Loader Interface

A loader is just a JavaScript module that exports a function. The loader runner calls this function and passes the result of the previous loader or the resource file into it. The `this` context of the function is filled-in by webpack and the loader runner with some useful methods that allow the loader (among other things) to change its invocation style to async, or get query parameters.

The first loader is passed one argument: the content of the resource file. The compiler expects a result from the last loader. The result should be a `String` or a `Buffer` (which is converted to a string), representing the JavaScript source code of the module. An optional SourceMap result (as a JSON object) may also be passed.

A single result can be returned in sync mode. For multiple results the `this.callback()` must be called. In async mode `this.async()` must be called to indicate that the loader runner should wait for an asynchronous result. It returns `this.callback()`. Then the loader must return `undefined` and call that callback.

```
/**
 *
 * @param {string|Buffer} content Content of the resource file
 * @param {object} [map] SourceMap data consumable by https://github.com/mozilla/source-map
 * @param {any} [meta] Meta data, could be anything
 */
function webpackLoader(content, map, meta) {
  // code of your webpack loader
}
```

Examples

The following sections provide some basic examples of the different types of loaders. Note that the `map` and `meta` parameters are optional, see `this.callback` below.

Synchronous Loaders

Either `return` or `this.callback` can be used to return the transformed `content` synchronously:

sync-loader.js

```
module.exports = function (content, map, meta) {
  return someSyncOperation(content);
};
```

The `this.callback` method is more flexible as it allows multiple arguments to be passed as opposed to just the `content`.

sync-loader-with-multiple-results.js

```
module.exports = function (content, map, meta) {
  this.callback(null, someSyncOperation(content), map, meta);
  return; // always return undefined when calling callback()
};
```

Asynchronous Loaders

For asynchronous loaders, `this.async` is used to retrieve the `callback` function:

async-loader.js

```
module.exports = function (content, map, meta) {
  var callback = this.async();
  someAsyncOperation(content, function (err, result) {
    if (err) return callback(err);
    callback(null, result, map, meta);
  });
};
```

async-loader-with-multiple-results.js

```
module.exports = function (content, map, meta) {
  var callback = this.async();
  someAsyncOperation(content, function (err, result, sourceMaps, meta) {
    if (err) return callback(err);
    callback(null, result, sourceMaps, meta);
  });
};
```

> ### Tip

> Loaders were originally designed to work in synchronous loader pipelines, like Node.js (using enhanced-require), *and* asynchronous pipelines, like in webpack. However, since expensive synchronous computations are a bad idea in a single-threaded environment like Node.js, we advise making your loader asynchronous if possible. Synchronous loaders are ok if the amount of computation is trivial.

"Raw" Loader

By default, the resource file is converted to a UTF-8 string and passed to the loader. By setting the `raw` flag to `true`, the loader will receive the raw `Buffer`. Every loader is allowed to deliver its result as a `String` or as a `Buffer`. The compiler converts them between loaders.

raw-loader.js

```
module.exports = function (content) {
  assert(content instanceof Buffer);
  return someSyncOperation(content);
  // return value can be a `Buffer` too
  // This is also allowed if loader is not "raw"
};
module.exports.raw = true;
```

Pitching Loader

Loaders are **always** called from right to left. There are some instances where the loader only cares about the **metadata** behind a request and can ignore the results of the previous loader. The `pitch` method on loaders is called from **left to right** before the loaders are actually executed (from right to left).

> ### Tip
>
> Loaders may be added inline in requests and disabled via inline prefixes, which will impact the order in which they are "pitched" and executed. See `Rule.enforce` for more details.

For the following configuration of `use`:

```
module.exports = {
  //...
  module: {
    rules: [
      {
        //...
        use: ['a-loader', 'b-loader', 'c-loader'],
      },
    ],
  },
};
```

These steps would occur:

```
|- a-loader `pitch`
  |- b-loader `pitch`
    |- c-loader `pitch`
      |- requested module is picked up as a dependency
    |- c-loader normal execution
  |- b-loader normal execution
|- a-loader normal execution
```

So why might a loader take advantage of the "pitching" phase?

First, the `data` passed to the `pitch` method is exposed in the execution phase as well under `this.data` and could be useful for capturing and sharing information from earlier in the cycle.

```
module.exports = function (content) {
  return someSyncOperation(content, this.data.value);
};

module.exports.pitch = function (remainingRequest, precedingRequest, data) {
  data.value = 42;
};
```

Second, if a loader delivers a result in the `pitch` method, the process turns around and skips the remaining loaders. In our example above, if the `b-loader`s `pitch` method returned something:

```
module.exports = function (content) {
  return someSyncOperation(content);
};

module.exports.pitch = function (remainingRequest, precedingRequest, data) {
  if (someCondition()) {
    return (
      'module.exports = require(' +
      JSON.stringify('-!' + remainingRequest) +
      ');'
    );
```

```
  }
};
```

The steps above would be shortened to:

```
|- a-loader `pitch`
  |- b-loader `pitch` returns a module
|- a-loader normal execution
```

The Loader Context

The loader context represents the properties that are available inside of a loader assigned to the `this` property.

Given the following example, this require call is used:

In `/abc/file.js`:

```
require('./loader1?xyz!loader2!./resource?rrr');
```

`this.addContextDependency`

```
addContextDependency(directory: string)
```

Add a directory as dependency of the loader result.

`this.addDependency`

```
addDependency(file: string)
dependency(file: string) // shortcut
```

Add a file as dependency of the loader result in order to make them watchable. For example, `sass-loader`, `less-loader` uses this to recompile whenever any imported `css` file changes.

`this.async`

Tells the loader-runner that the loader intends to call back asynchronously. Returns `this.callback`.

`this.cacheable`

A function that sets the cacheable flag:

```
cacheable(flag = true: boolean)
```

By default, loader results are flagged as cacheable. Call this method passing `false` to make the loader's result not cacheable.

A cacheable loader must have a deterministic result when inputs and dependencies haven't changed. This means the loader shouldn't have dependencies other than those specified with `this.addDependency`.

`this.callback`

A function that can be called synchronously or asynchronously in order to return multiple results. The expected arguments are:

```
this.callback(
  err: Error | null,
  content: string | Buffer,
  sourceMap?: SourceMap,
  meta?: any
);
```

1. The first argument must be an `Error` or `null`
2. The second argument is a `string` or a `Buffer`.
3. Optional: The third argument must be a source map that is parsable by this module.
4. Optional: The fourth option, ignored by webpack, can be anything (e.g. some metadata).

> **Tip**
>
> It can be useful to pass an abstract syntax tree (AST), like `ESTree`, as the fourth argument (`meta`) to speed up the build time if you want to share common ASTs between loaders.

In case this function is called, you should return undefined to avoid ambiguous loader results.

`this.clearDependencies`

```
clearDependencies();
```

Remove all dependencies of the loader result, even initial dependencies and those of other loaders. Consider using `pitch`.

`this.context`

The directory of the module. Can be used as a context for resolving other stuff.

In the example: `/abc` because `resource.js` is in this directory

`this.data`

A data object shared between the pitch and the normal phase.

`this.emitError`

```
emitError(error: Error)
```

Emit an error that also can be displayed in the output.

```
ERROR in ./src/lib.js (./src/loader.js!./src/lib.js)
Module Error (from ./src/loader.js):
Here is an Error!
 @ ./src/index.js 1:0-25
```

Tip

Unlike throwing an Error directly, it will NOT interrupt the compilation process of the current module.

`this.emitFile`

```
emitFile(name: string, content: Buffer|string, sourceMap: {...})
```

Emit a file. This is webpack-specific.

`this.emitWarning`

```
emitWarning(warning: Error)
```

Emit a warning that will be displayed in the output like the following:

```
WARNING in ./src/lib.js (./src/loader.js!./src/lib.js)
Module Warning (from ./src/loader.js):
Here is a Warning!
 @ ./src/index.js 1:0-25
```

Tip

Note that the warnings will not be displayed if `stats.warnings` is set to `false`, or some other omit setting is used to `stats` such as `none` or `errors-only`. See the stats presets configuration.

`this.fs`

Access to the `compilation`'s `inputFileSystem` property.

`this.getOptions(schema)`

Extracts given loader options. Optionally, accepts JSON schema as an argument.

Tip

Since webpack 5, `this.getOptions` is available in loader context. It substitutes `getOptions` method from loader-utils.

`this.getResolve`

```
getResolve(options: ResolveOptions): resolve
```

```
resolve(context: string, request: string, callback: function(err, result: string))
resolve(context: string, request: string): Promise<string>
```

Creates a resolve function similar to `this.resolve`.

Any options under webpack `resolve` options are possible. They are merged with the configured `resolve` options. Note that `"..."` can be used in arrays to extend the value from `resolve` options, e.g. `{ extensions: [".sass", "..."] }`.

`options.dependencyType` is an additional option. It allows us to specify the type of dependency, which is used to resolve `byDependency` from the `resolve` options.

All dependencies of the resolving operation are automatically added as dependencies to the current module.

this.hot

Information about HMR for loaders.

```
module.exports = function (source) {
  console.log(this.hot); // true if HMR is enabled via --hot flag or webpack configuration
  return source;
};
```

this.loaderIndex

The index in the loaders array of the current loader.

In the example: in loader1: `0`, in loader2: `1`

this.loadModule

```
loadModule(request: string, callback: function(err, source, sourceMap, module))
```

Resolves the given request to a module, applies all configured loaders and calls back with the generated source, the sourceMap and the module instance (usually an instance of `NormalModule`). Use this function if you need to know the source code of another module to generate the result.

`this.loadModule` in a loader context uses CommonJS resolve rules by default. Use `this.getResolve` with an appropriate `dependencyType`, e.g. `'esm'`, `'commonjs'` or a custom one before using a different semantic.

this.loaders

An array of all the loaders. It is writable in the pitch phase.

```
loaders = [{request: string, path: string, query: string, module: function}]
```

In the example:

```
[
  {
    request: '/abc/loader1.js?xyz',
    path: '/abc/loader1.js',
    query: '?xyz',
    module: [Function],
  },
  {
    request: '/abc/node_modules/loader2/index.js',
    path: '/abc/node_modules/loader2/index.js',
    query: '',
    module: [Function],
  },
];
```

this.mode

Read in which `mode` webpack is running.

Possible values: `'production'`, `'development'`, `'none'`

this.query

1. If the loader was configured with an `options` object, this will point to that object.
2. If the loader has no `options`, but was invoked with a query string, this will be a string starting with `?`.

this.request

The resolved request string.

In the example: `'/abc/loader1.js?xyz!/abc/node_modules/loader2/index.js!/abc/resource.js?rrr'`

`this.resolve`

```
resolve(context: string, request: string, callback: function(err, result: string))
```

Resolve a request like a require expression.

- `context` must be an absolute path to a directory. This directory is used as the starting location for the resolving.
- `request` is the request to be resolved. Usually either relative requests like `./relative` or module requests like `module/path` are used, but absolute paths like `/some/path` are also possible as requests.
- `callback` is a normal Node.js-style callback function giving the resolved path.

All dependencies of the resolving operation are automatically added as dependencies to the current module.

`this.resource`

The resource part of the request, including query.

In the example: `'/abc/resource.js?rrr'`

`this.resourcePath`

The resource file.

In the example: `'/abc/resource.js'`

`this.resourceQuery`

The query of the resource.

In the example: `'?rrr'`

`this.rootContext`

Since webpack 4, the formerly `this.options.context` is provided as `this.rootContext`.

`this.sourceMap`

Tells if source map should be generated. Since generating source maps can be an expensive task, you should check if source maps are actually requested.

`this.target`

Target of compilation. Passed from configuration options.

Example values: `'web'`, `'node'`

this.utils

5.27.0+

Access to `contextify` and `absolutify` utilities.

- `contextify`: Return a new request string avoiding absolute paths when possible.
- `absolutify`: Return a new request string using absolute paths when possible.

my-sync-loader.js

```
module.exports = function (content) {
  this.utils.contextify(
    this.context,
    this.utils.absolutify(this.context, './index.js')
  );
  this.utils.absolutify(this.context, this.resourcePath);
  // …
  return content;
};
```

`this.version`

Loader API version. Currently `2`. This is useful for providing backwards compatibility. Using the version you can specify custom logic or fallbacks for breaking changes.

`this.webpack`

This boolean is set to true when this is compiled by webpack.

> **Tip**
>
> Loaders were originally designed to also work as Babel transforms. Therefore, if you write a loader that works for both, you can use this property to know if there is access to additional loaderContext and webpack features.

Webpack specific properties

The loader interface provides all module relate information. However in rare cases you might need access to the compiler api itself.

> **Warning**
>
> Please note that using these webpack specific properties will have a negative impact on your loaders compatibility.
>
> Therefore you should only use them as a last resort. Using them will reduce the portability of your loader.

`this._compilation`

Access to the current Compilation object of webpack.

`this._compiler`

Access to the current Compiler object of webpack.

Deprecated context properties

> **Warning**
>
> The usage of these properties is highly discouraged since we are planning to remove them from the context. They are still listed here for documentation purposes.

`this.debug`

A boolean flag. It is set when in debug mode.

`this.inputValue`

Passed from the last loader. If you would execute the input argument as a module, consider reading this variable for a shortcut (for performance).

`this.minimize`

Tells if result should be minimized.

`this.value`

Pass values to the next loader. If you know what your result exports if executed as a module, set this value here (as an only element array).

`this._module`

Hacky access to the Module object being loaded.

Error Reporting

You can report errors from inside a loader by:

- Using this.emitError. Will report the errors without interrupting module's compilation.
- Using `throw` (or other uncaught exception). Throwing an error while a loader is running will cause current module compilation failure.
- Using `callback` (in async mode). Pass an error to the callback will also cause module compilation failure.

For example:

./src/index.js

```
require('./loader!./lib');
```

Throwing an error from loader:

./src/loader.js

```
module.exports = function (source) {
  throw new Error('This is a Fatal Error!');
};
```

Or pass an error to the callback in async mode:

./src/loader.js

```
module.exports = function (source) {
  const callback = this.async();
  //...
  callback(new Error('This is a Fatal Error!'), source);
};
```

The module will get bundled like this:

```
/***/ "./src/loader.js!./src/lib.js":
/*!*************************************!*\
  !*** ./src/loader.js!./src/lib.js ***!
  \*************************************/
/*! no static exports found */
/***/ (function(module, exports) {

throw new Error("Module build failed (from ./src/loader.js):\nError: This is a Fatal Error!\n    at Object.module.exports (/workspace/src/loader.js:3:9)");

/***/ })
```

Then the build output will also display the error (Similar to `this.emitError`):

```
ERROR in ./src/lib.js (./src/loader.js!./src/lib.js)
Module build failed (from ./src/loader.js):
Error: This is a Fatal Error!
    at Object.module.exports (/workspace/src/loader.js:2:9)
 @ ./src/index.js 1:0-25
```

As you can see below, not only error message, but also details about which loader and module are involved:

- the module path: `ERROR in ./src/lib.js`
- the request string: `(./src/loader.js!./src/lib.js)`
- the loader path: `(from ./src/loader.js)`
- the caller path: `@ ./src/index.js 1:0-25`

> **Warning**
>
> The loader path in the error is displayed since webpack 4.12

> **Tip**
>
> All the errors and warnings will be recorded into `stats`. Please see Stats Data.

Inline matchResource

A new inline request syntax was introduced in webpack v4. Prefixing `<match-resource>!=!` to a request will set the `matchResource` for this request.

> **Warning**
>
> It is not recommended to use this syntax in application code. Inline request syntax is intended to only be used by loader generated code. Not following this recommendation will make your code webpack-specific and non-standard.

> **Tip**
>
> A relative `matchResource` will resolve relative to the current context of the containing module.

When a `matchResource` is set, it will be used to match with the `module.rules` instead of the original resource. This can be useful if further loaders should be applied to the resource, or if the module type needs to be changed. It's also displayed in the stats and used for matching `Rule.issuer` and `test` in `splitChunks`.

Example:

file.js

```
/* STYLE: body { background: red; } */
console.log('yep');
```

A loader could transform the file into the following file and use the `matchResource` to apply the user-specified CSS processing rules:

file.js (transformed by loader)

```
import './file.js.css!=!extract-style-loader/getStyles!./file.js';
console.log('yep');
```

This will add a dependency to `extract-style-loader/getStyles!./file.js` and treat the result as `file.js.css`. Because `module.rules` has a rule matching `/\.css$/` and it will apply to this dependency.

The loader could look like this:

extract-style-loader/index.js

```
const stringifyRequest = require('loader-utils').stringifyRequest;
const getRemainingRequest = require('loader-utils').getRemainingRequest;
const getStylesLoader = require.resolve('./getStyle');

module.exports = function (source) {
  if (STYLES_REGEXP.test(source)) {
    source = source.replace(STYLES_REGEXP, '');
    const remReq = getRemainingRequest(this);
    return `import ${stringifyRequest(
      `${this.resource}.css!=!${getStylesLoader}!${remReq}`
    )};${source}`;
  }
  return source;
};
```

extract-style-loader/getStyles.js

```
module.exports = function (source) {
  const match = STYLES_REGEXP.match(source);
  return match[0];
};
```

Logging

Logging API is available since the release of webpack 4.37. When `logging` is enabled in `stats configuration` and/or when `infrastructure logging` is enabled, loaders may log messages which will be printed out in the respective logger format (stats, infrastructure).

- Loaders should prefer to use `this.getLogger()` for logging which is a shortcut to `compilation.getLogger()` with loader path and processed file. This kind of logging is stored to the Stats and formatted accordingly. It can be filtered and exported by the webpack user.
- Loaders may use `this.getLogger('name')` to get an independent logger with a child name. Loader path and processed file is still added.
- Loaders may use special fallback logic for detecting logging support `this.getLogger() ? this.getLogger() : console` to provide a fallback when an older webpack version is used which does not support `getLogger` method.