To get started writing a package, use the Meteor command line tool:

```
meteor create --package my-package
```

> *It is required that your `my-package` name take the form of `username:my-package`, where `username` is your Meteor Developer username, if you plan to publish your package to Atmosphere.*

If you run this inside an app, it will place the newly generated package in that app's `packages/` directory. Outside an app, it will just create a standalone package directory. The command also generates some boilerplate files for you:

```
my-package
├── README.md
├── package.js
├── my-package-tests.js
└── my-package.js
```

The `package.js` file is the main file in every Meteor package. This is a JavaScript file that defines the metadata, files loaded, architectures, npm packages, and Cordova packages for your Meteor package.

In this guide article, we will go over some important points for building packages, but we won't explain every part of the `package.js` API. To learn about all of the options, read about the `package.js` API in the Meteor docs.

> *Don't forget to run `meteor add [my-package]` once you have finished developing your package in order to use it; this applies if the package is a local package for internal use only or if you have published the package to Atmosphere.*

Adding files and assets

The main function of an Atmosphere package is to contain source code (JS, CSS, and any transpiled languages) and assets (images, fonts, and more) that will be shared across different applications.

Adding JavaScript

To add JavaScript files to a package, specify an entrypoint with `api.mainModule()` in the package's `onUse` block (this will already have been done by `meteor create --package` above):

```
Package.onUse(function(api) {
  api.mainModule('my-package.js');
});
```

From that entrypoint, you can `import` other files within your package, just as you would in an application.

If you want to include different files on the client and server, you can specify multiple entry points using the second argument to the function:

```
Package.onUse(function(api) {
  api.mainModule('my-package-client.js', 'client');
  api.mainModule('my-package-server.js', 'server');
});
```

You can also add any source file that would be compiled to a JS file (such as a CoffeeScript file) in a similar way, assuming you depend on an appropriate build plugin.

Adding CSS

To include CSS files with your package you can use `api.addFiles()`:

```
Package.onUse(function(api) {
  api.addFiles('my-package.css', 'client');
});
```

The CSS file will be automatically loaded into any app that uses your package.

Adding Sass, Less, or Stylus mixins/variables

Just like packages can export JavaScript code, they can export reusable bits of CSS pre-processor code. You can also have a package that doesn't actually include any CSS, but just exports different bits of reusable mixins and variables. To get more details see Meteor build tool CSS pre-processors:

```
Package.onUse(function(api) {
  api.addFiles('my-package.scss', 'client');
});
```

This Sass file will be eagerly evaluated and its compiled form will be added to the CSS of the app immediately.

```
Package.onUse(function(api) {
  api.addFiles([
    'stylesheets/_util.scss',
    'stylesheets/_variables.scss'
  ], 'client', {isImport: true});
});
```

These two Sass files will be lazily evaluated and only included in the CSS of the app if imported from some other file.

Adding other assets

You can include other assets, such as fonts, icons or images, to your package using `api.addAssets`:

```
Package.onUse(function(api) {
  api.addAssets([
    'font/OpenSans-Regular-webfont.eot',
    'font/OpenSans-Regular-webfont.svg',
    'font/OpenSans-Regular-webfont.ttf',
    'font/OpenSans-Regular-webfont.woff',
  ], 'client');
});
```

You can then access these files from the client from a URL `/packages/username_my-package/font/OpenSans-Regular-webfont.eot` or from the server using the Assets API.

Exporting

While some packages exist just to provide side effects to the app, most packages provide a reusable bit of code that can be used by the consumer with `import`. To export a symbol from your package, use the ES2015 `export` syntax in your `mainModule`:

```
// in my-package.js:
export const myName = 'my-package';
```

Now users of your package can import the symbol with:

```
import { myName } from 'meteor/username:my-package';
```

Dependencies

Chances are your package will want to make use of other packages. To ensure they are available, you can declare dependencies. Atmosphere packages can depend both on other Atmosphere packages, as well as packages from npm.

Atmosphere dependencies

To depend on another Atmosphere package, use `api.use`:

```
Package.onUse(function(api) {
  // This package depends on 1.2.0 or above of validated-method
  api.use('mdg:validated-method@1.2.0');
});
```

One important feature of the Atmosphere package system is that it is single-loading: no two packages in the same app can have dependencies on conflicting versions of a single package. Read more about that in the section about version constraints below.

Depending on Meteor version

Note that the Meteor release version number is mostly a marketing artifact - the core Meteor packages themselves typically don't share this version number. This means packages can only depend on specific versions of the packages inside a Meteor release, but can't depend on a specific release itself. We have a helpful shorthand api called `api.versionsFrom` that handles this for you by automatically filling in package version numbers from a particular release:

```
// Use versions of core packages from Meteor 1.2.1
api.versionsFrom('1.2.1');

api.use([
  // Don't need to specify version because of versionsFrom above
  'ecmascript',
  'check',

  // Still need to specify versions of non-core packages
```

```
  'mdg:validated-method@1.2.0',
  'mdg:validation-error@0.1.0'
]);
```

The above code snippet is equivalent to the code below, which specifies all of the version numbers individually:

```
api.use([
  'ecmascript@0.1.6',
  'check@1.1.0',
  'mdg:validated-method@1.2.0',
  'mdg:validation-error@0.1.0'
]);
```

Additionally, you can call `api.versionsFrom(<release>)` multiple times, or with an array (eg `api.versionsFrom([<release1>, <release2>])`. Meteor will interpret this to mean that the package will work with packages from all the listed releases.

```
api.versionsFrom('1.2.1');
api.versionsFrom('1.4');
api.versionsFrom('1.8');

// or

api.versionsFrom(['1.2.1', '1.4', '1.8']);
```

This usually isn't necessary, but can help in cases where you support more than one major version of a core package.

Semantic versioning and version constraints

Meteor's package system relies heavily on Semantic Versioning, or SemVer. When one package declares a dependency on another, it always comes with a version constraint. These version constraints are then solved by Meteor's industrial-grade Version Solver to arrive at a set of package versions that meet all of the requirements, or display a helpful error if there is no solution.

The mental model here is:

1. **The major version must always match exactly.** If package `a` depends on `b@2.0.0`, the constraint will only be satisfied if the version of package `b` starts with a `2`. This means that you can never have two different major versions of a package in the same app.
2. **The minor and patch version numbers must be greater or equal to the requested version.** If the dependency requests version `2.1.3`, then `2.1.4` and `2.2.0` will work, but `2.0.4` and `2.1.2` will not.

The constraint solver is necessary because Meteor's package system is **single-loading** - that is, you can never have two different versions of the same package loaded side-by-side in the same app. This is particularly useful for packages that include a lot of client-side code, or packages that expect to be singletons.

Note that the version solver also has a concept of "gravity" - when many solutions are possible for a certain set of dependencies, it always selects the oldest possible version. This is helpful if you are trying to develop a package to ship to lots of users, since it ensures your package will be compatible with the lowest common denominator of a dependency. If your package needs a newer version than is currently being selected for a certain dependency, you need to update your `package.js` to have a newer version constraint.

If your package supports multiple major versions of a dependency, you can supply both versions to `api.use` like so:

```
api.use('blaze@1.0.0 || 2.0.0');
```

Meteor will use whichever major version is compatible with your other packages, or the most recent of the options given.

npm dependencies

Meteor packages can include npm packages to use JavaScript code from outside the Meteor package ecosystem or to include JavaScript code with native dependencies. Use Npm.depends at the top level of your `package.js` file. For example, here's how you would include the `github` npm package:

```
Npm.depends({
  github: '0.2.4'
});
```

If you want to use a local npm package, for example during development, you can give a directory instead of a version number:

```
Npm.depends({
  my-package: 'file:///home/user/npms/my-package'
});
```

You can import the dependency from within you package code in the same way that you would inside an application:

```
import github from 'github';
```

Peer npm dependencies

`Npm.depends()` is fairly rigid (you can only depend on an exact version), and will typically result in multiple versions of a package being installed if many different Atmosphere packages depend on the same npm package. This makes it less than ideal to use on the client, where it's impractical to ship multiple copies of the same package code to the browser. Client-side packages are also often written with the assumption that only a single copy will be loaded. For example, React will complain if it is included more than once in an application bundle.

To avoid this problem as a package author, you can request that users of your package have installed the npm package you want to use at the application level. This is similar to a peer dependency of an npm package (although with less support in the tool). You can use the `tmeasday:check-npm-versions` package to ensure that they've done this, and to warn them if not.

For instance, if you are writing a React package, you should not directly depend on `react`, but instead use `check-npm-versions` to check the user has installed it:

```
import { checkNpmVersions } from 'meteor/tmeasday:check-npm-versions';

checkNpmVersions({
  'react': '0.14.x'
```

```
}, 'my:awesome-package');

// If you are using the dependency in the same file, you'll need to use require,
otherwise
// you can continue to `import` in another file.
const React = require('react');
```

> *Note that `checkNpmVersions` will only output a warning if the user has installed a incompatible version of the npm package. So your `require` call may not give you what you expect. This is consistent with npm's handling of peer dependencies.*

Cordova plugins

Atmosphere packages can include Cordova plugins to ship native code for the Meteor mobile app container. This way, you can interact with the native camera interface, use the gyroscope, save files locally, and more.

Include Cordova plugins in your Meteor package by using Cordova.depends.

Read more about using Cordova in the mobile guide.

Testing packages

Meteor has a test mode for packages called `meteor test-packages`. If you are in a package's directory, you can run

```
meteor test-packages ./ --driver-package meteortesting:mocha
```

This will run a special app containing only a "test" version of your package and start a Mocha test driver package.

When your package starts in test mode, rather than loading the `onUse` block, Meteor loads the `onTest` block:

```
Package.onTest(function(api) {
  // You almost definitely want to depend on the package itself,
  // this is what you are testing!
  api.use('my-package');

  // You should also include any packages you need to use in the test code
  api.use(['ecmascript', 'random', 'meteortesting:mocha']);

  // Finally add an entry point for tests
  api.mainModule('my-package-tests.js');
});
```

From within your test entry point, you can import other files as you would in the package proper.

You can read more about testing in Meteor in the Testing article.

Publishing your package

To publish your package to Atmosphere, run `meteor publish` from the package directory. To publish a package the package name must follow the format of `username:my-package` and the package must contain a SemVer

version number.

> *Note that if you have a local `node_modules` directory in your package, remove it before running `meteor publish`. While local `node_modules` directories are allowed in Meteor packages, their paths can collide with the paths of `Npm.depends` dependencies when published.*

Cache format

If you've ever looked inside Meteor's package cache at `~/.meteor/packages`, you know that the on-disk format of a built Meteor package is completely different from the way the source code looks when you're developing the package. The idea is that the target format of a package can remain consistent even if the API for development changes.

Local packages

As an alternative to publishing your package on Atmosphere, if you want to keep your package private, you can place it in your Meteor app in the `packages/` directory, for instance `packages/foo/`, and then add it to your app with `meteor add foo`.

Overriding published packages with a local version

If you need to modify an Atmosphere package to do something that the published version doesn't do, you can edit a local version of the package on your computer.

A Meteor app can load Atmosphere packages in one of three ways, and it looks for a matching package name in the following order:

1. Package source code in the `packages/` directory inside your app.
2. Package source code in directories indicated by setting a `METEOR_PACKAGE_DIRS` environment variable before running any `meteor` command. You can add multiple directories by separating the paths with a `:` on OSX or Linux, or a `;` on Windows. For example: `METEOR_PACKAGE_DIRS=../first/directory:../second/directory`, or on Windows: `set PACKAGE_DIRS=..\first\directory;..\second\directory`.

   > *Note: Prior to Meteor 1.4.2, `METEOR_PACKAGE_DIRS` was `PACKAGE_DIRS`. For compatibility reasons, developers should use `METEOR_PACKAGE_DIRS` going forward.*

3. Pre-built package from Atmosphere. The package is cached in `~/.meteor/packages` on Mac/Linux or `%LOCALAPPDATA%\.meteor\packages` on Windows, and only loaded into your app as it is built.

You can use (1) or (2) to override the version from Atmosphere. You can even do this to load patched versions of Meteor core packages - just copy the code of the package from Meteor's GitHub repository, and edit away.