@npmcli/arborist

Inspect and manage `node_modules` trees.

There's more documentation in the docs folder.

USAGE

```
const Arborist = require('@npmcli/arborist')

const arb = new Arborist({
  // options object
```

```
  // where we're doing stuff.  defaults to cwd.
  path: '/path/to/package/root',

  // url to the default registry.  defaults to npm's default registry
  registry: 'https://registry.npmjs.org',

  // scopes can be mapped to a different registry
  '@foo:registry': 'https://registry.foo.com/',

  // Auth can be provided in a couple of different ways.  If none are
  // provided, then requests are anonymous, and private packages will 404.
  // Arborist doesn't do anything with these, it just passes them down
  // the chain to pacote and npm-registry-fetch.

  // Safest: a bearer token provided by a registry:
  // 1. an npm auth token, used with the default registry
  token: 'deadbeefcafebad',
  // 2. an alias for the same thing:
  _authToken: 'deadbeefcafebad',

  // insecure options:
  // 3. basic auth, username:password, base64 encoded
  auth: 'aXNhYWNzOm5vdCBteSByZWFsIHBhc3N3b3Jk',
  // 4. username and base64 encoded password
  username: 'isaacs',
  password: 'bm90IG15IHJlYWwgcGFzc3dvcmQ=',

  // auth configs can also be scoped to a given registry with this
  // rather unusual pattern:
  '//registry.foo.com:token': 'blahblahblah',
  '//basic.auth.only.foo.com:_auth': 'aXNhYWNzOm5vdCBteSByZWFsIHBhc3N3b3Jk',
  '//registry.foo.com:always-auth': true,
})

// READING

// returns a promise.  reads the actual contents of node_modules
arb.loadActual().then(tree => {
  // tree is also stored at arb.virtualTree
})

// read just what the package-lock.json/npm-shrinkwrap says
// This *also* loads the yarn.lock file, but that's only relevant
// when building the ideal tree.
arb.loadVirtual().then(tree => {
  // tree is also stored at arb.virtualTree
  // now arb.virtualTree is loaded
  // this fails if there's no package-lock.json or package.json in the folder
  // note that loading this way should only be done if there's no
  // node_modules folder
})
```

```
// OPTIMIZING AND DESIGNING

// build an ideal tree from the package.json and various lockfiles.
arb.buildIdealTree(options).then(() => {
  // next step is to reify that ideal tree onto disk.
  // options can be:
  // rm: array of package names to remove at top level
  // add: Array of package specifiers to add at the top level.  Each of
  //   these will be resolved with pacote.manifest if the name can't be
  //   determined from the spec.  (Eg, `github:foo/bar` vs `foo@somespec`.)
  //   The dep will be saved in the location where it already exists,
  //   (or pkg.dependencies) unless a different saveType is specified.
  // saveType: Save added packages in a specific dependency set.
  //   - null (default) Wherever they exist already, or 'dependencies'
  //   - prod: definitely in 'dependencies'
  //   - optional: in 'optionalDependencies'
  //   - dev: devDependencies
  //   - peer: save in peerDependencies, and remove any optional flag from
  //     peerDependenciesMeta if one exists
  //   - peerOptional: save in peerDependencies, and add a
  //     peerDepsMeta[name].optional flag
  // saveBundle: add newly added deps to the bundleDependencies list
  // update: Either `true` to just go ahead and update everything, or an
  //   object with any or all of the following fields:
  //   - all: boolean.  set to true to just update everything
  //   - names: names of packages update (like `npm update foo`)
  // prune: boolean, default true.  Prune extraneous nodes from the tree.
  // preferDedupe: prefer to deduplicate packages if possible, rather than
  //   choosing a newer version of a dependency.  Defaults to false, ie,
  //   always try to get the latest and greatest deps.
  // legacyBundling: Nest every dep under the node requiring it, npm v2 style.
  //   No unnecessary deduplication.  Default false.

  // At the end of this process, arb.idealTree is set.
})

// WRITING

// Make the idealTree be the thing that's on disk
arb.reify({
  // write the lockfile(s) back to disk, and package.json with any updates
  // defaults to 'true'
  save: true,
}).then(() => {
  // node modules has been written to match the idealTree
})
```

DATA STRUCTURES

A `node_modules` tree is a logical graph of dependencies overlaid on a physical tree of folders.

A `Node` represents a package folder on disk, either at the root of the package, or within a `node_modules` folder. The physical structure of the folder tree is represented by the `node.parent` reference to the containing folder, and `node.children` map of nodes within its `node_modules` folder, where the key in the map is the name of the folder in `node_modules`, and the value is the child node.

A node without a parent is a top of tree.

A `Link` represents a symbolic link to a package on disk. This can be a symbolic link to a package folder within the current tree, or elsewhere on disk. The `link.target` is a reference to the actual node. Links differ from Nodes in that dependencies are resolved from the *target* location, rather than from the link location.

An `Edge` represents a dependency relationship. Each node has an `edgesIn` set, and an `edgesOut` map. Each edge has a `type` which specifies what kind of dependency it represents: `'prod'` for regular dependencies, `'peer'` for peerDependencies, `'dev'` for devDependencies, and `'optional'` for optionalDependencies. `edge.from` is a reference to the node that has the dependency, and `edge.to` is a reference to the node that requires the dependency.

As nodes are moved around in the tree, the graph edges are automatically updated to point at the new module resolution targets. In other words, `edge.from`, `edge.name`, and `edge.spec` are immutable; `edge.to` is updated automatically when a node's parent changes.

class Node

All arborist trees are `Node` objects. A `Node` refers to a package folder, which may have children in `node_modules`.

- `node.name` The name of this node's folder in `node_modules`.
- `node.parent` Physical parent node in the tree. The package in whose `node_modules` folder this package lives. Null if node is top of tree.

  Setting `node.parent` will automatically update `node.location` and all graph edges affected by the move.
- `node.meta` A `Shrinkwrap` object which looks up `resolved` and `integrity` values for all modules in this tree. Only relevant on `root` nodes.
- `node.children` Map of packages located in the node's `node_modules` folder.
- `node.package` The contents of this node's `package.json` file.
- `node.path` File path to this package. If the node is a link, then this is the path to the link, not to the link target. If the node is *not* a link, then this matches `node.realpath`.
- `node.realpath` The full real filepath on disk where this node lives.
- `node.location` A slash-normalized relative path from the root node to this node's path.
- `node.isLink` Whether this represents a symlink. Always `false` for Node objects, always `true` for Link objects.
- `node.isRoot` True if this node is a root node. (Ie, if `node.root === node`.)

- `node.root` The root node where we are working. If not assigned to some other value, resolves to the node itself. (Ie, the root node's `root` property refers to itself.)
- `node.isTop` True if this node is the top of its tree (ie, has no `parent`, false otherwise).
- `node.top` The top node in this node's tree. This will be equal to `node.root` for simple trees, but link targets will frequently be outside of (or nested somewhere within) a `node_modules` hierarchy, and so will have a different `top`.
- `node.dev`, `node.optional`, `node.devOptional`, `node.peer`, Indicators as to whether this node is a dev, optional, and/or peer dependency. These flags are relevant when pruning dependencies out of the tree or deciding what to reify. See **Package Dependency Flags** below for explanations.
- `node.edgesOut` Edges in the dependency graph indicating nodes that this node depends on, which resolve its dependencies.
- `node.edgesIn` Edges in the dependency graph indicating nodes that depend on this node.
- `extraneous` True if this package is not required by any other for any reason. False for top of tree.
- `node.resolve(name)` Identify the node that will be returned when code in this package runs `require(name)`
- `node.errors` Array of errors encountered while parsing package.json or version specifiers.

class Link

Link objects represent a symbolic link within the `node_modules` folder. They have most of the same properties and methods as `Node` objects, with a few differences.

- `link.target` A Node object representing the package that the link references. If this is a Node already present within the tree, then it will be the same object. If it's outside of the tree, then it will be treated as the top of its own tree.
- `link.isLink` Always true.
- `link.children` This is always an empty map, since links don't have their own children directly.

class Edge

Edge objects represent a dependency relationship a package node to the point in the tree where the dependency will be loaded. As nodes are moved within the tree, Edges automatically update to point to the appropriate location.

- `new Edge({ from, type, name, spec })` Creates a new edge with the specified fields. After instantiation, none of the fields can be changed directly.
- `edge.from` The node that has the dependency.
- `edge.type` The type of dependency. One of `'prod'`, `'dev'`, `'peer'`, or `'optional'`.
- `edge.name` The name of the dependency. Ie, the key in the relevant `package.json` dependencies object.
- `edge.spec` The specifier that is required. This can be a version, range, tag name, git url, or tarball URL. Any specifier allowed by npm is supported.
- `edge.to` Automatically set to the node in the tree that matches the `name` field.
- `edge.valid` True if `edge.to` satisfies the specifier.
- `edge.error` A string indicating the type of error if there is a problem, or `null` if it's valid. Values, in order of precedence:

    - `DETACHED` Indicates that the edge has been detached from its `edge.from` node, typically because a new edge was created when a dependency specifier was modified.
    - `MISSING` Indicates that the dependency is unmet. Note that this is *not* set for unmet dependencies of the `optional` type.
    - `PEER LOCAL` Indicates that a `peerDependency` is found in the node's local `node_modules` folder, and the node is not the top of the tree. This violates the `peerDependency` contract, because it means that the dependency is not a peer.
    - `INVALID` Indicates that the dependency does not satisfy `edge.spec`.
- `edge.reload()` Re-resolve to find the appropriate value for `edge.to`. Called automatically from the `Node` class when the tree is mutated.

Package Dependency Flags

The dependency type of a node can be determined efficiently by looking at the `dev`, `optional`, and `devOptional` flags on the node object. These are updated by arborist when necessary whenever the tree is modified in such a way that the dependency graph can change, and are relevant when pruning nodes from the tree.

```
| extraneous | peer | dev | optional | devOptional | meaning             | prune?
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |      |     |          |             | production dep      | never
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|     X      | N/A  | N/A |   N/A    |     N/A     | nothing depends on  | always
|
|            |      |     |          |             | this, it is trash   |
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |      |  X  |          |      X      | devDependency, or   | if pruning
dev    |
|            |      |     |          | not in lock | only depended upon  |
|
|            |      |     |          |             | by devDependencies  |
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |      |     |    X     |      X      | optionalDependency, | if pruning
|
|            |      |     |          | not in lock | or only depended on | optional
|
|            |      |     |          |             | by optionalDeps     |
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |      |  X  |    X     |      X      | Optional dependency | if pruning
EITHER |
|            |      |     |          | not in lock | of dep(s) in the    | dev OR
optional   |
```

```
|            |      |     |          |             | dev hierarchy       |
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |      |     |          |      X      | BOTH a non-optional | if pruning
BOTH   |
|            |      |     |          |   in lock   | dep within the dev  | dev AND
optional  |
|            |      |     |          |             | hierarchy, AND a    |
|
|            |      |     |          |             | dep within the      |
|
|            |      |     |          |             | optional hierarchy  |
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |  X   |     |          |             | peer dependency, or | if pruning
peers  |
|            |      |     |          |             | only depended on by |
|
|            |      |     |          |             | peer dependencies   |
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |  X   |  X  |          |      X      | peer dependency of  | if pruning
peer   |
|            |      |     |          | not in lock | dev node hierarchy  | OR dev deps
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |  X   |     |    X     |      X      | peer dependency of  | if pruning
peer   |
|            |      |     |          | not in lock | optional nodes, or  | OR optional
deps  |
|            |      |     |          |             | peerOptional dep    |
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |  X   |  X  |    X     |      X      | peer optional deps  | if pruning
peer   |
|            |      |     |          | not in lock | of the dev dep      | OR optional
OR    |
|            |      |     |          |             | hierarchy           | dev
|
|------------+------+-----+----------+-------------+---------------------+------------
-------|
|            |  X   |     |          |      X      | BOTH a non-optional | if pruning
peers  |
|            |      |     |          |   in lock   | peer dep within the | OR:
|
|            |      |     |          |             | dev hierarchy, AND  | BOTH
optional     |
```

```
|            |      |     |          |             | a peer optional dep | AND dev
deps      |
+------------+------+-----+----------+-------------+---------------------+-----------
-------+
```

- If none of these flags are set, then the node is required by the dependency and/or peerDependency hierarchy. It should not be pruned.
- If *both* `node.dev` and `node.optional` are set, then the node is an optional dependency of one of the packages in the devDependency hierarchy. It should be pruned if *either* dev or optional deps are being removed.
- If `node.dev` is set, but `node.optional` is not, then the node is required in the devDependency hierarchy. It should be pruned if dev dependencies are being removed.
- If `node.optional` is set, but `node.dev` is not, then the node is required in the optionalDependency hierarchy. It should be pruned if optional dependencies are being removed.
- If `node.devOptional` is set, then the node is a (non-optional) dependency within the devDependency hierarchy, *and* a dependency within the `optionalDependency` hierarchy. It should be pruned if *both* dev and optional dependencies are being removed.
- If `node.peer` is set, then all the same semantics apply as above, except that the dep is brought in by a peer dep at some point, rather than a normal non-peer dependency.

Note: `devOptional` is only set in the shrinkwrap/package-lock file if *neither* `dev` nor `optional` are set, as it would be redundant.

BIN

Arborist ships with a cli that can be used to run arborist specific commands outside of the context of the npm CLI. This script is currently not part of the public API and is subject to breaking changes outside of major version bumps.

To see the usage run:

```
npx @npmcli/arborist --help
```