Using npm Packages

Searching for packages

You can use the official search at npmjs.com or see results sorted by package quality (code quality, maintenance status, development velocity, popularity etc.) at npms.io. There are also sites that search certain types of packages, like js.coach's React and React Native sections.

npm on the client

Tools like browserify and webpack are designed to provide a Node-like environment on the client so that many npm packages, even ones originally intended for the server, can run unmodified. In most cases, you can import npm dependencies from a client file, just as you would on the server.

When creating a new application Meteor installs the `meteor-node-stubs` npm package to help provide this client browser compatibility. If you are upgrading an application to Meteor 1.3 you may have to run `meteor npm install --save meteor-node-stubs` manually.

The `meteor-node-stubs` npm package provides browser-friendly implementations of Node's built-in modules, like `path`, `buffer`, `util`, etc. Meteor's module system avoids actually bundling any stub modules (and their dependencies) if they are not used, so there is no cost to keeping `meteor-node-stubs` in the dependencies. In other words, leave `meteor-node-stubs` installed unless you really know what you're doing.

Installing npm packages

npm packages are configured in a `package.json` file at the root of your project. If you create a new Meteor project, you will have such a file created for you. If not you can run `meteor npm init` to create one.

To install a package into your app you run the `npm install` command with the `--save` flag:

```
meteor npm install --save moment
```

This will both update your `package.json` with information about the dependency and download the package into your app's local `node_modules` directory. Typically, you don't check the `node_modules` directory into source control and

your teammates run `meteor npm install` to get up to date when dependencies change:

```
meteor npm install
```

If the package is just a development dependency (i.e. it's used for testing, linting or the like) then you should use `--save-dev`. That way if you have some kind of build script, it can do `npm install --production` and avoid installing packages it doesn't need.

For more information about `npm install`, check out the official documentation.

> Meteor comes with npm bundled so that you can type `meteor npm` without worrying about installing it yourself. If you like, you can also use a globally installed npm to manage your packages.

Using npm packages

To use an npm package from a file in your application you `import` the name of the package:

```
import moment from 'moment';

// this is equivalent to the standard node require:
const moment = require('moment');
```

This imports the default export from the package into the symbol `moment`.

You can also import specific functions from a package using the destructuring syntax:

```
import { isArray } from 'lodash';
```

You can also import other files or JS entry points from a package:

```
import { parse } from 'graphql/language';
```

Some Meteor apps contain local Meteor packages (packages defined in the `packages/` directory of your app tree); this was an older recommendation from before Meteor had full ECMAScript support. If your app is laid out this way, you can also `require` or `import` npm packages installed in your app from within your local Meteor packages.

Importing styles from npm

Using any of Meteor's supported CSS pre-processors you can import other style files provided by an NPM into your application using both relative and absolute paths. However, this will only work for the top-level app and will not work inside an Atmosphere package.

Importing styles from an npm package with an absolute path using the `{}` syntax, for instance with Less:

```
@import '{}/node_modules/npm-package-name/button.less';
```

Importing styles from an npm package with a relative path:

```
@import '../../node_modules/npm-package-name/colors.less';
```

You can also import CSS directly from a JavaScript file to control load order if you have the `ecmascript` package installed:

```
import 'npm-package-name/stylesheets/styles.css';
```

> When importing CSS from a JavaScript file, that CSS is not bundled with the rest of the CSS processed with the Meteor build tool, but instead is put in your app's `<head>` tag inside `<style>...</style>` after the main concatenated CSS file.

Building with other assets from npm

Meteor also supports building other assets into your app, such as fonts, that are located in your `node_modules` directory by symbolic linking to those assets from either the `/public` or `/private` directories. For example, `font-awesome` is a very popular font library that provides lots of font-based icons. New icons appear frequently as the library is developed and it would be difficult to manage all the updates if you were to copy the entire `font-awesome` code base to your own app and git repository. Instead use the following to include these fonts:

```
cd /public
ln -ls ../node_modules/font-awesome/fonts ./fonts
```

Any assets made available via symlinks in the `/public` and `/private` directories of an application will be copied into the Meteor application bundles when using the `meteor build` command.

Recompiling npm packages

Meteor does not recompile packages installed in your `node_modules` by default. However, compilation of specific npm packages (for example, to support older browsers that the package author neglected), can be achieved through the `meteor.nodeModules.recompile` configuration object in your `package.json` file.

For example:

```
{
  "name": "your-application",
  ...
  "meteor": {
    "mainModule": ...,
    "testModule": ...,
    "nodeModules": {
      "recompile": {
        "very-modern-package": ["client", "server"],
        "alternate-notation-for-client-and-server": true,
        "somewhat-modern-package": "legacy",
```

```
        "another-package": ["legacy", "server"]
      }
    }
  }
}
```

The keys of the `meteor.nodeModules.recompile` configuration object are npm package names, and the values specify for which bundles those packages should be recompiled using the Meteor compiler plugins system, as if the packages were part of the Meteor application.

For example, if an npm package uses const/let syntax or arrow functions, that's fine for modern and server code, but you would probably want to recompile the package when building the legacy bundle. To accomplish this, specify `"legacy"` or `["legacy"]` as the value of the package's property, similar to `somewhat-modern-package` above. These strings and arrays of strings have the same meaning as the second argument to `api.addFiles(files, where)` in a `package.js` file.

This configuration serves pretty much the same purpose as symlinking an application directory into `node_modules/`, but without any symlinking: https://forums.meteor.com/t/litelement-import-litelement-html/45042/8?u=benjamn

The `meteor.nodeModules.recompile` configuration currently applies to the application `node_modules/` directory only (not to `Npm.depends` dependencies in Meteor packages). Recompiled packages must be direct dependencies of the application.

Meteor will compile the exposed code as if it was part of your application, using whatever compiler plugins you have installed. You can influence this compilation using `.babelrc` files or any other techniques you would normally use to configure compilation of application code.

npm Shrinkwrap

`package.json` typically encodes a version range, and so each `npm install` command can sometimes lead to a different result if new versions have been published in the meantime. In order to ensure that you and the rest of your team are using the same exact same version of each package, it's a good idea to use `npm shrinkwrap` after making any dependency changes to `package.json`:

```
# after installing
meteor npm install --save moment
meteor npm shrinkwrap
```

This will create an `npm-shrinkwrap.json` file containing the exact versions of each dependency, and you should check this file into source control. For even more precision (the contents of a given version of a package *can* change), and

to avoid a reliance on the npm server during deployment, you should consider using `npm shrinkpack`.

Asyncronous callbacks

Many npm packages rely on an asynchronous, callback or promise-based coding style. For several reasons, Meteor is currently built around a synchronous-looking but still non-blocking style using Fibers.

The global Meteor server context and every method and publication initialize a new fiber so that they can run concurrently. Many Meteor APIs, for example collections, rely on running inside a fiber. They also rely on an internal Meteor mechanism that tracks server "environment" state, like the currently executing method. This means you need to initialize your own fiber and environment to use asynchronous Node code inside a Meteor app. Let's look at an example of some code that won't work, using the code example from the node-github repository:

```
// Inside a Meteor method definition
updateGitHubFollowers() {
  github.user.getFollowingFromUser({
    user: 'stubailo'
  }, (err, res) => {
    // Using a collection here will throw an error
    // because the asynchronous code is not in a fiber
    Followers.insert(res);
  });
}
```

Let's look at a few ways to resolve this issue.

`Meteor.bindEnvironment`

In most cases, wrapping the callback in `Meteor.bindEnvironment` will do the trick. This function both wraps the callback in a fiber, and does some work to maintain Meteor's server-side environment tracking. Here's the same code with `Meteor.bindEnvironment`:

```
// Inside a Meteor method definition
updateGitHubFollowers() {
  github.user.getFollowingFromUser({
    user: 'stubailo'
  }, Meteor.bindEnvironment((err, res) => {
    // Everything is good now
    Followers.insert(res);
  }));
}
```

However, this won't work in all cases - since the code runs asynchronously, we can't use anything we got from an API in the method return value. We need a

different approach that will convert the async API to a synchronous-looking one that will allow us to return a value.

`Meteor.wrapAsync`

Many npm packages adopt the convention of taking a callback that accepts `(err, res)` arguments. If your asynchronous function fits this description, like the one above, you can use `Meteor.wrapAsync` to convert to a fiberized API that uses return values and exceptions instead of callbacks, like so:

```
// Setup sync API
const getFollowingFromUserFiber =
  Meteor.wrapAsync(github.user.getFollowingFromUser, github.user);

// Inside a Meteor method definition
updateGitHubFollowers() {
  const res = getFollowingFromUserFiber({
    user: 'stubailo'
  });

  Followers.insert(res);

  // Return how many followers we have
  return res.length;
}
```

If you wanted to refactor this and create a completely fiber-wrapper GitHub client, you could write some logic to loop over all of the methods available and call `Meteor.wrapAsync` on them, creating a new object with the same shape but with a more Meteor-compatible API.

Promises

Recently, a lot of npm packages have been moving to Promises instead of callbacks for their API. This means you actually get a return value from the asynchronous function, but it's just an empty shell where the real value is filled in later.

The good news is that Promises can be used with the new ES2015 `async/await` syntax (available in the `ecmascript` package since Meteor 1.3) in a natural and synchronous-looking style on both the client and the server.

If you declare your function `async` (which ends up meaning it returns a Promise itself), then you can use the `await` keyword to wait on other promise inside. This makes it very easy to serially call Promise-based libraries:

```
async function sendTextMessage(user) {
  const toNumber = await phoneLookup.findFromEmail(user.emails[0].address);
  return await client.sendMessage({
    to: toNumber,
    from: '+14506667788',
```

```
    body: 'Hello world!'
  });
}
```

Shrinkpack

Shrinkpack is a tool that gives you more bulletproof and repeatable builds than you get by using `npm shrinkwrap` alone.

Essentially it copies a tarball of the contents of each of your npm dependencies into your application source repository. This is essentially a more robust version of the `npm-shrinkwrap.json` file that shrinkwrap creates, because it means your application's npm dependencies can be assembled without the need or reliance on the npm servers being available or reliable. This is good for repeatable builds especially when deploying.

To use shrinkpack, first globally install it:

```
npm install -g shrinkpack
```

Then use it directly after you shrinkwrap

```
meteor npm install --save moment
meteor npm shrinkwrap
shrinkpack
```

You should then check the generated `node_shrinkwrap/` directory into source control, but ensure it is ignored by your text editor.

> NOTE: Although this is a good idea for projects with a lot of npm dependencies, it will not affect Atmosphere dependencies, even if they themselves have direct npm dependencies.