A package is a directory containing a package.js file, which contains roughly three major sections: a basic description, a package definition, and a test definition. By default, the directory name is the name of the package.

The `package.js` file below is an example of how to use the packaging API. The rest of this section will explain the specific API commands in greater detail.

```
// Information about this package:
Package.describe({
  // Short two-sentence summary
  summary: 'What this does',
  // Version number
  version: '1.0.0',
  // Optional, default is package directory name
  name: 'username:package-name',
  // Optional GitHub URL to your source repository
  git: 'https://github.com/something/something.git'
});

// This defines your actual package:
Package.onUse((api) => {
  // If no version is specified for an `api.use` dependency, use the one defined
  // in Meteor 1.4.3.1.
  api.versionsFrom('1.4.3.1');
  // Use the `underscore` package, but only on the server. Version not
  // specified, so it will be as of Meteor 1.4.3.1.
  api.use('underscore', 'server');
  // Use `kadira:flow-router`, version 2.12.1 or newer.
  api.use('kadira:flow-router@2.12.1');
  // Give users of this package access to active-route's JavaScript helpers.
  api.imply('zimme:active-route@2.3.2')
  // Export the object `Email` to packages or apps that use this package.
  api.export('Email', 'server');
  // Specify the source code for the package.
  api.addFiles('email.js', 'server');
  // When using `ecmascript` or `modules` packages, you can use this instead of
  // `api.export` and `api.addFiles`.
  api.mainModule('email.js', 'server');
});

// This defines the tests for the package:
Package.onTest((api) => {
  // Sets up a dependency on this package.
  api.use('username:package-name');
  // Use the Mocha test framework.
  api.use('practicalmeteor:mocha@2.4.5_2');
  // Specify the source code for the package tests.
  api.addFiles('email_tests.js', 'server');
});

// This lets you use npm packages in your package:
Npm.depends({
  simplesmtp: '0.3.10',
```

```
  'stream-buffers': '0.2.5'
});
```

`api.mainModule` is documented in the modules section.

Build plugins are created with `Package.registerBuildPlugin`. See the coffeescript package for an example. Build plugins are fully-fledged Meteor programs in their own right and have their own namespace, package dependencies, source files and npm requirements.

> *You can use local packages to define custom build plugins for your app, with one caveat. In published packages, build plugins are already bundled with their transitive dependencies. So if you want a dependency of a build plugin to be satisfied by a local package, you must use a local copy of the package that defines the plugin (even if you make no changes to that package) so that Meteor will pick up the local dependency.*

> *In a lifecycle of a package there might come time to end the development for various reasons, or it gets superseded. In either case Meteor allows you to easily notify the users of the package by setting the deprecated flag to true:* `deprecated: true` *in the package description. In addition, you replace it with a string that tells the users where to find replacement or what to do.*

Provide basic package information with `Package.describe(options)`. To publish a package, you must define `summary` and `version`.

{% apibox "PackageNamespace#describe" nested: %}

Define dependencies and expose package methods with the `Package.onUse` handler. This section lets you define what packages your package depends on, what packages are implied by your package, and what object your package is exported to.

{% apibox "PackageNamespace#onUse" nested: %}

{% apibox "PackageAPI#versionsFrom" %} {% apibox "PackageAPI#use" %} {% apibox "PackageAPI#imply" %} {% apibox "PackageAPI#export" %} {% apibox "PackageAPI#addFiles" %} {% apibox "PackageAPI#addAssets" %}

Set up your tests with the `Package.onTest` handler, which has an interface that's parallel to that of the `onUse` handler. The tests will need to depend on the package that you have just created. For example, if your package is the `email` package, you have to call `api.use('email')` in order to test the package.

If you used `meteor create` to set up your package, Meteor will create the required scaffolding in `package.js`, and you'll only need to add unit test code in the `_test.js` file that was created.

{% apibox "PackageNamespace#onTest" nested: %}

Meteor packages can include NPM packages and Cordova plugins by using `Npm.depends` and `Cordova.depends` in the `package.js` file.

{% apibox "PackageNpm#depends" nested: %} {% apibox "Npm.require" %} {% apibox "PackageCordova#depends" nested: %} {% apibox "PackageNamespace#registerBuildPlugin" nested: %}

Options

In some cases we need to offer options in packages where these options are not going to change in runtime.

We prefer to have these options defined in a configuration file instead of using JS code to call specific functions to define options in runtime.

For example, in `quave:collections` package you can force collections to be available only in the server like this:

```
"packages": {
  "quave:collections": {
    "isServerOnly": true
  }
}
```

We encourage every package author to follow this standard to offer options:

1. Use the official Meteor `settings` file
2. Inside the `settings` file read from a `Meteor` . `packages` . `<package name>` . `<your option name>`

   > *If it needs to be available in the client follow the same structure inside the* `public` *key.*

You can use quave:settings package to read options in the format above already merging the private and public options.

This way we avoid having to call a specific code before another specific code in a package as the setting is stored in the settings, and the package can load it when necessary instead of relying on a specific order of calls from the developer in the app code.

> *We've started to adopt this standard also in core packages on Meteor 1.10.2.*

{% apibox "Plugin.registerSourceHandler" nested:true %}

Build Plugins API

Meteor packages can provide build plugins - programs that integrate with the build tool Isobuild used to compile and bundle your application.

Starting with Meteor 1.2, the API used to plug into the build process is called "Build Plugins". There are 3 phases when a package's plugin can run: linting, compilation and minification. Here is an overview of operations Isobuild performs on the application and packages source:

1. Gather source files from the app folder or read `package.js` file for a package.
2. Lint all source files and print the linting warnings.
3. Compile the source files like CoffeeScript, ES2015, Less, or Templates to plain JavaScript and CSS.
4. Link the JavaScript files: wrap them into closures and provide necessary package imports.
5. Minify JavaScript and CSS files. Can also include concatenation of all files.

Build plugins fill the phases 2, 3 and 5.

Usually build plugins implement a class that is given a list of files to process. Commonly, such files have the following methods:

- `getContentsAsBuffer` - Returns the full contents of the file as a buffer.
- `getContentsAsString` - Returns the full contents of the file as a string.
- `getPackageName` - Returns the name of the package or `null` if the file is not in a package.
- `getPathInPackage` - Returns the relative path of file to the package or app root directory. The returned path always uses forward slashes.
- `getSourceHash` - Returns a hash string for the file that can be used to implement caching.

- `getArch` - Returns the architecture that is targeted while processing this file.
- `getBasename` - Returns the filename of the file.
- `getDirname` - Returns the directory path relative to the package or app root. The returned path always uses forward slashes.
- `error` - Call this method to raise a compilation or linting error for the file.

Linters

Linters are programs that check the code for undeclared variables or find code that doesn't correspond to certain style guidelines. Some of the popular examples of linters are JSHint and ESLint. Some of the non-JavaScript linter examples include CoffeeLint for CoffeeScript and CSSLint for CSS.

To register a linter build plugin in your package, you need to do a couple of things in your `package.js`:

- depend on the `isobuild:linter-plugin@1.0.0` package
- register a build plugin: `Package.registerBuildPlugin({ name, sources, ... });` (see docs)

In your build plugin sources, register a Linter Plugin: provide details such as a name, list of extensions and filenames the plugin will handle and a factory function that returns an instance of a linter class. Example:

```
Plugin.registerLinter({
  extensions: ['js'],
  filenames: ['.linterrc']
}, () => new MyLinter);
```

In this example, we register a linter that runs on all `js` files and also reads a file named `.linterrc` to get a configuration.

The `MyLinter` class should now implement the `processFilesForPackage` method. The method should accept two arguments: a list of files and an options object.

```
class MyLinter {
  processFilesForPackage(files, options) {
    files.forEach((file) => {
      // Lint the file.
      const lint = lintFile(file.getContentsAsString());

      if (lint) {
        // If there are linting errors, output them.
        const { message, line, column } = lint;
        file.error({ message, line, column });
      }
    });
  }
}
```

The globals are passed in the options object so that the linters can omit the warnings about the package imports that look like global variables.

Each file in the list is an object that has all the methods provided by all build plugins, described above.

See an example of a linting plugin implemented in Core: jshint.

Compilers

Compilers are programs that take the source files and output JavaScript or CSS. They also can output parts of HTML that is added to the `<head>` tag and static assets. Examples for the compiler plugins are: CoffeeScript, Babel.js, JSX compilers, Pug templating compiler and others.

To register a compiler plugin in your package, you need to do the following in your `package.js` file:

- depend on the `isobuild:compiler-plugin@1.0.0` package
- register a build plugin: `Package.registerBuildPlugin({ name, sources, ... });` (see docs)

In your build plugin source, register a Compiler Plugin: similar to other types of build plugins, provide the details, extensions and filenames and a factory function that returns an instance of the compiler. Ex.:

```
Plugin.registerCompiler({
  extensions: ['pug', 'tpl.pug'],
  filenames: []
}, () => new PugCompiler);
```

The compiler class must implement the `processFilesForTarget` method that is given the source files for a target (server or client part of the package/app).

```
class PugCompiler {
  processFilesForTarget(files) {
    files.forEach((file) => {
      // Process and add the output.
      const output = compilePug(file.getContentsAsString());

      file.addJavaScript({
        data: output,
        path: `${file.getPathInPackage()}.js`
      });
    });
  }
}
```

Besides the common methods available on the input files' class, the following methods are available:

- `getExtension` - Returns the extension that matched the compiler plugin. The longest prefix is preferred.
- `getDeclaredExports` - Returns a list of symbols declared as exports in this target. The result of `api.export('symbol')` calls in target's control file such as package.js.
- `getDisplayPath` Returns a relative path that can be used to form error messages or other display properties. Can be used as an input to a source map.
- `addStylesheet` - Web targets only. Add a stylesheet to the document. Not available for linter build plugins.
- `addJavaScript` - Add JavaScript code. The code added will only see the namespaces imported by this package as runtime dependencies using 'api.use'. If the file being compiled was added with the bare flag, the resulting JavaScript won't be wrapped in a closure.
- `addAsset` - Add a file to serve as-is to the browser or to include on the browser, depending on the target. On the web, it will be served at the exact path requested. For server targets, it can be retrieved using `Assets.getText` or `Assets.getBinary`.

- `addHtml` - Works in web targets only. Add markup to the `head` or `body` section of the document.
- `hmrAvailable` - Returns true if the file can be updated with HMR. Among other things, it checks if HMR supports the current architecture and build mode, and that the unibuild uses the `hot-module-replacement` package. There are rare situations where `hmrAvailable` returns true, but when more information is available later in the build process Meteor decides the file can not be updated with HMR.
- `readAndWatchFileWithHash` - Accepts an absolute path, and returns { contents, hash } Makes sure Meteor watches the file so any changes to it will trigger a rebuild

Meteor implements a couple of compilers as Core packages, good examples would be the Blaze templating package and the ecmascript package (compiles ES2015+ to JavaScript that can run in the browsers).

Minifiers

Minifiers run last after the sources has been compiled and JavaScript code has been linked. Minifiers are only ran for the client programs (`web.browser` and `web.cordova`).

There are two types of minifiers one can add: a minifier processing JavaScript (registered extensions: `['js']`) and a minifier processing CSS (extensions: `['css']`).

To register a minifier plugin in your package, add the following in your `package.js` file:

- depend on `isobuild:minifier-plugin@1.0.0` package
- register a build plugin: `Package.registerBuildPlugin({ name, sources, ... });` (see docs)

In your build plugin source, register a Minifier Plugin. Similar to Linter and Compiler plugin, specify the interested extensions (`css` or `js`). The factory function returns an instance of the minifier class.

```
Plugin.registerMinifier({
  extensions: ['js']
}, () => new UglifyJsMinifier);
```

The minifier class must implement the method `processFilesForBundle` . The first argument is a list of processed files and the options object specifies if the minifier is ran in production mode or development mode.

```
class UglifyJsMinifier {
  processFilesForBundle(files, options) {
    const { minifyMode } = options;

    if (minifyMode === 'development') {
      // Don't minify in development.
      file.forEach((file) => {
        file.addJavaScript({
          data: file.getContentsAsBuffer(),
          sourceMap: file.getSourceMap(),
          path: file.getPathInBundle()
        });
      });

      return;
    }

    // Minify in production.
```

```
    files.forEach((file) => {
      file.addJavaScript({
        data: uglifyjs.minify(file.getContentsAsBuffer()),
        path: file.getPathInBundle()
      });
    });
  }
}
```

In this example, we re-add the same files in the development mode to avoid unnecessary work and then we minify the files in production mode.

Besides the common input files' methods, these methods are available:

- `getPathInBundle` - returns a path of the processed file in the bundle.
- `getSourcePath` - returns absolute path of the input file if available, or null.
- `getSourceMap` - returns the source-map for the processed file if there is such.
- `addJavaScript` - same as compilers
- `addStylesheet` - same as compilers
- `readAndWatchFileWithHash` - only available for css minifiers. Same as compilers.

Right now, Meteor Core ships with the `standard-minifiers` package that can be replaced with a custom one. The source of the package is a good example how to build your own minification plugin.

In development builds, minifiers must meet these requirements to not prevent hot module replacement:

- Call `addJavasScript` once for each file to add the file's contents
- The contents of the files are not modified

In the future Meteor will allow minifiers to concatenate or modify files in development without affected hot module replacement.

Caching

Since the API allows build plugins to process multiple files at once, we encourage package authors to implement at least some in-memory caching for their plugins. Using the `getSourceHash` function for linters and compilers will allow quick incremental recompilations if the file is not reprocessed even when the contents didn't change.

For the fast rebuilds between the Isobuild process runs, plugins can implement on-disk caching. If a plugin implements the `setDiskCacheDirectory` method, it will be called from time to time with a new path on disk where the plugin can write its offline cache. The folder is correctly reset when the plugin is rebuilt or cache should be invalidated for any reason (for example, picked package versions set has changed).

Caching Compiler

There is a core package called `caching-compiler` that implements most of the common logic of keeping both in-memory and on-disk caches. The easiest way to implement caching correctly is to subclass the `CachingCompiler` or `MultiFileCachingCompiler` class from this package in your build plugin. `CachingCompiler` is for compilers that consider each file completely independently; `MultiFileCachingCompiler` is for compilers that allow files to reference each other. To get this class in your plugin namespace, add a dependency to the plugin definition:

```
Package.registerBuildPlugin({
  name: 'compileGG',
  use: ['caching-compiler@1.0.0'],
  sources: ['plugin/compile-gg.js']
});
```

Accessing File System

Since the build plugins run as part of the Meteor tool, they follow the same file-system access convention - all file system paths always look like a Unix path: using forward slashes and having a root at '/', even on Windows. For example: paths `/usr/bin/program` and `/C/Program Files/Program/program.exe` are valid paths, and `C:\Program Files\Program\program.exe` is not.

So whenever you get a path in your build plugin implementation, via `getPathInPackage` or in an argument of the `setDiskCacheDirectory` method, the path will be a Unix path.

Now, on running on Windows, the usual node modules `fs` and `path` expect to get a DOS path. To assist you to write correct code, the `Plugin` symbol provides its own versions of `fs` and `path` that you can use instead (note that all methods on `fs` are fiberized and sync versions prefer using Fibers rather than freezing the whole event loop).

Also `Plugin` provides helper functions `convertToStandardPath` and `convertToOSPath` to convert to a Unix path or to the path expected by the node libraries regardless of the path origin.

Example:

```
// On Windows
const fs = Plugin.fs;
const path = Plugin.path;

const filePath = path.join('/C/Program Files', 'Program/file.txt');
console.log(filePath); // Prints '/C/Program Files/Program/file.txt'

fs.writeFileSync(filePath, 'Hello.'); // Writes to 'C:\Program
Files\Program\file.txt'

console.log(Plugin.convertToOsPath(filePath)); // Prints 'C:\Program
Files\Program\file.txt'
```

Isobuild Feature Packages

Starting with Meteor 1.2, packages can declare that they need a version of the Meteor tool whose Isobuild build system supports a certain feature. For example, packages must write `api.use('isobuild:compiler-plugin@1.0.0')` in order to call `Plugin.registerCompiler`. This means that a package can transition from the old `registerSourceHandler` API to `registerCompiler` and Version Solver will properly prevent the `registerCompiler` version from being chosen by older tools that don't know how to handle it.

This is the known Isobuild feature "packages" sorted by the first release of Meteor which supports them.

Introduced in Meteor 1.2

- `compiler-plugin@1.0.0` : Allows use of `Plugin.registerCompiler` .
- `linter-plugin@1.0.0` : Allows use of `Plugin.registerLinter` .
- `minifier-plugin@1.0.0` : Allows use of `Plugin.registerMinifier` .
- `isopack-2@1.0.0` : This package is published only in `isopack-2` format and won't work in versions of Meteor that don't support that format.
- `prod-only@1.0.0` : Allows use of the `prodOnly` flag in `Package.describe` .
- `isobuild:cordova@5.4.0` : This package depends on a specific version of Cordova, most likely as a result of the Cordova plugins it depends on.