Webpack 5 release (2020-10-10)

webpack 4 was released in February 2018. Since then we shipped a lot of features without breaking changes. We know that people dislike major changes with breaking changes. Especially with webpack, which people usually only touch twice a year, and the remaining time it "just works". But shipping features without breaking changes also has a cost: We can't do major API or architectural improvements.

So from time to time, there is a point where the difficulties pile up and we are forced to do breaking changes to not mess everything up. That's the time for a new major version. So webpack 5 contains these architectural improvements and the features that were not possible to implement without them.

The major version was also the chance to revise some of the defaults and to align with proposals and specifications that come up in the meantime.

So today (2020-10-10) webpack 5.0.0 is released, but this doesn't mean it's done, bugfree or even feature-complete. As with webpack 4 we continue development by fixing problems and adding features. In the next days there will probably be a lot bugfixes. Features will come later.

Common Questions

So what does the release mean?

It means we finished doing breaking changes. Many refactorings have been done to up-level the architecture and create a good base for future features (and current features).

So when is the time to upgrade?

It depends. There is a good chance that upgrading fails and you would need to give it a second or 3rd try. If you are open to that, try to upgrade now and provide feedback to webpack, plugins and loaders. We are eager to fix those problems. Someone has to start and you would be one of the first ones benefiting from it.

Sponsoring Update

webpack is fully based upon sponsoring. It's not tied to (and paid by) a big company like some other Open Source projects. 99% of the earnings from sponsoring are distributed towards contributors and maintainers based on the contributions they do. We believe in investing the money towards making webpack better.

But there is a pandemic, and companies ain't that much open to sponsoring anymore. Webpack is suffering under these circumstances too (like many other companies and people).

We were never able to pay our contributors the amount we think they deserve, but now we only have half of the money available, so we need to make a more serious cut. Until the situation improves we will only pay contributors and maintainers the first 10 days of each month. The remaining days they could work voluntarily, paid by their employer, work on something else, or take some days off. This allows us to pay for their work in the first 10 days more equivalent to the invested time.

The biggest "Thank You" goes to trivago which has been sponsoring webpack a huge amount for the last 3 years. Sadly they are unable to continue their sponsorship this year, as they have been hit hard by Covid-19. I hope some other company steps up and follows these (gigantic) footsteps.

Thanks to all the sponsors.

General direction

This release focus on the following:

- Improve build performance with Persistent Caching.
- Improve Long Term Caching with better algorithms and defaults.
- Improve bundle size with better Tree Shaking and Code Generation.
- Improve compatibility with the web platform.
- Clean up internal structures that were left in a weird state while implementing features in v4 without introducing any breaking changes.
- Prepare for future features by introducing breaking changes now, allowing us to stay on v5 for as long as possible.

Migration Guide

See here for a **migration** guide

Major Changes: Removals

Removed Deprecated Items

All items deprecated in v4 were removed.

MIGRATION: Make sure that your webpack 4 build does not print deprecation warnings.

Here are a few things that were removed but did not have deprecation warnings in v4:

- IgnorePlugin and BannerPlugin must now be passed only one argument that can be an object, string or function.

Deprecation codes

New deprecations include a deprecation code so they are easier to reference.

Syntax deprecated

`require.include` has been deprecated and will emit a warning by default when used.

Behavior can be changed with `Rule.parser.requireInclude` to allowed, deprecated or disabled.

Automatic Node.js Polyfills Removed

In the early days, webpack's aim was to allow running most Node.js modules in the browser, but the module landscape changed and many module uses are now written mainly for frontend purposes. webpack <= 4 ships with polyfills for many of the Node.js core modules, which are automatically applied once a module uses any of the core modules (i.e. the `crypto` module).

While this makes using modules written for Node.js easy, it adds these huge polyfills to the bundle. In many cases these polyfills are unnecessary.

webpack 5 stops automatically polyfilling these core modules and focus on frontend-compatible modules. Our goal is to improve compatibility with the web platform, where Node.js core modules are not available.

MIGRATION:

- Try to use frontend-compatible modules whenever possible.
- It's possible to manually add a polyfill for a Node.js core module. An error message will give a hint on how to achieve that.
- Package authors: Use the `browser` field in `package.json` to make a package frontend-compatible. Provide alternative implementations/dependencies for the browser.

Major Changes: Long Term Caching

Deterministic Chunk, Module IDs and Export names

New algorithms were added for long term caching. These are enabled by default in production mode.

`chunkIds: "deterministic"` `moduleIds: "deterministic"` `mangleExports: "deterministic"`

The algorithms assign short (3 or 5 digits) numeric IDs to modules and chunks and short (2 characters) names to exports in a deterministic way. This is a trade-off between bundle size and long term caching.

`moduleIds/chunkIds/mangleExports: false` disables the default behavior and one can provide a custom algorithm via plugin. Note that in webpack 4 `moduleIds/chunkIds: false` without custom plugin resulted in a working build, while in webpack 5 you must provide a custom plugin.

MIGRATION: Best use the default values for `chunkIds`, `moduleIds` and `mangleExports`. You can also opt-in to the old defaults `chunkIds: "size", moduleIds: "size", mangleExports: "size"`, this will generate smaller bundles, but invalidate them more often for caching.

Note: In webpack 4 hashed module ids yielded reduced gzip performance. This was related to changed module order and has been fixed.

Note: In webpack 5, `deterministic` Ids are enabled by default in production mode

Real Content Hash

Webpack 5 will use a real hash of the file content when using `[contenthash]` now. Before it "only" used a hash of the internal structure. This can be positive impact on long term caching when only comments are changed or variables are renamed. These changes are not visible after minimizing.

Major Changes: Development Support

Named Chunk IDs

A new named chunk id algorithm enabled by default in development mode gives chunks (and filenames) human-readable names. A Module ID is determined by its path, relative to the `context`. A Chunk ID is determined by the chunk's content.

So you no longer need to use `import(/* webpackChunkName: "name" */ "module")` for debugging. But it would still make sense if you want to control the filenames for production environments.

It's possible to use `chunkIds: "named"` in production, but make sure not to accidentally expose sensitive information about module names.

MIGRATION: If you dislike the filenames being changed in development, you can pass `chunkIds: "natural"` to use the old numeric mode.

Module Federation

Webpack 5 adds a new feature called "Module Federation", which allows multiple webpack builds to work together. From runtime perspective modules from multiple builds will behave like a huge connected module graph. From developer perspective modules can be imported from specified remote builds and used with minimal restrictions.

For more details see this separate guide.

Major Changes: New Web Platform Features

JSON modules

JSON modules now align with the proposal and emit a warning when a non-default export is used. JSON modules no longer have named exports when importing from a strict ECMAScript module.

MIGRATION: Use the default export.

Even when using the default export, unused properties are dropped by the `optimization.usedExports` optimization and properties are mangled by the `optimization.mangleExports` optimization.

It's possible to specify a custom JSON parser in `Rule.parser.parse` to import JSON-like files (e.g. for toml, yaml, json5, etc.).

import.meta

- `import.meta.webpackHot` is an alias for `module.hot` which is also available in strict ESM
- `import.meta.webpack` is the webpack major version as number
- `import.meta.url` is the `file:` url of the current file (similar to `__filename` but as file url)

Asset modules

Webpack 5 has now native support for modules representing assets. These modules will either emit a file into the output folder or inject a DataURI into the javascript bundle. Either way they give a URL to work with.

They can be used via multiple ways:

- `import url from "./image.png"` and setting `type: "asset"` in `module.rules` when matching such import. (old way)
- `new URL("./image.png", import.meta.url)` (new way)

The "new way" syntax was chosen to allow running code without bundler too. This syntax is also available in native ECMAScript modules in the browser.

Native Worker support

When combining `new URL` for assets with `new Worker` / `new SharedWorker` / `navigator.serviceWorker.register` webpack will automatically create a new entrypoint for a web worker.

`new Worker(new URL("./worker.js", import.meta.url))`

The syntax was chosen to allow running code without bundler too. This syntax is also available in native ECMAScript modules in the browser.

URIs

Webpack 5 supports handling of protocols in requests.

- `data:` is supported. Base64 or raw encoding is supported. Mimetype can be mapped to loaders and module type in `module.rules`. Example: `import x from "data:text/javascript,export default 42"`
- `file:` is supported.

- `http(s):` is supported, but requires opt-in via `new webpack.experiments.schemesHttp(s)UriPlugin()`
    - By default when targeting "web", these URIs result in requests to external resource (they are externals)

Fragments in requests are supported: Example: `./file.js#fragment`

Async modules

Webpack 5 supports so called "async modules". That are modules that do not evaluate synchronously, but are async and Promise-based instead.

Importing them via `import` is automatically handled and no additional syntax is needed and difference is hardly noticeable.

Importing them via `require()` will return a Promise that resolves to the exports.

In webpack there are multiple ways to have async modules:

- async externals
- WebAssembly Modules in the new spec
- ECMAScript Modules that are using Top-Level-Await

Externals

Webpack 5 adds additional external types to cover more applications:

`promise`: An expression that evaluates to a Promise. The external module is an async module and the resolved value is used as module exports.

`import`: Native `import()` is used to load the specified request. The external module is an async module.

`module`: Not implemented yet, but planned to load modules via `import x from "..."`.

`script`: Loads a url via `<script>` tag and gets the exports from a global variable (and optionally properties of it). The external module is an async module.

Major Changes: New Node.js Ecosystem Features

Resolving

The `exports` and `imports` field in package.json is now supported.

Yarn PnP is supported natively.

See more details in package exports.

Major Changes: Development Experience

Improved target

Webpack 5 allows to pass a list of targets and also support versions of target.

Examples: `target: "node14"` `target: ["web", "es2020"]`

This is a simple way to provide webpack all the information it needs to determine:

- chunk loading mechanism, and
- supported syntax like arrow functions

Stats

The Stats test format has been improved regarding readability and verbosity. The defaults have been improved to be less verbose and also suitable for large builds.

- Chunk relations are hidden by default now. This can be toggled with `stats.chunkRelations`.
- Stats differentiate between `files` and `auxiliaryFiles` now.
- Stats hide module and chunk ids by default now. This can be toggled with `stats.ids`.
- The list of all modules is sorted by distance to entrypoint now. This can be changed with `stats.modulesSort`.
- The list of chunk modules is sorted by module name now. This can be changed with `stats.chunkModulesSort`.
- The list of nested modules in concatenated modules is sorted topologically now. This can be changed with `stats.nestedModulesSort`.
- Chunks and Assets show chunk id hints now.
- Assets and modules will display in a tree instead of a list/table.
- General information is shown in a summary at the end now. It shows webpack version, config name and warnings/errors count.
- Hash is hidden by default now. This can be changed with `stats.hash`.
- Timestamp of build is no longer shown by default. This can be enabled with `stats.builtAt`. It will show the timestamp in the summary.
- Child compilations will no longer shown by default. They can be displayed with `stats.children`.

Progress

A few improvements have been done to the `ProgressPlugin` which is used for `--progress` by the CLI, but can also be used manually as plugin.

It used to only count the processed modules. Now it can count `entries` `dependencies` and `modules`. All of them are shown by default now.

It used to display the currently processed module. This caused much stderr output and yielded a performance problem on some consoles. This is now disabled by default (`activeModules` option). This also reduces the amount of spam on the console. Now writing to stderr during building modules is throttled to 500ms.

The profiling mode also got an upgrade and will display timings of nested progress messages. This makes it easier to figure out which plugin is causing performance problems.

A newly added `percentBy` -option tells `ProgressPlugin` how to calculate progress percentage.

```
new webpack.ProgressPlugin({ percentBy: 'entries' });
```

To make progress percentage more accurate `ProgressPlugin` caches the last known total modules count and reuses this value on the next build. The first build will warm the cache but the following builds will use and update this value.

Automatic unique naming

In webpack 4 multiple webpack runtimes could conflict on the same HTML page, because they use the same global variable for chunk loading. To fix that it was needed to provide a custom name to the `output.jsonpFunction` configuration.

Webpack 5 does automatically infer a unique name for the build from `package.json` `name` and uses this as default for `output.uniqueName` .

This value is used to make all potential conflicting globals unique.

MIGRATION: Remove `output.jsonpFunction` in favor of a unique name in your `package.json` .

Automatic public path

Webpack 5 will determine the `output.publicPath` automatically when possible.

Typescript typings

Webpack 5 generates typescript typings from source code and exposes them via the npm package.

MIGRATION: Remove `@types/webpack` . Update references when names differ.

Major Changes: Optimization

Nested tree-shaking

webpack is now able to track access to nested properties of exports. This can improve Tree Shaking (Unused export elimination and export mangling) when reexporting namespace objects.

```js
// inner.js
export const a = 1;
export const b = 2;

// module.js
export * as inner from './inner';
// or import * as inner from './inner'; export { inner };

// user.js
import * as module from './module';
console.log(module.inner.a);
```

In this example, the export `b` can be removed in production mode.

Inner-module tree-shaking

webpack 4 didn't analyze dependencies between exports and imports of a module. webpack 5 has a new option `optimization.innerGraph`, which is enabled by default in production mode, that runs an analysis on symbols in a module to figure out dependencies from exports to imports.

In a module like this:

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

function usingSomething() {
  return something;
}

export function test() {
  return usingSomething();
}
```

The inner graph algorithm will figure out that `something` is only used when the `test` export is used. This allows to flag more exports as unused and to omit more code from the bundle.

When `"sideEffects": false` is set, this allows to omit even more modules. In this example `./something` will be omitted when the `test` export is unused.

To get the information about unused exports `optimization.unusedExports` is required. To remove side-effect-free modules `optimization.sideEffects` is required.

The following symbols can be analysed:

- function declarations
- class declarations
- `export default` with or variable declarations with
  - function expressions
  - class expressions
  - sequence expressions

    - `/*#__PURE__*/` expressions
    - local variables
    - imported bindings

FEEDBACK: If you find something missing in this analysis, please report an issue and we consider adding it.

Using `eval()` will bail-out this optimization for a module, because evaled code could reference any symbol in scope.

This optimization is also known as Deep Scope Analysis.

CommonJs Tree Shaking

webpack used to opt-out from used exports analysing for CommonJs exports and `require()` calls.

webpack 5 adds support for some CommonJs constructs, allows to eliminate unused CommonJs exports and track referenced export names from `require()` calls.

The following constructs are supported:

- `exports|this|module.exports.xxx = ...`
- `exports|this|module.exports = require("...")` (reexport)
- `exports|this|module.exports.xxx = require("...").xxx` (reexport)
- `Object.defineProperty(exports|this|module.exports, "xxx", ...)`
- `require("abc").xxx`
- `require("abc").xxx()`
- importing from ESM
- `require()` a ESM
- flagged exportType (special handling for non-strict ESM import):
    - `Object.defineProperty(exports|this|module.exports, "__esModule", { value: true|!0 })`
    - `exports|this|module.exports.__esModule = true|!0`
- It's planned to support more constructs in future

When detecting not analysable code, webpack bails out and doesn't track export information at all for these modules (for performance reasons).

Side-Effect analysis

The `"sideEffects"` flag in package.json allows to manually flag modules as side-effect-free, which allows to drop them when unused.

webpack 5 can also automatically flag modules as side-effect-free according to a static analysis of the source code.

Optimization per runtime

Webpack 5 is now able (and does by default) to analyse and optimize modules per runtime (A runtime is often equal to an entrypoint). This allows to only exports in these entrypoints where they are really needed. Entrypoints doesn't affect each other (as long as using a runtime per entrypoint)

Module Concatenation

Module Concatenation also works per runtime to allow different concatenation for each runtime.

Module Concatenation has become a first class citizen and any module and dependency is now allowed to implement it. Initially webpack 5 already added support for ExternalModules and json modules, more will likely ship soonish.

General Tree Shaking improvements

`export *` has been improved to track more info and do no longer flag the `default` export as used.

`export *` will now show warnings when webpack is sure that there are conflicting exports.

`import()` allows to manually tree shake the module via `/* webpackExports: ["abc", "default"] */` magic comment.

Development Production Similarity

We try to find a good trade-off between build performance in development mode and avoiding production-only problems by improving the similarity between both modes.

Webpack 5 enables the `sideEffects` optimization by default in both modes. In webpack 4 this optimization lead to some production-only errors because of an incorrect `"sideEffects"` flag in package.json. Enabling this optimization in development allows to find these problems faster and easier.

In many cases development and production happen on different OS with different case-sensitivity of filesystem, so webpack 5 adds a few more warnings/errors when there is something weird casing-wise.

Improved Code Generation

webpack detects when ASI happens and generates shorter code when no semicolons are inserted. `Object(...)` -> `(0, ...)`

webpack merges multiple export getters into a single runtime function call: `r.d(x, "a", () => a); r.d(x, "b", () => b);` -> `r.d(x, {a: () => a, b: () => b});`

There are additional options in `output.environment` now. They allows specifying which ECMAScript feature can be used for runtime code generated by webpack.

One usually do not specify this option directly, but would use the `target` option instead.

webpack 4 used to only emit ES5 code. webpack 5 can generate both ES5 and ES6/ES2015 code now.

Supporting only modern browsers will generate shorter code using arrow functions and more spec-conform code using `const` declarations with TDZ for `export default`.

Improved `target` option

In webpack 4 the `target` was a rough choice between `"web"` and `"node"` (and a few others). Webpack 5 gives you more options here.

The `target` option now influences more things about the generated code than before:

- method of chunk loading
- format of chunks
- method of wasm loading
- method of chunk and wasm loading in workers
- global object used
- if publicPath should be determined automatically
- ECMAScript features/syntax used in the generated code
- `externals` enabled by default
- Behavior of some Node.js compat layers (`global` , `__filename` , `__dirname`)
- Resolving of modules (`browser` field, `exports` and `imports` conditions)
- Some loaders might change behavior based on that

For some of these things the choice between `"web"` and `"node"` is too rough and we need more information. Therefore we allow to specify the minimum version e.g. like `"node10.13"` and infer more properties about the target environment.

It's now also allowed to combined multiple targets with an array and webpack will determine the minimum properties of all targets. Using an array is also useful when using targets that doesn't give full information like `"web"` or `"node"` (without version number). E. g. `["web", "es2020"]` combines these two partial targets.

There is a target `"browserslist"` which will use browserslist data to determine properties of the environment. This target is also used by default when there is a browserslist config available in the project. When none such config is available, the `"web"` target will be used by default.

Some combinations and features are not yet implemented and will result in errors. They are preparations for future features. Examples:

- `["web", "node"]` will lead to an universal chunk loading method, which is not implemented yet
- `["web", "node"]` + `output.module: true` will lead to a module chunk loading method, which is not implemented yet
- `"web"` will lead to `http(s):` imports being treated as `module` externals, which are not implemented yet (Workaround: `externalsPresets: { web: false, webAsync: true }`, which will use `import()` instead).

SplitChunks and Module Sizes

Modules now express size in a better way than a single number. There are different types of sizes now.

The SplitChunksPlugin now knows how to handle these different sizes and uses them for `minSize` and `maxSize`. By default, only `javascript` size is handled, but you can now pass multiple values to manage them:

```
module.exports = {
  optimization: {
    splitChunks: {
      minSize: {
        javascript: 30000,
        webassembly: 50000,
      },
    },
  },
};
```

You can still use a single number for sizes. In this case webpack will automatically use the default size types.

The `mini-css-extract-plugin` uses `css/mini-extra` as size type, and adds this size type to the default types automatically.

Major Changes: Performance

Persistent Caching

There is now a filesystem cache. It's opt-in and can be enabled with the following configuration:

```
module.exports = {
  cache: {
    // 1. Set cache type to filesystem
    type: 'filesystem',

    buildDependencies: {
      // 2. Add your config as buildDependency to get cache invalidation on config change
      config: [__filename],

      // 3. If you have other things the build depends on you can add them here
      // Note that webpack, loaders and all modules referenced from your config are automatically added
    },
  },
};
```

Important notes:

By default, webpack assumes that the `node_modules` directory, which webpack is inside of, is **only** modified by a package manager. Hashing and timestamping is skipped for `node_modules`. Instead, only the package name and version is used for performance reasons. Symlinks (i. e. `npm/yarn link`) are fine as long `resolve.symlinks: false` is not specified (avoid that anyway). Do not edit files in `node_modules` directly unless you opt-out of this optimization with `snapshot.managedPaths: []`. When using Yarn PnP webpack assumes that the yarn cache is immutable (which it usually is). You can opt-out of this optimization with `snapshot.immutablePaths: []`

The cache will be stored into `node_modules/.cache/webpack` (when using node_modules) resp. `.yarn/.cache/webpack` (when using Yarn PnP) by default. You probably never have to delete it manually, when all plugins handle caching correctly.

Many internal plugins will use the Persistent Cache too. Examples: `SourceMapDevToolPlugin` (to cache the SourceMap generation) or `ProgressPlugin` (to cache the number of modules)

The Persistent Cache will automatically create multiple cache files depending on usage to optimize read and write access to and from the cache.

By default timestamps will be used for snapshotting in development mode and file hashes in production mode. File hashes allow to use Persistent Caching on CI too.

Compiler Idle and Close

Compilers now need to be closed after being used. Compilers now enter and leave the idle state and have hooks for these states. Plugins may use these hooks to do unimportant work. (i. e. the Persistent cache slowly stores the cache to disk). On compiler close - All remaining work should be finished as fast as possible. A callback signals the closing as done.

Plugins and their respective authors should expect that some users may forget to close the Compiler. So, all work should eventually be finishing while in idle too. Processes should be prevented from exiting when the work is being done.

The `webpack()` façade automatically calls `close` when being passed a callback.

MIGRATION: While using the Node.js API, make sure to call `Compiler.close` when done.

File Emitting

webpack used to always emit all output files during the first build but skipped writing unchanged files during incremental (watch) builds. It is assumed that nothing else changes output files while webpack is running.

With Persistent Caching added a watch-like experience should be given even when restarting the webpack process, but it would be a too strong assumption to think that nothing else changes the output directory even when webpack is not running.

So webpack will now check existing files in the output directory and compares their content with the output file in memory. It will only write the file when it has been changed. This is only done on the first build. Any incremental build will always write the file when a new asset has been generated in the running webpack process.

We assume that webpack and plugins only generate new assets when content has been changed. Caching should be used to ensure that no new asset is generated when input is equal. Not following this advice will degrade performance.

Files that are flagged as `[immutable]` (including a content hash), will never be written when a file with the same name already exists. We assume that the content hash will change when file content changes. This is true in general, but might not be always true during webpack or plugin development.

Major Changes: Long outstanding problems

Code Splitting for single-file-targets

Targets that only allow to startup a single file (like node, WebWorker, electron main) now supports loading the dependent pieces required for bootstrapping automatically by the runtime.

This allows using `opimization.splitChunks` for these targets with `chunks: "all"` and also `optimization.runtimeChunk`

Note that with targets where chunk loading is async, this makes initial evaluation async too. This can be an issue when using `output.library`, since the exported value is a Promise now.

Updated Resolver

`enhanced-resolve` was updated to v5. This has the following improvements:

- The resolve tracks more dependencies, like missing files
- aliasing may have multiple alternatives
- aliasing to `false` is possible now
- support for features like `exports` and `imports` fields
- Increased performance

Chunks without JS

Chunks that contain no JS code, will no longer generate a JS file. This allows to have chunks that contain only CSS.

Major Changes: Future

Experiments

Not all features are stable from the beginning. In webpack 4 we added experimental features and noted in the changelog that they are experimental, but it was not always clear from the configuration that these features are experimental.

In webpack 5 there is a new `experiments` config option which allows to enable experimental features. This makes it clear which ones are enabled/used.

While webpack follows semantic versioning, it will make an exception for experimental features. Experimental features might contain breaking changes in minor webpack versions. When this happens we will add a clear note into the changelog. This will allow us to iterate faster for experimental features, while also allowing us to stay longer on a major version for stable features.

The following experiments will ship with webpack 5:

- Old WebAssembly support like in webpack 4 (`experiments.syncWebAssembly`)
- New WebAssembly support according to the updated spec (`experiments.asyncWebAssembly`)
  - This makes a WebAssembly module an async module
- Top Level Await Stage 3 proposal (`experiments.topLevelAwait`)
  - Using `await` on top-level makes the module an async module
- Emitting bundle as module (`experiments.outputModule`)
  - This removed the wrapper IIFE from the bundle, enforces strict mode, lazy loads via `<script type="module">` and minimized in module mode

Note that this also means WebAssembly support is now disabled by default.

Minimum Node.js Version

The minimum supported Node.js version has increased from 6 to 10.13.0(LTS).

MIGRATION: Upgrade to the latest Node.js version available.

Changes to the Configuration

Changes to the Structure

- `entry: {}` allows an empty object now (to allow to use plugins to add entries)
- `target` supports an array, versions and browserslist
- `cache: Object` removed: Setting to a memory-cache object is no longer possible
- `cache.type` added: It's now possible to choose between `"memory"` and `"filesystem"`
- New configuration options for `cache.type = "filesystem"` added:
  - `cache.cacheDirectory`
  - `cache.name`
  - `cache.version`
  - `cache.store`
  - `cache.hashAlgorithm`
  - `cache.idleTimeout`
  - `cache.idleTimeoutForInitialStore`
  - `cache.buildDependencies`
- `snapshot.resolveBuildDependencies` added

- `snapshot.resolve` added
- `snapshot.module` added
- `snapshot.managedPaths` added
- `snapshot.immutablePaths` added
- `resolve.cache` added: Allows to disable/enable the safe resolve cache
- `resolve.concord` removed
- `resolve.alias` values can be arrays or `false` now
- `resolve.restrictions` added: Allows to restrict potential resolve results
- `resolve.fallback` added: Allow to alias requests that failed to resolve
- `resolve.preferRelative` added: Allows to resolve modules requests are relative requests too
- Automatic polyfills for native Node.js modules were removed
  - `node.Buffer` removed
  - `node.console` removed
  - `node.process` removed
  - `node.*` (Node.js native module) removed
  - **MIGRATION**: `resolve.alias` and `ProvidePlugin` . Errors will give hints. (Refer to node-libs-browser for polyfills & mocks used in v4)
- `output.filename` can now be a function
- `output.assetModuleFilename` added
- `output.jsonpScriptType` renamed to `output.scriptType`
- `devtool` is more strict
  - Format: `false | eval | [inline-|hidden-|eval-][nosources-][cheap-[module-]]source-map`
- `optimization.chunkIds: "deterministic"` added
- `optimization.moduleIds: "deterministic"` added
- `optimization.moduleIds: "hashed"` deprecated
- `optimization.moduleIds: "total-size"` removed
- Deprecated flags for module and chunk ids were removed
  - `optimization.hashedModuleIds` removed
  - `optimization.namedChunks` removed (`NamedChunksPlugin` too)
  - `optimization.namedModules` removed (`NamedModulesPlugin` too)
  - `optimization.occurrenceOrder` removed
  - **MIGRATION**: Use `chunkIds` and `moduleIds`
- `optimization.splitChunks` `test` no longer matches chunk name

  - **MIGRATION**: Use a test function `(module, { chunkGraph }) => chunkGraph.getModuleChunks(module).some(chunk => chunk.name === "name")`
- `optimization.splitChunks` `minRemainingSize` was added
- `optimization.splitChunks` `filename` can now be a function
- `optimization.splitChunks` sizes can now be objects with a size per source type
  - `minSize`
  - `minRemainingSize`
  - `maxSize`
  - `maxAsyncSize`
  - `maxInitialSize`
- `optimization.splitChunks` `maxAsyncSize` and `maxInitialSize` added next to `maxSize`: allows to specify different max sizes for initial and async chunks
- `optimization.splitChunks` `name: true` removed: Automatic names are no longer supported
  - **MIGRATION**: Use the default. `chunkIds: "named"` will give your files useful names for debugging
- `optimization.splitChunks.cacheGroups[].idHint` added: Gives a hint how the named chunk id should be chosen
- `optimization.splitChunks` `automaticNamePrefix` removed
  - **MIGRATION**: Use `idHint` instead
- `optimization.splitChunks` `filename` is no longer restricted to initial chunks
- `optimization.splitChunks` `usedExports` added to include used exports when comparing modules
- `optimization.splitChunks.defaultSizeTypes` added: Specified the size types when using numbers for sizes
- `optimization.mangleExports` added
- `optimization.minimizer` `"..."` can be used to reference the defaults
- `optimization.usedExports` `"global"` value added to allow to disable the analysis per runtime and instead do it globally (better performance)
- `optimization.noEmitOnErrors` renamed to `optimization.emitOnErrors` and logic inverted
- `optimization.realContentHash` added
- `output.devtoolLineToLine` removed
  - **MIGRATION**: No replacement
- `output.chunkFilename: Function` is now allowed
- `output.hotUpdateChunkFilename: Function` is now forbidden: It never worked anyway.
- `output.hotUpdateMainFilename: Function` is now forbidden: It never worked anyway.
- `output.importFunctionName: string` specifies the name used as replacement for `import()` to allow polyfilling for non-supported environments

- `output.charset` added: setting it to false omit the `charset` property on script tags
- `output.hotUpdateFunction` renamed to `output.hotUpdateGlobal`
- `output.jsonpFunction` renamed to `output.chunkLoadingGlobal`
- `output.chunkCallbackFunction` renamed to `output.chunkLoadingGlobal`
- `output.chunkLoading` added
- `output.enabledChunkLoadingTypes` added
- `output.chunkFormat` added
- `module.rules` `resolve` and `parser` will merge in a different way (objects are deeply merged, array may include `"..."` to reference to prev value)
- `module.rules` `parser.worker` added: Allows to configure the worker supported
- `module.rules` `query` and `loaders` were removed
- `module.rules` `options` passing a string is deprecated
  - **MIGRATION**: Pass an options object instead, open an issue on the loader when this is not supported
- `module.rules` `mimetype` added: allows to match a mimetype of a DataURI
- `module.rules` `descriptionData` added: allows to match a data from package.json
- `module.defaultRules` `"..."` can be used to reference the defaults
- `stats.chunkRootModules` added: Show root modules for chunks
- `stats.orphanModules` added: Show modules which are not emitted
- `stats.runtime` added: Show runtime modules
- `stats.chunkRelations` added: Show parent/children/sibling chunks
- `stats.errorStack` added: Show webpack-internal stack trace of errors
- `stats.preset` added: select a preset
- `stats.relatedAssets` added: show assets that are related to other assets (e.g. SourceMaps)
- `stats.warningsFilter` deprecated in favor of `ignoreWarnings`
- `BannerPlugin.banner` signature changed
  - `data.basename` removed
  - `data.query` removed
  - **MIGRATION**: extract from `filename`
- `SourceMapDevToolPlugin` `lineToLine` removed
  - **MIGRATION**: No replacement
- `[hash]` as hash for the full compilation is now deprecated
  - **MIGRATION**: Use `[fullhash]` instead or better use another hash option
- `[modulehash]` is deprecated

  - **MIGRATION**: Use `[hash]` instead
- `[moduleid]` is deprecated
  - **MIGRATION**: Use `[id]` instead
- `[filebase]` removed
  - **MIGRATION**: Use `[base]` instead
- New placeholders for file-based templates (i. e. SourceMapDevToolPlugin)
  - `[name]`
  - `[base]`
  - `[path]`
  - `[ext]`
- `externals` when passing a function, it has now a different signature `({ context, request }, callback)`
  - **MIGRATION**: Change signature
- `externalsPresets` added
- `experiments` added (see Experiments section above)
- `watchOptions.followSymlinks` added
- `watchOptions.ignored` can now be a RegExp
- `webpack.util.serialization` is now exposed.

Changes to the Defaults

- `target` is now `"browserslist"` by default when a browserslist config is available
- `module.unsafeCache` is now only enabled for `node_modules` by default
- `optimization.moduleIds` defaults to `deterministic` in production mode, instead of `size`
- `optimization.chunkIds` defaults to `deterministic` in production mode, instead of `total-size`
- `optimization.nodeEnv` defaults to `false` in `none` mode
- `optimization.splitChunks.minSize` defaults to `20k` in production
- `optimization.splitChunks.enforceSizeThreshold` defaults to `50k` in production
- `optimization.splitChunks` `minRemainingSize` defaults to `minSize`
  - This will lead to less splitted chunks created in cases where the remaining part would be too small
- `optimization.splitChunks` `maxAsyncRequests` and `maxInitialRequests` defaults was been increased to 30
- `optimization.splitChunks.cacheGroups.vendors` has be renamed to `optimization.splitChunks.cacheGroups.defaultVendors`
- `optimization.splitChunks.cacheGroups.defaultVendors.reuseExistingChunk` now defaults to `true`
- `optimization.minimizer` target default uses `compress.passes: 2` in terser options now

- `resolve(Loader).cache` defaults to `true` when `cache` is used
- `resolve(Loader).cacheWithContext` defaults to `false`
- `resolveLoader.extensions` remove `.json`
- `node.global` `node.__filename` and `node.__dirname` defaults to `false` in node-`target`s
- `stats.errorStack` defaults to `false`

Loader related Changes

`this.getOptions`

This new API should simplify the usage for options in loaders. It allows to pass a JSON schema for validation. See PR for details

`this.exec`

This has been removed from loader context

MIGRATION: This can be implemented in the loader itself

`this.getResolve`

`getResolve(options)` in the loader API will merge options in a different way, see `module.rules` `resolve`.

As webpack 5 differs between different issuing dependencies so it might make sense to pass a `dependencyType` as option (e.g. `"esm"`, `"commonjs"`, or others).

Major Internal Changes

Todo

This section might need a bit more refinement

The following changes are only relevant for plugin authors:

New plugin order

Plugins in webpack 5 are now applies **before** the configuration defaults has been applied. This allows plugins to apply their own defaults, or act as configuration presets.

But this is also a breaking change as plugins can't rely on configuration values to be set when they are applied.

MIGRATION: Access configuration only in plugin hooks. Or best avoid accessing configuration at all and take options via constructor.

Runtime Modules

A large part of the runtime code was moved into the so-called "runtime modules". These special modules are in-charge of adding runtime code. They can be added into any chunk, but are currently always added to the runtime chunk. "Runtime Requirements" control which runtime modules (or core runtime parts) are added to the bundle. This ensures that only runtime code that is used is added to the bundle. In the future, runtime modules could also be added to an on-demand-loaded chunk, to load runtime code when needed.

In most cases, the core runtime allows to inline the entry module instead of calling it with `__webpack_require__`. If there is no other module in the bundle, no `__webpack_require__` is needed at all. This combines well with Module Concatenation where multiple modules are concatenated into a single module.

In the best case, no runtime code is needed at all.

MIGRATION: If you are injecting runtime code into the webpack runtime in a plugin, consider using RuntimeModules instead.

Serialization

A serialization mechanism was added to allow serialization of complex objects in webpack. It has an opt-in semantic, so classes that should be serialized need to be explicitly flagged (and their serialization implemented). This has been done for most Modules, all Dependencies and some Errors.

MIGRATION: When using custom Modules or Dependencies, it is recommended to make them serializable to benefit from persistent caching.

Plugins for Caching

A `Cache` class with a plugin interface has been added. This class can be used to write and read to the cache. Depending on the configuration, different plugins can add functionality to the cache. The `MemoryCachePlugin` adds in-memory caching. The `FileCachePlugin` adds persistent (file-system) caching.

The `FileCachePlugin` uses the serialization mechanism to persist and restore cached items to/from the disk.

Hook Object Frozen

Classes with `hooks` have their `hooks` object frozen, so adding custom hooks is no longer possible this way.

MIGRATION: The recommended way to add custom hooks is using a WeakMap and a static `getXXXHooks(XXX)` (i. e. `getCompilationHook(compilation)`) method. Internal classes use the same mechanism used for custom hooks.

Tapable Upgrade

The compat layer for webpack 3 plugins has been removed. It had already been deprecated for webpack 4.

Some less used tapable APIs were removed or deprecated.

MIGRATION: Use the new tapable API.

Staged Hooks

For several steps in the sealing process, there had been multiple hooks for different stages. i. e. `optimizeDependenciesBasic` `optimizeDependencies` and `optimizeDependenciesAdvanced`. These have been removed in favor of a single hook which can be used with a `stage` option. See `OptimizationStages` for possible stage values.

MIGRATION: Hook into the remaining hook instead. You may add a `stage` option.

Main/Chunk/ModuleTemplate deprecation

Bundle templating has been refactored. MainTemplate/ChunkTemplate/ModuleTemplate were deprecated and the JavascriptModulesPlugin takes care of JS templating now.

Before that refactoring, JS output was handled by Main/ChunkTemplate while another output (i. e. WASM, CSS) was handled by plugins. This looks like JS is first class, while another output is second class. The refactoring changes that and all output is handled by their plugins.

It's still possible to hook into parts of the templating. The hooks are in JavascriptModulesPlugin instead of Main/ChunkTemplate now. (Yes plugins can have hooks too. I call them attached hooks.)

There is a compat-layer, so Main/Chunk/ModuleTemplate still exist, but only delegate tap calls to the new hook locations.

MIGRATION: Follow the advice in the deprecation messages. Mostly pointing to hooks at different locations.

Entry point descriptor

If an object is passed as entry point the value might be a string, array of strings or a descriptor:

```
module.exports = {
  entry: {
    catalog: {
      import: './catalog.js',
    },
  },
};
```

Descriptor syntax might be used to pass additional options to an entry point.

Entry point output filename

By default, the output filename for the entry chunk is extracted from `output.filename` but you can specify a custom output filename for a specific entry:

```
module.exports = {
  entry: {
    about: { import: './about.js', filename: 'pages/[name][ext]' },
  },
};
```

Entry point dependency

By default, every entry chunk stores all the modules that it uses. With `dependOn` -option you can share the modules from one entry chunk to another:

```
module.exports = {
  entry: {
    app: { import: './app.js', dependOn: 'react-vendors' },
    'react-vendors': ['react', 'react-dom', 'prop-types'],
  },
};
```

The app chunk will not contain the modules that `react-vendors` has.

Entry point library

The entry descriptor allows to pass a different `library` option for each entrypoint.

```
module.exports = {
  entry: {
    commonjs: {
      import: './lib.js',
      library: {
        type: 'commonjs-module',
      },
    },
    amd: {
      import: './lib.js',
      library: {
        type: 'amd',
      },
    },
  },
};
```

Entry point runtime

The entry descriptor allows to specify a `runtime` per entry. When specified a chunk with this name is created which contains only the runtime code for the entry. When multiple entries specify the same `runtime` , that chunk will contain a common runtime for all these entry. This means they could be used together on the same HTML page.

```
module.exports = {
  entry: {
    app: {
      import: './app.js',
      runtime: 'app-runtime',
    },
  },
};
```

Entry point chunk loading

The entry descriptor allows to specify a `chunkLoading` per entry. The runtime for this entry will use this to load chunks.

```
module.exports = {
  entry: {
    app: {
      import: './app.js',
    },
    worker: {
      import: './worker.js',
      chunkLoading: 'importScripts',
    },
  },
};
```

Order and IDs

webpack used to order modules and chunks in the Compilation phase, in a specific way, to assign IDs in incremental order. This is no longer the case. The order will no longer be used for id generation, instead, the full control of ID generation is in the plugin.

Hooks to optimize the order of module and chunks have been removed.

MIGRATION: You cannot rely on the order of modules and chunks in the compilation phase no more.

Arrays to Sets

- Compilation.modules is now a Set
- Compilation.chunks is now a Set
- Chunk.files is now a Set

There is a compat-layer which prints deprecation warnings.

MIGRATION: Use Set methods instead of Array methods.

Compilation.fileSystemInfo

This new class can be used to access information about the filesystem in a cached way. Currently, it allows asking for both file and directory timestamps. Information about timestamps is transferred from the watcher if possible, otherwise determined by filesystem access.

In the future, asking for file content hashes will be added and modules will be able to check validity with file contents instead of file hashes.

MIGRATION: Instead of using `file/contextTimestamps` use the `compilation.fileSystemInfo` API instead.

Timestamping for directories is possible now, which allows serialization of ContextModules.

`Compiler.modifiedFiles` has been added (next to `Compiler.removedFiles`) to make it easier to reference the changed files.

Filesystems

Next to `compiler.inputFileSystem` and `compiler.outputFileSystem` there is a new `compiler.intermediateFileSystem` for all fs actions that are not considered as input or output, like writing records, cache or profiling output.

The filesystems have now the `fs` interface and do no longer demand additional methods like `join` or `mkdirp`. But if they have methods like `join` or `dirname` they are used.

Hot Module Replacement

HMR runtime has been refactored to Runtime Modules. `HotUpdateChunkTemplate` has been merged into `ChunkTemplate`. ChunkTemplates and plugins should also handle `HotUpdateChunk`s now.

The javascript part of HMR runtime has been separated from the core HMR runtime. Other module types can now also handle HMR in their own way. In the future, this will allow i. e. HMR for the mini-css-extract-plugin or for WASM modules.

MIGRATION: As this is a newly introduced functionality, there is nothing to migrate.

`import.meta.webpackHot` exposes the same API as `module.hot`. This is also usable from strict ESM modules (.mjs, type: "module" in package.json) which do not have access to `module`.

Work Queues

webpack used to handle module processing by functions calling functions, and a `semaphore` which limits parallelism. The `Compilation.semaphore` has been removed and async queues now handle work queuing and processing. Each step has a separate queue:

- `Compilation.factorizeQueue`: calling the module factory for a group of dependencies.
- `Compilation.addModuleQueue`: adding the module to the compilation queue (may restore module from cache).
- `Compilation.buildQueue`: building the module if necessary (may stores module to cache).
- `Compilation.rebuildQueue`: building a module again if manually triggered.
- `Compilation.processDependenciesQueue`: processing dependencies of a module.

These queues have some hooks to watch and intercept job processing.

In the future, multiple compilers may work together and job orchestration can be done by intercepting these queues.

MIGRATION: As this is a newly introduced functionality, there is nothing to migrate.

Logging

webpack internals includes some logging now. `stats.logging` and `infrastructureLogging` options can be used to enabled these messages.

Module and Chunk Graph

webpack used to store a resolved module in the dependency, and store the contained modules in the chunk. This is no longer the case. All information about how modules are connected in the module graph are now stored in a ModuleGraph class. All information about how modules are connected with chunks are now stored in the ChunkGraph class. The information which depends on i. e. the chunk graph, is also stored in the related class.

That means the following information about modules has been moved:

- Module connections -> ModuleGraph
- Module issuer -> ModuleGraph
- Module optimization bailout -> ModuleGraph (TODO: check if it should ChunkGraph instead)
- Module usedExports -> ModuleGraph
- Module providedExports -> ModuleGraph
- Module pre order index -> ModuleGraph
- Module post order index -> ModuleGraph
- Module depth -> ModuleGraph
- Module profile -> ModuleGraph
- Module id -> ChunkGraph
- Module hash -> ChunkGraph
- Module runtime requirements -> ChunkGraph
- Module is in chunk -> ChunkGraph
- Module is entry in chunk -> ChunkGraph
- Module is runtime module in chunk -> ChunkGraph
- Chunk runtime requirements -> ChunkGraph

webpack used to disconnect modules from the graph when restored from the cache. This is no longer necessary. A Module stores no info about the graph and can technically be used in multiple graphs. This makes caching easier.

There is a compat-layer for most of these changes, which prints a deprecation warning when used.

MIGRATION: Use the new APIs on ModuleGraph and ChunkGraph

Init Fragments

`DependenciesBlockVariables` has been removed in favor of InitFragments. `DependencyTemplates` can now add `InitFragments` to inject code to the top of the module's source. `InitFragments` allows deduplication.

MIGRATION: Use `InitFragments` instead of inserting something at a negative index into the source.

Module Source Types

Modules now have to define which source types they support via `Module.getSourceTypes()`. Depending on that, different plugins call `source()` with these types. i. e. for source type `javascript` the `JavascriptModulesPlugin` embeds the source code into the bundle. Source type `webassembly` will make the `WebAssemblyModulesPlugin` emit a wasm file. Custom source types are also supported, i. e. the mini-css-extract-plugin will probably use the source type `stylesheet` to embed the source code into a css file.

There is no relationship between module type and source type. i. e. module type `json` also uses source type `javascript` and module type `webassembly/experimental` uses source types `javascript` and `webassembly`.

MIGRATION: Custom modules need to implement these new interface methods.

Plugins for Stats

Stats `preset`, `default`, `json` and `toString` are now baked in by a plugin system. Converted the current Stats into plugins.

MIGRATION: Instead of replacing the whole Stats functionality, you can now customize it. Extra information can now be added to the stats json instead of writing a separate file.

New Watching

The watcher used by webpack was refactored. It was previously using `chokidar` and the native dependency `fsevents` (only on macOS). Now it's only based on native Node.js `fs`. This means there is no native dependency left in webpack.

It also captures more information about filesystem while watching. It now captures mtimes and watches event times, as well as information about missing files. For this, the `WatchFileSystem` API changed a little bit. While on it we also converted Arrays to Sets and Objects to Maps.

SizeOnlySource after emit

webpack now replaces the Sources in `Compilation.assets` with `SizeOnlySource` variants to reduce memory usage.

Emitting assets multiple times

The warning `Multiple assets emit different content to the same filename` has been made an error.

ExportsInfo

The way how information about exports of modules is stored has been refactored. The ModuleGraph now features an `ExportsInfo` for each `Module`, which stores information per export. It also stores information about unknown exports and if the module is used in a side-effect-only way.

For each export the following information is stored:

- Is the export used? yes, no, not statically known, not determined. (see also `optimization.usedExports`)
- Is the export provided? yes, no, not statically known, not determined. (see also `optimization.providedExports`)
- Can be export name be renamed? yes, no, not determined.
- The new name, if the export has been renamed. (see also `optimization.mangleExports`)
- Nested ExportsInfo, if the export is an object with information attached itself
  - Used for reexporting namespace objects: `import * as X from "..."; export { X };`
  - Used for representing structure in JSON modules

Code Generation Phase

The Compilation features Code Generation as separate compilation phase now. It no longer runs hidden in `Module.source()` or `Module.getRuntimeRequirements()`.

This should make the flow much cleaner. It also allows to report progress for this phase and makes Code Generation more visible when profiling.

MIGRATION: `Module.source()` and `Module.getRuntimeRequirements()` are deprecated now. Use `Module.codeGeneration()` instead.

DependencyReference

webpack used to have a single method and type to represent references of dependencies (`Compilation.getDependencyReference` returning a `DependencyReference`). This type used to include all information about this reference like the referenced Module, which exports have been imported, if it's a weak reference and also some ordering related information.

Bundling all these information together makes getting the reference expensive and it's also called often (every time somebody needs one piece of information).

In webpack 5 this part of the codebase was refactored and the method has been split up.

- The referenced module can be read from the ModuleGraphConnection
- The imported export names can be get via `Dependency.getReferencedExports()`
- There is a `weak` flag on the `Dependency` class
- Ordering is only relevant to `HarmonyImportDependencies` and can be get via `sourceOrder` property

Presentational Dependencies

There is now a new type of dependency in `NormalModules`: Presentational Dependencies

These dependencies are only used during the Code Generation phase but are not used during Module Graph building. So they can never have referenced modules or influence exports/imports.

These dependencies are cheaper to process and webpack uses them when possible

Deprecated loaders

- `null-loader`

  It will be deprecated. Use

  ```
  module.exports = {
    resolve: {
      alias: {
        xyz$: false,
      },
    },
  };
  ```

  or use an absolute path

  ```
  module.exports = {
    resolve: {
      alias: {
        [path.resolve(__dirname, '....')]: false,
      },
    },
  };
  ```

Minor Changes

- `Compiler.name`: When generating a compiler name with absolute paths, make sure to separate them with `|` or `!` on both parts of the name.
  - Using space as a separator is now deprecated. (Paths could contain spaces)
  - Hint: `|` is replaced with space in Stats string output.
- `SystemPlugin` is now disabled by default.
  - **MIGRATION**: Avoid using it as the spec has been removed. You can re-enable it with `Rule.parser.system: true`
- `ModuleConcatenationPlugin`: concatenation is no longer prevented by `DependencyVariables` as they have been removed

  - This means it can now concatenate in cases of `module`, `global`, `process` or the ProvidePlugin
- `Stats.presetToOptions` removed
  - **MIGRATION:** Use `compilation.createStatsOptions` instead
- `SingleEntryPlugin` and `SingleEntryDependency` removed
  - **MIGRATION:** use `EntryPlugin` and `EntryDependency`
- Chunks can now have multiple entry modules
- `ExtendedAPIPlugin` removed
  - **MIGRATION:** No longer needed, `__webpack_hash__` and `__webpack_chunkname__` can always be used and runtime code is injected where needed.
- `ProgressPlugin` no longer uses tapable context for `reportProgress`
  - **MIGRATION:** Use `ProgressPlugin.getReporter(compiler)` instead
- `ProvidePlugin` is now re-enabled for `.mjs` files
- `Stats` json `errors` and `warnings` no longer contain strings but objects with information splitted into properties.
  - **MIGRATION:** Access the information on the properties. i. e. `message`
- `Compilation.hooks.normalModuleLoader` is deprecated
  - **MIGRATION:** Use `NormalModule.getCompilationHooks(compilation).loader` instead
- Changed hooks in `NormalModuleFactory` from waterfall to bailing, changed and renamed hooks that return waterfall functions
- Removed `compilationParams.compilationDependencies`
  - Plugins can add dependencies to the compilation by adding to `compilation.file/context/missingDependencies`
  - Compat layer will delegate `compilationDependencies.add` to `fileDependencies.add`
- `stats.assetsByChunkName[x]` is now always an array
- `__webpack_get_script_filename__` function added to get the filename of a script file
- `"sideEffects"` in package.json will be handled by `glob-to-regex` instead of `micromatch`
  - This may have changed semantics in edge-cases
- `checkContext` was removed from `IgnorePlugin`
- New `__webpack_exports_info__` API allows export usage introspection
- SourceMapDevToolPlugin applies to non-chunk assets too now
- EnvironmentPlugin shows an error now when referenced env variable is missing and has no fallback
- Remove `serve` property from schema

Other Minor Changes

- removed builtin directory and replaced builtins with runtime modules
- Removed deprecated features
  - BannerPlugin now only support one argument that can be an object, string or function
- removed `CachePlugin`
- `Chunk.entryModule` is deprecated, use ChunkGraph instead
- `Chunk.hasEntryModule` is deprecated
- `Chunk.addModule` is deprecated
- `Chunk.removeModule` is deprecated
- `Chunk.getNumberOfModules` is deprecated
- `Chunk.modulesIterable` is deprecated
- `Chunk.compareTo` is deprecated
- `Chunk.containsModule` is deprecated
- `Chunk.getModules` is deprecated
- `Chunk.remove` is deprecated
- `Chunk.moveModule` is deprecated
- `Chunk.integrate` is deprecated
- `Chunk.canBeIntegrated` is deprecated
- `Chunk.isEmpty` is deprecated
- `Chunk.modulesSize` is deprecated
- `Chunk.size` is deprecated
- `Chunk.integratedSize` is deprecated
- `Chunk.getChunkModuleMaps` is deprecated
- `Chunk.hasModuleInGraph` is deprecated
- `Chunk.updateHash` signature changed
- `Chunk.getChildIdsByOrders` signature changed (TODO: consider moving to `ChunkGraph`)
- `Chunk.getChildIdsByOrdersMap` signature changed (TODO: consider moving to `ChunkGraph`)
- `Chunk.getChunkModuleMaps` removed
- `Chunk.setModules` removed
- deprecated Chunk methods removed
- `ChunkGraph` added
- `ChunkGroup.setParents` removed
- `ChunkGroup.containsModule` removed

- `Compilation.cache` was removed in favor of `Compilation.getCache()`
- `ChunkGroup.remove` no longer disconnected the group from block
- `ChunkGroup.compareTo` signature changed
- `ChunkGroup.getChildrenByOrders` signature changed
- `ChunkGroup` index and index renamed to pre/post order index
  - old getter is deprecated
- `ChunkTemplate.hooks.modules` signature changed
- `ChunkTemplate.hooks.render` signature changed
- `ChunkTemplate.updateHashForChunk` signature changed
- `Compilation.hooks.optimizeChunkOrder` removed
- `Compilation.hooks.optimizeModuleOrder` removed
- `Compilation.hooks.advancedOptimizeModuleOrder` removed
- `Compilation.hooks.optimizeDependenciesBasic` removed
- `Compilation.hooks.optimizeDependenciesAdvanced` removed
- `Compilation.hooks.optimizeModulesBasic` removed
- `Compilation.hooks.optimizeModulesAdvanced` removed
- `Compilation.hooks.optimizeChunksBasic` removed
- `Compilation.hooks.optimizeChunksAdvanced` removed
- `Compilation.hooks.optimizeChunkModulesBasic` removed
- `Compilation.hooks.optimizeChunkModulesAdvanced` removed
- `Compilation.hooks.optimizeExtractedChunksBasic` removed
- `Compilation.hooks.optimizeExtractedChunks` removed
- `Compilation.hooks.optimizeExtractedChunksAdvanced` removed
- `Compilation.hooks.afterOptimizeExtractedChunks` removed
- `Compilation.hooks.stillValidModule` added
- `Compilation.hooks.statsPreset` added
- `Compilation.hooks.statsNormalize` added
- `Compilation.hooks.statsFactory` added
- `Compilation.hooks.statsPrinter` added
- `Compilation.fileDependencies`, `Compilation.contextDependencies` and `Compilation.missingDependencies` are now LazySets
- `Compilation.entries` removed
  - **MIGRATION:** Use `Compilation.entryDependencies` instead

- `Compilation._preparedEntrypoints` removed
- `dependencyTemplates` is now a `DependencyTemplates` class instead of a raw `Map`
- `Compilation.fileTimestamps` and `contextTimestamps` removed
  - **MIGRATION**: Use `Compilation.fileSystemInfo` instead
- `Compilation.waitForBuildingFinished` removed
  - **MIGRATION**: Use the new queues
- `Compilation.addModuleDependencies` removed
- `Compilation.prefetch` removed
- `Compilation.hooks.beforeHash` is now called after the hashes of modules are created
  - **MIGRATION**: Use `Compiliation.hooks.beforeModuleHash` instead
- `Compilation.applyModuleIds` removed
- `Compilation.applyChunkIds` removed
- `Compiler.root` added, which points to the root compiler
  - it can be used to cache data in WeakMaps instead of statically scoped
- `Compiler.hooks.afterDone` added
- `Source.emitted` is no longer set by the Compiler
  - **MIGRATION**: Check `Compilation.emittedAssets` instead
- `Compiler/Compilation.compilerPath` added: It's a unique name of the compiler in the compiler tree. (Unique to the root compiler scope)
- `Module.needRebuild` deprecated
  - **MIGRATION**: use `Module.needBuild` instead
- `Dependency.getReference` signature changed
- `Dependency.getExports` signature changed
- `Dependency.getWarnings` signature changed
- `Dependency.getErrors` signature changed
- `Dependency.updateHash` signature changed
- `Dependency.module` removed
- There is now a base class for `DependencyTemplate`
- `MultiEntryDependency` removed
- `EntryDependency` added
- `EntryModuleNotFoundError` removed
- `SingleEntryPlugin` removed
- `EntryPlugin` added

- `Generator.getTypes` added
- `Generator.getSize` added
- `Generator.generate` signature changed
- `HotModuleReplacementPlugin.getParserHooks` added
- `Parser` was moved to `JavascriptParser`
- `ParserHelpers` was moved to `JavascriptParserHelpers`
- `MainTemplate.hooks.moduleObj` removed
- `MainTemplate.hooks.currentHash` removed
- `MainTemplate.hooks.addModule` removed
- `MainTemplate.hooks.requireEnsure` removed
- `MainTemplate.hooks.globalHashPaths` removed
- `MainTemplate.hooks.globalHash` removed
- `MainTemplate.hooks.hotBootstrap` removed
- `MainTemplate.hooks` some signatures changed
- `Module.hash` deprecated
- `Module.renderedHash` deprecated
- `Module.reasons` removed
- `Module.id` deprecated
- `Module.index` deprecated
- `Module.index2` deprecated
- `Module.depth` deprecated
- `Module.issuer` deprecated
- `Module.profile` removed
- `Module.prefetched` removed
- `Module.built` removed
- `Module.used` removed
  - **MIGRATION**: Use `Module.getUsedExports` instead
- Module.usedExports deprecated
  - **MIGRATION**: Use `Module.getUsedExports` instead
- `Module.optimizationBailout` deprecated
- `Module.exportsArgument` removed
- `Module.optional` deprecated
- `Module.disconnect` removed

- `Module.unseal` removed
- `Module.setChunks` removed
- `Module.addChunk` deprecated
- `Module.removeChunk` deprecated
- `Module.isInChunk` deprecated
- `Module.isEntryModule` deprecated
- `Module.getChunks` deprecated
- `Module.getNumberOfChunks` deprecated
- `Module.chunksIterable` deprecated
- `Module.hasEqualsChunks` removed
- `Module.useSourceMap` moved to `NormalModule`
- `Module.addReason` removed
- `Module.removeReason` removed
- `Module.rewriteChunkInReasons` removed
- `Module.isUsed` removed
  - **MIGRATION**: Use `isModuleUsed` , `isExportUsed` and `getUsedName` instead
- `Module.updateHash` signature changed
- `Module.sortItems` removed
- `Module.unbuild` removed
  - **MIGRATION**: Use `invalidateBuild` instead
- `Module.getSourceTypes` added
- `Module.getRuntimeRequirements` added
- `Module.size` signature changed
- `ModuleFilenameHelpers.createFilename` signature changed
- `ModuleProfile` class added with more data
- `ModuleReason` removed
- `ModuleTemplate.hooks` signatures changed
- `ModuleTemplate.render` signature changed
- `Compiler.dependencies` removed
  - **MIGRATION**: Use `MultiCompiler.setDependencies` instead
- `MultiModule` removed
- `MultiModuleFactory` removed

- `NormalModuleFactory.fileDependencies` , `NormalModuleFactory.contextDependencies` and `NormalModuleFactory.missingDependencies` are now LazySets
- `RuntimeTemplate` methods now take `runtimeRequirements` arguments
- `serve` property is removed
- `Stats.jsonToString` removed
- `Stats.filterWarnings` removed
- `Stats.getChildOptions` removed
- `Stats` helper methods removed
- `Stats.toJson` signature changed (second argument removed)
- `ExternalModule.external` removed
- `HarmonyInitDependency` removed
- `Dependency.getInitFragments` deprecated
  - **MIGRATION**: Use `apply` `initFragements` instead
- DependencyReference now takes a function to a module instead of a Module
- HarmonyImportSpecifierDependency.redirectedId removed
  - **MIGRATION**: Use `setId` instead
- acorn 5 -> 8
- Testing
  - HotTestCases now runs for multiple targets `async-node` `node` `web` `webworker`
  - TestCases now also runs for filesystem caching with `store: "instant"` and `store: "pack"`
  - TestCases now also runs for deterministic module ids
- Tooling added to order the imports (checked in CI)
- Chunk name mapping in runtime no longer contains entries when chunk name equals chunk id
- add `resolvedModuleId` `resolvedModuleIdentifier` and `resolvedModule` to reasons in Stats which point to the module before optimizations like scope hoisting
- show `resolvedModule` in Stats toString output
- loader-runner was upgraded: https://github.com/webpack/loader-runner/releases/tag/v3.0.0
- `file/context/missingDependencies` in `Compilation` are no longer sorted for performance reasons
  - Do not rely on the order
- webpack-sources was upgraded to version 2: https://github.com/webpack/webpack-sources/releases/tag/v2.0.1
- webpack-command support was removed
- Use schema-utils@2 for schema validation
- `Compiler.assetEmitted` has an improved second argument with more information
- BannerPlugin omits trailing whitespace

- removed `minChunkSize` option from `LimitChunkCountPlugin`
- reorganize from javascript related files into sub-directory
  - `webpack.JavascriptModulesPlugin` -> `webpack.javascript.JavascriptModulesPlugin`
- Logger.getChildLogger added
- change the default of entryOnly of the DllPlugin to true
- remove special request shortening logic and use single relative paths for readable module names
- allow webpack:// urls in SourceMaps to provided paths relative to webpack root context
- add API to generate and process CLI arguments targeting webpack configuration
- add `__system_context__` as context from System.js when using System.js as libraryTarget
- add bigint support for the DefinePlugin
- add bigint support for basic evaluations like maths
- remove ability to modify the compilation hash after the hash has been created
- remove HotModuleReplacementPlugin multiStep mode
- `assetInfo` from `emitAsset` will now merge when nested objects or arrays are used
- `[query]` is now a valid placeholder when for paths based on a `filename` like assets
- add `Compilation.deleteAsset` to correctly delete an assets and non-shared related assets
- expose `require("webpack-sources")` as `require("webpack").sources`
- terser 5
- Webpack can be written with a capital W when at the start of a sentence

« Previous
Roadmap 2021 (2020-12-08)