# Build System

The Meteor build system is the actual command line tool that you get when you install Meteor. You run it by typing the `meteor` command in your terminal, possibly followed by a set of arguments. Read the docs about the command line tool or type `meteor help` in your terminal to learn about all of the commands.

What does it do?

The Meteor build tool is what compiles, runs, deploys, and publishes all of your Meteor apps and packages. It's Meteor's built-in solution to the problems also solved by tools like Grunt, Gulp, Webpack, Browserify, Nodemon, and many others, and uses many popular Node.js tools like Babel and UglifyJS internally to enable a seamless experience.

Reloads app on file change

After executing the `meteor` command to start the build tool you should leave it running while further developing your app. The build tool automatically detects any relevant file changes using a file watching system and recompiles the necessary changes, restarting your client or server environment as needed. Hot module replacement can optionally be used so you can view and test your changes even quicker.

Compiles files with build plugins

The main function of the Meteor build tool is to run "build plugins". These plugins define different parts of your app build process. Meteor puts heavy emphasis on reducing or removing build configuration files, so you won't see any large build process config files like you would in Gulp or Webpack. The Meteor build process is configured almost entirely through adding and removing packages to your app and putting files in specially named directories. For example, to get all of the newest stable ES2015 JavaScript features in your app, you add the `ecmascript` package. This package provides support for ES2015 modules, which gives you even more fine grained control over file load order using ES2015 `import` and `export`. As new Meteor releases add new features to this package you get them for free.

Controlling which files to build

By default Meteor will build certain files as controlled by your application file structure and Meteor's default file load order rules. However, you may override

the default behavior using `.meteorignore` files, which cause the build system to ignore certain files and directories using the same pattern syntax as `.gitignore` files. These files may appear in any directory of your app or package, specifying rules for the directory tree below them. These `.meteorignore` files are also fully integrated with Meteor's file watching system, so they can be added, removed, or modified during development.

Combines and minifies code

Another important feature of the Meteor build tool is that it automatically concatenates your application asset files, and in production minifies these bundles. This lets you add all of the comments and whitespace you want to your source code and split your code into as many files as necessary, all without worrying about app performance and load times. This is enabled by the `standard-minifier-js` and `standard-minifier-css` packages, which are included in all Meteor apps by default. If you need different minification behavior, you can replace these packages (see zodern:standard-minifier-js as an example).

Development vs. production

Running an app in development is all about fast iteration time. All kinds of different parts of your app are handled differently and instrumented to enable better reloads and debugging. In production, the app is reduced to the necessary code and functions just like any standard Node.js app. Therefore, you shouldn't run your app in production by executing the `meteor run` command. Instead, follow the directions in Deploying Meteor Applications. If you find an error in production that you suspect is related to minification, you can run the minified version of your app locally for testing with `meteor --production`.

JavaScript transpilation

These days, the landscape of JavaScript tools and frameworks is constantly shifting, and the language itself is evolving just as rapidly. It's no longer reasonable to wait for web browsers to implement the language features you want to use. Most JavaScript development workflows rely on compiling code to work on the lowest common denominator of environments, while letting you use the newest features in development. Meteor has support for some of the most popular tools out of the box.

ES2015+ (recommended)

The `ecmascript` package (which is installed into all new apps and packages by default, but can be removed), allows support for many ES2015 features. We recommend using it. You can read more about it in the Code Style article.

Babel

Babeljs is a configurable transpiler, which allows you write code in the latest version of JavaScript even when your supported environments don't support certain features natively. Babel will compile those features down to a supported version.

Meteor provides a set appropriate core plugins for each environment (Node 8, modern browsers, and legacy browsers) and React to support most modern Javascript code practices. In addition, Meteor (as of 1.3.3) supports custom .babelrc files which allows developers to further customise their Babel configuration to suit there needs (e.g. Stage 0 proposals).

Developers are encouraged to avoid adding large presets (such as babel-preset-env & babel-preset-react) and adding specific plugins as needed (even though it seems to work). You will avoid unnecessary Babel compilation and you'll be less likely to experience plugin ordering issues.

CoffeeScript

While we recommend using ES2015 with the `ecmascript` package as the best development experience for Meteor, everything in the platform is 100% compatible with CoffeeScript and many people in the Meteor community prefer it.

All you need to do to use CoffeeScript is add the right Meteor package:

```
meteor add coffeescript
```

All code written in CoffeeScript compiles to JavaScript under the hood, and is completely compatible with any code in other packages that is written in JS or ES2015.

TypeScript

TypeScript is modern JavaScript with optional types and more.

Adding types will make your code more readable and less prone to runtime errors.

TypeScript can be installed with:

```
meteor add typescript
```

It is necessary to configure the TypeScript compiler with a `tsconfig.json` file. Here's the one generated by `meteor create --typescript`:

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es2018",
    "module": "esNext",
    "lib": ["esnext", "dom"],
    "allowJs": true,
    "checkJs": false,
    "jsx": "preserve",
    "incremental": true,
    "noEmit": true,

    /* Strict Type-Checking Options */
```

```
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,

    /* Additional Checks */
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": false,
    "noFallthroughCasesInSwitch": false,

    /* Module Resolution Options */
    "baseUrl": ".",
    "paths": {
      /* Support absolute /imports/* with a leading '/' */
      "/*": ["*"]
    },
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "types": ["node", "mocha"],
    "esModuleInterop": true,
    "preserveSymlinks": true
  },
  "exclude": [
    "./.meteor/**",
    "./packages/**"
  ]
}
```

If you want to add TypeScript from the point of project creation, as of Meteor 1.8.2, you can run the create command with the –typescript flag:

```
meteor create --typescript name-of-my-new-typescript-app
```

Conditional imports

TypeScript does not support nested `import` statements, therefore conditionally importing modules requires you to use the `require` statement (see Using `require`).

To maintain type safety, you can take advantage of TypeScript's import elision and reference the types using the `typeof` keyword. See the TypeScript handbook article for details or this blog post for a concrete Meteor example.

Templates and HTML

Since Meteor uses client-side rendering for your app's UI, all of your HTML code, UI components, and templates need to be compiled to JavaScript. There are a few options at your disposal to write your UI code.

Blaze HTML templates

The aptly named `blaze-html-templates` package that comes with every new Meteor app by default compiles your `.html` files written using Spacebars into Blaze-compatible JavaScript code. You can also add `blaze-html-templates` to any of your packages to compile template files located in the package.

Read about how to use Blaze and Spacebars in the Blaze article.

Blaze Jade templates

If you don't like the Spacebars syntax Meteor uses by default and want something more concise, you can give Jade a try by using `pacreach:jade`. This package will compile all files in your app with the `.jade` extension into Blaze-compatible code, and can be used side-by-side with `blaze-html-templates` if you want to have some of your code in Spacebars and some in Jade.

JSX for React

If you're building your app's UI with React, currently the most popular way to write your UI components involves JSX, an extension to JavaScript that allows you to type HTML tags that are converted to React DOM elements. JSX code is handled automatically by the `ecmascript` package.

Other options for React

If you want to use React but don't want to deal with JSX and prefer a more HTML-like syntax, there are a few community options available. One that stands out in particular is Blaze-React, which simulates the entire Blaze API using React as a rendering engine.

CSS processing

All your CSS style files will be processed using Meteor's default file load order rules along with any import statements and concatenated into a single stylesheet, `merged-stylesheets.css`. In a production build this file is also minified. By default this single stylesheet is injected at the beginning of the HTML `<head />` section of your application.

However, this can potentially be an issue for some applications that use a third party UI framework, such as Bootstrap, which is loaded from a CDN. This could cause Bootstrap's CSS to come after your CSS and override your user-defined styles.

To get around this problem Meteor supports the use of a pseudo tag `<meteor-bundled-css />` that if placed anywhere in the `<head />` section your app will be replaced by a link to this concatenated CSS file. If this pseudo tag isn't used, the CSS file will be placed at the beginning of the

section as before.

CSS pre-processors

It's no secret that writing plain CSS can often be a hassle as there's no way to share common CSS code between different selectors or have a consistent color

scheme between different elements. CSS compilers, or pre-processors, solve these issues by adding extra features on top of the CSS language like variables, mixins, math, and more, and in some cases also significantly change the syntax of CSS to be easier to read and write.

Here are three example CSS pre-processors supported by Meteor:

1. Sass
2. Less.js
3. Stylus

They all have their pros and cons, and different people have different preferences, just like with JavaScript transpiled languages. Sass with the SCSS syntax is quite popular as CSS frameworks like Bootstrap 4 have switched to Sass, and the C++ LibSass implementation appears to be faster than some of the other compilers available.

CSS framework compatibility should be a primary concern when picking a pre-processor, because a framework written with Less won't be compatible with one written in Sass.

Source vs. import files

An important feature shared by all of the available CSS pre-processors is the ability to import files. This lets you split your CSS into smaller pieces, and provides a lot of the same benefits that you get from JavaScript modules:

1. You can control the load order of files by encoding dependencies through imports, since the load order of CSS matters.
2. You can create reusable CSS "modules" that only have variables and mixins and don't actually generate any CSS.

In Meteor, each of your `.scss`, `.less`, or `.styl` source files will be one of two types: "source" or "import".

A "source" file is evaluated eagerly and adds its compiled form to the CSS of the app immediately.

An "import" file is evaluated only if imported from some other file and can be used to share common mixins and variables between different CSS files in your app.

Read the documentation for each package listed below to see how to indicate which files are source files vs. imports.

Importing styles

In all three Meteor supported CSS pre-processors you can import other style files from both relative and absolute paths in your app and from both npm and Meteor Atmosphere packages.

```
@import '../stylesheets/colors.less';   // a relative path
@import '{}/imports/ui/stylesheets/button.less';   // absolute path with `{}` syntax
```

You can also import CSS from a JavaScript file if you have the `ecmascript` package installed:

```
import '../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.

Importing styles from an Atmosphere package using the `{}` package name syntax:

```
@import '{my-package:pretty-buttons}/buttons/styles.import.less';
```

> CSS files in an Atmosphere package are declared with `api.addFiles`, and therefore will be eagerly evaluated, and automatically bundled with all the other CSS in your app.

Importing styles from an npm package using the `{}` syntax:

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

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

For more examples and details on importing styles and using `@imports` with packages see the Using Packages article.

Sass

The best Sass build plugin for Meteor is `fourseven:scss`.

Less

Less is maintained as a Meteor core package called `less`.

Stylus

The best Stylus build plugin for Meteor is coagmano:stylus

PostCSS and Autoprefixer

In addition to CSS pre-processors like Sass, Less, and Stylus, there is now an ecosystem of CSS post-processors. Regardless of which CSS pre-processor you use, a post-processor can give you additional benefits like cross-browser compatibility.

The most popular CSS post-processor right now is PostCSS, which supports a variety of plugins. Autoprefixer is perhaps the most useful plugin, since it enables you to stop worrying about browser prefixes and compatibility and write standards-compliant CSS. No more copying 5 different statements every time you want a CSS gradient - you can write a standard gradient without any prefixes and Autoprefixer handles it for you.

Meteor automatically runs PostCSS for you once you've configured it. Learn more about enabling it in the docs for standard-minifier-css.

Hot Module Replacement

In Meteor apps, javascript, typescript, css files that are dynamically imported, and many other types of files are converted into javascript modules during the build process. Instead of reloading the client after a rebuild, Meteor is able to update the javascript modules within the running application that were modified. This reduces the feedback cycle while developing by allowing you to view and test your changes quicker.

Hot module replacement (HMR) can be enabled by adding the hot-module-replacement package to your app:

```
meteor add hot-module-replacement
```

Many types of javascript modules can not be updated with HMR, so HMR has to be configured to know which modules can be replaced and how to replace them. Most apps never need to do this manually. Instead, you can use integrations that configure HMR for you:

- React components are automatically updated using React Fast Refresh. This integration is enabled for all Meteor apps that use HMR and a supported react version.
- An integration for Blaze templates is in beta.
- Svelte files can be automatically updated with HMR by using the zodern:melte compiler package.
- akryum:vue-component uses its own implementation of HMR to update vue components.
- Some packages are able to help automatically dispose old versions of modules. For example, zodern:pure-admin removes menu items and pages added in the old version of the module so you don't end up with duplicate or outdated items when the new version of the module is ran.

To further control how HMR applies updates in your app, you can use the hot API. This can be used to accept updates for additional types of files, help dispose a module so the old version no longer affects the app (such as stopping Tracker.autorun computations), or creating your own integrations with other view layers or libraries.

If a change was made to the app that can not be applied with HMR, it reloads the page with hot code push, as is done when HMR is not enabled. It currently only supports app code in the modern client architecture. Future versions of Meteor will add support for packages and other architectures.

Build plugins

The most powerful feature of Meteor's build system is the ability to define custom build plugins. If you find yourself writing scripts that mangle one type of file into another, merge multiple files, or something else, it's likely that these scripts would be better implemented as a build plugin. The `ecmascript`, `templating`,

and `coffeescript` packages are all implemented as build plugins, so you can replace them with your own versions if you want to!

Read the documentation about build plugins.

Types of build plugins

There are three types of build plugins supported by Meteor today:

1. Compiler plugin - compiles source files (LESS, CoffeeScript) into built output (JS, CSS, asset files, and HTML). Only one compiler plugin can handle a single file extension.
2. Minifier plugin - compiles lots of built CSS or JS files into one or more minified files, for example `standard-minifiers`. Only one minifier can handle each of `js` and `css`.
3. Linter plugin - processes any number of files, and can print lint errors. Multiple linters can process the same files.

Writing your own build plugin

Writing a build plugin is a very advanced task that only the most advanced Meteor users should get into. The best place to start is to copy a different plugin that is the most similar to what you are trying to do. For example, if you wanted to make a new CSS compiler plugin, you could fork the `less` package; if you wanted to make your own JS transpiler, you could fork `ecmascript`. A good example of a linter is the `jshint` package, and for a minifier you can look at `standard-minifiers-js` and `standard-minifiers-css`.

Caching

The best way to make your build plugin fast is to use caching anywhere you can - the best way to save time is to do less work! Check out the documentation about CachingCompiler to learn more. It's used in all of the above examples, so you can see how to use it by looking at them.