guide

repl

chat

github

Script which compiles small pieces of code into something larger and pplication. It uses the new standardized format for code modules Script, instead of previous idiosyncratic solutions such as CommonJS and AMD. ES6 modules let you freely and seamlessly combine the most useful individual functions from your favorite libraries. This will eventually be possible natively, but Rollup lets you do it today.

Quick start

Install with `npm install --global rollup`. Rollup can be used either through a command line interface with an optional configuration file, or else through its JavaScript API. Run `rollup --help` to see the available options and parameters.

> ⓘ See rollup-starter-lib and rollup-starter-app to see example library and application projects using Rollup

These commands assume the entry point to your application is named main.js, and that you'd like all imports compiled into a single file named bundle.js.

For browsers:

```
# compile to a <script> containing a self-executing function ('iife')
$ rollup main.js --file bundle.js --format iife
```

For Node.js:

```
# compile to a CommonJS module ('cjs')
$ rollup main.js --file bundle.js --format cjs
```

For both browsers and Node.js:

```
# UMD format requires a bundle name
$ rollup main.js --file bundle.js --format umd --name "myBundle"
```

Why

Developing software is usually easier if you break your project into smaller separate pieces, since that often removes unexpected interactions and dramatically reduces the complexity of the problems you'll need to solve, and simply writing smaller projects in the first place isn't necessarily the answer. Unfortunately, JavaScript has not historically included this capability as a core feature in the language.

This finally changed with the ES6 revision of JavaScript, which includes a syntax for importing and exporting functions and data so they can be shared between separate scripts. The specification is now fixed, but it is not yet implemented in browsers or Node.js. Rollup allows you to write your code using the new module system, and will then compile it back down to existing supported formats such as CommonJS modules, AMD modules, and IIFE-style scripts. This means that you get to *write future-proof code*, and you also get the tremendous benefits of...

Tree-shaking

In addition to enabling the use of ES6 modules, Rollup also statically analyzes the code you are importing, and will exclude anything that isn't actually used. This allows you to build on top of existing tools and modules without adding extra dependencies or bloating the size of your project.

For example, with CommonJS, the *entire tool or library must be imported*.

```
// import the entire utils object with CommonJS
var utils = require( './utils' );
var query = 'Rollup';
// use the ajax method of the utils object
utils.ajax( 'https://api.example.com?search=' + query ).then( handleResponse );
```

But with ES6 modules, instead of importing the whole `utils` object, we can just import the one `ajax` function we need:

```
// import the ajax function with an ES6 import statement
import { ajax } from './utils';
var query = 'Rollup';
// call the ajax function
ajax( 'https://api.example.com?search=' + query ).then( handleResponse );
```

Because Rollup includes the bare minimum, it results in lighter, faster, and less complicated libraries and applications. Since this approach is based on explicit `import` and `export` statements, it is more effective than simply running an automated minifier to detect unused variables in the compiled output code.

Compatibility

Importing CommonJS

Rollup can import existing CommonJS modules through a plugin.

Publishing ES6 Modules

To make sure your ES6 modules are immediately usable by tools that work with CommonJS such as Node.js and webpack, you can use Rollup to compile to UMD or CommonJS format, and then point to that compiled version with the `main` property in your `package.json` file. If your `package.json` file also has a `module` field, ES6-aware tools like Rollup and webpack 2 will import the ES6 module version directly.

Links

- step-by-step tutorial video series, with accompanying written walkthrough
- miscellaneous issues in the wiki

Frequently asked questions

Why are ES modules better than CommonJS modules?

ES modules are an official standard and the clear path forward for JavaScript code structure, whereas CommonJS modules are an idiosyncratic legacy format that served as a stopgap solution before ES modules had been proposed. ES modules allow static analysis that enables optimizations like tree-shaking, and provide advanced features like circular references and live bindings.

What is "tree-shaking?"

Tree-shaking, also known as "live code inclusion," is the process of eliminating code that is not actually used in a given project. It is similar to dead code elimination but can be much more efficient.

How do I use Rollup in Node.js with CommonJS modules?

Rollup strives to implement the specification for ES modules, not necessarily the behaviors of Node.js, npm, `require()`, and CommonJS. Consequently, loading of CommonJS modules and use of Node's module location resolution logic are both implemented as optional plugins, not included by default in the Rollup core. Just `npm install` the CommonJS and node-resolve plugins and then enable them using a `rollup.config.js` file and you should be all set.

Is Rollup meant for building libraries or applications?

Rollup is already used by many major JavaScript libraries, and can also be used to build the vast majority of applications. However if you want to use code-splitting or dynamic imports with older browsers, you will need an additional runtime to handle loading missing chunks. We recommend using the SystemJS Production Build as it integrates nicely with Rollup's system format output and is capable of properly handling all the ES module live bindings and re-export edge cases. Alternatively, an AMD loader can be used as well.

Who made the Rollup logo? It's lovely.

Julian Lloyd!

Tutorial

Creating your first bundle

Before we begin, you'll need to have Node.js installed so that you can use npm. You'll also need to know how to access the command line on your machine.

The easiest way to use Rollup is via the Command Line Interface (or CLI). For now, we'll install it globally (later on we'll learn how to install it locally to your project so that your build process is portable, but don't worry about that yet). Type this into the command line:

```
npm install rollup --global # or `npm i rollup -g` for short
```

You can now run the `rollup` command. Try it!

```
rollup
```

Because no arguments were passed, Rollup prints usage instructions. This is the same as running `rollup --help`, or `rollup -h`.

Let's create a simple project:

```
mkdir -p my-rollup-project/src
cd my-rollup-project
```

First, we need an *entry point*. Paste this into a new file called `src/main.js`:

```
// src/main.js
import foo from './foo.js';
export default function () {
  console.log(foo);
}
```

Then, let's create the `foo.js` module that our entry point imports:

```
// src/foo.js
export default 'hello world!';
```

Now we're ready to create a bundle:

```
rollup src/main.js -f cjs
```

The `-f` option (short for `--format`) specifies what kind of bundle we're creating — in this case, CommonJS (which will run in Node.js). Because we didn't specify an output file, it will be printed straight to `stdout`:

```
'use strict';

var foo = 'hello world!';

var main = function () {
  console.log(foo);
};

module.exports = main;
```

You can save the bundle as a file like so:

```
rollup src/main.js -o bundle.js -f cjs
```

(You could also do `rollup src/main.js -f cjs > bundle.js`, but as we'll see later, this is less flexible if you're generating sourcemaps.)

Try running the code:

```
node
> var myBundle = require('./bundle.js');
> myBundle();
'hello world!'
```

Congratulations! You've created your first bundle with Rollup.

Using config files

So far, so good, but as we start adding more options it becomes a bit of a nuisance to type out the command.

To save repeating ourselves, we can create a config file containing all the options we need. A config file is written in JavaScript and is more flexible than the raw CLI.

Create a file in the project root called `rollup.config.js`, and add the following code:

```
// rollup.config.js
export default {
  input: 'src/main.js',
  output: {
    file: 'bundle.js',
    format: 'cjs'
  }
};
```

To use the config file, we use the `--config` or `-c` flag:

```
rm bundle.js # so we can check the command works!
rollup -c
```

You can override any of the options in the config file with the equivalent command line options:

```
rollup -c -o bundle-2.js # `-o` is equivalent to `--file` (formerly "output")
```

(Note that Rollup itself processes the config file, which is why we're able to use `export default` syntax – the code isn't being transpiled with Babel or anything similar, so you can only use ES2015 features that are supported in the version of Node.js that you're running.)

You can, if you like, specify a different config file from the default `rollup.config.js`:

```
rollup --config rollup.config.dev.js
rollup --config rollup.config.prod.js
```

Using plugins

So far, we've created a simple bundle from an entry point and a module imported via a relative path. As you build more complex bundles, you'll often need more flexibility – importing modules installed with npm, compiling code with Babel, working with JSON files and so on.

For that, we use *plugins*, which change the behaviour of Rollup at key points in the bundling process. A list of available plugins is maintained on the Rollup wiki.

For this tutorial, we'll use rollup-plugin-json, which allows Rollup to import data from a JSON file.

Create a file in the project root called `package.json`, and add the following content:

```
{
  "name": "rollup-tutorial",
  "version": "1.0.0",
  "scripts": {
    "build": "rollup -c"
  }
}
```

Install rollup-plugin-json as a development dependency:

```
npm install --save-dev rollup-plugin-json
```

(We're using `--save-dev` rather than `--save` because our code doesn't actually depend on the plugin when it runs – only when we're building the bundle.)

Update your `src/main.js` file so that it imports from your package.json instead of `src/foo.js`:

```
// src/main.js
import { version } from '../package.json';

export default function () {
  console.log('version ' + version);
}
```

Edit your `rollup.config.js` file to include the JSON plugin:

```
// rollup.config.js
import json from 'rollup-plugin-json';

export default {
  input: 'src/main.js',
  output: {
    file: 'bundle.js',
    format: 'cjs'
  },
  plugins: [ json() ]
};
```

Run Rollup with `npm run build`. The result should look like this:

```
'use strict';

var version = "1.0.0";

var main = function () {
  console.log('version ' + version);
};

module.exports = main;
```

(Notice that only the data we actually need gets imported –`name` and `devDependencies` and other parts of `package.json` are ignored. That's tree-shaking in action!)

Experimental Code Splitting

To use the new experimental code splitting feature, we add a second*entry point* called `src/main2.js` that itself dynamically loads main.js:

```
// src/main2.js
export default function () {
  return import('./main.js').then(({ default: main }) => {
    main();
  });
}
```

We can then pass both entry points to the rollup build, and instead of an output file we set a folder to output to with the `--dir` option (also passing the experimental flags):

```
rollup src/main.js src/main2.js -f cjs --dir dist --experimentalCodeSplitting
```

Either built entry point can then be run in NodeJS without duplicating any code between the modules:

```
node -e "require('./dist/main2.js')()"
```

You can build the same code for the browser, for native ES modules, an AMD loader or SystemJS.

For example, with `-f es` for native modules:

```
rollup src/main.js src/main2.js -f es --dir dist --experimentalCodeSplitting
```

```
<!doctype html>
<script type="module">
  import main2 from './dist/main2.js';
  main2();
</script>
```

Or alternatively, for SystemJS with `-f system`:

```
rollup src/main.js src/main2.js -f system --dir dist --experimentalCodeSplitting
```

install SystemJS via

```
npm install --save-dev systemjs
```

and then load either or both entry points in an HTML page as needed:

```
<!doctype html>
<script src="node_modules/systemjs/dist/system-production.js"></script>
<script>
  System.import('./dist/main2.js')
  .then(({ default: main }) => main());
</script>
```

Command Line Interface

Rollup should typically be used from the command line. You can provide an optional Rollup configuration file to simplify command line usage and enable advanced Rollup functionality.

Configuration files

Rollup configuration files are optional, but they are powerful and convenient and thus recommended.

A config file is an ES6 module that exports a default object with the desired options. Typically, it is called `rollup.config.js` and sits in the root directory of your project.

Consult the big list of options for details on each option you can include in your config file.

```
// rollup.config.js
export default { // can be an array (for multiple inputs)
  // core input options
  input,     // required
  external,
  plugins,

  // advanced input options
  onwarn,
  perf,

  // danger zone
  acorn,
  acornInjectPlugins,
  treeshake,
  context,
  moduleContext,

  // experimental
  experimentalCodeSplitting,
  manualChunks,
  optimizeChunks,
  chunkGroupingSize,

  output: {  // required (can be an array, for multiple outputs)
    // core output options
    format,  // required
    file,
    dir,
    name,
    globals,

    // advanced output options
    paths,
    banner,
    footer,
    intro,
    outro,
    sourcemap,
    sourcemapFile,
    interop,
    extend,

    // danger zone
    exports,
    amd,
    indent,
    strict,
    freeze,
    legacy,
    namespaceToStringTag

    // experimental
    entryFileNames,
    chunkFileNames,
    assetFileNames
  },

  watch: {
    chokidar,
    include,
    exclude,
    clearScreen
  }
};
```

You can export an array from your config file to build bundles from several different unrelated inputs at once, even in watch mode. To build different bundles with the same input, you supply an array of output options for each input:

```
// rollup.config.js (building more than one bundle)
export default [{
  input: 'main-a.js',
  output: {
    file: 'dist/bundle-a.js',
    format: 'cjs'
  }
}, {
  input: 'main-b.js',
  output: [
    {
      file: 'dist/bundle-b1.js',
      format: 'cjs'
    },
    {
      file: 'dist/bundle-b2.js',
      format: 'es'
    }
  ]
}];
```

If you want to create your config asynchronously, Rollup can also handle a `Promise` which resolves to an object or an array.

```
// rollup.config.js
import fetch from  'node-fetch';
export default fetch('/some-remote-service-or-file-which-returns-actual-config');
```

Similarly, you can do this as well:

```
// rollup.config.js (Promise resolving an array)
export default Promise.all([
  fetch('get-config-1'),
  fetch('get-config-2')
])
```

You *must* use a configuration file in order to do any of the following:

- bundle one project into multiple output files
- use Rollup plugins, such as rollup-plugin-node-resolve and rollup-plugin-commonjs which let you load CommonJS modules from the Node.js ecosystem

To use Rollup with a configuration file, pass the `--config` or `-c` flags.

```
# use Rollup with a rollup.config.js file
$ rollup --config

# alternatively, specify a custom config file location
$ rollup --config my.config.js
```

You can also export a function that returns any of the above configuration formats. This function will be passed the current command line arguments so that you can dynamically adapt your configuration to respect e.g. `--silent`. You can even define your own command line options if you prefix them with `config`:

```
// rollup.config.js
import defaultConfig from './rollup.default.config.js';
import debugConfig from './rollup.debug.config.js';

export default commandLineArgs => {
  if (commandLineArgs.configDebug === true) {
    return debugConfig;
  }
  return defaultConfig;
}
```

If you now run `rollup --config --configDebug`, the debug configuration will be used.

Command line flags

Many options have command line equivalents. Any arguments passed here will override the config file, if you're using one. See the big list of options for details.

```
-c, --config               Use this config file (if argument is used but value
                             is unspecified, defaults to rollup.config.js)
-i, --input                Input (alternative to <entry file>)
-o, --file <output>        Output (if absent, prints to stdout)
-f, --format [es]          Type of output (amd, cjs, es, iife, umd)
-e, --external             Comma-separate list of module IDs to exclude
-g, --globals              Comma-separate list of `module ID:Global` pairs
                             Any module IDs defined here are added to external
-n, --name                 Name for UMD export
-m, --sourcemap            Generate sourcemap (`-m inline` for inline map)
-l, --legacy               Support IE8
--amd.id                   ID for AMD module (default is anonymous)
--amd.define               Function to use in place of `define`
--no-strict                Don't emit a `"use strict";` in the generated modules.
--no-indent                Don't indent result
--environment <values>     Environment variables passed to config file
--noConflict               Generate a noConflict method for UMD globals
--no-treeshake             Disable tree-shaking
--intro                    Content to insert at top of bundle (inside wrapper)
--outro                    Content to insert at end of bundle (inside wrapper)
--banner                   Content to insert at top of bundle (outside wrapper)
--footer                   Content to insert at end of bundle (outside wrapper)
--no-interop               Do not include interop block
```

In addition, the following arguments can be used:

`-h/--help`

Print the help document.

`-v/--version`

Print the installed version number.

`-w/--watch`

Rebuild the bundle when its source files change on disk.

`--silent`

Don't print warnings to the console.

`--environment <values>`

Pass additional settings to the config file via`process.ENV`.

@@8 will set`process.env.INCLUDE_DEPS === 'true'` and `process.env.BUILD === 'production'`. You can use this option several times. In that case, subsequently set variables will overwrite previous definitions. This enables you for instance to overwrite environment variables in package.json scripts:

```
// in package.json
{
  "scripts": {
    "build": "rollup -c --environment INCLUDE_DEPS,BUILD:production"
  }
}
```

If you call this script via @@10

then the config file will receive `process.env.INCLUDE_DEPS === 'true'` and `process.env.BUILD === 'development'`.

JavaScript API

Rollup provides a JavaScript API which is usable from Node.js. You will rarely need to use this, and should probably be using the command line API unless you are extending Rollup itself or using it for something esoteric, such as generating bundles programmatically.

rollup.rollup

The `rollup.rollup` function returns a Promise that resolves to a `bundle` object with various properties and methods shown here:

```
const rollup = require('rollup');

// see below for details on the options
const inputOptions = {...};
const outputOptions = {...};

async function build() {
  // create a bundle
  const bundle = await rollup.rollup(inputOptions);

  console.log(bundle.imports); // an array of external dependencies
  console.log(bundle.exports); // an array of names exported by the entry point
  console.log(bundle.modules); // an array of module objects

  // generate code and a sourcemap
  const { code, map } = await bundle.generate(outputOptions);

  // or write the bundle to disk
  await bundle.write(outputOptions);
}

build();
```

inputOptions

The `inputOptions` object can contain the following properties (see the big list of options for full details on these):

```
const inputOptions = {
  // core options
  input, // the only required option
  external,
  plugins,

  // advanced options
  onwarn,
  cache,
  perf,

  // danger zone
  acorn,
  acornInjectPlugins,
  treeshake,
  context,
  moduleContext,

  // experimental
  experimentalCodeSplitting,
  manualChunks,
  optimizeChunks,
  chunkGroupingSize
};
```

outputOptions

The `outputOptions` object can contain the following properties (see the big list of options for full details on these):

```
const outputOptions = {
  // core options
  format, // required
  file,
  dir,
  name,
  globals,

  // advanced options
  paths,
  banner,
  footer,
  intro,
  outro,
  sourcemap,
  sourcemapFile,
  interop,
  extend,

  // danger zone
  exports,
  amd,
  indent,
  strict,
  freeze,
  legacy,
  namespaceToStringTag,

  // experimental
  entryFileNames,
  chunkFileNames,
  assetFileNames
};
```

rollup.watch

Rollup also provides a `rollup.watch` function that rebuilds your bundle when it detects that the individual modules have changed on disk. It is used internally when you run Rollup from the command line with the `--watch` flag.

```
const rollup = require('rollup');

const watchOptions = {...};
const watcher = rollup.watch(watchOptions);

watcher.on('event', event => {
  // event.code can be one of:
  //   START        — the watcher is (re)starting
  //   BUNDLE_START — building an individual bundle
  //   BUNDLE_END   — finished building a bundle
  //   END          — finished building all bundles
  //   ERROR        — encountered an error while bundling
  //   FATAL        — encountered an unrecoverable error
});

// stop watching
watcher.close();
```

watchOptions

The `watchOptions` argument is a config (or an array of configs) that you would export from a config file.

```
const watchOptions = {
  ...inputOptions,
  output: [outputOptions],
  watch: {
    chokidar,
    include,
    exclude,
    clearScreen
  }
};
```

See above for details on `inputOptions` and `outputOptions`, or consult the big list of options for info on `chokidar`, `include` and `exclude`.

TypeScript Declarations

If you'd like to use the API in a TypeScript environment you can do so, as now we ship TypeScript declarations.

You need to install some dependencies in case you have skipLibCheck turned off.

```
npm install @types/acorn @types/chokidar source-map magic-string --only=dev
```

Integrating Rollup with other tools

npm packages

At some point, it's very likely that your project will depend on packages installed from npm into your

`node_modules` folder. Unlike other bundlers like Webpack and Browserify, Rollup doesn't know 'out of the box' how to handle these dependencies - we need to add some configuration.

Let's add a simple dependency called the-answer, which exports the answer to the question of life, the universe and everything:

```
npm install the-answer # or `npm i the-answer`
```

If we update our `src/main.js` file...

```
// src/main.js
import answer from 'the-answer';

export default function () {
  console.log('the answer is ' + answer);
}
```

...and run Rollup...

```
npm run build
```

...we'll see a warning like this:

```
(!) Unresolved dependencies
https://github.com/rollup/rollup/wiki/Troubleshooting#treating-module-as-external-dependency
the-answer (imported by main.js)
```

The resulting `bundle.js` will still work in Node.js, because the `import` declaration gets turned into a CommonJS `require` statement, but `the-answer` does *not* get included in the bundle. For that, we need a plugin.

rollup-plugin-node-resolve

The rollup-plugin-node-resolve plugin teaches Rollup how to find external modules. Install it...

```
npm install --save-dev rollup-plugin-node-resolve
```

...and add it to your config file:

```
// rollup.config.js
import resolve from 'rollup-plugin-node-resolve';

export default {
  input: 'src/main.js',
  output: {
    file: 'bundle.js',
    format: 'cjs'
  },
  plugins: [ resolve() ]
};
```

This time, when you `npm run build`, no warning is emitted — the bundle contains the imported module.

rollup-plugin-commonjs

Some libraries expose ES6 modules that you can import as-is — `the-answer` is one such module. But at the moment, the majority of packages on npm are exposed as CommonJS modules instead. Until that changes, we need to convert CommonJS to ES2015 before Rollup can process them.

The rollup-plugin-commonjs plugin does exactly that.

Note that `rollup-plugin-commonjs` should go *before* other plugins that transform your modules — this is to prevent other plugins from making changes that break the CommonJS detection.

Peer dependencies

Let's say that you're building a library that has a peer dependency, such as React or Lodash. If you set up externals as described above, your rollup will bundle *all* imports:

```
import answer from 'the-answer';
import _ from 'lodash';
```

You can finely tune which imports are bundled and which are treated as external. For this example, we'll treat `lodash` as external, but not `the-answer`.

Here is the config file:

```
// rollup.config.js
import resolve from 'rollup-plugin-node-resolve';

export default {
  input: 'src/main.js',
  output: {
    file: 'bundle.js',
    format: 'cjs'
  },
  plugins: [resolve({
    // pass custom options to the resolve plugin
    customResolveOptions: {
      moduleDirectory: 'node_modules'
    }
  })],
  // indicate which modules should be treated as external
  external: ['lodash']
};
```

Voila, `lodash` will now be treated as external, and not be bundled with your library.

The `external` key accepts either an array of module names or a function which takes the module name and returns true if it should be treated as external. For example:

```
export default {
  // ...
  external: id => /lodash/.test(id)
}
```

You might use this form if you're using babel-plugin-lodash to cherry-pick lodash modules. In this case, Babel will convert your import statements to look like this:

```
import _merge from 'lodash/merge';
```

The array form of `external` does not handle wildcards, so this import will only be treated as external in the functional form.

Babel

Many developers use Babel in their projects, so that they can use futuristic JavaScript features that aren't yet supported by browsers and Node.js.

The easiest way to use both Babel and Rollup is with rollup-plugin-babel. Install it:

```
npm i -D rollup-plugin-babel
```

Add it to `rollup.config.js`:

```
// rollup.config.js
import resolve from 'rollup-plugin-node-resolve';
import babel from 'rollup-plugin-babel';

export default {
  input: 'src/main.js',
  output: {
    file: 'bundle.js',
    format: 'cjs'
  },
  plugins: [
    resolve(),
    babel({
      exclude: 'node_modules/**' // only transpile our source code
    })
  ]
};
```

Before Babel will actually compile your code, it needs to be configured. Create a new file `src/.babelrc`:

```
{
  "presets": [
    ["env", {
      "modules": false
    }]
  ],
  "plugins": ["external-helpers"]
}
```

There are a few unusual things about this setup. First, we're setting `"modules": false`, otherwise Babel will convert our modules to CommonJS before Rollup gets a chance to do its thing, causing it to fail.

Secondly, we're using the `external-helpers` plugin, which allows Rollup to include any 'helpers' just once at the top of the bundle, rather than including them in every module that uses them (which is the default behaviour).

Thirdly, we're putting our `.babelrc` file in `src`, rather than the project root. This allows us to have a different `.babelrc` for things like tests, if we need that later – it's generally a good idea to have separate configuration for separate tasks.

Now, before we run rollup, we need to install the `env` preset and the `external-helpers` plugin:

```
npm i -D babel-preset-env babel-plugin-external-helpers
```

Running Rollup now will create a bundle... except we're not actually using any ES2015 features. Let's change that. Edit `src/main.js`:

```
// src/main.js
import answer from 'the-answer';

export default () => {
  console.log(`the answer is ${answer}`);
}
```

Run Rollup with `npm run build`, and check the bundle:

```
'use strict';

var index = 42;

var main = (function () {
  console.log('the answer is ' + index);
});

module.exports = main;
```

Gulp

Rollup returns promises which are understood by gulp so integration is easy.

The syntax is very similar to the configuration file, but the properties are split across two different operations, corresponding to the JavaScript API:

```
const gulp = require('gulp');
const rollup = require('rollup');
const rollupTypescript = require('rollup-plugin-typescript');

gulp.task('build', () => {
  return rollup.rollup({
    input: './src/main.ts',
    plugins: [
      rollupTypescript()
    ]
  }).then(bundle => {
    return bundle.write({
      file: './dist/library.js',
      format: 'umd',
      name: 'library',
      sourcemap: true
    });
  });
});
```

Also you may use new async/await syntax

```
const gulp = require('gulp');
const rollup = require('rollup');
const rollupTypescript = require('rollup-plugin-typescript');

gulp.task('build', async function () {
  const bundle = await rollup.rollup({
    input: './src/main.ts',
    plugins: [
      rollupTypescript()
    ]
  });

  await bundle.write({
    file: './dist/library.js',
    format: 'umd',
    name: 'library',
    sourcemap: true
  });
});
```

ES module syntax

The following is intended as a lightweight reference for the module behaviors defined in the ES2015 specification, since a proper understanding of the import and export statements are essential to successful use of Rollup.

Importing

Imported values cannot be reassigned, though imported objects and arrays *can* be mutated (and the exporting module, and any other importers, will be affected by the mutation). In that way, they behave similarly to `const` declarations.

Named Imports

Import a specific item from a source module, with its original name.

```
import { something } from './module.js';
```

Import a specific item from a source module, with a custom name assigned upon import.

```
import { something as somethingElse } from './module.js';
```

Namespace Imports

Import everything from the source module as an object which exposes all the source module's named exports as properties and methods. Default exports are excluded from this object.

```
import * as module from './module.js'
```

The `something` example from above would then be attached to the imported object as a property, e.g. `module.something`.

Default Import

Import the default export of the source module.

```
import something from './module.js';
```

Empty Import

Load the module code, but don't make any new objects available.

```
import './module.js';
```

This is useful for polyfills, or when the primary purpose of the imported code is to muck about with prototypes.

Dynamic Import

Import modules using the dynamic import API. This API is experimental and available under the `experimentalDynamicImport` flag.

```
import('./modules.js').then(({ default: DefaultExport, NamedExport })=> {
  // do something with modules.
})
```

This is useful for code-splitting applications and using modules on-the-fly.

Exporting

Named exports

Export a value that has been previously declared:

```
var something = true;
export { something };
```

Rename on export:

```
export { something as somethingElse };
```

Export a value immediately upon declaration:

```
// this works with `var`, `let`, `const`, `class`, and `function`
export var something = true;
```

Default Export

Export a single value as the source module's default export:

```
export default something;
```

This practice is only recommended if your source module only has one export.

It is bad practice to mix default and named exports in the same module, though it is allowed by the specification.

How bindings work

ES modules export *live bindings*, not values, so values can be changed after they are initially imported as per this demo:

```
// incrementer.js
export let count = 0;

export function increment() {
  count += 1;
}
```

```
// main.js
import { count, increment } from './incrementer.js';

console.log(count); // 0
increment();
console.log(count); // 1

count += 1; // Error — only incrementer.js can change this
```

Big list of options

Core functionality

input `-i/--input`

`String` / `String[]` / `{ [entryName: String]: String }` The bundle's entry point(s) (e.g. your `main.js` or `app.js` or `index.js`). If you enable `experimentalCodeSplitting`, you can provide an array of entry points or object of named entry points which will be bundled to separate output chunks.

output.file `-o/--file`

`String` The file to write to. Will also be used to generate sourcemaps, if applicable. If `experimentalCodeSplitting` is enabled and `input` is an array, you must specify `dir` instead of `file`.

output.dir `--dir`

`String` The directory in which all generated chunks are placed. Only used if `experimentalCodeSplitting` is enabled and `input` is an array. In these cases this option replaces `file`.

output.format `-f/--format`

`String` The format of the generated bundle. One of the following:

- `amd` – Asynchronous Module Definition, used with module loaders like RequireJS
- `cjs` – CommonJS, suitable for Node and Browserify/Webpack
- `es` – Keep the bundle as an ES module file
- `iife` – A self-executing function, suitable for inclusion as a `<script>` tag. (If you want to create a bundle for your application, you probably want to use this, because it leads to smaller file sizes.)
- `umd` – Universal Module Definition, works as `amd`, `cjs` and `iife` all in one
- `system` – Native format of the SystemJS loader

output.name `-n/--name`

`String` The variable name, representing your `iife`/`umd` bundle, by which other scripts on the same page can access it.

```
// rollup.config.js
export default {
  ...,
  output: {
    file: 'bundle.js',
    format: 'iife',
    name: 'MyBundle'
  }
};

// -> var MyBundle = (function () {...
```

Namespaces are supported, so your name can contain dots. The resulting bundle will contain the setup necessary for the namespacing.

```
$ rollup -n "a.b.c"

/* ->
this.a = this.a || {};
this.a.b = this.a.b || {};
this.a.b.c = ...
*/
```

plugins

`Array` of plugin objects (or a single plugin object) – see Getting started with plugins for more information. Remember to call the imported plugin function (i.e. `commonjs()`, not just `commonjs`).

```
// rollup.config.js
import resolve from 'rollup-plugin-node-resolve';
import commonjs from 'rollup-plugin-commonjs';

export default {
  entry: 'main.js',
  plugins: [
    resolve(),
    commonjs()
  ]
};
```

external `-e/--external`

Either a `Function` that takes an `id` and returns `true` (external) or `false` (not external), or an `Array` of module IDs that should remain external to the bundle. The IDs should be either:

1. the name of an external dependency
2. a resolved ID (like an absolute path to a file)

```
// rollup.config.js
import path from 'path';

export default {
  ...,
  external: [
    'some-externally-required-library',
    path.resolve( './src/some-local-file-that-should-not-be-bundled.js' )
  ]
};
```

When given as a command line argument, it should be a comma-separated list of IDs:

```
rollup -i src/main.js ... -e foo,bar,baz
```

When providing a function, it is actually called with three parameters `(id, parent, isResolved)` that can give you more fine-grained control:

- `id` is the id of the module in question
- `parent` is the id of the module doing the import
- `isResolved` signals whether the `id` has been resolved by e.g. plugins

output.globals `-g/--globals`

`Object` of `id: name` pairs, used for `umd`/`iife` bundles. For example, in a case like this...

```
import $ from 'jquery';
```

...we want to tell Rollup that the `jquery` module ID equates to the global `$` variable:

```
// rollup.config.js
export default {
  ...,
  output: {
    format: 'iife',
    name: 'MyBundle',
    globals: {
      jquery: '$'
    }
  }
};

/*
var MyBundle = (function ($) {
  // code goes here
}(window.jQuery));
*/
```

Alternatively, supply a function that will turn an external module ID into a global.

When given as a command line argument, it should be a comma-separated list of`id:name` pairs:

```
rollup -i src/main.js ... -g jquery:$,underscore:_
```

Advanced functionality

output.paths

`Function` that takes an ID and returns a path, or`Object` of `id: path` pairs. Where supplied, these paths will be used in the generated bundle instead of the module ID, allowing you to (for example) load dependencies from a CDN:

```
// app.js
import { selectAll } from 'd3';
selectAll('p').style('color', 'purple');
// ...

// rollup.config.js
export default {
  input: 'app.js',
  external: ['d3'],
  output: {
    file: 'bundle.js',
    format: 'amd',
    paths: {
      d3: 'https://d3js.org/d3.v4.min'
    }
  }
};

// bundle.js
define(['https://d3js.org/d3.v4.min'], function (d3) {

  d3.selectAll('p').style('color', 'purple');
  // ...

});
```

output.banner/output.footer `-banner/--footer`

`String` A string to prepend/append to the bundle. You can also supply a`Promise` that resolves to a `String` to generate it asynchronously (Note: `banner` and `footer` options will not break sourcemaps)

```
// rollup.config.js
export default {
  ...,
  output: {
    ...,
    banner: '/* my-library version ' + version + ' */',
    footer: '/* follow me on Twitter! @rich_harris */'
  }
};
```

output.intro/output.outro `--intro/--outro`

`String` Similar to `banner` and `footer`, except that the code goes *inside* any format-specific wrapper. As with `banner` and `footer`, you can also supply a `Promise` that resolves to a `String`.

```
export default {
  ...,
  output: {
    ...,
    intro: 'var ENVIRONMENT = "production";'
  }
};
```

cache

`Object` A previously-generated bundle. Use it to speed up subsequent builds — Rollup will only reanalyse the modules that have changed.

onwarn

`Function` that will intercept warning messages. If not supplied, warnings will be deduplicated and printed to the console.

Warnings are objects with at minimum a `code` and a `message` property, meaning you can control how different kinds of warnings are handled:

```
onwarn (warning) {
  // skip certain warnings
  if (warning.code === 'UNUSED_EXTERNAL_IMPORT') return;

  // throw on others
  if (warning.code === 'NON_EXISTENT_EXPORT') throw new Error(warning.message);

  // console.warn everything else
  console.warn(warning.message);
}
```

Many warnings also have a `loc` property and a `frame` allowing you to locate the source of the warning:

```
onwarn ({ loc, frame, message }) {
  // print location if applicable
  if (loc) {
    console.warn(`${loc.file} (${loc.line}:${loc.column}) ${message}`);
    if (frame) console.warn(frame);
  } else {
    console.warn(message);
  }
}
```

output.sourcemap `-m/--sourcemap`

If `true`, a separate sourcemap file will be created. If `inline`, the sourcemap will be appended to the resulting `output` file as a data URI.

output.sourcemapFile `--sourcemapFile`

`String` The location of the generated bundle. If this is an absolute path, all the`sources` paths in the sourcemap will be relative to it. The `map.file` property is the basename of `sourcemapFile`, as the location of the sourcemap is assumed to be adjacent to the bundle.

`sourcemapFile` is not required if `output` is specified, in which case an output filename will be inferred by adding ".map" to the output filename for the bundle.

output.interop `--interop`/`--no-interop`*

`true` or `false` (defaults to `true`) – whether or not to add an 'interop block'. By default `interop: true`), for safety's sake, Rollup will assign any external dependencies' `default` exports to a separate variable if it's necessary to distinguish between default and named exports. This generally only applies if your external dependencies were transpiled (for example with Babel) – if you're sure you don't need it, you can save a few bytes with `interop: false`.

output.extend `--extend`/`--no-extend`*

`true` or `false` (defaults to `false`) – whether or not to extend the global variable defined by the`name` option in `umd` or `iife` formats. When `true`, the global variable will be defined as`(global.name = global.name || {})`. When false, the global defined by `name` will be overwritten like `(global.name = {})`.

perf `--perf`

`true` or `false` (defaults to `false`) – whether to collect performance timings. When used from the command line or a configuration file, detailed measurements about the current bundling process will be displayed. When used from the JavaScript API, the returned bundle object will contain an aditional `getTimings()` function that can be called at any time to retrieve all accumulated measurements.

Danger zone

You probably don't need to use these options unless you know what you're doing!

treeshake `--treeshake`/`--no-treeshake`

Can be `true`, `false` or an object (see below), defaults to`true`. Whether or not to apply tree-shaking and to fine-tune the tree-shaking process. Setting this option to `false` will produce bigger bundles but may improve build performance. If you discover a bug caused by the tree-shaking algorithm, please file an issue! Setting this option to an object implies tree-shaking is enabled and grants the following additional options:

treeshake.pureExternalModules `true`/`false` (default: `false`). If `true`, assume external dependencies from which nothing is imported do not have other side-effects like mutating global variables or logging.

```
// input file
import {unused} from 'external-a';
import 'external-b';
console.log(42);
```

```
// output with treeshake.pureExternalModules === false
import 'external-a';
import 'external-b';
console.log(42);
```

```
// output with treeshake.pureExternalModules === true
console.log(42);
```

treeshake.propertyReadSideEffects `true`/`false` (default: `true`). If `false`, assume reading a property of an object never has side-effects. Depending on your code, disabling this option can significantly reduce bundle size but can potentially break functionality if you rely on getters or errors from illegal property access.

```
// Will be removed if treeshake.propertyReadSideEffects === false
const foo = {
  get bar() {
    console.log('effect');
    return 'bar';
  }
}
const result = foo.bar;
const illegalAccess = foo.quux.tooDeep;
```

acorn

Any options that should be passed through to Acorn, such as `allowReserved: true`. Cf. the Acorn documentation for more available options.

acornInjectPlugins

An array of plugins to be injected into Acorn. In order to use a plugin, you need to pass its inject function here and enable it via the `acorn.plugins` option. For instance, to use async iteration, you can specify @@18

in your rollup configuration.

context `--context`

`String` By default, the context of a module – i.e., the value of `this` at the top level – is `undefined`. In rare cases you might need to change this to something else, like `'window'`.

moduleContext

Same as `options.context`, but per-module – can either be an object of `id: context` pairs, or an `id => context` function.

output.legacy `-l/--legacy`

`true` or `false` (defaults to `false`) – adds support for very old environments like IE8 by stripping out more modern code that might not work reliably, at the cost of deviating slightly from the precise specifications required of ES6 module environments.

output.exports `--exports`

`String` What export mode to use. Defaults to `auto`, which guesses your intentions based on what the `entry` module exports:

- `default` – suitable if you're only exporting one thing using `export default ...`
- `named` – suitable if you're exporting more than one thing
- `none` – suitable if you're not exporting anything (e.g. you're building an app, not a library)

The difference between `default` and `named` affects how other people can consume your bundle. If you use `default`, a CommonJS user could do this, for example:

```
var yourLib = require( 'your-lib' );
```

With `named`, a user would do this instead:

```
var yourMethod = require( 'your-lib' ).yourMethod;
```

The wrinkle is that if you use `named` exports but *also* have a `default` export, a user would have to do something like this to use the default export:

```
var yourMethod = require( 'your-lib' ).yourMethod;
var yourLib = require( 'your-lib' )['default'];
```

output.amd `--amd.id` and `--amd.define`

`Object` Can contain the following properties:

amd.id `String` An ID to use for AMD/UMD bundles:

```
// rollup.config.js
export default {
  ...,
  format: 'amd',
  amd: {
    id: 'my-bundle'
  }
};

// -> define('my-bundle', ['dependency'], ...
```

amd.define `String` A function name to use instead of `define`:

```
// rollup.config.js
export default {
  ...,
  format: 'amd',
  amd: {
    define: 'def'
  }
};

// -> def(['dependency'],...
```

output.indent `--indent/--no-indent`

`String` the indent string to use, for formats that require code to be indented `amd`, `iife`, `umd`). Can also be `false` (no indent), or `true` (the default – auto-indent)

```
// rollup.config.js
export default {
  ...,
  output: {
    ...,
    indent: false
  }
};
```

output.strict `--strict/--no-strict`*****

`true` or `false` (defaults to `true`) – whether to include the 'use strict' pragma at the top of generated non-ES6 bundles. Strictly-speaking (geddit?), ES6 modules are *always* in strict mode, so you shouldn't disable this without good reason.

output.freeze `--freeze/--no-freeze`

`true` or `false` (defaults to `true`) – wether to `Object.freeze()` namespace import objects (i.e. `import * as namespaceImportObject from...`) that are accessed dynamically.

output.namespaceToStringTag `--namespaceToStringTag/--no-namespaceToStringTag`*****

`true` or `false` (defaults to `false`) – whether to add spec compliant `.toString()` tags to namespace objects. If this option is set,

```
import * as namespace from './file.js';
console.log(String(namespace));
```

will always log `[object Module]`;

Experimental options

These options reflect new features that have not yet been fully finalized. Specific behaviour and usage may therefore be subject to change.

experimentalDynamicImport `--experimentalDynamicImport`

`true` or `false` (defaults to `false`) – adds the necessary acorn plugin to enable parsing dynamic imports, e.g. @@26 When used without `experimentalCodeSplitting`, statically resolvable dynamic imports will be automatically inlined into your bundle. Also enables the `resolveDynamicImport` plugin hook.

experimentalCodeSplitting `--experimentalCodeSplitting`

`true` or `false` (defaults to `false`) – enables you to specify multiple entry points. If this option is enabled, `input` can be set to an array of entry points to be built into the folder at the provided `output.dir`.

- Filenames of generated chunks in the `output.dir` folder correspond to the entry point filenames.
- Shared chunks are generated automatically to avoid code duplication between chunks.
- Enable the `experimentalDynamicImport` flag to generate new chunks for dynamic imports as well.

`output.dir` and input as an array must both be provided for code splitting to work, the `output.file` option is not compatible with code splitting workflows.

output.entryFileNames `--entryFileNames`

`String` the pattern to use for naming entry point output files within `dir` when code splitting. Defaults to `"[alias].js"`.

output.chunkFileNames `--chunkFileNames`

`String` the pattern to use for naming shared chunks created when code-splitting. Defaults to `"[alias]-[hash].js"`.

output.assetFileNames `--assetFileNames`

`String` the pattern to use for naming custom emitted assets to include in the build output when code-splitting. Defaults to `"assets/[name]-[hash][extname]"`.

manualChunks `--manualChunks`

`{ [chunkAlias: String]: String[] }` allows creating custom shared commmon chunks. Provides an alias for the chunk and the list of modules to include in that chunk. Modules are bundled into the chunk along with their dependencies. If a module is already in a previous chunk, then the chunk will reference it there. Modules defined into chunks this way are considered to be entry points that can execute independently to any parent importers.

optimizeChunks `--optimizeChunks`

`true` or `false` (defaults to `false`) - experimental feature to optimize chunk groupings. When a large number of chunks are generated in code-splitting, this allows smaller chunks to group together as long as they are within the `chunkGroupingSize` limit. It results in unnecessary code being loaded in some cases in order to have a smaller number of chunks overall. Disabled by default as it may cause unwanted side effects when loading unexpected code.

chunkGroupingSize `--chunkGroupingSize`

`number` (defaults to 5000) - the total source length allowed to be loaded unnecessarily when applying chunk grouping optimizations.

Watch options

These options only take effect when running Rollup with the `--watch` flag, or using `rollup.watch`.

watch.chokidar

A `Boolean` indicating that chokidar should be used instead of the built-in `fs.watch`, or an `Object` of options that are passed through to chokidar.

You must install chokidar separately if you wish to use it.

watch.include

Limit the file-watching to certain files:

```
// rollup.config.js
export default {
  ...,
  watch: {
    include: 'src/**'
  }
};
```

watch.exclude

Prevent files from being watched:

```
// rollup.config.js
export default {
  ...,
  watch: {
    exclude: 'node_modules/**'
  }
};
```

watch.clearScreen

`true` or `false` (defaults to `true`) – whether to clear the screen when a rebuild is triggered.