Table of Contents

Parcel Bundler PDF

This is the PDF version of Parcel web site parceljs.org The document is generated by gitbook CLI tool.

Getting Started

Parcel is a web application bundler, differentiated by its developer experience. It offers blazing fast performance utilizing multicore processing, and requires zero configuration.

First install Parcel using Yarn or npm:

Yarn:

```
yarn global add parcel-bundler
```

npm:

```
npm install -g parcel-bundler
```

Create a package.json file in your project directory using:

```
yarn init -y
```

or

```
npm init -y
```

Parcel can take any type of file as an entry point, but an HTML or JavaScript file is a good place to start. If you link your main JavaScript file in the HTML using a relative path, Parcel will also process it for you, and replace the reference with a URL to the output file.

Next, create an index.html and index.js file.

```
<html>
<body>
  <script src="./index.js"></script>
</body>
</html>
```

```
console.log("hello world");
```

Parcel has a development server built in, which will automatically rebuild your app as you change files and supports hot module replacement for fast development. Just point it at your entry file:

```
parcel index.html
```

Now open http://localhost:1234/ in your browser. If hot module replacement isn't working you may need to configure your editor. You can also override the default port with the `-p <port number>` option.

Use the development server when you don't have your own server, or your app is entirely client rendered. If you do have your own server, you can run Parcel in `watch` mode instead. This still automatically rebuilds as files change and supports hot module replacement, but doesn't start a web server.

```
parcel watch index.html
```

Multiple entry files

In case you have more than one entry file, let's say `index.html` and `about.html`, you have 2 ways to run the bundler:

Specifing the file names:

```
parcel index.html about.html
```

Use tokens and create a glob:

```
parcel *.html
```

NOTE: In case you have a file structure like this:

```
- folder-1
-- index.html
- folder-2
-- index.html
```

Going to http://localhost:1234/folder-1/ won't work, instead you will need to explicity point to the file http://localhost:1234/folder-1/index.html.

Building for production

When you're ready to build for production, the `build` mode turns off watching and only builds once. See the Production section for more details.

Assets

Parcel is based around assets. An asset can represent any file, but Parcel has special support for certain types of assets like JavaScript, CSS, and HTML files. Parcel automatically analyzes the dependencies referenced in these files and includes them in the output bundle. Assets of similar types are grouped together into the same output bundle. If you import an asset of a different type (for example, if you imported a CSS file from JS), it starts a child bundle and leaves a reference to it in the parent. This will be illustrated in the following sections.

JavaScript

The most traditional file type for web bundlers is JavaScript. Parcel supports both CommonJS and ES6 module syntax for importing files. It also supports dynamic `import()` function syntax to load modules asynchronously, which is discussed in the Code Splitting section.

```
// Import a module using CommonJS syntax
const dep = require('./path/to/dep');

// Import a module using ES6 import syntax
import dep from './path/to/dep';
```

You can also import non-JavaScript assets from a JavaScript file, e.g. CSS or even an image file. When you import one of these files, it is not inlined as in some other bundlers. Instead, it is placed in a separate bundle (e.g. a CSS file) along with all of its dependencies. When using CSS Modules, the exported classes are placed in the JavaScript bundle. Other asset types export a URL to the output file in the JavaScript bundle so you can reference them in your code.

```
// Import a CSS file
import './test.css';

// Import a CSS file with CSS modules
import classNames from './test.css';

// Import the URL to an image file
import imageURL from './test.png';
```

If you want to inline a file into the JavaScript bundle instead of reference it by URL, you can use the Node.js `fs.readFileSync` API to do that. The URL must be statically analyzable, meaning it cannot have any variables in it (other than `__dirname` and `__filename`).

```
import fs from 'fs';

// Read contents as a string
const string = fs.readFileSync(__dirname + '/test.txt', 'utf8');

// Read contents as a Buffer
const buffer = fs.readFileSync(__dirname + '/test.png');

// Convert Buffer contents to an image
<img  src={`${buffer.toString('base64')}`}/>
```

CSS

CSS assets can be imported from a JavaScript or HTML file, and can contain dependencies referenced by `@import` syntax as well as references to images, fonts, etc. via the `url()` function. Other CSS files that are `@import`ed are inlined into the same CSS bundle, and `url()` references are rewritten to their output filenames. All filenames should be relative to the current CSS file.

```
/* Import another CSS file */
@import './other.css';

.test {
  /* Reference an image file */
  background: url('./images/background.png');
}
```

In addition to plain CSS, other compile-to-CSS languages like LESS, SASS, and Stylus are also supported, and work the same way.

SCSS

SCSS compilation needs `node-sass` module. To install it with npm:

```
npm install node-sass
```

Once you have `node-sass` installed you can import SCSS files from JavaScript files.

```
import './custom.scss'
```

Dependencies in the SCSS files can be used with the `@import` statements.

HTML

HTML assets are often the entry file that you provide to Parcel, but can also be referenced by JavaScript files, e.g. to provide links to other pages. URLs to scripts, styles, media, and other HTML files are extracted and compiled as described above. The references are rewritten in the HTML so that they link to the correct output files. All filenames should be relative to the current HTML file.

```
<html>
<body>
  <!-- reference an image file -->
  <img src="./images/header.png">

  <a href="./other.html">Link to another page</a>

  <!-- import a JavaScript bundle -->
  <script src="./index.js"></script>
</body>
</html>
```

Transforms

While many bundlers require you to install and configure plugins to transform assets, Parcel has support for many common transforms and transpilers built in out of the box. You can transform JavaScript using Babel, CSS using PostCSS, and HTML using PostHTML. Parcel automatically runs these transforms when it finds a configuration file (e.g. `.babelrc`, `.postcssrc`) in a module.

This even works in third-party `node_modules`: if a configuration file is published as part of the package, the transform is automatically turned on for that module only. This keeps bundling fast since only modules that need to be transformed are processed. It also means that you don't need to manually configure the transforms to include and exclude certain files, or know how third party code is built in order to use it in your application.

Babel

Babel is a popular transpiler for JavaScript, with a large plugin ecosystem. Using Babel with Parcel works the same way as using it standalone or with other bundlers.

Install presets and plugins in your app:

```
yarn add babel-preset-react
```

Then, create a `.babelrc`:

```
{
  "presets": [
    "react"
  ]
}
```

Default babel transforms

Parcel transpiles your code with `babel-preset-env` by default, this is to transpile every module both internal (local requires) and external (node_modules) to match the defined target.

For the `browser` target it utilises browserlist, the target browserlist can be defined in `package.json` (`engines.browsers` or `browserslist`) or using a configuration file (`browserslist` or `.browserslistrc`).

The browserlist target defaults to: `> 0.25%` (Meaning, support every browser that has 0.25% or more of the total amount of active web users)

For the `node` target, Parcel uses the `engines.node` defined in `package.json`, this default to *node 8*.

PostCSS

PostCSS is a tool for transforming CSS with plugins, like autoprefixer, cssnext, and CSS Modules. You can configure PostCSS with Parcel by creating a configuration file using one of these names: `.postcssrc` (JSON), `.postcssrc.js`, or `postcss.config.js`.

Install plugins in your app:

```
yarn add postcss-modules autoprefixer
```

Then, create a `.postcssrc`:

```
{
  "modules": true,
  "plugins": {
    "autoprefixer": {
      "grid": true
    }
  }
}
```

Plugins are specified in the `plugins` object as keys, and options are defined using object values. If there are no options for a plugin, just set it to `true` instead.

Target browsers for Autoprefixer, cssnext and other tools can be specified in `.browserslistrc` file:

```
> 1%
last 2 versions
```

CSS Modules are enabled slightly differently using the a top-level `modules` key. This is because Parcel needs to have special support for CSS Modules since they export an object to be included in the JavaScript bundle as well. Note that you still need to install `postcss-modules` in your project.

Usage with existing CSS libraries

For CSS Modules to work properly with existing modules they need to specify this support in their own `.postcssrc`.

Set cssnano minify config

Parcel adds cssnano to postcss in order to minify css in production build, where custom configuration can be set by creating `cssnano.config.js` file:

```
module.exports = {
  preset: ['default', {
    calc: false,
    discardComments: {
      removeAll: true,
    }
  }]
};
```

PostHTML

PostHTML is a tool for transforming HTML with plugins. You can configure PostHTML with Parcel by creating a configuration file using one of these names: `.posthtmlrc` (JSON), `posthtmlrc.js`, or `posthtml.config.js`.

Install plugins in your app:

```
yarn add posthtml-img-autosize
```

Then, create a `.posthtmlrc`:

```
{
  "plugins": {
    "posthtml-img-autosize": {
      "root": "./images"
```

```
    }
  }
}
```

Plugins are specified in the `plugins` object as keys, and options are defined using object values. If there are no options for a plugin, just set it to `true` instead.

TypeScript

TypeScript is a typed superset of JavaScript that compiles down to plain JavaScript, which also supports modern ES2015+ features. Transforming TypeScript works out of the box without any additional configuration.

```
<!-- index.html -->
<html>
<body>
  <script src="./index.ts"></script>
</body>
</html>
```

```
// index.ts
import message from "./message";
console.log(message);
```

```
// message.ts
export default "Hello, world";
```

ReasonML/BuckleScript

ReasonML compiles OCaml to JavaScript with the help of BuckleScript. You can use ReasonML by installing dependencies and creating `bsconfig.json`:

```
$ yarn add bs-platform --dev
```

```
// bsconfig.json
// from https://github.com/BuckleScript/bucklescript/blob/master/jscomp/bsb/templates/basic-reason/bsconfig.jso
n

{
  "name": "whatever",
  "sources": {
    "dir": "src",
    "subdirs": true
  },
  "package-specs": {
    "module": "commonjs",
    "in-source": true
  },
  "suffix": ".bs.js",
  "bs-dependencies": [
  ],
  "warnings": {
    "error": "+101"
  },
  "namespace": true,
  "refmt": 3
}
```

```
<!-- index.html -->
<!doctype html>
<html>
<body>
  <script src="./src/index.re"></script>
</body>
</html>
```

```
/* src/index.re */
print_endline("Hello World");
```

ReasonReact

ReasonReact is React binding for ReasonML. You can use it with Parcel too:

```
$ yarn add react react-dom reason-react
```

```
// bsconfig.json

{
  "name": "whatever",
+ "reason": {
+   "react-jsx": 2
+ },
  "sources": {
    "dir": "src",
    "subdirs": true
  },
  "package-specs": {
    "module": "commonjs",
    "in-source": true
  },
  "suffix": ".bs.js",
  "bs-dependencies": [
+   "reason-react"
  ],
  "warnings": {
    "error": "+101"
  },
  "namespace": true,
  "refmt": 3
}
```

```
<!-- index.html -->
<html>
<body>
+  <div id="app"></div>
  <script src="./src/index.re"></script>
</body>
</html>
```

```
/* src/Greeting.re */

let component = ReasonReact.statelessComponent("Greeting");

let make = (~name, _children) => {
  ...component,
  render: _self => <div> (ReasonReact.string("Hello! " ++ name)) </div>,
};
```

```
/* src/index.re */

ReactDOMRe.renderToElementWithId(<Greeting name="Parcel" />, "app");
```

✂ Code Splitting

Parcel supports zero configuration code splitting out of the box. This allows you to split your application code into separate bundles which can be loaded on demand, which means smaller initial bundle sizes and faster load times. As the user navigates around in your application and modules are required, Parcel automatically takes care of loading child bundles on demand.

Code splitting is controlled by use of the dynamic `import()` function syntax proposal, which works like the normal `import` statement or `require` function, but returns a Promise. This means that the module is loaded asynchronously.

The following example shows how you might use dynamic imports to load a sub-page of your application on demand.

```
// pages/about.js
export function render() {
  // Render the page
}
```

```
import('./pages/about').then(function (page) {
  // Render page
  page.render();
});
```

Because `import()` returns a Promise, you can also use async/await syntax. You probably need to configure Babel to transpile the syntax though, until it is more widely supported by browsers.

```
const page = await import('./pages/about');
// Render page
page.render();
```

Dynamic imports are also lazily loaded in Parcel, so you can still put all your `import()` calls at the top of your file and the child bundles won't be loaded until they are used. The following example shows how you might lazily load sub-pages of an application dynamically.

```
// Setup a map of page names to dynamic imports.
// These are not loaded until they are used.
const pages = {
  about: import('./pages/about'),
  blog: import('./pages/blog')
};

async function renderPage(name) {
  // Lazily load the requested page.
  const page = await pages[name];
  return page.render();
}
```

Note: If you would like to use async/await in browsers that don't support it natively, remember to include `babel-polyfill` in your app or `babel-runtime` + `babel-plugin-transform-runtime` in libraries.

```
yarn add babel-polyfill
```

```
import "babel-polyfill";
import "./app";
```

Read the docs on babel-polyfill and babel-runtime.

Hot Module Replacement

Hot Module Replacement (HMR) improves the development experience by automatically updating modules in the browser at runtime without needing a whole page refresh. This means that application state can be retained as you change small things. Parcel's HMR implementation supports both JavaScript and CSS assets out of the box. HMR is automatically disabled when bundling in production mode.

As you save files, Parcel rebuilds what changed and sends an update to any running clients containing the new code. The new code then replaces the old version, and is re-evaluated along with all parents. You can hook into this process using the `module.hot` API, which can notify your code when a module is about to be disposed, or when a new version comes in. Projects like react-hot-loader can help with this process, and work out of the box with Parcel.

There are two methods to know about: `module.hot.accept` and `module.hot.dispose`. You call `module.hot.accept` with a callback function which is executed when that module or any of its dependencies are updated. `module.hot.dispose` accepts a callback which is called when that module is about to be replaced.

```
if (module.hot) {
  module.hot.dispose(function () {
    // module is about to be replaced
  });

  module.hot.accept(function () {
    // module or one of its dependencies was just updated
  });
}
```

Automagically installed dependencies

Whenever Parcel comes across a dependency that fits the `node_module` pattern and can't find it, Parcel tries to install this dependency using `yarn` or `npm` depending on finding a `yarn.lock` file or not. This prevents the developer from having to exit parcel or having multiple terminal windows open.

This only occurs in *development* (using `serve` or `watch`), however in production (using `build`) autoinstall is disabled to prevent unwanted side-effects on deployment.

You can disable this feature using `--no-autoinstall`.

Safe Write

Some text editors and IDE's have a feature called `safe write` this basically prevents data loss, by taking a copy of the file and renaming it when saved.

When using Hot Module Reload (HMR) this feature blocks the automatic detection of file updates, to disable `safe write` use the options provided below:

- `Sublime Text 3` add `atomic_save: "false"` to your user preferences.
- `IntelliJ` use search in the preferences to find "safe write" and disable it.
- `Vim` add `:set backupcopy=yes` to your settings.
- `WebStorm` uncheck `Use "safe write"` in Preferences > Appearance & Behavior > System Settings.

Production

When it comes time to bundle your application for production, you can use Parcel's production mode.

```
parcel build entry.js
```

Optimisations

This disables watch mode and hot module replacement so it will only build once. It also enables the minifier for all output bundles to reduce file size. The minifiers used by Parcel are terser for JavaScript, cssnano for CSS, and htmlnano for HTML.

Enabling production mode also sets the `NODE_ENV=production` environment variable. Large libraries like React have development only debugging features which are disabled by setting this environment variable, which results in smaller and faster builds for production.

File naming strategy

To allow setting very aggresive caching rules to your cdn, for optimal performance and efficiency, Parcel hashes the file names of most bundles (according to whether the bundle should have a readable/rememberable name or not, mainly for SEO).

Parcel follows the following table, when it comes to naming bundles. (Entrypoints are never hashed)

Bundle Type	Type	Content hashed
Any	Entrypoint	✘
JavaScript	`<script>`	✓
JavaScript	Dynamic import	✘
JavaScript	Service worker	✘
HTML	iframe	✘
HTML	anchor link	✘
Raw (Images, text files, ...)	Import/Require/...	✓

The file hash follows the following naming pattern: `<directory name>-<hash>.<extension>`

Cross platform gotchas

In an effort to optimize production build performance, Parcel will try to determine the number of CPUs available at the machine running the build command so it can distribute the work accordingly. To do so, Parcel relies on the physical-cpu-count library.

Be aware that this module assumes you have the `lscpu` program available in your system.

Recipes

React

First we need to install the dependencies for React.

Blog Post

```
npm install --save react
npm install --save react-dom
npm install --save-dev parcel-bundler
```

Or if you have the optional Yarn package manager installed

```
yarn add react
yarn add react-dom
yarn add --dev parcel-bundler
```

Add Start script to `package.json`

```
// package.json
"scripts": {
  "start": "parcel index.html"
}
```

Preact

First we need to install the dependencies for Preact.

```
npm install --save preact
npm install --save preact-compat
npm install --save-dev parcel-bundler
npm install --save-dev babel-preset-preact
```

Or if you have the optional Yarn package manager installed

```
yarn add preact
yarn add preact-compat
yarn add --dev parcel-bundler
yarn add --dev babel-preset-preact
```

Then make sure the following Babel config is present.

```
// .babelrc
{
  "presets": [
    "preact"
  ]
}
```

Add Start script to `package.json`

```
// package.json
```

```
  "scripts": {
    "start": "parcel index.html"
  }
```

Vue

First we need to install the dependencies for Vue.

```
npm install --save vue
npm install --save-dev parcel-bundler
```

Or if you have the optional Yarn package manager installed

```
yarn add vue
yarn add --dev parcel-bundler
```

Add Start script to `package.json`

```
// package.json
"scripts": {
  "start": "parcel index.html"
}
```

CLI

Commands

Serve

Starts up a development server, which will automatically rebuild your app as you change files and supports hot module replacement for fast development.

```
parcel index.html
```

Build

Builds the assets once, it also enabled minification and sets the `NODE_ENV=production` environment variable. See Production for more details.

```
parcel build index.html
```

Watch

The `watch` command is similar to `serve`, with the main difference being it doesn't start up a server.

```
parcel watch index.html
```

Help

Displays all possible cli options

```
parcel help
```

Version

Displays Parcel version number

```
parcel --version
```

Options

Output directory

Default: "dist"

Available in: `serve`, `watch`, `build`

```
parcel build entry.js --out-dir build/output
# or
parcel build entry.js -d build/output
```

```
root
- build
- - output
- - - entry.js
```

Set the public URL to serve on

Default: the same as the --out-dir option

Available in: `serve` , `watch` , `build`

```
parcel entry.js --public-url ./dist/
```

will output:

```
<link rel="stylesheet" type="text/css" href="/dist/entry.1a2b3c.css">
<!-- or -->
<script src="/dist/entry.e5f6g7.js"></script>
```

Target

Default: browser

Available in: `serve` , `watch` , `build`

```
parcel build entry.js --target node
```

Possible targets: `node` , `browser` , `electron`

Cache directory

Default: ".cache"

Available in: `serve` , `watch` , `build`

```
parcel build entry.js --cache-dir build/cache
```

Port

Default: 1234

Available in: `serve`

```
parcel serve entry.js --port 1111
```

Change Log level

Default: 3

Available in: `serve` , `watch` , `build`

```
parcel entry.js --log-level 1
```

Loglevel	Effect
0	Logging disabled
1	Only log errors
2	Log errors and warnings
3	Log everything

HMR Hostname

Default: `location.hostname` of current window

Available in: `serve`, `watch`

```
parcel entry.js --hmr-hostname parceljs.org
```

HMR Port

Default: A random available port

Available in: `serve`, `watch`

```
parcel entry.js --hmr-port 8080
```

Output filename

Default: Original filename

Available in: `serve`, `watch`, `build`

```
parcel build entry.js --out-file output.html
```

This changes the output filename of the entrypoint bundle

Print a detailed report

Default: Minimal report

Available in: `build`

```
parcel build entry.js --detailed-report
```

Enable https

Default: https disabled

Available in: `serve`, `watch` (listen on HTTPS for HMR connections)

```
parcel build entry.js --https
```

⚠ This flag generates a self-signed certificate, you might have to configure your browser to allow self-signed certificates for localhost.

Set a custom certificate

Default: https disabled

Available in: `serve`, `watch`

```
parcel entry.js --cert certificate.cert --key private.key
```

Open in browser

Default: open disabled

Available in: `serve`

```
parcel entry.js --open
```

Disable source-maps

Default: source-maps enabled

Available in: `serve`, `watch`, `build`

```
parcel build entry.js --no-source-maps
```

Disable autoinstall

Default: autoinstall enabled

Available in: `serve`, `watch`

```
parcel entry.js --no-autoinstall
```

Disable HMR

Default: HMR enabled

Available in: `serve`, `watch`

```
parcel entry.js --no-hmr
```

Disable minification

Default: minification enabled

Available in: `build`

```
parcel build entry.js --no-minify
```

Disable the filesystem cache

Default: cache enabled

Available in: `serve`, `watch`, `build`

```
parcel build entry.js --no-cache
```

Expose modules as UMD

Default: disabled

Available in: `serve`, `watch`, `build`

```
parcel serve entry.js --global myvariable
```

Enable experimental scope hoisting/tree shaking support

Default: disabled

Available in: `build`

```
parcel build entry.js --experimental-scope-hoisting
```

For more information, see the Tree Shaking section of Devon Govett's post on Parcel 1.9.

How It Works

Parcel transforms a tree of **assets** to a tree of **bundles**. Many other bundlers are fundamentally based around JavaScript assets, with other formats tacked on - e.g. inlined as strings into JS files. Parcel is file-type agnostic - it will work with any type of assets the way you'd expect, with no configuration. There are three steps to Parcel's bundling process.

1. Constructing the Asset Tree

Parcel takes as input a single entry asset, which could be any type: a JS file, HTML, CSS, image, etc. There are various asset types defined in Parcel which know how to handle specific file types. The assets are parsed, their dependencies are extracted, and they are transformed to their final compiled form. This creates a tree of assets.

2. Constructing the Bundle Tree

Once the asset tree has been constructed, the assets are placed into a bundle tree. A bundle is created for the entry asset, and child bundles are created for dynamic `import()` s, which cause code splitting to occur.

Sibling bundles are created when assets of a different type are imported, for example if you imported a CSS file from JavaScript, it would be placed into a sibling bundle to the corresponding JavaScript.

If an asset is required in more than one bundle, it is hoisted up to the nearest common ancestor in the bundle tree so it is not included more than once.

3. Packaging

After the bundle tree is constructed, each bundle is written to a file by a packager specific to the file type. The packagers know how to combine the code from each asset together into the final file that is loaded by a browser.

Asset Types

As described in the Assets documentation, Parcel represents each input file as an `Asset`. Asset types are represented as classes inheriting from the base `Asset` class and implementing the required interface to parse, analyze dependencies, transform, and code generate.

Because Parcel processes assets in parallel across multiple processor cores, the transforms that asset types can perform are limited to those that operate on a single file at a time. For transforms across multiple files, a custom Packager can be used.

Asset Interface

```
const {Asset} = require('parcel-bundler');

class MyAsset extends Asset {
  type = 'foo'; // set the main output type.

  async parse(code) {
    // parse code to an AST
    return ast;
  }

  async pretransform() {
    // optional. transform prior to collecting dependencies.
  }

  collectDependencies() {
    // analyze dependencies
    this.addDependency('my-dep');
  }

  async transform() {
    // optional. transform after collecting dependencies.
  }

  async generate() {
    // code generate. you can return multiple renditions if needed.
    // results are passed to the appropriate packagers to generate final bundles.
    return [
      {
        type: 'foo',
        value: 'my stuff here' // main output
      },
      {
        type: 'js',
        value: 'some javascript', // alternative rendition to be placed in JS bundle if needed
        sourceMap
      }
    ];
  }

  async postProcess(generated) {
    // Process after all code generating has been done
    // Can be used for combining multiple asset types
  }
}

module.exports = MyAsset
```

Registering an Asset Type

You can register your asset type with a bundler using the `addAssetType` method. It accepts a file extension to register, and the path to your asset type module. It is a path rather than the actual object so that it can be passed to worker processes.

```
const Bundler = require('parcel-bundler');

let bundler = new Bundler('input.js');
bundler.addAssetType('.ext', require.resolve('./MyAsset'));
```

API

Bundler

Instead of the CLI you can also use the API to initialise a bundler, for more advanced use-cases (e.g. custom processing after every build). A watch example with every option explained:

```
const Bundler = require('parcel-bundler');
const Path = require('path');

// Single entrypoint file location:
const entryFiles = Path.join(__dirname, './index.html');
// OR: Multiple files with globbing (can also be .js)
// const entryFiles = './src/*.js';
// OR: Multiple files in an array
// const entryFiles = ['./src/index.html', './some/other/directory/scripts.js'];

// Bundler options
const options = {
  outDir: './dist', // The out directory to put the build files in, defaults to dist
  outFile: 'index.html', // The name of the outputFile
  publicUrl: './', // The url to server on, defaults to dist
  watch: true, // whether to watch the files and rebuild them on change, defaults to process.env.NODE_ENV !== '
production'
  cache: true, // Enabled or disables caching, defaults to true
  cacheDir: '.cache', // The directory cache gets put in, defaults to .cache
  contentHash: false, // Disable content hash from being included on the filename
  minify: false, // Minify files, enabled if process.env.NODE_ENV === 'production'
  scopeHoist: false, // turn on experimental scope hoisting/tree shaking flag, for smaller production bundles
  target: 'browser', // browser/node/electron, defaults to browser
  https: { // Define a custom {key, cert} pair, use true to generate one or false to use http
    cert: './ssl/c.crt', // path to custom certificate
    key: './ssl/k.key' // path to custom key
  },
  logLevel: 3, // 3 = log everything, 2 = log warnings & errors, 1 = log errors
  hmrPort: 0, // The port the HMR socket runs on, defaults to a random free port (0 in node.js resolves to a ra
ndom free port)
  sourceMaps: true, // Enable or disable sourcemaps, defaults to enabled (not supported in minified builds yet)
  hmrHostname: '', // A hostname for hot module reload, default to ''
  detailedReport: false // Prints a detailed report of the bundles, assets, filesizes and times, defaults to fa
lse, reports are only printed if watch is disabled
};

async function runBundle() {
  // Initializes a bundler using the entrypoint location and options provided
  const bundler = new Bundler(entryFiles, options);

  // Run the bundler, this returns the main bundle
  // Use the events if you're using watch mode as this promise will only trigger once and not for every rebuild
  const bundle = await bundler.bundle();
}

runBundle();
```

Events

This is a list of all bundler events

- `bundled` gets called once Parcel has successfully finished bundling **for the first time**, the main bundle instance gets passed to the callback

```
const bundler = new Bundler(...);
bundler.on('bundled', (bundle) => {
// bundler contains all assets and bundles, see documentation for details
});
// Call this to start bundling
bundler.bundle();
```

- `buildEnd` gets called after each build (aka **including every rebuild**), this also emits if an error occurred

```
const bundler = new Bundler(...);
bundler.on('buildEnd', () => {
// Do something...
});
// Call this to start bundling
bundler.bundle();
```

- `buildStart` gets called at the start of the first build, the `entryFiles` Array gets passed to the callback

```
const bundler = new Bundler(...);
bundler.on('buildStart', entryPoints => {
// Do something...
});
// Call this to start bundling
bundler.bundle();
```

- `buildError` gets called every time an error occurs during builds, the `Error` Object gets passed to the callback

```
const bundler = new Bundler(...);
bundler.on('buildError', error => {
// Do something...
});
// Call this to start bundling
bundler.bundle();
```

Bundle

A `Bundle` is what Parcel uses to bundle assets together, this also contains child and sibling bundles to be able to build a bundle tree.

Properties

- `type` : The type of assets it contains (e.g. js, css, map, ...)
- `name` : The name of the bundle (generated using `Asset.generateBundleName()` of `entryAsset`)
- `parentBundle` : The parent bundle, is null in case of the entry bundle
- `entryAsset` : The entryPoint of the bundle, used for generating the name and gathering assets.
- `assets` : A `Set` of all assets inside the bundle
- `childBundles` : A `Set` of all child bundles
- `siblingBundles` : A `Set` of all sibling bundles
- `siblingBundlesMap` : A `Map<String(Type: js, css, map, ...), Bundle>` of all sibling bundles
- `offsets` : A `Map<Asset, number(line number inside the bundle)>` of all the locations of the assets inside the bundle, used to generate accurate source maps

Tree

The `Bundle` contains a `parentBundle`, `childBundles` and `siblingBundles`, all these properties together create a fast to iterate bundle tree.

A very basic example of an asset tree and it's generated bundle Tree

Asset tree:

`index.html` requires `index.js` and `index.css`.

`index.js` requires `test.js` and `test.txt`

```
index.html
-- index.js
 |--- test.js
 |--- test.txt
-- index.css
```

Bundle Tree:

`index.html` gets used as an entry asset for the main bundle, this main bundle creates two child bundles one for `index.js` and one for `index.css` this because they both are different from the `html` type.

`index.js` requires two files, `test.js` and `test.txt`.

`test.js` gets added to the assets of the `index.js` bundle, as it is of the same type as `index.js`

`test.txt` creates a new bundle and gets added as a child of the `index.js` bundle as it is a different assetType than `index.js`

`index.css` has no requires and therefore only contains it's entry Asset.

`index.css` and `index.js` bundles are siblingBundles of each other as they share the same parent.

```
index.html
-- index.js (includes index.js and test.js)
 |--- test.txt (includes test.txt)
-- index.css (includes index.css)
```

Middleware

Middleware can be used to hook into an http server (e.g. `express` or node `http`).

An example of using the Parcel middleware with express

```
const Bundler = require('parcel-bundler');
const app = require('express')();

const file = 'index.html'; // Pass an absolute path to the entrypoint here
const options = {}; // See options section of api docs, for the possibilities

// Initialize a new bundler using a file and options
const bundler = new Bundler(file, options);

// Let express use the bundler middleware, this will let Parcel handle every request over your express server
app.use(bundler.middleware());

// Listen on port 8080
app.listen(8080);
```

Plugins

Parcel takes a slightly different approach from many other tools in that many common formats are included out of the box without the need to install and configure additional plugins. However, there are cases where you might want to extend Parcel in a nonstandard way, and for those times, plugins are supported. Installed plugins are automatically detected and loaded based on `package.json` dependencies.

When adding support for a new file format to Parcel, you should first consider how widespread it is, and how standardized the implementation is. If it is sufficiently widespread and standard, the format should probably be added to Parcel core rather than as a plugin that users need to install. If you have any doubts, GitHub is the right place to discuss.

Plugin API

Parcel plugins are very simple. They are simply modules that export a single function, which is called by Parcel automatically during initialization. The function receives as input the `Bundler` object, and can do configuration such as registering asset types and packagers.

```
module.exports = function (bundler) {
  bundler.addAssetType('ext', require.resolve('./MyAsset'));
  bundler.addPackager('foo', require.resolve('./MyPackager'));
};
```

Publish this package on npm using the `parcel-plugin-` prefix, and it will be automatically detected and loaded as described below.

Using Plugins

Using plugins in Parcel could not be any simpler. All you need to do is install them and save in your `package.json`. Plugins should be named with the prefix `parcel-plugin-`, e.g. `parcel-plugin-foo`. Any dependencies listed in `package.json` with this prefix will be automatically loaded during initialization.