0" bench supervisor
```

The `--tag` argument will apply to all of the specified install targets. If a tag with the given name exists, the tagged version is preferred over newer versions.

The `--dry-run` argument will report in the usual way what the install would have done without actually installing anything.

The `--package-lock-only` argument will only update the `package-lock.json` , instead of checking `node_modules` and downloading dependencies.

The `-f` or `--force` argument will force npm to fetch remote resources even if a local copy exists on disk.

```
npm install sax --force
```

The `-g` or `--global` argument will cause npm to install the package globally rather than locally. See `npm-folders` .

The `--global-style` argument will cause npm to install the package into your local `node_modules` folder with the same layout it uses with the global `node_modules` folder. Only your direct dependencies will show in `node_modules` and everything they depend on will be flattened in their `node_modules` folders. This obviously will eliminate some deduping.

The `--ignore-scripts` argument will cause npm to not execute any scripts defined in the package.json. See `npm-scripts` .

The `--legacy-bundling` argument will cause npm to install the package such that versions of npm prior to 1.4, such as the one included with node 0.8, can install the package. This eliminates all automatic deduping.

The `--link` argument will cause npm to link global installs into the local space in some cases.

The `--no-bin-links` argument will prevent npm from creating symlinks for any binaries the package might contain.

The `--no-optional` argument will prevent optional dependencies from being installed.

The `--no-shrinkwrap` argument, which will ignore an available package lock or shrinkwrap file and use the package.json instead.

The `--no-package-lock` argument will prevent npm from creating a `package-lock.json` file. When running with package-lock's disabled npm will not automatically prune your node modules when installing.

The `--nodedir=/path/to/node/source` argument will allow npm to find the node source code so that npm can compile native modules.

The `--only={prod[uction]|dev[elopment]}` argument will cause either only `devDependencies` or only non- `devDependencies` to be installed regardless of the `NODE_ENV` .

The `--no-audit` argument can be used to disable sending of audit reports to the configured registries. See `npm-audit` for details on what is sent.

See `npm-config` . Many of the configuration params have some effect on installation, since that's most of what npm does.

ALGORITHM

To install a package, npm uses the following algorithm:

```
load the existing node_modules tree from disk
clone the tree
fetch the package.json and assorted metadata and add it to the clone
walk the clone and add any missing dependencies
  dependencies will be added as close to the top as is possible
  without breaking any other modules
compare the original tree with the cloned tree and make a list of
actions to take to convert one to the other
execute all of the actions, deepest first
  kinds of actions are install, update, remove and move
```

For this `package{dep}` structure: `A{B,C}, B{C}, C{D}` , this algorithm produces:

```
A
+-- B
+-- C
+-- D
```

That is, the dependency from B to C is satisfied by the fact that A already caused C to be installed at a higher level. D is still installed at the top level because nothing conflicts with it.

For `A{B,C}, B{C,D@1}, C{D@2}` , this algorithm produces:

```
A
+-- B
+-- C
   `-- D@2
+-- D@1
```

Because B’s D@1 will be installed in the top level, C now has to install D@2 privately for itself. This algorithm is deterministic, but different trees may be produced if two dependencies are requested for installation in a different order.

See npm-folders for a more detailed description of the specific folder structures that npm creates.

Limitations of npm’s Install Algorithm

npm will refuse to install any package with an identical name to the current package. This can be overridden with the `--force` flag, but in most cases can simply be addressed by changing the local package name.

There are some very rare and pathological edge-cases where a cycle can cause npm to try to install a never-ending tree of packages. Here is the simplest case:

```
A -> B -> A' -> B' -> A -> B -> A' -> B' -> A -> ...
```

where `A` is some version of a package, and `A'` is a different version of the same package. Because `B` depends on a different version of `A` than the one that is already in the tree, it must install a separate copy. The same is true of `A'` , which must install `B'` . Because `B'` depends on the original version of `A` , which has been overridden, the cycle falls into infinite regress.

To avoid this situation, npm flat-out refuses to install any `name@version` that is already present anywhere in the tree of package folder ancestors. A more correct, but more complex, solution would be to symlink the existing version into the new location. If this ever affects a real use-case, it will be investigated.