New issue Jump to bottom

Merge Proposal: Module federation and code sharing between bundles. Many builds act as one #10352

Closed ScriptedAlchemy opened this issue on 7 Feb 2020 · 398 comments

Labels Send a PR

ScriptedAlchemy commented on 7 Feb 2020 • edited

This is a proposal to merge my existing work into the Webpack core. The base concept is federated application orchestration at runtime. What I hope will provide a new era for how we build apps at scale.

Due to the demand for more information, we have created a site that consolidates all content

https://module-federation.github.io/

https://github.com/module-federation/module-federation-examples

https://github.com/webpack/changelog-v5/blob/master/guides/module-federation.md

Feature request

@sokra as requested, I've opened a new issue for us to discuss planning and implementation.

This is the evolution of my original issue: #8524
edit: a better way to explain the request. I want the webpack equivalent of Apollos federation concept. I want to share resources between separate bundles at runtime. Similar to DLL plugin but without needing to provide Static build context , and I want to do this at runtime in the browser. Not at build time. Thus enabling tech like micro-frontends to be efficient, easy to manage, and most importantly - erase page reloads between separate apps and Enable independent deploys of standalone applications.

For context, I've already implemented this feature request with great success. I feel this would have a major impact on frontend applications. I am proposing to refactor/rewrite my project and introduce it into the Webpack core. https://github.com/ScriptedAlchemy/webpack-external-import

What is the expected behavior?

```
import('website-one/MegaNav')
require('website-one/MegaNav')
```

- I don't want a single point of failure, like a commons chunk
- I want to load code from another build, like dynamic imports and code-splitting
- I want multiple builds to look and feel like a monolith in the client
- I am able to deploy frontend apps independently and expect Webpack to orchestrate at runtime
- I don't want to use less integrated/framework oriented solutions like SingleSPA, browser events, service workers - I want to use Webpack as the host container for foreign chunks and modules, not load other app entry points, but rather load other apps *chunks* and use those modules
- If desired, an entire company should be able to federate code across every UI, from user-facing to backend. Effectively turning a multi build, multi team, multi mono-repo company into one SPA in the browser. Removing the need for page reloads or downloading additional code and bundles containing mostly the same node modules

What is motivation or use case for adding/changing the behavior?
This would offer a major shift in frontend architecture.
Apps are getting larger and code-splitting on its own is not as effective with things such as micro-frontend technology.

All current solutions on the market are substandard - working around the problem instead of addressing it. Application interleaving at runtime enables many new avenues for engineering.

Heres some:

- Applications can be self-healing, in the event there is a network failure - one could query other interleaved apps and load their copy of the missing dependency. This would solve deploy issues where pages break for a moment as the new code replaced old
- Code can be distributed and managed by the team maintaining it, with options for evergreen code. Navigation, Pages, Routers - could be managed and deployed by the team who owns it - others would not have to redeploy their applications to get updates
- Smaller bundles - interleaved apps wouldn't download chunks if Webpack already has the modules required by the interleaved app or fragment.
- Easy and feasible micro-frontends - the tech right now involves some serious management and time to build supporting infrastructure
- Shareable configs - interleave FE configurations without the need to rebundle or expose multiple configs.
- Avoid long & slow builds due to the size of the monorepo. This would enable smaller builds and faster deploys with less coordination across teams
- Introduction of "micro-service functions" modeling something similar to what backend engineers get to enjoy.
- Better AB testing and marketing tooling. Right now AB tests are managed via a tag manager, plastering vanilla JS onto the dom which is usually not optimized, pollifilled, or compatible with how much javascript apps are written today. Markering teams and AB teams would write native components without having to get in the way of team delivery.
- Better analytics - Tag managers and analytics engineers could write system native modules that could be woven into codebases. Most depend on some plugin or just reading the DOM. Interleaving would open a new world of slick integration.

- no global pollution of the window, globals can be accessed via the Webpack runtime
- as more apps are interleaved, they too can supply their modules to another interleaved app, resulting in less and less code needing to be downloaded as more and more modules are added to webpack

The use cases are limitless, I've only begun to imagine possibilities.

Webpack is more than capable of doing this without needing any big changes to the core. Both 4 and 5 already have what I need.

How should this be implemented in your opinion?
Seeing as I've already done it. I have a clear picture of how to implement it. However, it can be improved with some guidance. I hold little opinion over the solution, as long as it offers a similar outcome.

1. Add an additional require extension which can support interleaving. I'm currently using a slightly modified version of `requireEnsure` which already works very well with Webpacks chunk and module loading / caching. when requiring an interleaved module - users would specify a pattern like /<module.id> - Because my current solution is based on requireEnsure, it's pretty robust and not hacky. To handle CSS I've also taken the code template from mini-css to support side-effect loading.
2. Output an unhashed JS file that can be embedded onto other apps. This file would contain a small mapping of `chunk.id` to the cache busted file name.
3. Hashed module ids based on contents + package.json version of dependency + usedExports to support tree-shaken code. Hashing needs to be effective at avoiding collisions as well as to support tree-shaken code. Hashing enables effective code sharing and prevents overwriting existing modules. Ideally, if there's an option to update module id after optimization - it would likely improve the reliability of hashing. However, I have not encountered any problems. Open to alternative solutions. This hashing mechanism I am using is used by amazon who built a CLI to orchestrate code sharing across the company.
4. A slight addition to the chunk template to include some registration data or another function similar to webpackJsonp.push which registers chunks and provides some metadata to Webpack
   Heres what it would include:

- compilation hash: to detect if this chunk is being loaded by the build that created it, if so, no additional processing is needed
- an array of additional `chunk.id` 's required by the current chunk
- a list of module ids which allow Webpack to determine if it has all modules required by the chunk about to be interleaved. Webpack would use a lazy find to search modules for the first instance of a missing module.id. At which point Webpack would load the extra chunk needed to fulfill any missing modules it does not already have

5. an interface similar to externals, where a developer can specify what files/modules they intend to make available for interleaving. During the build, Webpack would not hash these module.ids, but instead, make the id be whatever the key is. Allowing humans to call the module by a predictable name. All dependencies of a module and all modules, in general, would be hashed - ensuring that nested dependencies can be tree shaken and will only use existing modules in Webpack if they are an exact match.

into the main chunk. In order to prevent downloading a massive 200kb chunk, I am suggesting that there be some default limit for maxSize set on cache groups. Something along the lines of enabling aggressive code-splitting by default if interleaving is enabled. The goal is a happy medium between not downloading dozens of chunks and not downloading a massive chunk for one or two modules the host build cannot supply.

```
"interleave": {
  "TitleComponent": "src/components/Title/index.js",
  "SomeExternalModule": "src/components/hello-world/index.js"
}
```

```
plugins: [
  new WebpackExternalImport({
    manifestName: "website-one"
  })
];
```

Which would be used in consumer apps like this:
`require.interleaved('website-one/TitleComponeent')`

For better understanding, I suggest reading the links below. Please note that I am aware of many of the less elegant parts of the codebase and eager to improve the implementation to meet internal standards of Webpack (perf, bundle size, implementation strategy, better hooks to use)

- https://github.com/ScriptedAlchemy/webpack-external-import
- https://github.com/ScriptedAlchemy/webpack-external-import/blob/master/src/webpack/interleaveFn.js
- https://github.com/ScriptedAlchemy/webpack-external-import/blob/master/src/webpack/chunkSplitting.js
- https://github.com/ScriptedAlchemy/webpack-external-import/blob/master/src/webpack/optimizeChunk.js

Are you willing to work on this yourself?
yes - I have been contributing to Webpack since v1 and have a deep understanding of its API.
I'd like to be the one to bring this feature to the wider JS community as well as to Webpack. I've built multiple large scale MFEs, some consisting of over 100 MFE's at one company. I am very familiar with the challenges and alternative methods to chunk federation/interleaving - all of them are terrible.

While this solution might not be for everyone, it will have major value to massive corporations consisting of hundreds of teams of FE stacks. I am also able to test implementation updates against my own userbase - providing additional assurance to Webpack. I used the same approach when merging `extract-css-chunks` with `mini-css` (bringing HMR to mini-css)

(1.2mb uncompressed) compared to 4mb using traditional solutions available on the market.

Extra considerations

- Whitelisting instructions that would enable developers to tell Webpack to use the "host build's" version of a dependency. Cases like React where I can't have two on the page - I want to offer the option to have a chunk use the version already installed, even if its a patch version different.
- Hashing bloats builds: I'm looking for an alternative solution to hashing to give the community options, however, the savings of code sharing drastically outweigh any overhead in my opinion. 4mb to 1.2mb speaks for itself.
- Chunk manifests are inside the chunk - this could be in one single file to produce a single manifest, however, I like the idea of code splitting chunk needs into the chunk themselves. This would make the actual manifest file very small as all it looks up is chunk names to URL paths. Seeing as the manifest is either cache busted with Date.now or 304 cache control - I wanted to keep it lean in the event developers don't want to deal with cache headers and just cache bust the 2kb file on each page load. This would introduce one extra loading wave, as once the chunk is downloaded, it would specify what it needs, triggering Webpack to then download any additional files in one call. Download chunk->download any chunks containing modules Webpack host build cannot provide. Nested chunks would not create more loading waves as the chunk metadata contains a thorough map of what it needs in totality

SSR
Update: because this is **not only a browser implementation** - you are more than able to federate modules via common js and shared directory or volumes on s3 or some other store that system. Alternatively, one could npm install a node version of the dependency at CI, treating the federated code (which is bundled so there's no need for nested npm installs) to install the latest copy at deploy rendering most recent deploy of it - but at runtime the most updated Versions are available.

Ultimately- if you use module federation for the node build and were to have a shared disk that a lambda could read from, you'd be able to require other code deployed, in the exact same manner as you do on the frontend.

👍 287 🎉 120 ❤️ 134 🚀 88 👀 62

ScriptedAlchemy changed the title ~~Module federation and code sharing between bundles. Many builds act as one~~ **Merge Proposal: Module federation and code sharing between bundles. Many builds act as one** on 7 Feb 2020

sokra commented on 7 Feb 2020 • edited

> What is the expected behavior?

I try to generalize your proposal a little bit. It intentionally don't mention chunks and always speaks about modules being loaded. That modules are transferred via chunks is left as implementation detail here.

In the host build code:

- you want to have some startup code that is executed on startup
- you want to use `import("<scope>/<request>")` (plus nice to have: `import "<scope>/<request>"`).

In the host build configuration:

- you want define a name for `<scope>`
- you want to define a remote URL for `<scope>` (alternative: this is set at runtime with some special instruction)
- you want to define some modules which are forced to be used by the remote build runtime instead of their own modules (this could either happen by request string or by a global unique name, in most cases probably npm name)
  - optionally you may want to specify a list of exports used in these modules (as it's not known at build time which exports are used by the remote build)

In the remote build code:

- You want to have some startup code which is used on normal startup, but not if the remote build is consumed by an host build.

In the remote build configuration:

- you want to define this build as consumable by a host build, by exposing some modules for a host build.
- you want to define which `<request>` from the host build points to which modules
- you want to define which modules are override-able by host builds (This is needed to keep these modules as modules at runtime, and prevent some optimizations e. g. scope hoisting or side-effects which would remove modules)

Delivery:

- For `target: "web"` : the remote build may be loaded cross-origin. (e. g. via script tag and JSONP similar to how chunk loading works)
- For `target: "node"` : the remote build should be loaded with `require()`
- For `target: "async-node"` : the remote build should be loaded with `fs + vm`

Recursion:

- Host builds may define multiple `<scopes>` and remote builds
- The remote build may use the same feature again. It could be a host build for another remote build. This is possible in a circular way. In the smallest case A loads a modules from B which loads a module from A.
  - Override-able modules are inherited and apply recursively e. g. For A -> B -> C: Here A is able to override modules in C too.

Optimizations:

- Host build:
  - A module used to override a module in the remote build, which is not used by the remote build should not be downloaded.
- Remote build
  - A override-able module which is overriden by the host build should not be downloaded.
- Exceptions may be made when download is very small, when this leads to reduced number of requests made in some cases (similar to splitChunks.minSize)

Host build and remote build may be updated independently. The following information is shared off the band:

- The exposed modules by the remote build
  - The host build may use these modules as `<request>` in code
  - The shared list may not be complete or change over time
  - Using an non-exposed `<request>` will lead to a runtime error resp promise rejection.
- The override-able modules
  - The host build may use these modules in configuration to override them
  - Trying to override a non-override-able module will be ignored
  - Optionally a list of exports used in these modules
- The remote URL.
  - The URL must not change when the remote build is updated.
  - User should be hinted towards correct caching of the remote URL.
    - revalidate, etag
    - Alternative: Redirect to hashed URL
    - May be cached for a short timespan, delayed effect of deployment should be considered.

Did I miss something? Would you consider these requirements as full-filling for your use cases?

ScriptedAlchemy commented on 7 Feb 2020

> optionally you may want to specify a list of exports used in these modules (as it's not known at build time which exports are used by the remote build)

Currently, I mark the module as usedInUnknownWays to prevent tree shaking the exports on exposed moduled/files

I do not see anything missed, I believe this captures the requirements perfectly.
I did not think about async-node or node targets - but this sounds like a great addition id love to support

Your understanding is correct and meets requirements plus adds new aspects i had not yet considered 👍

sokra commented on 7 Feb 2020

Yep, that makes sense.

I intentionally removed some points from your proposal:

- The ability to automatically share modules if they match by hash.
  - I think this is too unsafe. webpack usually don't choose ways that may break in some cases.
  - Instead modules are only shared when opt-in into making them override-able and override them from the host build.
  - A more aggressive way of sharing modules may be provided later as separate plugin or opt-in
- `require.interleaved` syntax
  - I think this should integrate with ESM imports instead.
  - This makes it possible to use native ESM and import maps without changing the code
  - More aligned by the spec
  - Users don't have to care much about the location of modules
  - Migrating from and to single build should be without code change
- The ability to share modules between sibling remote builds.
  - I think this could lead to weird runtime errors, because loaded version depends on load order of remote builds.
  - So host build is in control of everything and sibling remote builds no not effect each other.
  - This should prevent problems that only occur in cases e. g. when page a is visited before page b.
  - If two sibling remote builds should share modules, the host build must override them.

I think I have a good idea how to solve these requirements in a very clean way. I can share that on Monday.

ScriptedAlchemy commented on 7 Feb 2020 • edited

Okay, I'm on board with the proposed changes.

I believe, with your suggestions included, it will be a much smoother integration which would greatly benefit the js community!

Have a good weekend! Appreciate your input

ScriptedAlchemy mentioned this issue on 9 Feb 2020

Child getting loaded as empty module in browser when made react as external module in child MFE ScriptedAlchemy/webpack-external-import#58

ethersage mentioned this issue on 9 Feb 2020

More info on this plugin and the MFEs ScriptedAlchemy/webpack-external-import#13

Closed

ScriptedAlchemy commented on 10 Feb 2020 • edited

@sokra Let me know your thoughts on how we can solve these in a clean way.

I was working with supporting externals yesterday and a lot of your proposed changes made sense. In terms of how to opt-in and out of sharing code.

It sounds similar to what you were talking about.

Host Build

```
new WebpackExternalImport({
  provideExternals: {
    react: "React"
  }
});
```

Remote Build

```
new WebpackExternalImport({
  useExternals: {
    react: "React"
  }
});
```

To make this work, I modified `ExternalModuleFactoryPlugin`

I then added the following, which changed the external module to a raw request like `require('react');` which would be supplied by the host to the remote. This however only works for externals and does not allow the remote build to function on its own as `require('react') = module.exports = React` on the remote build

```
            const externalModule = new ExternalModule(
              value,
              type || globalType,
              dependency.request
            );
            externalModule.id = dependency.userRequest;
```

hashed module ids

sokra commented on 11 Feb 2020 • edited

So here are my idea:

host and remote runtimes are two complete separate webpack runtimes with separate `__webpack_modules__` etc. There is no manifest json file, instead the remote runtime is compiled with an special entrypoints which exposes an interface like this:

```
{
    get(module: string) => Promise<(Module) => void>,
    override(module: string, getter: () => Promise<(Module) => void>) => void
}
```

It basically consists of two parts: a kind of exposed `import()` function (get), which allows to load modules from the runtime build. And an `override` function which allows to override override-able modules from the remote build. The entrypoint contains all information which would be in a manifest json.

Here are more details:

Remote build

Plugin

A new plugin (names up to discussion):

```
new ContainerPlugin({
  expose: {
    "Dashboard": "./src/DashboardComponent.js",
    "themes/dark": "./src/DarkTheme.css"
  },
  overridable: {
    "react": "react"
  },
  name: "remoteBuildABC",
  library: "remoteBuildABC",
  libraryTarget: "var"
})
```

- `expose` : modules exposed to the host build. keys is the request the host build must use to access to module. value is request string which is resolved during remote build.

- In addition to a object syntax an array of request string may be given as shortcut. In this case the plugin determines the keys automatically: `["./src/Dashboard", "react"] => { "src/Dashboard": "./src/Dashboard", "react": "react" }` (This is especially useful for the `overridable` s)
- `name` : A name for the entry.
- `library` : A library name, defaults to `name` or nothing depending on `libraryTarget` . See also `output.library` .
- `libraryTarget` : The used libary target for the entrypoint. See `output.libraryTarget` . This is how the remote build is exposed. Defaults to `"var"` .

When the plugin is used it adds an additional entrypoint which represents the remote build. The remote url points to the generated file from this entrypoint.

It also switches the mode of all `overridable` modules. Later more.

The generated entry

```
export function get(module) {
  switch(module) {
    case "themes/dark":
      return __webpack_require__.e(12).then(() => __webpack_require__(34));
    case "Dashboard":
      return Promise.all([__webpack_require__.e(23), __webpack_require__.e(24)]).then(() => __web
    default:
      return Promise.resolve().then(() => { throw new Error(...); });
  }
};
export function override(module, getter) {
  __webpack_require__.overrides[module] = getter;
  // foreach child container, call override too
};
```

Overridables

Getting the overridable modules correct is a little bit tricky. Their code should only load when used and the overriden module should be used when set. So loading a overridable module is async, as it either has to load the separately bundled module or the module provided by the override function.

All overridable modules are hidden behind a wrapper module (by replacing them after resolving). The wrapper module has a special module type (e. g. "overrideable module"). The wrapper module has an async dependency on the original module (which puts it into a separate chunk during chunk graph building).

Modules with the "overrideable module" type generate extra code for the chunk loading, which loads the nested chunk (group) too (in parallel to the normal chunk loading). So far this should restore normal operation, but puts overridable modules into separate chunks.

loading.

Host build

Plugin

```
new ContainerReferencePlugin({
  remotesType: "var",
  remotes: ["remoteBuildABC", "remoteBuildDEF"],
  override: {
    "react": "react"
  }
})
```

- `remotesType` : An default external type used for remotes
- `remotes` : A list of remote builds as `externals` with the same `syntax` .
- `override` : A list of overrides used for the remote builds. key is a name chosen by `overridable` in the `ContainerPlugin` and value is a request resolved in the host build.
  - An advanced configuration e. g. `{"react": { import: "react", usedExports: ["createElement", ["default", "createElement"]]}}` might be used to only include specified exports.

Types of operation:

- `remotesType: "var"` . The remote url is used as already existing `<script>` tag. It sets a global variable with the specified name. This variable is used to access the remote build.
- `remotesType: "commonjs2"` . The remote build is access via `require()` . This makes sense for node.js applications
- `remotesType: "module"` . The remote url is used via `import remote from "remote-url"` . (This kind of external is not yet implemented, but planned.)
- `remotesType: "import"` . The remote url is used via `import("remote-url")` . (This kind of external is not yet implemented, but planned.)
- `remotesType: "jsonp"` . The remote url is injected as `<script>` tag. It is expected to set a global variable on load. This variable is used to access the remote build. (This kind of external is not yet implemented, but doable.)
- `remotesType: "amd"` . The remote url is used via `define(["remote-url"], (remote) => {})` . Depends on an AMD loader on page and `output.libraryTarget: "amd"` .
- `remotesType: "umd"` . Either commonjs, amd or global. Depends on `output.libraryTarget: "umd"` .
- etc. (see ExternalsPlugin)

Some types allow to load the container on demand when used, some expect it to be loaded on bootstrap. Depending on usage both makes sense.

When using this plugin all requests starting with one of the `remotes` keys are handled in special way: "remote modules" used. And the module has a dependency to the "external module".

During chunk loading all "remote modules" are created by executing the function exposed by the remote build as received from the external module.

Loading and execution happen in two steps. Loading happens during chunk loading and execution happens in the normal evaluation flow. This makes sure that remote modules are not executed to early. Execution happens by assinging to the `exports` property of a given Module object (and not by returning the exports). This is important to handle circular dependencies between module modules and host overriden modules.

Summary

The ContainerPlugin adds a container entrypoint which exposed modules to the consumer. Technically this creates a separate root to the module graph. In practice splitChunks deduplicates chunks between the container entry and other entries so most chunks are shared between applications and container.

The ContainerReferencePlugin allows to consume a container entrypoint, referenced as external. This allows to reference modules from the container via a specified scope.

An overriding mechnism allows the container to use modules from the host build. This is useful for multiple reasons:

- Sharing modules: Shared modules do not need to be downloaded/executed multiple times. So for performance reasons.
- Singleton modules: Some modules must only be instatiated once to be functional. e. g. react
- State sharing: A store might be shared to have shared state between host and containers.
- Functional changes: A module might be overriden to change the behavior of the containers modules. This might be better solved with arguments, but anyway.

Questions

- Should `overridables` and `override` be renamed to `shared` ?
- How to ensure that each webpack build has a unique `output.jsonpFunction` ?
- How to load a remote module that is needed in the initial page load? There is no chunk loading in this case and loading would delay bootstrapping. Should we force to use `import()` in this case?
- Should there be a validation step at initialization which checks if all expected remote modules are exposed from the remote build? This would make it fail earlier.

❤️ 18

ScriptedAlchemy commented on 11 Feb 2020

❤️ 1

sokra commented on 11 Feb 2020

> I definitely like your implementation idea though - I just wanted to make 1 comment however... It would be quite fantastic for the `expose` property to be expressed via code somehow.

Note that the config file is code and you can use any javascript code to generate the config for the plugin. e. g. you could glob your source code for all `*.public.js` files and expose them.

If you talking about a in source code annotation or somewhat like that: I don't think that's a good idea for multiple reasons: 1. It would depend on the file parsed by webpack in first place. Meaning it must be including in some other entrypoint. 2. The request on which a module is exposes is a somewhat global information. Having global information in modules is not that pretty. 3. It would add webpack-special code into your source code, which is something we want to avoid. 4. It's difficult to figure out the "interface" of a remote build, because it's no longer in the config file in one place, but distributed across all source files.

But anyway, if you like to have something like that you write a custom plugin for that.

❤️ 2

ScriptedAlchemy commented on 11 Feb 2020

Getting you your response - thank you for the detail provided! I'm going to begin rewriting it tonight, will create a new branch that ill use for merging progress into on my repo. This also will help get reviews.

I like this idea. This solves some problems I was thinking about today with external-import I was wondering how to offer fallbacks on overridable modules.

It makes a lot of sense - though I will have questions on implementation detail. I know your API well, but I've not utilized some of the parts that I might need to build this. I may DM on slack when rebuilding it.

It looks like I could also have a build that has both reference plugin and container plugin. In the event, you want to enable bidirectional host capabilities.

I really like your idea about using an entry point with a custom getter, the switch statement being both the map and the loader which hooks into require functions the runtime already has. Pretty much a webpack-to-webpack coordination instead of webpack-to-chunk like I have proposed.

right location. Would the reference plugin be specifying the scope of a remote so when the request has `<scope>/<id>` it knows to treat it as a request to the scopes webpackJsonP belonging to a remote?

`remotesType` : would this be the scope? `website-one` for example.

I'll need to fiddle around to understand how to implement overrides, but it sounds like the host would push module references and the remote would use these over its own.

In response to your questions

- let's call it `shared` its easier to understand the intent.
- the unique webpackJsonP could be inferred from the package name. We could also throw an error if a remote build detects another webpackJsonP already on the window. Similar to how React throws errors if more than one copy is included. I use `namespace` on the plugin when is required to enforce a scope to be set. We could do the same thing for webpackJsonP
- Initial loading, this sounds like my `corsImport` concept. I would say we start off with forcing dynamic `import()` for loading the remote entry point (`switch` statement function). Developers could delay the application or dynamic import it elsewhere in the application, close to when a remote might be used. What to do if a request is attempted without the remote entry point? How would we allow Webpack to `import()` a remote entry point without failing the build?
- Yes, I think validation should be done - however depending on importance/complexity, it could be a follow-up modification?

Additional comments

- This specification is very thorough and covers multiple implementation methods with the `remotesType` Ill definitely have some questions on how to model all of this.

- Creating new RemoteModules, this is something ill also have questions on. I've tinkered with mini-CSS CssModule but not really created entirely new ones.

- This looks like a very large rewrite - I'm going to create a new branch on my repo and scrap most of the implementation. I'm going to start with setting up the plugin options interface and will likely request some form of review as I reach a milestone.

- I'm also likely to ask questions on which hooks are optimal to use - there are many ways to do this but id like for it to be written as correctly as possible.

- When you create new plugins, do you follow any specific patterns or have a small boilerplate reference that has schema validation and a few other repetitive steps already made?

- What plugins should I read over within Webpack that would help me mimic existing functionality?

- I'll definitely want to speak with you in detail about how to create cache groups and create async imports on the fly. I've not dynamically created an entrypoint, but I think that's pretty straight-forward.

maraisr mentioned this issue on 12 Feb 2020

The future of v3 🚀 ScriptedAlchemy/webpack-external-import#123

Merged

3 tasks

sokra commented on 12 Feb 2020

> 'm not sure how the import mechanism would look though. How would we differentiate a local import from a remote one inside the host build? Require.interleaved was the indicator to let Webpack know this was remote. I like not using a non-standard syntax, but not quite seeing how to route requests to the right location. Would the reference plugin be specifying the scope of a remote so when the request has `<scope>/<id>` it knows to treat it as a request to the scopes webpackJsonP belonging to a remote?

Yep, based on the start of the request. You can check how the ExternalsPlugin replaces modules. Basically you hook into afterResolve of the NormalModuleFactory and replace the module with a different one.

> `remotesType` : would this be the scope? `website-one` for example.

`remotes` would be a list of scopes. `remotesType` is the way how the remote is referenced. Basically this `remotesType` is passed along to the ExternalsPlugin. So the remote is referenced as external, similar to the DllPlugin references the Dll. This allows to use the different external types to have different ways to load or reference the remote bundle.

> - let's call it `shared` its easier to understand the intent.

👍

> - How would we allow Webpack to `import()` a remote entry point without failing the build?

This should not be an issue as webpack sees the stub modules for remote modules.

> - What to do if a request is attempted without the remote entry point?

If the remote build is not loaded/loadable at runtime we would reject the Promise returned from `import()`.

> - Yes, I think validation should be done - however depending on importance/complexity, it could be a follow-up modification?

Yep.

> - This specification is very thorough and covers multiple implementation methods with the `remotesType` Ill definitely have some questions on how to model all of this.

👍 1

ppiyush13 commented on 12 Feb 2020

@sokra , @ScriptedAlchemy
This looks fantastic. I am currently working on similar tool https://github.com/ppiyush13/qubix
Right now this tool is very much minimal and fragile. I would wait for this issue to be resolved before doing any further development.
Based on my experience I have few points for considerations.

1. We may need to devise a way to override public path of remote build within the host build.
   For efficient caching, remote build may be composed of multiple chunks.
   Entry chunk must be loaded with script tag. Then all the dependent chunks must be loaded from same remote URL from which entry chunk was loaded.

   publicPath overriding must only be done for for assests loaded from JS files.
   For assets loaded from CSS, publicPath should **not** overriden because partial URLs are interpreted relative to the source of the style sheet, not relative to the document.

   For the tool I have created, publiPath is overidden by **webpack-require-from** plugin. Please read the tool's source to understand how it is being done. I honestly feels that this mechanism can be improved further.

2. I aggree with **@ScriptedAlchemy** for deriving unique webpackJsonP id from package name or as as another option for ContainerReferencePlugin constructor.

3. dynamic `import` for loading remote build sounds like a good idea. In the tool I have created, I provide custom load function as "qubix.load" which loads the entry and all the required chunks.

👍 1

ScriptedAlchemy commented on 12 Feb 2020 • edited

@sokra
With that initial milestone near complete.

https://github.com/ScriptedAlchemy/webpack-external-import/pull/124/files#diff-0eb7c718f9460448d676917db1cad175

1. What would be the next logical step
2. What's missing / what should I correct on `addEntry` - as you see, its hard-coded.

The good news is that I am emitting an entry point that was dynamically created.

```
    compilation.dependencyFactories.set(
      ContainerEntryDependency,
      containerEntryModuleFactory
    );

    compilation.addEntry(
      "./src", // what should go here?
      new ContainerEntryDependency(),
      "remoteEntry", // what should this be?
      () => {
        return new ContainerEntryModule(); // what else needs to go in here, if anything?
      }
    );
  });
```

👍 1

ScriptedAlchemy commented on 14 Feb 2020

Posting here in public space for visibility.

We had some challenges in implementation once switching to Webpack 5. This is an API that I've not actually worked against in development until this week.

In order to make this work - `ContainerExposedDependency` also needed a dependency factory.

We have separated these into a hook that happens earlier on in the process

```
  compiler.hooks.compilation.tap(PLUGIN_NAME, compilation => {
    compilation.dependencyFactories.set(
      ContainerEntryDependency,
      new ContainerEntryModuleFactory()
    );
    compilation.dependencyFactories.set(
      ContainerExposedDependency,
      normalModuleFactory
    );
  });
```

As for `make` there were some challenges - so a different route was taken. Now that `remoteEntry` is emitting - I think ill review this code block and reconstitute it back into `ContainerEntryModule` - Ideally, ill re-address this when I've got some free time. Id likes to make up for the lost time and keep our momentum going. Refactoring implementation details will be a little easier once we have the `ContainerPlugin` near completion, but before starting on the `ContainerReferencePlugin`

@sokra if you have any objections - I'll follow your advice if this is implemented in such a way that it will block further progress.

```
          compilation.context,
          new ContainerEntryDependency(
            Object.entries(this.options.expose).map(([name, request], idx) => {
              const dep = new ContainerExposedDependency(name, request);
              dep.loc = {
                name,
                index: idx
              };
              return dep;
            })
          ),
          this.options.name,
          callback
        );
      });
```

Next stages

We are currently working on the switch statement. So ill see how much progress we make on that portion.
I'll send a link to the PR file blob shortly.

@sokra - What would you suggest the next logical progression here is? `shared` ?
How do we connect `library` and `libraryTarget` within this plugin appropriately?

Last comments

I didn't address this earlier - we got lost in other chatter. Regarding the naming of this plugin.

Id like to propose something like `FederationPlugin` `FederationReferencePlugin` or `ModuleFederationPlugin` `ModuleFederationReferencePlugin`

The reasoning here:

- This is a high-impact feature for webpack and the JS community. Something easier to recognize/search on the internet would likely be beneficial.
- I, and I'm sure others will write additional plugins, tools, and frameworks built on top of this foundation. Being able to easily find projects specifically designed to leverage this development pattern would help with the distribution and recognition of third-party tools.
- Federation naming aligns with Apollos term for GraphQL federation - which effectively is the API version of this. It would align us more closely with the distributed architecture naming that's picked up in the community
- It will be easy to find documentation and things like "Awesome Federation Tools List" could aggregate collections of tools.

I've already built a whole micro frontend platform, AB testing, and soon analytics tooling + tag management based on external-import that will now be based on `FederationPlugin` . This might be a good opportunity to introduce a more consistent language than the blanket "micro frontend" buzzword.

Tooling wise -
`AutomaticFederationPlugin` based on my initial idea but will revise stability.
`FederatedABTests`, `FederatedAnalyticsIntegration`, `FederatedReactRouter`, and so on.

Pretty much everything i build will be open-sourced and leveraging this - It would be a fantastic opportunity to establish strong brand recognition with this core concept, and as of right now - webpack is the only one in possession of something like this.

Open to ideas on naming, but id like to have a name recognizable enough to build a sub-culture / new technology branch around that aligns with distributed systems that work as one.

Much like how every third party plugin usually has `some-name-webpack-plugin` instantly recognizable and easy to search for

❤️ 2

ScriptedAlchemy commented on 14 Feb 2020 • edited

here's the blob for the plugin.
https://github.com/ScriptedAlchemy/webpack-external-import/pull/124/files#diff-0eb7c718f9460448d676917db1cad175

search for : `src/webpack-core/ContainerPlugin.js` - ill merge this tomorrow to you dont have to scroll through so much noise in the PR

🎉 2

ScriptedAlchemy commented on 14 Feb 2020

Fully agree - it’s been worth the wait. I intend to become a regular maintainer once I’m familiar with the changes. It’s much easier to build with compared to v4

👍 1

ScriptedAlchemy commented on 15 Feb 2020

Following update: we have more or less completed ContainerPlugin and will be working on ContainerReferencePlugin in order to test end to end flows.

Both will need a few additional updates by we have achieved basic mvp capabilities

ScriptedAlchemy mentioned this issue on 16 Feb 2020

How to prevent tree shaking on some files #10025

Closed

vankop commented on 17 Feb 2020

Is this feature only for webpack 5?

👍 2

casperle commented on 17 Feb 2020 • edited

Hi guys,
great work so far! I have one question accordingly to your solution. What will happen if two micro apps will use different react versions?
This may happen in the time of the migration, when one team is faster than the other.
Also it may happen that one of the micro apps will introduce a breaking change in theirs shared component?
How you solve this?

👍 3

sokra pinned this issue on 17 Feb 2020

sokra commented on 17 Feb 2020

> What will happen if two micro apps will use different react versions?

The host app can override the react version for all remote apps. As only a single react version must exist on page, you must do that for react. For other libraries to have the choice to override and force one version for all, or to not override in which case each micro app keeps it's own version.

Anyway the host app is in control.

> This may happen in the time of the migration, when one team is faster than the other.

In the case of a circular dependencies between two micro-apps:
Both need to upgrade the version and deploy the new version.
When all micro-apps are using the upgraded version, they may start to use new features.

That's bad and they may break the consumer apps. That need to be coordinated. It could make sense to publish a new major version of a component under a different name and expose both versions of the component at the same time to give consumers time to migrate.

👍 5

ScriptedAlchemy commented on 17 Feb 2020

@vankop ill be porting it back to Webpack 4 - but will be as a separate project, not in the core

❤️ 6

ScriptedAlchemy commented on 18 Feb 2020

On the `ReferencePlugin` side - I have it working with also a few caveats.

One issue is when importing a remote module, i end up with an unwanted module.default -> Promise -> RemoteModule.

It looks like I might need some kind of custom context dependency? **@sokra** what are your thoughts?

```
componentDidMount() {
  import("websiteTwo/Title")
    .then(Module => {
      return Module.default; // returns a promise
    })
    .then(Module => {
      console.log(Module.Title)
    });
}
```

I will also need some guidance on UMD/AMD modules and what the RemoteModule should look like for them. I've not used AMD/UMD and seeing as the variables require and ID reference, I don't know how one would be generated as its unknown. Unless i use the request and just point it to the variable similar to how `module.exports =`

sokra commented on 18 Feb 2020

> cjs,var,this,global,window etc.. it isnt an issue, but for umd,amd's they are.
>
> What I mean by that is, If I warp our new remoteEntry in `library: 'test', libraryTarget: 'global'` , in both ContainerPlugin and the root output options. You'll end up with
>
> ```
> global['test'] =
> global['test'] = (() => {...})()
> ```
>
> which work, however for var, umd, amd's that obviously doesnt work. Just wondering if you can point me in the right direction? I was thinking to `intercept` the `JavascriptModulesPlugin.render` hook for `SetVarTemplatePlugin` and co. To not apply the template over our remote chunk created by this plugin. (And quite hopefully give me an example).
>
> I couldn't just `new LibraryTemplatePlugin().apply(compiler)` in this plugin, as that'll wrap every entry, where the main bundle might not want that to happen. I could investigate pulling that plugin apart? To look at potential config on the chunk or something...??

On the core we are working on allowing target, library, etc. to be configured per entrypoint. Currently that's not possible -> https://github.com/webpack/webpack/projects/5#card-29122473

> - should our new entryChunk be marked as `preventIntegration = true` ?

no

> - `CachedConstDependency` seeing as that extends `NullDependency` , does that mean it doesnt emit? I can see a Template is defined for it, so perhaps I need to call something in my `codeGeneration` method to *print* that dep?

By defining your own codeGeneration these templates are not being used. The NormalModule uses the JavascriptGenerator in its codeGeneration, which calls these dependencyTemplates.

These DependencyTemplates also refer to ranges in the original source code. This is not something you have.

> - when `expose` is sent in as an array, we assume the keys right? Are you okay with this to derive those keys? `exposeMap[exp.replace(/(^(?:[^\w])+)/, "")] = exp;` eg: `['./src/test.js'] => {'src/test.js': './src/test.js'}`

Seems reasonable. The Regexp could be simplified to `/^[^\w]+/` . Maybe to handle `"../../test.js" -> /^([^\w]+/)+` ?

> - to keep jsonp methods unique, should that be a "warning" that we flick up when the output options hasnt deviated that from the default? Or be cleaver, and do something like `compiler.options.output.jsonpFunction = compiler.options.output.jsonpFunction + remoteEntryName`

> - I've opted for a `new Map` to house those mappings. Which I wanted your opinion on. Is a switch case, with 80 cases performant/memory efficient compared to a Map (and object for that matter) with 80 entries. The switch branches obviously will be stack allocated yeah? Whereas the Map, would be Heap allocated? I'm just thinking in a use case for my company where every component in our design system will be exposes, and 80+ cases could become quite expensive, when it really just a dictionary lookup?

Ok. Maybe use an Object instead of a Map. That's similar performance-wise, but allows the code to work in older browsers.

> One issue is when importing a remote module, i end up with an unwanted module.default -> Promise -> RemoteModule.

That seem to be a bug in the implementation. All Promises should be resolved during chunk loading.

❤️ 3

amclin mentioned this issue on 18 Feb 2020

add microfront-end loader amclin/react-project-boilerplate#193

Open

Jordan-Gilliam mentioned this issue on 18 Feb 2020

Feasibility of micro frontends vercel/next.js#6040

Closed

380 hidden items

Load more...

brendonco commented on 23 Sep 2020 • edited

@sokra How do we test expose component? e.g. snapshot

Host1:
webpack.config.js

```
    exposes: {
      './Widget': './src/components/Widget.js'
    }
```

ShoppingCart

```
import Widget from 'name/Widget';   --> this will be undefined in unit test

const ShoppingCart = () => <Widget />;
```

test

```
import { render, fireEvent } from '@testing-library/react';
import ShoppingCart from './shoppingCart';

test('ShoppingCart', () => {
 const {container} = render(<ShoppingCart />

expect(container).toMatchSnapshot();

});
```

sokra commented on 23 Sep 2020

Mock it in unit tests

brendonco commented on 23 Sep 2020

how do you mock this import Widget from 'name/Widget'; --> this will be undefined in unit test

csvan commented on 23 Sep 2020

@brendonco depends on what unit test system you are using. In Jest, you can do e.g.

```
jest.mock('name/widget', () => ({
  myExport: 'blabla',
}));
```

brendonco commented on 23 Sep 2020

test

```
import { render, fireEvent } from '@testing-library/react';
import ShoppingCart from './shoppingCart';

jest.mock('name/Widget', () => content => content);

test('ShoppingCart', () => {
 const {container} = render(<ShoppingCart />

expect(container).toMatchSnapshot();

});
```

csvan commented on 23 Sep 2020 • edited

@brendonco you will need to refer to the docs for Jest or whatever you are using for testing. Mocking is not a webpack feature and your problem is not related to it.

FishOrBear commented on 24 Sep 2020

I want to make a package that depends on another package (for example,'three'). When I want to publish the package, I don't include `three` , but let the user download it. Is there any way?

brendonco commented on 24 Sep 2020

> **@brendonco** you will need to refer to the docs for Jest or whatever you are using for testing. Mocking is not a webpack feature and your problem is not related to it.

That's true, this is not part of webpack feature. But eventually once webpack 5 is out, someone will ask how do you mock and test expose components again.

brendonco commented on 24 Sep 2020

> I want to make a package that depends on another package (for example,'three'). When I want to publish the package, I don't include `three` , but let the user download it. Is there any way?

I don't think this is relevant to webpack feature.

Well, I can use rollup.js

👎 1

csvan commented on 24 Sep 2020 • edited

> That's true, this is not part of webpack feature. But eventually once webpack 5 is out, someone will ask how do you mock and test expose components again.

In the code itself (which is what you should be unit testing), exposed components are imported just like other components, so the same mocking patterns apply. MF does not change how modules work, only how they are generated, shared and served. None of these things should affect unit tests.

brendonco commented on 24 Sep 2020

Understand **@csvan**. Maybe **@ScriptedAlchemy** can include sample unit test/mock in this example https://github.com/module-federation/module-federation-examples/tree/master/shared-routes2/app1 ? That would be helpful.

csvan commented on 24 Sep 2020 • edited

@brendonco for which test library? Different libraries do mocking in different ways, and not everyone uses Jest. Again, *exposed modules are just modules as far as the code is concerned*. There is literally no difference between mocking an exposed module and a "regular" module since the import syntax is the same for both, so everything you need should already be in the docs of the test library you are using. No need for an explicit example for MF.

(I apologize for taking this thread on a tangent, this will be my last reply regarding mocking)

csvan commented on 2 Oct 2020

Is it possible to have a *dynamic* list of remotes, so that I can add new modules on the fly?

Say for example that my main app has something like:

```
const myRemotes = await fetchListOfRemotes();
Promise.all(myRemotes.map(remote => import(myRemote.path));
```

Sorry if this already is answered above, I've been trying to keep up but it seems we are soon over 400 (rather meaty) comments in this thread. A reference would be appreciated in that case.

👍 1

csvan commented on 2 Oct 2020

Answering my own question: #10352 (comment)

@ScriptedAlchemy regarding:

> I'm working on an idea for an additional plugin that enables dynamic definition of remotes at runtime. Webpack prefers explicit over implicit. So dynamic remotes at runtime will be a plugin that hooks into RemoteModule and dependency factories

Is there an update regarding this plugin?

sokra commented on 2 Oct 2020 • edited

Containers have an API to load modules from them at runtime.

```
const container = getItSomehow();
await __webpack_initialize_sharing__("default");
await container.init(__webpack_share_scopes__.default);
const factory = await container.get("./module");
const exports = factory();
```

👍 4 ❤️ 1

matrunchyk commented on 5 Oct 2020

> **@vankop** ill be porting it back to Webpack 4 - but will be as a separate project, not in the core

@ScriptedAlchemy Hi! Do you have some updates here? Very keen to try it with Webpack4 (vue-cli and Vue3 to be precise)

sokra commented on 7 Oct 2020

sokra closed this on 7 Oct 2020

sokra unpinned this issue on 7 Oct 2020

Domiii mentioned this issue on 17 Oct 2020

Official Module Federation Documentation? module-federation/module-federation.github.io#20

Closed

oakland mentioned this issue on 4 Dec 2020

webpack✨ oakland/tecblog#80

Open

knagurski commented on 22 Jan 2021

> **@vankop** ill be porting it back to Webpack 4 - but will be as a separate project, not in the core

@ScriptedAlchemy did you ever make any headway on this? I'm all-in on module federation, but our main app is stuck at Webpack 4. I tried https://module-federation.github.io/videos/using-module-federation-with-webpack-4 but it doesn't work past v5.0.0-beta.16

🚀 1

matrunchyk commented on 22 Jan 2021

Yeah, this is the main stopper for our projects to move forward with Module Federation. It would be very very cool to have MF on WP4 as well, ideally with some example project! ❤️

fivethreeo commented on 1 Feb 2021

Or switch to webpack5, may I boast about razzle? ;)

https://github.com/jaredpalmer/razzle/tree/canary/examples/with-module-federation

neenhouse commented on 1 Feb 2021

> Yeah, this is the main stopper for our projects to move forward with Module Federation. It would be very very cool to have MF on WP4 as well, ideally with some example project!

Just my 2 cents, but moving up to WP5 would be much less pain in the long run since so many internals have been re-written to support this feature (like 40+% of internals?). Especially features like run-time observability of how your dependencies resolve are going to be features you absolutely want if you are running things in production.

👍 1

BlackDark mentioned this issue on 26 Feb 2021

Evaluate similar features to Webpack Module Federation mercedes-benz/mo360-ftk#32

Open

somsharp commented on 13 Apr 2021

> Containers have an API to load modules from them at runtime.
>
> ```
> const container = getItSomehow();
> await __webpack_initialize_sharing__("default");
> await container.init(__webpack_share_scopes__.default);
> const factory = await container.get("./module");
> const exports = factory();
> ```

Is there a way to pass parameters to the modules with the above runtime approach? Looking for an sample code.

clarkwinkelmann mentioned this issue 22 days ago

Unify javascript imports, exports and compat from core and extensions flarum/issue-archive#165

Open

clarkwinkelmann mentioned this issue 22 days ago

Unify javascript imports, exports and compat from core and extensions flarum/issue-archive#166

Assignees

No one assigned

Labels

Send a PR

Projects

None yet

Milestone

No milestone

Development

No branches or pull requests

65 participants

and others