After reading this article, you'll know:

1. How a Meteor application compares to other types of applications in terms of file structure.
2. How to organize your application both for small and larger applications.
3. How to format your code and name the parts of your application in consistent and maintainable ways.

Universal JavaScript

Meteor is a *full-stack* framework for building JavaScript applications. This means Meteor applications differ from most applications in that they include code that runs on the client, inside a web browser or Cordova mobile app, code that runs on the server, inside a Node.js container, and *common* code that runs in both environments. The Meteor build tool allows you to specify what JavaScript code, including any supporting UI templates, CSS rules, and static assets, to run in each environment using a combination of ES2015 `import` and `export` and the Meteor build system default file load order rules.

ES2015 modules

As of version 1.3, Meteor ships with full support for ES2015 modules. The ES2015 module standard is the replacement for CommonJS and AMD, which are commonly used JavaScript module format and loading systems.

In ES2015, you can make variables available outside a file using the `export` keyword. To use the variables somewhere else, you must `import` them using the path to the source. Files that export some variables are called "modules", because they represent a unit of reusable code. Explicitly importing the modules and packages you use helps you write your code in a modular way, avoiding the introduction of global symbols and "action at a distance".

Since this is a new feature introduced in Meteor 1.3, you will find a lot of code online that uses the older, more centralized conventions built around packages and apps declaring global symbols. This old system still works, so to opt-in to the new module system, code must be placed inside the `imports/` directory in your application. We expect a future release of Meteor will turn on modules by default for all code, because this is more aligned with how developers in the wider JavaScript community write their code.

You can read about the module system in detail in the `modules` package README. This package is automatically included in every new Meteor app as part of the `ecmascript` meta-package, so most apps won't need to do anything to start using modules right away.

Introduction to using `import` and `export`

Meteor allows you to `import` not only JavaScript in your application, but also CSS and HTML to control load order:

```
import '../../api/lists/methods.js';  // import from relative path
import '/imports/startup/client';     // import module with index.js from absolute path
import './loading.html';              // import Blaze compiled HTML from relative path
import '/imports/ui/style.css';       // import CSS from absolute path
```

> *For more ways to import styles, see the Build System article.*

Meteor also supports the standard ES2015 modules `export` syntax:

```
export const listRenderHold = LaunchScreen.hold();  // named export
export { Todos };                                   // named export
export default Lists;                          // default export
export default new Collection('lists');        // default export
```

Importing from packages

In Meteor, it is also simple and straightforward to use the `import` syntax to load npm packages on the client or server and access the package's exported symbols as you would with any other module. You can also import from Meteor Atmosphere packages, but the import path must be prefixed with `meteor/` to avoid conflict with the npm package namespace. For example, to import `moment` from npm and `HTTP` from Atmosphere:

```
import moment from 'moment';          // default import from npm
import { HTTP } from 'meteor/http';   // named import from Atmosphere
```

For more details using `imports` with packages see [Using Packages]() in the Meteor Guide.

Using `require`

In Meteor, `import` statements compile to CommonJS `require` syntax. However, as a convention we encourage you to use `import`.

With that said, in some situations you may need to call out to `require` directly. One notable example is when requiring client or server-only code from a common file. As `import`s must be at the top-level scope, you may not place them within an `if` statement, so you'll need to write code like:

```
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.

If you need to `require` from an ES2015 module with a `default` export, you can grab the export with `require("package").default`.

Using CoffeeScript

See the Docs: [Modules » Syntax » CoffeeScript]()

```
// lists.coffee

export Lists = new Collection 'lists'
```

```
import { Lists } from './lists.coffee'
```

File structure

To fully use the module system and ensure that our code only runs when we ask it to, we recommend that all of your application code should be placed inside the `imports/` directory. This means that the Meteor build system will only bundle and include that file if it is referenced from another file using an `import` (also called "lazy evaluation or loading").

Meteor will load all files outside of any directory named `imports/` in the application using the [default file load order](#) rules (also called "eager evaluation or loading"). It is recommended that you create exactly two eagerly loaded files, `client/main.js` and `server/main.js`, in order to define explicit entry points for both the client and the server. Meteor ensures that any file in any directory named `server/` will only be available on the server, and likewise for files in any directory named `client/`. This also precludes trying to `import` a file to be used on the server from any directory named `client/` even if it is nested in an `imports/` directory and vice versa for importing client files from `server/`.

These `main.js` files won't do anything themselves, but they should import some *startup* modules which will run immediately, on client and server respectively, when the app loads. These modules should do any configuration necessary for the packages you are using in your app, and import the rest of your app's code.

Example directory layout

To start, let's look at our [Todos example application](#), which is a great example to follow when structuring your app. Below is an overview of its directory structure. You can generate a new app with this structure using the command `meteor create appName --full`.

```
imports/
  startup/
    client/
      index.js                 # import client startup through a single index entry
point
      routes.js                # set up all routes in the app
      useraccounts-configuration.js # configure login templates
    server/
      fixtures.js              # fill the DB with example data on startup
      index.js                 # import server startup through a single index entry
point

  api/
    lists/                     # a unit of domain logic
      server/
        publications.js        # all list-related publications
        publications.tests.js  # tests for the list publications
      lists.js                 # definition of the Lists collection
      lists.tests.js           # tests for the behavior of that collection
      methods.js               # methods related to lists
      methods.tests.js         # tests for those methods

  ui/
    components/                # all reusable components in the application
                               # can be split by domain if there are many
    layouts/                   # wrapper components for behaviour and visuals
    pages/                     # entry points for rendering used by the router

client/
  main.js                      # client entry point, imports all client code
```

```
server/
  main.js                      # server entry point, imports all server code
```

Structuring imports

Now that we have placed all files inside the `imports/` directory, let's think about how best to organize our code using modules. It makes sense to put all code that runs when your app starts in an `imports/startup` directory. Another good idea is splitting data and business logic from UI rendering code. We suggest using directories called `imports/api` and `imports/ui` for this logical split.

Within the `imports/api` directory, it's sensible to split the code into directories based on the domain that the code is providing an API for --- typically this corresponds to the collections you've defined in your app. For instance in the Todos example app, we have the `imports/api/lists` and `imports/api/todos` domains. Inside each directory we define the collections, publications and methods used to manipulate the relevant domain data.

> *Note: in a larger application, given that the todos themselves are a part of a list, it might make sense to group both of these domains into a single larger "list" module. The Todos example is small enough that we need to separate these only to demonstrate modularity.*

Within the `imports/ui` directory it typically makes sense to group files into directories based on the type of UI side code they define, i.e. top level `pages`, wrapping `layouts`, or reusable `components`.

For each module defined above, it makes sense to co-locate the various auxiliary files with the base JavaScript file. For instance, a Blaze UI component should have its template HTML, JavaScript logic, and CSS rules in the same directory. A JavaScript module with some business logic should be co-located with the unit tests for that module.

Startup files

Some of your code isn't going to be a unit of business logic or UI, it's some setup or configuration code that needs to run in the context of the app when it starts up. In the Todos example app, the `imports/startup/client/useraccounts-configuration.js` file configures the `useraccounts` login templates (see the Accounts article for more information about `useraccounts`). The `imports/startup/client/routes.js` configures all of the routes and then imports *all* other code that is required on the client:

```
import { FlowRouter } from 'meteor/ostrio:flow-router-extra';
import { BlazeLayout } from 'meteor/kadira:blaze-layout';
import { AccountsTemplates } from 'meteor/useraccounts:core';

// Import to load these templates
import '../../ui/layouts/app-body.js';
import '../../ui/pages/root-redirector.js';
import '../../ui/pages/lists-show-page.js';
import '../../ui/pages/app-not-found.js';

// Import to override accounts templates
import '../../ui/accounts/accounts-templates.js';

// Below here are the route definitions
```

We then import both of these files in `imports/startup/client/index.js` :

```
import './useraccounts-configuration.js';
import './routes.js';
```

This makes it easy to then import all the client startup code with a single import as a module from our main eagerly loaded client entry point `client/main.js` :

```
import '/imports/startup/client';
```

On the server, we use the same technique of importing all the startup code in `imports/startup/server/index.js` :

```
// This defines a starting set of data to be loaded if the app is loaded with an
empty db.
import '../imports/startup/server/fixtures.js';

// This file configures the Accounts package to define the UI of the reset password
email.
import '../imports/startup/server/reset-password-email.js';

// Set up some rate limiting and other important security settings.
import '../imports/startup/server/security.js';

// This defines all the collections, publications and methods that the application
provides
// as an API to the client.
import '../imports/api/api.js';
```

Our main server entry point `server/main.js` then imports this startup module. You can see that here we don't actually import any variables from these files - we import them so that they execute in this order.

Importing Meteor "pseudo-globals"

For backwards compatibility Meteor 1.3 still provides Meteor's global namespacing for the Meteor core package as well as for other Meteor packages you include in your application. You can also still directly call functions such as `Meteor.publish` , as in previous versions of Meteor, without first importing them. However, it is recommended best practice that you first load all the Meteor "pseudo-globals" using the `import { Name } from 'meteor/package'` syntax before using them. For instance:

```
import { Meteor } from 'meteor/meteor';
import { EJSON } from 'meteor/ejson';
```

Default file load order

Even though it is recommended that you write your application to use ES2015 modules and the `imports/` directory, Meteor 1.3 continues to support eager evaluation of files, using these default load order rules, to provide backwards compatibility with applications written for Meteor 1.2 and earlier. For a description of the difference between eager evaluation, lazy evaluation, and lazy loading, please see this Stack Overflow article.

You may combine both eager evaluation and lazy loading using `import` in a single app. Any import statements are evaluated in the order they are listed in a file when that file is loaded and evaluated using these rules.

There are several load order rules. They are *applied sequentially* to all applicable files in the application, in the priority given below:

1. HTML template files are **always** loaded before everything else
2. Files beginning with `main.` are loaded **last**
3. Files inside **any** `lib/` directory are loaded next
4. Files with deeper paths are loaded next
5. Files are then loaded in alphabetical order of the entire path

```
nav.html
main.html
client/lib/methods.js
client/lib/styles.js
lib/feature/styles.js
lib/collections.js
client/feature-y.js
feature-x.js
client/main.js
```

For example, the files above are arranged in the correct load order. `main.html` is loaded second because HTML templates are always loaded first, even if it begins with `main.` , since rule 1 has priority over rule 2. However, it will be loaded after `nav.html` because rule 2 has priority over rule 5.

`client/lib/styles.js` and `lib/feature/styles.js` have identical load order up to rule 4; however, since `client` comes before `lib` alphabetically, it will be loaded first.

> *You can also use Meteor.startup to control when run code is run on both the server and the client.*

Special directories

By default, any JavaScript files in your Meteor application folder are bundled and loaded on both the client and the server. However, the names of the files and directories inside your project can affect their load order, where they are loaded, and some other characteristics. Here is a list of file and directory names that are treated specially by Meteor:

- **imports**

  Any directory named `imports/` is not loaded anywhere and files must be imported using `import` .

- **node_modules**

  Any directory named `node_modules/` is not loaded anywhere. node.js packages installed into `node_modules` directories must be imported using `import` or by using `Npm.depends` in `package.js` .

- **client**

  Any directory named `client/` is not loaded on the server. Similar to wrapping your code in `if (Meteor.isClient) { ... }` . All files loaded on the client are automatically concatenated and minified when in production mode. In development mode, JavaScript and CSS files are not minified, to make debugging easier. CSS files are still combined into a single file for consistency between production and development, because changing the CSS file's URL affects how URLs in it are processed.

> *HTML files in a Meteor application are treated quite a bit differently from a server-side framework. Meteor scans all the HTML files in your directory for three top-level elements:* `<head>`, `<body>`, *and* `<template>`. *The head and body sections are separately concatenated into a single head and body, which are transmitted to the client on initial page load.*

- **server**

  Any directory named `server/` is not loaded on the client. Similar to wrapping your code in `if (Meteor.isServer) { ... }`, except the client never even receives the code. Any sensitive code that you don't want served to the client, such as code containing passwords or authentication mechanisms, should be kept in the `server/` directory.

  Meteor gathers all your JavaScript files, excluding anything under the `client`, `public`, and `private` subdirectories, and loads them into a Node.js server instance. In Meteor, your server code runs in a single thread per request, not in the asynchronous callback style typical of Node.

- **public**

  All files inside a top-level directory called `public/` are served as-is to the client. When referencing these assets, do not include `public/` in the URL, write the URL as if they were all in the top level. For example, reference `public/bg.png` as `<img src='/bg.png' />`. This is the best place for `favicon.ico`, `robots.txt`, and similar files.

- **private**

  All files inside a top-level directory called `private/` are only accessible from server code and can be loaded via the `Assets` API. This can be used for private data files and any files that are in your project directory that you don't want to be accessible from the outside.

- **client/compatibility**

  This folder is for compatibility with JavaScript libraries that rely on variables declared with var at the top level being exported as globals. Files in this directory are executed without being wrapped in a new variable scope. These files are executed before other client-side JavaScript files.

  > *It is recommended to use npm for 3rd party JavaScript libraries and use* `import` *to control when files are loaded.*

- **tests**

  Any directory named `tests/` is not loaded anywhere. Use this for any test code you want to run using a test runner outside of Meteor's built-in test tools.

The following directories are also not loaded as part of your app code:

- Files/directories whose names start with a dot, like `.meteor` and `.git`
- `packages/`: Used for local packages
- `cordova-build-override/`: Used for advanced mobile build customizations
- `programs`: For legacy reasons

Files outside special directories

All JavaScript files outside special directories are loaded on both the client and the server. Meteor provides the variables `Meteor.isClient` and `Meteor.isServer` so that your code can alter its behavior depending on whether it's running on the client or the server.

CSS and HTML files outside special directories are loaded on the client only and cannot be used from server code.

Splitting into multiple apps

If you are writing a sufficiently complex system, there can come a time where it makes sense to split your code up into multiple applications. For example you may want to create a separate application for the administration UI (rather than checking permissions all through the admin part of your site, you can check once), or separate the code for the mobile and desktop versions of your app.

Another very common use case is splitting a worker process away from your main application so that expensive jobs do not impact the user experience of your visitors by locking up a single web server.

There are some advantages of splitting your application in this way:

- Your client JavaScript bundle can be significantly smaller if you separate out code that a specific type of user will never use.
- You can deploy the different applications with different scaling setups and secure them differently (for instance you might restrict access to your admin application to users behind a firewall).
- You can allow different teams at your organization to work on the different applications independently.

However there are some challenges to splitting your code in this way that should be considered before jumping in.

Sharing code

The primary challenge is properly sharing code between the different applications you are building. The simplest approach to deal with this issue is to deploy the *same* application on different web servers, controlling the behavior via different settings. This approach allows you to deploy different versions with different scaling behavior but doesn't enjoy most of the other advantages stated above.

If you want to create Meteor applications with separate code, you'll have some modules that you'd like to share between them. If those modules are something the wider world could use, you should consider publishing them to a package system, either npm or Atmosphere, depending on whether the code is Meteor-specific or otherwise.

If the code is private, or of no interest to others, it typically makes sense to include the same module in both applications (you *can* do this with private npm modules). There are several ways to do this:

- a straightforward approach is to include the common code as a git submodule of both applications.
- alternatively, if you include both applications in a single repository, you can use symbolic links to include the common module inside both apps.

Sharing data

Another important consideration is how you'll share the data between your different applications.

The simplest approach is to point both applications at the same `MONGO_URL` and allow both applications to read and write from the database directly. This works well thanks to Meteor's support for reactivity through the database. When one app changes some data in MongoDB, users of any other app connected to the database will see the changes immediately thanks to Meteor's livequery.

However, in some cases it's better to allow one application to be the master and control access to the data for other applications via an API. This can help if you want to deploy the different applications on different schedules and need to be conservative about how the data changes.

The simplest way to provide a server-server API is to use Meteor's built-in DDP protocol directly. This is the same way your Meteor client gets data from your server, but you can also use it to communicate between different applications. You can use `DDP.connect()` to connect from a "client" server to the master server, and then use the connection object returned to make method calls and read from publications.

Sharing accounts

If you have two servers that access the same database and you want authenticated users to make DDP calls across the both of them, you can use the *resume token* set on one connection to login on the other.

If your user has connected to server A, then you can use `DDP.connect()` to open a connection to server B, and pass in server A's resume token to authenticate on server B. As both servers are using the same DB, the same server token will work in both cases. The code to authenticate looks like this:

```
// This is server A's token as the default `Accounts` points at our server
const token = Accounts._storedLoginToken();

// We create a *second* accounts client pointing at server B
const app2 = DDP.connect('url://of.server.b');
const accounts2 = new AccountsClient({ connection: app2 });

// Now we can login with the token. Further calls to `accounts2` will be
authenticated
accounts2.loginWithToken(token);
```

You can see a proof of concept of this architecture in an example repository.