- Start Date: 13 Sep 18
- RFC PR: https://github.com/yarnpkg/rfcs/pull/101
- Yarn Issue: https://github.com/yarnpkg/yarn/issues/6382
- Champion: Maël Nison (@arcanis - Twitter)

Plug'n'Play Whitepaper

1. Summary

We propose in this RFC a new alternative and entirely optional way to resolve dependencies installed on the disk, in order to solve issues caused by the incomplete knowledge Node has regarding the dependency tree. We also detail the actual implementation we went with, describing the rational behind the design choice we made.

I'll keep it short in this summary since the document is already large, but here are some highlights:

- Installs ran using Plug'n'Play are up to 70% faster than regular ones (sample app)
- Starting from this PR, Yarn will now be on the path to make yarn install a no-op on CI
- Yarn will now be able to tell you precisely when you forgot to list packages in your dependencies
- Your applications will boot faster through a hybrid approach of static resolutions

This is but a high-level description of some of the benefits unlocked by Plug'n'Play, I encourage you to give a look at the document for more information about the specific design choices - and in case anything is missing, please ask and I'll do my best to explain them more in depth!

2. Motivation

When the first Javascript package managers appeared, they had to use the tools available to them. As a consequence, they were implemented on top of the Node resolution algorithm, copying the packages in such a layout that Node would be able to find them without external tool. This allowed the ecosystem to thrive, which interestingly revealed some of the scalability flaws in this approach:

- The Node resolution algorithm isn't aware of what packages are, and as a result doesn't know anything about dependencies either. When a module makes a require call, Node simply traverses the filesystem until it finds something that matches the request, and uses it.

- Installations take time, and the time required to copy files from the package managers' caches to the `node_modules` folders is one of the main bottleneck as it is heavily I/O bound, which cannot be easily optimized. Copy-on-Write filesystems can alleviate this issue to an extent but come with their own set of drawbacks, such as a lack of global support.

We at Facebook suffered from these issues, and decided to find a way to solve them cleanly while staying compatible with the current ecosystem. The solution we found has been put to the test inside our infrastructure for a few weeks now, and we now feel confident enough about its merits that we want to share with the community at large, and continue iterating on it in the open.

3. Detailed Design

In its current state, running `yarn install` does the following under the hood - assuming a cold setup:

1. Dependency ranges are resolved into pinned versions
2. Packages for each version are fetched from their sources and stored in the offline mirror
3. The offline mirror is unpacked into the cache
4. The cache is copied into the `node_modules` folders

Our solution aims to trim the fourth step from the equation. Instead of copying each and every file from the cache to the `node_modules` folder (which can take a lot of time, amplified by the sheer number of files), we will instead generate a single file that will contain static resolutions tables that list:

- What packages are available in the dependency tree
- How they are linked together
- Where they are located on the disk

A special resolver is then able to leverage the knowledge extracted from those tables to guide Node and help it figure out the location where each package has been installed (in our case, the Yarn cache). Since all packages from the dependency tree can be statically found inside the tables, the whole filesystem traversal required by the `node_modules` resolution can be skipped - bringing a free performance win at runtime by decreasing the amount of filesystem I/O needed to boot Node applications.

> **Why a special resolver rather than symlinks/hardlinks?**
>
> While quite useful, we believe both symlinks and hardlinks are solving the wrong problem. Symlinks require tooling support and have ambiguous semantics (Node has no way to know whether a symlink has been created through yarn link, through a workspace, or through a special installation). Hardlinks have surprising behaviors (they unexpectedly corrupt your cache if you change them), don't play

well with cross-volumes installs, and also require tooling support (for example when trying to persist the `node_modules` folders). Both of them require heavy I/O at install time, don't decrease the I/O at runtime, and more generally try to workaround the Node resolution rather than change it.

In the end, while some package managers had some success with these strategies (shoutout to pnpm in particular which was one of the first package managers experimenting to solve those problems!), we envision an alternative approach that's working for us at large scale and that we hope will work for others as well.

generated api

A point needs to be made regarding what Plug'n'Play is. While the previous section described the way Yarn would now generate "static resolution tables", those tables are not what make Plug'n'Play what it is - instead, they are an implementation detail amongst others. Instead, Plug'n'Play is first and foremost an API meant to abstract the resolution process and provide strict behavioral guarantees. In consequence, the resolution tables are useless without the resolver itself, since it's the resolver that is standardized, not the tables.

For this reason, we decided to generate a single Javascript file (called `.pnp.js`, for "Plug'n'Play") that would contain both the static resolution tables and the resolver itself. This has multiple advantages:

- The first one is encapsulation. By keeping the static resolution tables private to the resolver, we also prevent third-party scripts from relying on them, thus making it possible for us to change their underlying implementation as we see fit. A real-life case study lies within the `findPackageLocator` function. Its current implementation is somewhat naive and would be much improved through the use of a trie. Since the static resolution tables are hidden, we can easily replace their format to match this new data structure, which wouldn't necessarily be possible if we had publicized the underlying data structures.
- Project dependencies are traditionally versioned through the `dependencies` field, and installed using `yarn install`. But assuming that the resolver would be kept within Yarn, how would we make sure that the generated file is always compatible with the current Yarn version? In fact, how would we make sure that Yarn itself was available? As detailed in a later section, one of the future improvements we've planned is to make Yarn entirely optional on CI. In this context, where would the "runtime" required for such a resolver be located? The easiest answer is to keep the static resolution tables tied to the resolver itself, removing the risk of unexpected incompatibilities.

As an executable Javascript file, the generated `.pnp.js` file presents the following features:

- It doesn't have any dependency other than the built-in Node modules - it doesn't even depend on Yarn itself.
- It exposes an API that can be used to programmatically query resolutions without having to parse the file or to load the full resolver. As mentioned, the implementation itself is entirely free as long as both the interface specified in Annex B and the contract listed in Annex C are fulfilled. This should leave enough room for experimentation while providing a consistent API that matches the current expectations of most packages.
- It's an executable script that can act as a resolution daemon that other processes can communicate with through standard input / standard output using the protocol listed in Annex D (based on JSON). This makes it suitable for integration with third-party tools not written in Javascript - this allowed us for example to introduce the `module.resolver` option into Flow. Flow is written in OCaml and cannot use the Javascript API, but thanks to this small bridge, it was only a matter of parsing a few lines of JSON.
- If loaded as a preloaded module (`node -r ./.pnp.js`), it will inject itself into the Node environment and will transparently cause the `require` function to load files directly from the right location. The current implementation overrides `Module._load`, but Node 10 recently released a new API that we plan to use to register into the resolver.
- While not guaranteed strictly speaking (should it?), the `.pnp.js` file implemented by Yarn is stable through the use of relative paths. It means that the file can be moved from a computer to another and will still work as expected (provided that the cache folder on the new environment is both hot and located in the same path relative to the `.pnp.js` file). Through smart uses of the `.yarnrc` argument options, it becomes possible to store both the cache and the `.pnp.js` files together, allowing to skip the installs altogether and making Yarn optional.

workspaces & links

Yarn supports adding persistent symlinks to your projects through two means: the first one, which we recommend, is to use the workspaces feature in order to create automatic links between your packages. The second one, which is a bit older, is to use the `link:` protocol and force Yarn to create a symlink to a location, regardless what's located there.

Both of those will continue working with Plug'n'Play, but will be slightly repurposed: when operating under a Plug'n'Play environment, the links between the packages will be kept inside the `.pnp.js` file, and only there. It means that we won't be creating actual symlinks anymore - we don't need them, since their main goal was to hook into the require process! In case a user needs to access something "through the symlink", they just have to use `require.resolve`, which will query the Plug'n'Play resolution and return the right path.

A third way of adding symlinks also exists in traditional installs: `yarn link`. While it would be possible to implement something similar, we decided not to rush it since it comes with additional issues. In case you need to use yarn link:

- If needed for debugging a project, we recommend to use `yarn unplug`, which will copy a package from your cache into the `.pnp/unplugged` folder. This "unplugged" copy is entirely free for you to alter as you see fit. Once you're done, just run `yarn unplug --purge-all` and the modifications you've made will be forgotten. Note that **this feature is not meant to be used as a permanent trick.**
- If you were using `yarn link` as part of your actual install process, we recommend you to either use the `link:` protocol (but be aware of the issues caused by split dependency trees, cf Section 5.B) or, much better, to port your project to use workspaces (which don't suffer from the issues described in Section 5.B since Yarn is then aware of the whole dependency tree).

Virtual packages

Packages instantiations

As a reminder, a module instance is the representation in memory of this module (it's usually the `module.exports` value exported by this value). Node caches each require call so that if a same module is required multiple times, it will only be instantiated once. The way it does this is by comparing the realpath of the files. Unfortunately, this heuristic doesn't work when a same package is located in multiple different locations - which can happen when a package cannot be hoisted, for example. In this instance, Node will create one separate cache entry for each time the package has been duplicated, increasing the time needed for the application to start and messing with `instanceof` checks.

Plug'n'Play guarantees that each combination of package name / version will only be instantiated once by Node, **except in one documented case**: if a package has any number of peer dependencies, Node will instantiate them exactly once for each time it is found in the dependency tree - and this regardless of whether the packages are strictly identical or not.

So for example if you have `react` and `react-dom` (which has a peer dependency on `react`), a same version of `react` will be guaranteed to only ever be instantiated once by Node (because it doesn't have any peer dependencies), but `react-dom` will be instantiated exactly once for each package that depends on it.

Why does it work this way?

Let's say you have `package-a` and `package-b`. Both of them depending on the same package `child`, which has a peer dependency

on `peer`. Now, imagine that `package-a` depends on `peer@1` while `package-b` depends on `peer@2`. In this instance, the `child` package will have to be instantiated twice in order for us to satisfy the peer dependency contract for both `package-a` and `package-b`. We do this by computing a "unique identifier" (also called a virtual package) for each package instance. Now, how can we standardize the wording for this behavior?

First solution would be to say something similar to: "a package with peer dependencies must have exactly one unique identifier per each set of inherited dependencies". So in our example, since `child` has two sets of inherited dependencies, it would get two unique identifiers, would be instantiated twice, and everything would work. But unfortunately it's not so simple.

Problems arise when you consider circular dependencies. Let's imagine a different scenario: `package` depends on `child-a` and `child-b`. The `child-a` package has a peer dependency on `child-b`, and `child-b` has a peer dependency on `child-a`. In this situation, per the wording described above, we would need to generate the unique identifier for `child-a` based on the set of its dependencies, which includes `child-b`. The problem is that the unique identifier for `child-b` has not been generated yet, so we cannot compute the unique identifier for `child-a`, and vice-versa! The loop cannot be broken.

The solution to this issue is to say that the unique identifier for a package with peer dependencies is based on the unique identifier of its direct parent. Since a package unique identifier is always computed before its children, we cannot have a cyclic dependency.

Install config

Since this proposal is still experimental we decided not to enable Plug'n'Play by default for the time being. As a result, a new key has been added to the `package.json`: `installConfig` (we selected this name to mirror the `publishConfig` settings that already existed). If `installConfig.pnp` contains a truthy value, Plug'n'Play will be used if the current version of Yarn supports it. Otherwise, the regular install will be used:

```
{
  "installConfig": {
    "pnp": true
  }
}
```

Why not store this configuration value within the `.yarnrc`?

> While the `.yarnrc` files would have made a fine candidate, we believe that checking whether a project is Plug'n'Play compatible or not can be extremely interesting for various tools. Listing it into the `package.json` means that virtually any tool can quickly decide whether they want to take advantage of the guarantees provided by Plug'n'Play.

4. Solved Issues

A. INSTALL SPEED NOW REACHES NEW ALL-TIME-HIGHS

Probably the most obvious win is that by keeping the dependencies files stored into the cache, Yarn doesn't have to copy them around anymore. This allows the link step to be skipped almost entirely - the last remaining actions being done only because of how Yarn is currently architectured and will be removed later on (we're still creating a deep tree before flattening it in a later step, which is unnecessary).

B. Installs can now be efficiently cached even on ci

Now that files don't have to be copied anymore, efficient caching becomes a breeze. The actual Yarn cache can be persisted into locations shared between all CI instances, making it possible to skip installs altogether provided both the cache and the `.pnp.js` file are made available.

Where some projects were spending more than two minutes running (like is the case for react-native, for example), Plug'n'Play now allows them to spend this time actually running their tests, decreasing the load on the CIs and making for a better developer experience - we all hate waiting for the tests to end.

C. users working on multiple projects across a system won't pay increasing install costs

A common occurence in the Javascript world is developers working on multiple disconnected projects sharing similar dependencies (for example all projects created through `create-react-app`). Due to how package managers currently work, the files used by those projects were typically copied from the cache into multiple `node_modules`, multiplying both the total size of the installs on the disk and the time wasted running installs.

Now that the files are read directly from the cache, no matter your system, you'll only ever pay the cost of having multiple projects once. This multi-megs project is much more bearable now that you know that its dependencies will be reused by all other projects on your machine now.

D. "perfect" hoisting due to the removal of the filesystem limitations

An example will be worth a thousand words: let's say you have the following dependency tree:

- `top-level`
  - `package-a`
    - `package-c@1.0.0`
  - `package-b`
    - `package-c@1.0.0`
  - `package-c@2.0.0`

In this situation, even though it is found multiple times inside the dependency tree, `package-c@1.0.0` cannot be hoisted in such a way that it can be installing only once. This is because doing so would conflict with the strict requirement of `package-c@2.0.0` declared by the top-level.

Since Plug'n'Play flattens the dependency tree while still preserving the links between the nodes, the paths Node will get will be the same for any `package-c@1.0.0` inside the dependency tree, causing the package to be instantiated a single time.

A. users cannot require dependencies that aren't listed in their dependencies

A common problem was that it was extremely easy for a library author to start relying on a package and forget listing it inside the dependencies. Because these broken dependencies were being pulled by dev dependencies before being hoisted to the top level, they often happened to work fine in development environments and break in production.

This problem isn't possible anymore with Plug'n'Play, because Yarn is aware of the whole dependency tree and can instantly decide whether a resolution request is valid or not for a given package. As an added bonus, it is able to know the exact reason why a require cannot succeed, which further improves the developer experience:

```
Error: You cannot require a package ("foobar") that is not declared in your
dependencies (via "/Users/mael/app/some-script.js")
```

```
Error: Package "foo@0.0.0" (via "/path/to/foo/0.0.0/node_modules/foo/index.js") is
trying to require the package "foobar" (via "foobar") without it being listed in
its dependencies (react, react-dom, foo)
```

F. package instantiations now obey strict & predictable rules

As mentioned in the previous point, it happens that the hoisting may not be applied fully. But the thing is, it often can be. Which makes it impossible for a project to know for sure how many times it will be instantiated by Node, and plan accordingly.

The Plug'n'Play API makes it a goal to provide strong guarantees that library authors can rely on. Anything that would cause the contract to be broken in some circumstances is unacceptable, and as a result cannot be used as guarantee.

> **Peer dependencies: The Return**
>
> This is for example why Plug'n'Play guarantees that *ALL* packages with peer dependencies are instantiated exactly once for each time they are found in the dependency tree: while it would be possible to optimize it some of the packages with peer dependencies in some specific cases, we wouldn't be able to guarantee it and would have to make it an undefined behavior.
>
> Since it's been proven in the past that such undefined behaviors were still leading some libraries to make incorrect assumptions (as happened with packages crossing the `node_modules` boundaries, for example), we deemed it safer to enforce a stricter but entirely predictable and consistent behavior that library authors can rely on.

G. enforcing the boundaries leaves room for different implementations

As detailed in section 6, Plug'n'Play is but the beginning of a long term project. As a result, we worked hard to make sure that the guarantees exposed in section 1 and the APIs detailed in section 3 weren't overly vast and would make it possible for us to change the way the Yarn Plug'n'Play resolver is implemented while still honoring the contract we set up.

Moreover, enforcing correctness will also make it easier for third-parties to write their own resolvers, because they'll know from the get go what are the rules they need to implement. We describe in Section 7.A a generalized testsuite that we wrote to make this work easier.

5. Potential New Issues

A. post install scripts considered harmful

Post-install scripts are likely the biggest technical issue of Plug'n'Play. Since all packages are kept in the cache, build artifacts need to be stored there as well. While it might work for a single project, modifying files into the cache

would still lead to cache corruptions, and would thus be unacceptable: it would cause issues when working on multiple projects sharing the same package with a post-install script, since they would each overwrite the files the others generated (which might be different since they can depend on a dependency of the package, which might be locked to different versions across multiple projects - think about a project using a Node-4-compiled version of node-sass that would conflict with a project using a Node-10-compiled version of this same package).

There's two ways this issue can be solved:

- First we've started to implement a `yarn unplug --persist` command that put specific packages outside of the cache, inside a specific project directory (`pnp-packages`, which wouldn't be too different from `node_modules` except that it would be entirely flat, even when the package names would usually cause a conflict).
- On the long term we believe that post-install scripts are a problem by themselves (they still need an install step, and can lead to security issues), and we would suggest relying less on this feature in the future, possibly even deprecating it in the long term. While native modules have their usefulness, WebAssembly is becoming a more and more serious candidate for a portable bytecode as the months pass.

As a data point, we encountered no problem at Facebook with adding `--ignore-scripts` to all of our Yarn invocations. The two main projects we're aware of that still use postinstall scripts are `fsevents` (which is optional, and whose absence didn't cause us any harm), and node-sass (which is currently working on a WebAssembly port).

B. cross-installs break the model

Plug'n'Play relies on the fact that it knows the whole dependency tree. This causes issues when a package tries to require a file located in another entirely different part of the filesystem.

The current implementation partially solves this by having a fallback on the regular Node resolution when files located outside of the dependency tree make require calls. Unfortunately, this doesn't work well with other projects that have themselves been installed using Plug'n'Play. We think this shouldn't happen under normal circumstances, and as a result have decided it wasn't blocking the proposal.

C. the package manager becomes the hub to run the project

This is more a philosophical issue than a technical one. In order to make it easier for users to work with Plug'n'Play, Yarn recommends calling scripts through `yarn run`, and running Javascript files using `yarn node` (both of which are commands

that have been available for some time now). They both automatically insert the Plug'n'Play hook if needed, making the whole thing transparent to the user. As a downside, it also means that the package manager also becomes the preferred way to run scripts.

The easiest solution would be for Node to implement native support for loading Plug'n'Play if detected. This is obviously not something that can be taken lightly, so this will only become viable once Plug'n'Play will have proven its value.

D. tools relying on crossing package boundaries in order to load their plugins need help

Some packages try to require packages they don't directly depend on for legit reasons. Most of those are trying to do this in order to reference plugins that their users declared in their configuration. Since they don't list those plugins in their dependencies (nor should they have to), Plug'n'Play is supposed to deny them access.

In order to solve this, Plug'n'Play details a special case when a package makes a require call to a package it doesn't own but that the top-level has listed as one of its dependencies. In such a case, the require call will succeed, and the path that will be returned will be the exact same one as the one that would be obtained if the top-level package was making the call.

e. edit-in-place workflows need different tools

A quite common debug pattern is to manually edit the `node_modules` files in order to alter the behavior of the program and debug it more easily (by adding `console.log` statements, for example). Since the `node_modules` folders don't exist anymore, this isn't directly possible anymore.

In order to solve this use case, we've implemented the `yarn unplug` command that can temporarily put a package from your cache into a special folder inside your project. You're then free to edit this local copy as you see fit, then once you're done just run `yarn install` again and it will be removed. This also provides a safety mechanism that avoids you from having to remove your whole `node_modules` hierarchy when you want to revert your changes.

6. Future Work

A. require.resolve

The `require.resolve` function is problematic in that it does two things in one:

- On one hand it returns an identifier that, when passed to `require`, will allow any module to load a file that would typically only be accessible from another module.
- On the other hand it converts an identifier into a path on the filesystem.

The reason this is a problem is that both of those actions don't have the same meaning and as such interfere with each other. The symlinks used for the virtual packages implementation referenced in Section 3 are a direct consequence of this: while it would be possible to implement this concept by making `require.resolve` create and return special in-memory identifiers that `require` would be able to understand, it wouldn't be possible to use those identifiers as paths (unless we were to patch the `fs` module, which is totally unacceptable).

A fix would be to split require.resole in two:

- `require.resolve.virtual`: would convert a request into an implementation-defined object ready for consumption by `require` and `require.resolve.physical`
- `require.resolve.physical`: would convert a request (or one of the values returned by `require.resolve.virtual`) into a filesystem path

B. tarball unpacking

The classic Yarn installs copy files from the cache to the `node_modules` folder. The Plug'n'Play Yarn installs instead ensure that the cache becomes the one and single source of truth. But what if we were to go one step further? What if the Yarn offline mirror (this folder that contains the tarballs of each package found in the project, and used to populate the cache without querying the network) was this one and only source of truth? What if we didn't have to unpack them anymore?

Plug'n'Play makes this easy: since the Plug'n'Play implementation is fluid (as long as the guarantees listed above are met), it becomes possible to implement it from various different way. One of these ways could be to return an opaque object that would contain an combination of an archive path and a file path, which `require` would then be able to interpret in order to load the file from the given archive on demand!

Deployments would then only have to ship the `.pnp.js` file along with the offline mirror and the project could then be run from anywhere without needing any extra install.

7. Annexes

A. generalized testsuite

In order to make it easier for everyone to experiment with different Plug'n'Play implementations, we've written a generalized testsuite that can be found on the Yarn repository. It contains acceptance tests that validate the high-level behavior of the feature, and aren't tied to any specific implementation detail. While it's still a work-in-progress, we've invested in it a lot and hope it'll prove valuable to the community.

Adapting the testsuite to a different manager is a matter of implementing an adapter matching the package manager you want to test, then enabling the specs you want to validate. You can take a look at the Yarn adapter for an example on how we implemented this for our package manager.

B. Plug'n'Play api

```
interface {

  VERSIONS: {std: 1, [extension]: number};

  findPackageLocator(path: string): {name: null, reference: null};
  findPackageLocator(path: string): {name: string, reference: string};

  getPackageInformation({name, reference}: {name: null, reference: null}): {packageLocation:
  getPackageInformation({name, reference}: {name: string, reference: string}): {packageLocat

  resolveToUnqualified(request: string, issuer: string): string;
  resolveUnqualified(unqualified: string, {extensions?: Array<string>}): string;
  resolveRequest(request: string, issuer: string): string;

  setup(void): void;

}
```

C. formal plug'n'play guarantees

- A package **MUST** be able to get the value exported by the main entries of its dependencies using the `require` function, or through an import statement.
- A package **MUST** be able to get the filesystem path to a file stored in one of its dependencies by using the `require.resolve` function.

- A package listing a peer dependency **MUST** obtain the exact same instance of this peer dependency when using `require` than its immediate parent in the dependency tree would. This process is applied recursively.
- A package **MUST NOT** be able to require a package that isn't listed in its dependency detail. The dependency detail is the sum of `dependencies` and `peerDependencies` for all packages, plus `devDependencies` for the top level package if running in development mode.
  - An exception is made if the package being required is listed in the dependency detail of the top-level. In this case, the package making the request will obtain the exact same **instance** than if the top-level package had made the require call (note the emphasis on instance rather than version).
  - We however discourage packages from relying on this exception, since it's only been implemented to lower the adoption cost and help plugin systems. Packages should prefer using `peerDependencies` if applicable.
- Two packages depending on the same reference of the same dependency that itself has a transitive peer dependency **MUST** get the exact same instance of this dependency, whatever their locations in the dependency tree are.
- Two packages depending on the same reference of the same dependency that itself has a transitive peer dependency **MUST** get different instances of this dependency.

The comprehensive list of guarantees can also be extracted from the "it should" statements that can be found on the Plug'n'Play test suite.

D. daemon-mode communications

The Plug'n'Play daemon communicates with the outside world by the mean of a very simple stdin / stdout loop. It doesn't spawn a server. The protocol is quite simple. In its most basic form, it's the following:

```
> JSON [request: string, issuer: string]
< JSON [error: ?{code: string, message: string, data: Object}, resolution: string]
```

Note that any execution is synchronous (multiple requests cannot be handled simultaneously), but the daemon can be spawn multiple times and pooled for a similar effect (there's no lock). Well behaving applications should watch for the resolver being modified, and act accordingly when they detect changes (such as clearing the resolution cache and restarting the daemon processes).