Client-side npm proposal

*(author:* [*iminar@google.com*](mailto:iminar@google.com)*, <if you contribute, add your email here>*

*created: 2015-02-17, last update: 2015-02-19, status: draft,*

*url:* [*http://goo.gl/aBbojw*](http://goo.gl/aBbojw)*)*

*This document is published to the web as part of the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk) *folder*

[Goal](#heading=h.mwcioe49obu8)

[The current situation](#heading=h.fe10de6ddq7d)

[Targeted scenarios](#heading=h.hnn9gbgrmyw8)

[Known unresolved issues and todos for this proposal](#heading=h.agnn1sblqs5t)

[Proposed Changes](#heading=h.bjbam3gn5do7)

[Registry](#heading=h.3xsh4rp1tax7)

[Communication protocol](#heading=h.8r7qp7iuxpar)

[npm client](#heading=h.8u39a1gjgpjn)

[package.json manifest changes](#heading=h.zfs8jp6f0tjf)

[Reason for this change](#heading=h.zi4b9ap3m5o)

[Future considerations](#heading=h.bej7nhjcf562)

[Dependency and version constraint resolution changes](#heading=h.gnkhpwtxccuf)

[Reason for change](#heading=h.f1q7m4x48zdx)

[Cache changes](#heading=h.fiquv1qklxv0)

[File storage and directory layout changes](#heading=h.cnjnj2rvw7ar)

[Reason for change](#heading=h.fk4t72tqipdu)

[Future considerations](#heading=h.7k0loy27aewj)

[Shrinkwrap changes](#heading=h.ov3m3wlqrmd7)

[Reason for change](#heading=h.251tu6ol8nz3)

[Unresolved issues](#heading=h.eiu1h4urq0vv)

[Changes intentionally omitted from proposal](#heading=h.1gfzbqhddhfp)

[Module format and module loader integration](#heading=h.ncy40grlc56x)

[Support for transpilation of any kind](#heading=h.gqajxxqe649q)

[Build related support](#heading=h.qax26yte4cqi)

# Goal

This document contains a proposal for improving [npm](https://www.npmjs.com/) to be client-side friendly. It is meant to be the starting point for the discussion between the JS community and npm inc.

The primary objective of this proposal is to make client-side development with npm smoother without making too many changes to npm. We are looking for the minimal amount of changes that will give us the maximum benefit.

These requested changes should create a foundation on top of which additional features can be built via tooling. Because of this, every feature should be requested only with an explanation as to why it is not possible to provide the feature on top of what is in npm already.

# The current situation

<todo>

* primarily aimed at server-side w/ CJS
* client-side w/ vanilla npm
* npm + browserify
* npm + webpack

</todo>

# Targeted scenarios

<todo>

* simple flat dependency tree (no transitive dependencies)
* dependency tree with shared transitive dependencies
* dependency tree with peer dependencies (angular core + angular component)
* dependency tree with conflicting requirements on shared dependencies
* dependency tree with duplicates that developer opted-in to
* complex projects using shrinkwrap

</todo>

# Known unresolved issues and todos for this proposal

1. the shrinkwrap behavior for transitive dependencies with shrinkwrap is not well defined
2. the dependency version constraint resolution algorithm has *lots* of gaps
3. we don't have a solid story for dealing with packages that need to provide source, minified version and a CDN url of the dependency - this is a common use-case and must be tackled
4. attempt to generalize the client(Dev)Dependencies into a concept of environments/universes - isolated dependency graphs within a single project.

# 

# Proposed Changes

## Registry

No changes are expected.

## Communication protocol

No changes are expected.

## npm client

### package.json manifest changes

Add new top level section into the package.json manifest file, called browserDependencies, which would similar to [dependencies](https://docs.npmjs.com/files/package.json#dependencies), and hold information about the dependencies of the client-side app.

Example:

{

"name": "my-clientside-app",

"version": "1.0.0",

"devDependencies": {

"broccoli": "^0.13.3"

},

"browserDependencies": {

"someLib": "^0.3.2",

"someOtherLib": ^1.5.0"

}

...

}

#### Reason for this change

npm already distinguishes between dependencies and development dependencies (devDependencies), because devDependencies are used differently from production dependencies. Because a package can be either or both dependencies and development dependencies, it is not possible to flag the package in its package.json. The situation with client-side dependencies is similar and therefore it warrants similar treatment.

#### Future considerations

In the future a need might arise for browser development dependencies (for example a live-reload-like library or debugging library, that is needed only during development). Once this need is well understood, we should discuss adding browserDevDependencies section to package.json. The current proposal for adding browserDependencies is fully compatible with this possibility.

### Dependency and version constraint resolution changes

The proposal is to create a browser-friendly dependency resolver that will not allow duplicates of any package unless the developer explicitly opts-in. This will raise the awareness of the issues with duplicate dependencies and will require developers to deal with them, rather than blindly hope for the best.

The implementation could be similar to the one that [Dart's pub package manager uses](https://www.dartlang.org/tools/pub/versioning.html#constraint-solving).

This is just a high level pseudo algorithm that doesn't yet deal with infinite cycles, peer dependencies, opt-in duplicates and other corner-cases.

1. start with the dependency tree as defined in browserDependencies section of package.json
2. recursively walk the dependency tree
3. is the next dependency in the tree a previously unseen package?
   1. yes:
      1. find the latest version matching semver requirements for this dependency
      2. move the package dependency to the top level of the dependency tree (keeping the portion of the tree that we already processed flat)
      3. remember the dependency requirements that resulted in this version being picked.
   2. no: does the version of this package picked previously match semver requirements of this dependency?
      1. yes: we are good, go to #2
      2. no:
         1. does another version matching all the known requirements exist?
            1. yes:

update the version number in our dependency tree and add the new dependency requirement to the list

is there a diff between the dependencies of the previously selected version and the newly selected version? If so, recursively update the dependency tree.

* + - * 1. no: fail with an error stating what dependencies couldn't be satisfied

#### Reason for change

Unlike with server-side JavaScript, on the client-side we need much more aggressive deduplication of dependencies. This requirement is due to high sensitivity to payload size, as well as global state issues in the browser platform and legacy userland code.

### Cache changes

No changes are expected.

### 

### File storage and directory layout changes

We are proposing that a new directory called browser\_modules is created as sibling of the node\_modules directory and the flat dependency tree is serialized there.

#### Reason for change

In order for client-side build tools to be able to process packages fetched via npm, we need the flattened dependency tree to be serialized onto the file system.

Since the current node\_modules directory contains serialized representation of the node.js dependency tree which is not flat, it is not suitable for storage of the browser dependencies.

#### Future considerations

If in the future browserDevDependencies section is added to the package.json file, the flattened dependency tree will contain also browser development dependencies. These can then be serialized to the browser\_modules directory and mixed with browser dependencies.

### Shrinkwrap changes

In order to support locking of browser dependencies, the format of the npm-shrinkwrap.json file should be extended to contain additional top level section called browserDependencies, which will contained the flattened dependency tree that resulted from running the browser-friendly dependency resolution algorithm on the root dependencies specified via browserDependencies section in package.json.

The npm client should then be modified to take this section into account when doing performing operations in the current package.

#### Reason for change

dependency locking is one of the most important way to achieve deterministic builds across members of teams and build environments. It is crucial that the client-side dependencies have the same guarantees from npm as the dev and server dependencies have and for this reason, this feature must be implemented in npm.

#### Unresolved issues

* what to do about shrinkwrap constraints of transitive dependencies?
  + if they behave as today, we'll almost never be able to find suitable combination of dependencies for any nontrivial project
  + if we ignore them during version constraint resolution and honor only top level shrinkwrap file, we risk breaking transitive dependencies. This sounds like a better option because it can be mitigated by using stricter versioning for packages with fragile dependencies that are often transitively depended on.

# Changes intentionally omitted from proposal

## Module format and module loader integration

The client-side JavaScript is on the brink of massive migration to standards-based module format (ES Modules as specified in ES 2015 spec). But there is a lot of existing code that doesn't use this format yet and either uses formats like CJS or AMD, or uses globals as the export and import mechanism.

Because of this wide variety of formats out there, we recommended that the package manager focuses on delivering bits to the development/ci environment from where a build tool can take care of unifying the module formats across dependencies.

To signify the module system used by each package, a property in package.json can be used to specify the published module format. e.g. "moduleFormat": "es"

Alternatively, as a parallel effort to this proposal, a new support for package "dialects" could be added to npm. Dialect, is a variation of a package at a given version number. There can be more than one variation published for each package and version number. For example: someLibrary@2.0.0#module=es, someLibrary@2.0.0#module=cjs. The dialects are managed by the publisher of the package and are generated before publishing to npm. Dialect can have several axis, the most common being module format (es, cjs, amd, ..), language (es3, es5, es6, coffeescript, clojurescript, typescript) and obfuscation state (source, minified).

The interesting thing is that the dialect feature could be used by the server-side packages as well and enable migration to ES module format in the future once this support is available in node.js / io.js.

Given that there is a lot of contention expected around the module format and module loader topics, it is recommended that at least in the short term, this area is sidestepped by npm and left up to additional tooling to deal with.

## Support for transpilation of any kind

Similar to module format, opening up discussion about transpilation can quickly result in a dead-lock. Doing this properly for all possible use-cases would require significant investment and would result in bloating npm.

We recommend that this issue is at least for now sidestepped and left up to additional tooling to deal with.

## Build related support

Unlike server-side JavaScript, client-side JavaScript and asset files usually requires some sort of preprocessing before they are deployed to production.

Because of high variance in the community today and continuously changing trends in this area, it is our recommendation that npm does not try to take on any responsibilities in this area and instead leaves it up to other tools to deal with these tasks.