Module Federation

Motivation

Multiple separate builds should form a single application. These separate builds should not have dependencies between each other, so they can be developed and deployed individually.

This is often known as Micro-Frontends, but is not limited to that.

> **Live Preview**
>
> Check out this live module federation example on StackBlitz.
>
> Open in StackBlitz

Low-level concepts

We distinguish between local and remote modules. Local modules are normal modules which are part of the current build. Remote modules are modules that are not part of the current build and loaded from a so-called container at the runtime.

Loading remote modules is considered an asynchronous operation. When using a remote module these asynchronous operations will be placed in the next chunk loading operation(s) that is between the remote module and the entrypoint. It's not possible to use a remote module without a chunk loading operation.

A chunk loading operation is usually an `import()` call, but older constructs like `require.ensure` or `require([...])` are supported as well.

A container is created through a container entry, which exposes asynchronous access to the specific modules. The exposed access is separated into two steps:

1. loading the module (asynchronous)
2. evaluating the module (synchronous).

Step 1 will be done during the chunk loading. Step 2 will be done during the module evaluation interleaved with other (local and remote) modules. This way, evaluation order is unaffected by converting a module from local to remote or the other way around.

It is possible to nest a container. Containers can use modules from other containers. Circular dependencies between containers are also possible.

High-level concepts

Each build acts as a container and also consumes other builds as containers. This way each build is able to access any other exposed module by loading it from its container.

Shared modules are modules that are both overridable and provided as overrides to nested container. They usually point to the same module in each build, e.g. the same library.

The `packageName` option allows setting a package name to look for a `requiredVersion`. It is automatically inferred for the module requests by default, set `requiredVersion` to `false` when automatic infer should be disabled.

Building blocks

ContainerPlugin (low level)

This plugin creates an additional container entry with the specified exposed modules.

ContainerReferencePlugin (low level)

This plugin adds specific references to containers as externals and allows to import remote modules from these containers. It also calls the `override` API of these containers to provide overrides to them. Local overrides (via `__webpack_override__` or `override` API when build is also a container) and specified overrides are provided to all referenced containers.

ModuleFederationPlugin (high level)

`ModuleFederationPlugin` combines `ContainerPlugin` and `ContainerReferencePlugin`.

Concept goals

- It should be possible to expose and use any module type that webpack supports.
- Chunk loading should load everything needed in parallel (web: single round-trip to server).
- Control from consumer to container
  - Overriding modules is a one-directional operation.
  - Sibling containers cannot override each other's modules.
- Concept should be environment-independent.

  - Usable in web, Node.js, etc.
- Relative and absolute request in shared:
  - Will always be provided, even if not used.
  - Will resolve relative to `config.context` .
  - Does not use a `requiredVersion` by default.
- Module requests in shared:
  - Are only provided when they are used.
  - Will match all used equal module requests in your build.
  - Will provide all matching modules.
  - Will extract `requiredVersion` from package.json at this position in the graph.
  - Could provide and consume multiple different versions when you have nested node_modules.
- Module requests with trailing `/` in shared will match all module requests with this prefix.

Use cases

Separate builds per page

Each page of a Single Page Application is exposed from container build in a separate build. The application shell is also a separate build referencing all pages as remote modules. This way each page can be separately deployed. The application shell is deployed when routes are updated or new routes are added. The application shell defines commonly used libraries as shared modules to avoid duplication of them in the page builds.

Components library as container

Many applications share a common components library which could be built as a container with each component exposed. Each application consumes components from the components library container. Changes to the components library can be separately deployed without the need to re-deploy all applications. The application automatically uses the up-to-date version of the components library.

Dynamic Remote Containers

The container interface supports `get` and `init` methods. `init` is an `async` compatible method that is called with one argument: the shared scope object. This object is used as a shared scope in the remote container and is filled with the provided modules from a host. It can be leveraged to connect remote containers to a host container dynamically at runtime.

init.js

```
(async () => {
  // Initializes the shared scope. Fills it with known provided modules from this build and all r
  await __webpack_init_sharing__('default');
  const container = window.someContainer; // or get the container somewhere else
  // Initialize the container, it may provide shared modules
  await container.init(__webpack_share_scopes__.default);
  const module = await container.get('./module');
})();
```

The container tries to provide shared modules, but if the shared module has already been used, a warning and the provided shared module will be ignored. The container might still use it as a fallback.

This way you could dynamically load an A/B test which provides a different version of a shared module.

Tip

Ensure you have loaded the container before attempting to dynamically connect a remote container.

Example:

init.js

```
function loadComponent(scope, module) {
  return async () => {
    // Initializes the shared scope. Fills it with known provided modules from this build and all
    await __webpack_init_sharing__('default');
    const container = window[scope]; // or get the container somewhere else
    // Initialize the container, it may provide shared modules
    await container.init(__webpack_share_scopes__.default);
    const factory = await window[scope].get(module);
    const Module = factory();
    return Module;
  };
}

loadComponent('abtests', 'test123');
```

See full implementation

Promise Based Dynamic Remotes

Generally, remotes are configured using URL's like in this example:

```
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'host',
      remotes: {
        app1: 'app1@http://localhost:3001/remoteEntry.js',
      },
    }),
  ],
};
```

But you can also pass in a promise to this remote, which will be resolved at runtime. You should resolve this promise with any module that fits the `get/init` interface described above. For example, if you wanted to pass in which version of a federated module you should use, via a query parameter you could do something like the following:

```
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'host',
      remotes: {
        app1: `promise new Promise(resolve => {
      const urlParams = new URLSearchParams(window.location.search)
      const version = urlParams.get('app1VersionParam')
      // This part depends on how you plan on hosting and versioning your federated modules
      const remoteUrlWithVersion = 'http://localhost:3001/' + version + '/remoteEntry.js'
      const script = document.createElement('script')
      script.src = remoteUrlWithVersion
      script.onload = () => {
        // the injected script has loaded and is available on window
        // we can now resolve this Promise
        const proxy = {
          get: (request) => window.app1.get(request),
          init: (arg) => {
            try {
              return window.app1.init(arg)
            } catch(e) {
              console.log('remote container already initialized')
            }
          }
        }
        resolve(proxy)
      }
      // inject this script with the src set to the versioned remoteEntry.js
      document.head.appendChild(script);
    })
    `,
      },
      // ...
    }),
  ],
};
```

Note that when using this API you *have* to resolve an object which contains the get/init API.

Dynamic Public Path

Offer a host API to set the publicPath

One could allow the host to set the publicPath of a remote module at runtime by exposing a method from that remote module.

This approach is particularly helpful when you mount independently deployed child applications on the sub path of the host domain.

Scenario:

You have a host app hosted on `https://my-host.com/app/*` and a child app hosted on `https://foo-app.com`. The child app is also mounted on the host domain, hence, `https://foo-app.com` is expected to be accessible via `https://my-host.com/app/foo-app` and `https://my-host.com/app/foo-app/*` requests are redirected to `https://foo-app.com/*` via a proxy.

Example:

webpack.config.js (remote)

```
module.exports = {
  entry: {
    remote: './public-path',
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'remote', // this name needs to match with the entry name
      exposes: ['./public-path'],
      // ...
    }),
  ],
};
```

public-path.js (remote)

```
export function set(value) {
  __webpack_public_path__ = value;
}
```

src/index.js (host)

```
const publicPath = await import('remote/public-path');
publicPath.set('/your-public-path');

//boostrap app  e.g. import('./boostrap.js')
```

Infer publicPath from script

One could infer the publicPath from the script tag from `document.currentScript.src` and set it with the `__webpack_public_path__` module variable at runtime.

Example:

webpack.config.js (remote)

```
module.exports = {
  entry: {
    remote: './setup-public-path',
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'remote', // this name needs to match with the entry name
      // ...
    }),
  ],
};
```

setup-public-path.js (remote)

```
// derive the publicPath with your own logic and set it with the __webpack_public_path__ API
__webpack_public_path__ = document.currentScript.src + '/../';
```

Tip

There is also an `'auto'` value available to `output.publicPath` which automatically determines the publicPath for you.

Troubleshooting

`Uncaught Error: Shared module is not available for eager consumption`

The application is eagerly executing an application that is operating as an omnidirectional host. There are options to choose from:

You can set the dependency as eager inside the advanced API of Module Federation, which doesn't put the modules in an async chunk, but provides them synchronously. This allows us to use these shared modules in the initial chunk. But be careful as all provided and fallback modules will always be downloaded. It's recommended to provide it only at one point of your application, e.g. the shell.

We strongly recommend using an asynchronous boundary. It will split out the initialization code of a larger chunk to avoid any additional round trips and improve performance in general.

For example, your entry looked like this:

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(<App />, document.getElementById('root'));
```

Let's create `bootstrap.js` file and move contents of the entry into it, and import that bootstrap into the entry:

index.js

```
+ import('./bootstrap');
- import React from 'react';
- import ReactDOM from 'react-dom';
- import App from './App';
- ReactDOM.render(<App />, document.getElementById('root'));
```

bootstrap.js

```
+ import React from 'react';
+ import ReactDOM from 'react-dom';
+ import App from './App';
+ ReactDOM.render(<App />, document.getElementById('root'));
```

This method works but can have limitations or drawbacks.

Setting `eager: true` for dependency via the `ModuleFederationPlugin`

webpack.config.js

```
// ...
new ModuleFederationPlugin({
  shared: {
    ...deps,
    react: {
      eager: true,
    },
  },
});
```

`Uncaught Error: Module "./Button" does not exist in container.`

It likely does not say `"./Button"`, but the error message will look similar. This issue is typically seen if you are upgrading from webpack beta.16 to webpack beta.17.

Within ModuleFederationPlugin. Change the exposes from:

```
new ModuleFederationPlugin({
  exposes: {
-   'Button': './src/Button'
+   './Button':'./src/Button'
  }
});
```

`Uncaught TypeError: fn is not a function`

You are likely missing the remote container, make sure it's added. If you have the container loaded for the remote you are trying to consume, but still see this error, add the host container's remote container file to the HTML as well.

Collision between modules from different remotes

If you're going to load multiple modules from different remotes, it's advised to set the `output.uniqueName` option for your remote builds to avoid collisions between multiple webpack runtimes.