modules

This document explains the usage and key features of the module system used by Meteor.

> Meteor 1.2 introduced support for many new ECMAScript 2015 features, one of the most notable omissions was ES2015 `import` and `export` syntax.
>
> Meteor 1.3 filled the gap with a fully standards-compliant module system that works on both the client and the server.
>
> Meteor 1.7 introduced `meteor.mainModule` and `meteor.testModule` to `package.json` so Meteor doesn't need special folders anymore for js resources. Also doesn't need to eager load js resources.

By design, `meteor.mainModule` only affect js resources. For non-js resources, there are still some things that can only be done within imports:

- only stylesheets within imports can be dynamically imported
- you can only control the load order of stylesheets by importing them in js if the stylesheets are within imports

Any non-js resource outside of imports (and some other special folders) are still eagerly loaded.

> You can read more about these differences in this comment.

Enabling modules

It is installed by default for all new apps and packages. Nevertheless, the `modules` package is totally optional.

If you want to add it to existent apps or packages:

For apps, this is as easy as `meteor add modules`, or (even better) `meteor add ecmascript`, since the `ecmascript` package *implies* the `modules` package.

For packages, you can enable `modules` by adding `api.use('modules')` to the `Package.onUse` or `Package.onTest` sections of your `package.js` file.

Now, you might be wondering what good the `modules` package is without the `ecmascript` package, since `ecmascript` enables `import` and `export` syntax. By

itself, the `modules` package provides the CommonJS `require` and `exports` primitives that may be familiar if you've ever written Node code, and the `ecmascript` package simply compiles `import` and `export` statements to CommonJS. The `require` and `export` primitives also allow Node modules to run within Meteor application code without modification. Furthermore, keeping `modules` separate allows us to use `require` and `exports` in places where using `ecmascript` is tricky, such as the implementation of the `ecmascript` package itself.

While the `modules` package is useful by itself, we very much encourage using the `ecmascript` package (and thus `import` and `export`) instead of using `require` and `exports` directly. If you need convincing, here's a presentation that explains the differences.

Basic syntax

ES2015

Although there are a number of different variations of `import` and `export` syntax, this section describes the essential forms that everyone should know.

First, you can `export` any named declaration on the same line where it was declared:

```
// exporter.js
export var a = ...;
export let b = ...;
export const c = ...;
export function d() { ... }
export function* e() { ... }
export class F { ... }
```

These declarations make the variables `a`, `b`, `c` (and so on) available not only within the scope of the `exporter.js` module, but also to other modules that `import` from `exporter.js`.

If you prefer, you can `export` variables by name, rather than prefixing their declarations with the `export` keyword:

```
// exporter.js
function g() { ... }
let h = g();

// At the end of the file
export { g, h };
```

All of these exports are *named*, which means other modules can import them using those names:

```
// importer.js
import { a, c, F, h } from './exporter';
```

```
new F(a, c).method(h);
```

If you'd rather use different names, you'll be glad to know `export` and `import` statements can rename their arguments:

```
// exporter.js
export { g as x };
g(); // Same as calling `y()` in importer.js

// importer.js
import { x as y } from './exporter';
y(); // Same as calling `g()` in exporter.js
```

As with CommonJS `module.exports`, it is possible to define a single *default* export:

```
// exporter.js
export default any.arbitrary(expression);
```

This default export may then be imported without curly braces, using any name the importing module chooses:

```
// importer.js
import Value from './exporter';
// Value is identical to the exported expression
```

Unlike CommonJS `module.exports`, the use of default exports does not prevent the simultaneous use of named exports. Here is how you can combine them:

```
// importer.js
import Value, { a, F } from './exporter';
```

In fact, the default export is conceptually just another named export whose name happens to be "default":

```
// importer.js
import { default as Value, a, F } from './exporter';
```

These examples should get you started with `import` and `export` syntax. For further reading, here is a very detailed explanation by Axel Rauschmayer of every variation of `import` and `export` syntax.

CommonJS

You don't need to use the `ecmascript` package or ES2015 syntax in order to use modules. Just like Node.js in the pre-ES2015 days, you can use `require` and `module.exports`—that's what the `import` and `export` statements are compiling into, anyway.

ES2015 `import` lines like these:

```
import { AccountsTemplates } from 'meteor/useraccounts:core';
import '../imports/startup/client/routes.js';
```

can be written with CommonJS like this:

```
var UserAccountsCore = require('meteor/useraccounts:core');
require('../imports/startup/client/routes.js');
```

and you can access `AccountsTemplates` via `UserAccountsCore.AccountsTemplates`.

Note that files don't need a `module.exports` if they're required like `routes.js` is in this example, without assignment to any variable. The code in `routes.js` will simply be included and executed in place of the above `require` statement.

ES2015 `export` statements like these:

```
export const insert = new ValidatedMethod({ ... });
export default incompleteCountDenormalizer;
```

can be rewritten to use CommonJS `module.exports`:

```
module.exports.insert = new ValidatedMethod({ ... });
module.exports.default = incompleteCountDenormalizer;
```

You can also simply write `exports` instead of `module.exports` if you prefer. If you need to `require` from an ES2015 module with a `default` export, you can access the export with `require('package').default`.

There is a case where you might *need* to use CommonJS, even if your project has the `ecmascript` package: if you want to conditionally include a module. `import` statements must be at top-level scope, so they cannot be within an `if` block. If you're writing a common file, loaded on both client and server, you might want to import a module in only one or the other environment:

```
if (Meteor.isClient) {
  require('./client-only-file.js');
}
```

Note that dynamic calls to `require()` (where the name being required can change at runtime) cannot be analyzed correctly and may result in broken client bundles. This is also discussed in the guide.

CoffeeScript

CoffeeScript has been a first-class supported language since Meteor's early days. Even though today we recommend ES2015, we still intend to support CoffeeScript fully.

As of CoffeeScript 1.11.0, CoffeeScript supports `import` and `export` statements natively. Make sure you are using the latest version of the CoffeeScript package in your project to get this support. New projects created today will get this version with `meteor add coffeescript`. Make sure you don't forget to include the `ecmascript` and `modules` packages: `meteor add ecmascript`. (The `modules` package is implied by `ecmascript`.)

CoffeeScript `import` syntax is nearly identical to the ES2015 syntax you see above:

```
import { Meteor } from 'meteor/meteor'
import SimpleSchema from 'simpl-schema'
import { Lists } from './lists.coffee'
```

You can also use traditional CommonJS syntax with CoffeeScript.

Modular application structure

Use in your application `package.json` file the section `meteor`.

> This is available since Meteor 1.7

```
{
  "meteor": {
    "mainModule": {
      "client": "client/main.js",
      "server": "server/main.js"
    }
  }
}
```

When specified, these entry points will define in which files Meteor is going to start the evaluation process for each architecture (client and server).

This way Meteor is not going to eager load any other js files.

There is also an architecture for the `legacy` client, which is useful if you want to load polyfills or other code for old browsers before importing the main module for the modern client.

In addition to `meteor.mainModule`, the `meteor` section of `package.json` may also specify `meteor.testModule` to control which test modules are loaded by `meteor test` or `meteor test --full-app`:

```
{
  "meteor": {
    "mainModule": {
      "client": "client/main.js",
      "server": "server/main.js"
    },
    "testModule": "tests.js"
  }
}
```

If your client and server test files are different, you can expand the testModule configuration using the same syntax as mainModule:

```
{
  "meteor": {
    "mainModule": {
      "client": "client/main.js",
      "server": "server/main.js"
    },
    "testModule": {
      "client": "client/tests.js",
      "server": "server/tests.js"
    }
  }
}
```

The same test module will be loaded whether or not you use the `--full-app` option.

Any tests that need to detect `--full-app` should check `Meteor.isAppTest`.

The module(s) specified by `meteor.testModule` can import other test modules at runtime, so you can still distribute test files across your codebase; just make sure you import the ones you want to run.

To disable eager loading of modules on a given architecture, simply provide a mainModule value of false:

```
{
  "meteor": {
    "mainModule": {
      "client": false,
      "server": "server/main.js"
    }
  }
}
```

Historic behind Modular application structure

If you want to understand how Meteor works without `meteor.mainModule` on `package.json` keep reading this section, but we don't recommend this approach anymore.

Before the release of Meteor 1.3, the only way to share values between files in an application was to assign them to global variables or communicate through shared variables like `Session` (variables which, while not technically global, sure do feel syntactically identical to global variables). With the introduction of modules, one module can refer precisely to the exports of any other specific module, so global variables are no longer necessary.

If you are familiar with modules in Node, you might expect modules not to be evaluated until the first time you import them. However, because earlier versions

of Meteor evaluated all of your code when the application started, and we care about backwards compatibility, eager evaluation is still the default behavior.

If you would like a module to be evaluated *lazily* (in other words: on demand, the first time you import it, just like Node does it), then you should put that module in an `imports/` directory (anywhere in your app, not just the root directory), and include that directory when you import the module: `import {stuff} from './imports/lazy'`. Note: files contained by `node_modules/` directories will also be evaluated lazily (more on that below).

Modular package structure

If you are a package author, in addition to putting `api.use('modules')` or `api.use('ecmascript')` in the `Package.onUse` section of your `package.js` file, you can also use a new API called `api.mainModule` to specify the main entry point for your package:

```
Package.describe({
  name: 'my-modular-package'
});

Npm.depends({
  moment: '2.10.6'
});

Package.onUse((api) => {
  api.use('modules');
  api.mainModule('server.js', 'server');
  api.mainModule('client.js', 'client');
  api.export('Foo');
});
```

Now `server.js` and `client.js` can import other files from the package source directory, even if those files have not been added using the `api.addFiles` function.

When you use `api.mainModule`, the exports of the main module are exposed globally as `Package['my-modular-package']`, along with any symbols exported by `api.export`, and thus become available to any code that imports the package. In other words, the main module gets to decide what value of `Foo` will be exported by `api.export`, as well as providing other properties that can be explicitly imported from the package:

```
// In an application that uses 'my-modular-package':
import { Foo as ExplicitFoo, bar } from 'meteor/my-modular-package';
console.log(Foo); // Auto-imported because of `api.export`.
console.log(ExplicitFoo); // Explicitly imported, but identical to `Foo`.
console.log(bar); // Exported by server.js or client.js, but not auto-imported.
```

Note that the `import` is `from 'meteor/my-modular-package'`, not `from 'my-modular-package'`. Meteor package identifier strings must include the prefix `meteor/...` to disambiguate them from npm packages.

Finally, since this package is using the new `modules` package, and the package `Npm.depends` on the "moment" npm package, modules within the package can `import moment from 'moment'` on both the client and the server. This is great news, because previous versions of Meteor allowed npm imports only on the server, via `Npm.require`.

Lazy loading modules from a package

Packages can also specify a *lazy* main module:

```
Package.onUse(function (api) {
  api.mainModule("client.js", "client", { lazy: true });
});
```

This means the `client.js` module will not be evaluated during app startup unless/until another module imports it, and will not even be included in the client bundle if no importing code is found.

To import a method named `exportedPackageMethod`, simply:

```
import { exportedPackageMethod } from "meteor/<package name>";
```

> Note: Packages with `lazy` main modules cannot use `api.export` to export global symbols to other packages/apps. Also, prior to Meteor 1.4.4.2 it is neccessary to explicitly name the file containing the module: `import "meteor/<package name>/client.js"`.

Local `node_modules`

Before Meteor 1.3, the contents of `node_modules` directories in Meteor application code were completely ignored. When you enable `modules`, those useless `node_modules` directories suddenly become infinitely more useful:

```
meteor create modular-app
cd modular-app
mkdir node_modules
npm install moment
echo "import moment from 'moment';" >> modular-app.js
echo 'console.log(moment().calendar());' >> modular-app.js
meteor
```

When you run this app, the `moment` library will be imported on both the client and the server, and both consoles will log output similar to: `Today at 7:51 PM`. Our hope is that the possibility of installing Node modules directly within an app will reduce the need for npm wrapper packages such as https://atmospherejs.com/momentjs/moment.

A version of the `npm` command comes bundled with every Meteor installation, and (as of Meteor 1.3) it's quite easy to use: `meteor npm ...` is synonymous with `npm ...`, so `meteor npm install moment` will work in the example above. (Likewise, if you don't have a version of `node` installed, or you want to be sure you're using the exact same version of `node` that Meteor uses, `meteor node ...` is a convenient shortcut.) That said, you can use any version of `npm` that you happen to have available. Meteor's module system only cares about the files installed by `npm`, not the details of how `npm` installs those files.

File load order

Before Meteor 1.3, the order in which application files were evaluated was dictated by a set of rules described in the Application Structure - Default file load order section of the Meteor Guide. These rules could become frustrating when one file depended on a variable defined by another file, particularly when the first file was evaluated after the second file.

Thanks to modules, any load-order dependency you might imagine can be resolved by adding an `import` statement. So if `a.js` loads before `b.js` because of their file names, but `a.js` needs something defined by `b.js`, then `a.js` can simply `import` that value from `b.js`:

```
// a.js
import { bThing } from './b';
console.log(bThing, 'in a.js');
```

```
// b.js
export var bThing = 'a thing defined in b.js';
console.log(bThing, 'in b.js');
```

Sometimes a module doesn't actually need to import anything from another module, but you still want to be sure the other module gets evaluated first. In such situations, you can use an even simpler `import` syntax:

```
// c.js
import './a';
console.log('in c.js');
```

No matter which of these modules is imported first, the order of the `console.log` calls will always be:

```
console.log(bThing, 'in b.js');
console.log(bThing, 'in a.js');
console.log('in c.js');
```