

ScriptedAlchemy

PRACTICAL MODULE FEDERATION

JACK HERRINGTON & ZACK JACKSON

Book Version 2.0

PROLOGUE

We've all remarked at some point how rapidly the world of Javascript changes, or how frequently a new framework arises. But rarely does a completely new architectural paradigm arise. Module Federation is a paradigm shift in how we organize and build our applications. This book, Practical Module Federation, seeks to give you not just a solid understanding of what Module Federation is and how to configure, build, and deploy it, but to do so in the context of the practical applications you will be building with it.

This book is copyrighted © 2020-2021, Zack Jackson and Jack Herrington, all rights are reserved.

CONTENTS

Preface	3
Micro-Frontends And Module Federation	6
Getting Started	18
Getting Started With Create React App	36
How Module Federation Works	44
Deploying Federated Modules	49
Resilient Sharing of React Components	55
React State Sharing Options	64
Resilient Function and Static Data Sharing	104
Dynamically Loading Federated Modules	113
Shared Libraries And Versioning	122
Non-React View Frameworks	130
Resilient Header/Footer	138
A/B Tests	145
Live Preview Between Applications	153
Fully Federated Sites	161
Alternative Deployment Options	175
Medusa	182
Frequently Asked Questions	185
Troubleshooting	188
Reference	194
Glossary	201
About the Authors	204

PREFACE

Module Federation is an advanced use topic for the Webpack bundler starting with version 5. In order to get the most out of this book you should have a passing familiarity with Webpack up to and including version 4. You should know what it's useful for (bundling Javascript), how it's configured (through configuration files written in Javascript) and how it's extended (through plugins). This book also deals extensively with the React view framework and with web development technologies in general.

SETUP

You will want to have node running on your local machine at version 14 or greater to run Webpack builds and execute the server code. You should also have git and yarn installed for build tooling.

HOW TO READ THIS BOOK

The first three chapters of this book provide a step-by-step walkthrough of the architectural role of Module Federation, how to use it in a simple case, and then a walk through of the mechanics of how it works internally. The chapters that follow on from that weave through a number of related topics to give you lots of detail and options on how to get the most out of Module Federation in practical real world projects at scale. It's recommended that for those chapters you have the code downloaded and experiment with the projects as you run them.

As you read through the book you might be asking yourself why the chapters are not given sequential numbers but are instead in [section] . [chapter] form. The book is structured that way so that it's easier to update.

SUBSCRIPTION

This book is more than just a book. This original intention of this book was to be a year long subscription after its initial release. In that time Module Federation went from a beta feature into full blown production, and the book dutifully tracked along with that journey. We still honor that subscription. If you bought a subscription to 1.x you will get updates to the 2.x version of the book **for free**.

After the release of Module Federation as a full fledged feature of Webpack 5 there weren't many updates to the book. This was primarily because the focus of Module Federation efforts were directed towards getting support into the major frameworks, which has been slow going.

At the end of 2021 we feel that there is enough new material in the world of Module Federation that it's time for a refresh of the book, and thus, the 2.0 release. Some more updates should follow with additional information. Particularly if there is a shift in NextJS support for the feature.

PART ONE

GETTING TO KNOW MODULE FEDERATION

11

MICRO-FRONTENDS AND MODULE FEDERATION

Micro-Frontends are often the first way that folks encounter Module Federation. So let's start off our investigation of Module Federation by first learning about what Micro-Frontends are, and then how Module Federation can be applied to them, and more!

WHAT IS A MICRO-FRONTEND?

Micro-Frontends (MFEs) are often described as “Micro-services for the frontend”. Where micro-services architecture allows individual teams to deploy independent services that work together to create a complete backend systems architecture, MFEs allow multiple teams to independently deploy frontend experience code to one or more applications that provide a complete experience to the frontend consumer.

If that's a bit too much jargon for, let me provide some examples. The header and footer of a website are MFEs, as they are usually functionally isolated from the content that sits between the header and footer. Another example would be a carousel of products. Where the carousel knows how to manage it's on view state and make requests to the backend micro-services to get the list of products to display.

The key attributes of an MFE are that they are **self contained** and **independently deployable**. They are self-contained because they contain all of the business logic that is specific to their function and behavior. That being said MFEs are *hosted* on a page,

and that page often includes an **Application Shell** (or just Shell for short) that provides global services such as user identity, authenticated API access and so on.

The second key attribute, independent deploy, is achievable in multiple ways. You can either deploy MFEs as build-time dependencies, or as a runtime dependencies. There are advantages and disadvantages to both techniques.

- **Build time deployment** - Build time deployment means that the MFEs are maintained in separate modules but are still available at build-time and the host applications need to be re-deployed for a change to go live. Technology options here include using NPM modules through a private artifact repository, or through [Bit](#). The big advantage here is that your application deploys as a complete unit. The disadvantage is that different versions of the same MFE might be deployed on individual applications at any given point in time.
- **Runtime deployment** - Runtime deployment means that the MFEs are independently deployed to production, and are consumed at runtime by the host applications. Module Federation is one way to achieve runtime dependencies. EcmaScript Modules (ESMs) are another. Host applications do not need to re-deploy to see a change go live, and all application have the same version of the MFE code immediately. The downside is that that host applications are no longer complete at the end of the build, there are runtime dependencies that need to be loaded in order for the host application to be complete.

There are strong advantages and disadvantages to each approach. But the key takeaway here is that MFEs can be either build-time or runtime deployed.

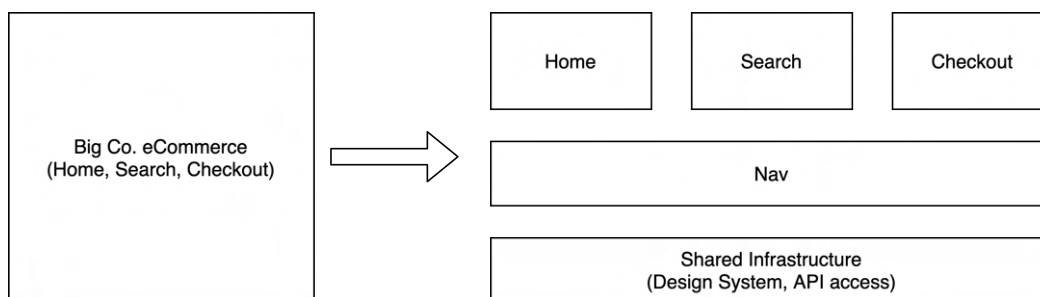
WHY MICRO-FRONTENDS?

We've talked about what MFEs are, let's now talk about why they are important.

Most applications start as “monoliths”. A monolithic application is a single application that contains all of the pages, components, business logic, etc. to render every single route on a website. They get the name “monolith” once they get to the point where it takes a considerable amount of time to deploy either for production, or often just during development.

Once they get to that stage a company will often “monolith-bust” by taking features out of the monolith one-by-one, most often route by route, to create a set of applications that are independently deployable and maintainable. The resulting architecture is sometimes called a “polyolith” or “multi-application architecture”.

Shown below is a monolithic eCommerce application that became a polyolith.

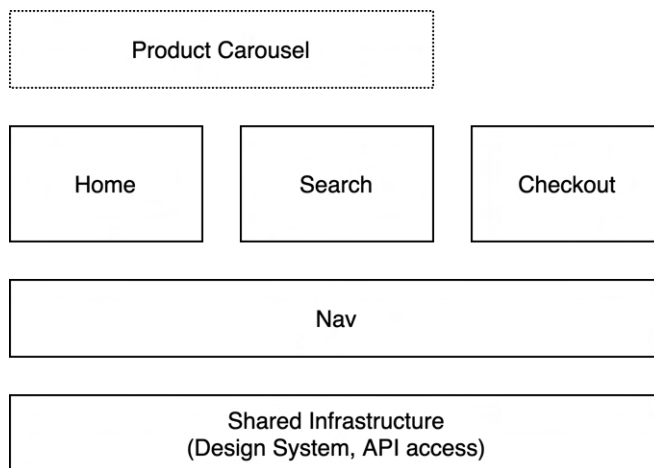


The original application that started out with forty developers became three frontend application teams with ten developers each, managing the Home, Search and Checkout experience. Those three teams share a global header footer provided by the five person “Nav” team. And they are all backed up by a five person Shared Infrastructure team that manages Design System components, API access libraries, and so on.

This architecture already has its first Micro-Frontend (MFE), the Nav teams Header and Footer, which is, in this case, build-time deployed through an NPM module.

The big advantage of this move from a monolith to a multi-application architecture is to get back to the speed of independent deployment that the team had at the start of the monolith's development. Instead of everyone having to wait for deployment on a "train", each team, Home, Search and Checkout, can now deploy their changes as often as they like.

As the name "polyolith" suggests, these applications even broken up can become unweildy. Which is one reason by MFEs deployed as runtime dependencies becomes more attractive. For example a team could develop a Product Carousel and deploy it as a runtime dependency as shown below:

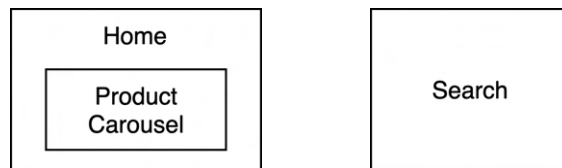


And now that Product Carousel team can deploy updates to their carousel both the Home and Search applications without requiring those applications to re-deploy. And this is where we get into the value of Module Federation.

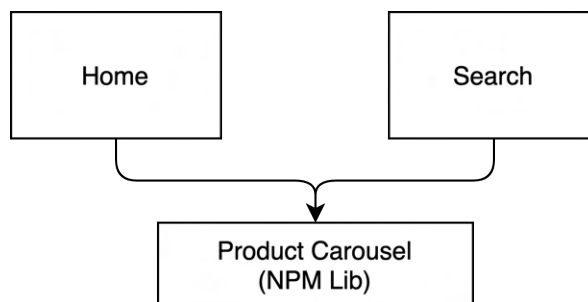
MODULE FEDERATION AND RUNTIME DEPLOYED MFES

Module Federation is a feature built into Webpack 5 that makes it very easy to share code between applications. But let's start by looking at what is familiar, NPM libraries, and work forward to Module Federation in steps.

The figure below shows two applications, Home and Search. The Home page team currently has a Product Carousel on their page that we would like to reuse as an MFE on the Search page.



To share this Product Carousel at build-time we could extract that code into a shared NPM library, as shown below. Then link it to both applications as a shared dependency.

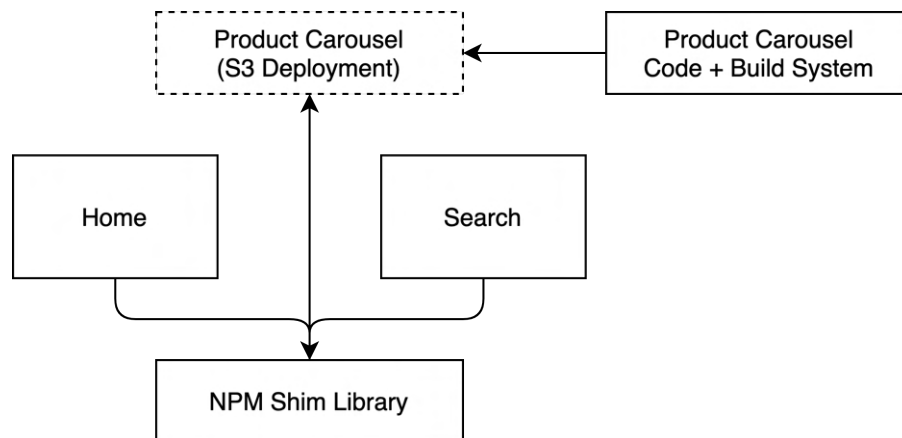


This is a very reliable way to go, but there are two drawbacks. First, for the Home page team, fixing issues in the Product Carousel becomes laborious. They need to change projects, make the changes, fix the tests, bump the version and deploy the library. Then

they need to bump the version of the library in both Home and Search and get those applications deployed.

The second drawback is that when the Product Carousel is updated there is no guarantee that both the Home and Search pages will show the same versions of the carousel, because either or both, could have deployed with older versions of the library.

So what we want is a runtime dependency, and before Module Federation we would have had to do something like this:

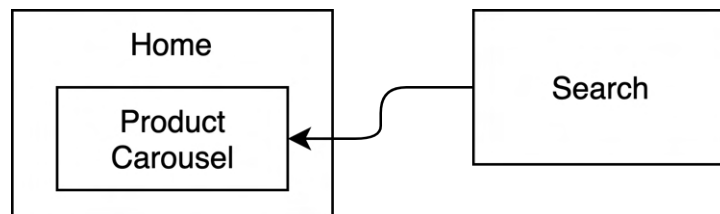


We would have extracted the Product Carousel into its own project that would package the code up and deploy it to S3. There are lots of ways to do that but there are no standard ways to do it. In addition we would create a shared shim library and use NPM to deploy that to Home and Search. And that shim library would manage loading the code from S3 and embedding it on the page.

The advantage here is that now the code is independently deployable. Once the Product Carousel team deploys the Home and Search applications will get the new code immediately.

There are several downsides. As with the NPM approach the code for the Carousel needs to be extracted from its original application. In addition a custom deployment infrastructure needs to be built. And once that's done the integration isn't seamless because the shim library will need to manage runtime loading and running the code.

By contrast, Module Federation makes runtime deployment look super easy:



The Home page application simply exposes the Product Carousel as a runtime dependency (called a remote) and the Search application points to that dependency in its Webpack configuration file. At that point the Search application imports and uses the Product Carousel as it would any other component.

This Wasn't A Problem Before Bundlers

Ironically, runtime dependencies weren't a problem before bundlers. Before bundlers, like Webpack, Parcel, Rollup (and more), JavaScript was deployed as individual files, and dependencies were managed, if they were managed at all, by loaders like `requirejs` that would ensure that global dependencies were available before your code would load.

Bundlers gave us the advantage of having a single file to deploy. Or later a set of split files to deploy that the system would manage as you performed asynchronous imports.

But with that advantage came the loss of runtime loadable dependencies, which it appears, are as important now as they were then. I guess you only realize something's value once it's gone.

MODULE FEDERATION ADVANTAGES

There are several advantages to the Module Federation approach.

- **Code remains in-place** - For one of the applications, the rendering code remains in-place and is not modified.
- **No framework** - As long as both applications are using the same view framework, then they can both use the same code.
- **No code loaders** - Micro-FE frameworks (i.e. Single-SPA) are often coupled with code loaders, like SystemJS, that work in parallel with the babel and Webpack imports that engineers are used to. Importing a federated module works just like any normal import. It just happens to be remote. Unlike other approaches, Module Federation does not require any alterations to existing codebases.
- **Applies to any Javascript** - Where Micro-Frontend frameworks work only on UI components, Module Federation can be used for any type of Javascript; UI components, business logic, i18n strings, etc. Any Javascript can be shared.
- **Applies beyond Javascript** - While many frameworks focus heavily on the Javascript aspects. Module Federation will work with file that Webpack is currently able to process today *into JavaScript* today.
- **Universal** - Module Federation can be used on any platform that uses the Javascript runtime. Browser, Node, Electron, Web Worker. It also does not require a specific module type. Many frameworks require use of SystemJS or UMD. Module Federation will work with any type currently available. AMD, UMD, CommonJS, SystemJS, window variable and so on.

There is one more key benefit to Micro-Frontends that we have yet to discuss; MFEs across frameworks.

FRAMEWORK NEUTRALITY

So far we have assumed that both the host application (or shell) and the Micro-Frontends are using the same view framework (e.g. React, Vue, Svelte, etc.). But, what if they aren't, is there a way to manage that? Absolutely. The industry standard solution to that problem is a system named Single-SPA.

Single-SPA packages components written in one of many different supported frameworks into a platform agnostic “parcel”. Then host applications can use a Single-SPA client to embed a parceled component on the page. In this way a Vue application can host React components or vice versa. Single-SPA manages all of that interaction.

This means that as you are thinking about migrating experiences from a monolith to a multiple application architecture you can stop thinking about that migration in terms of complete routes and instead start thinking about in terms of porting portions of a page.

One thing that Single-SPA does not handle is loading the parceled code into the application. This can be done as either a build-time dependency, or as a runtime dependency using Module Federation or SystemJS.

Now that you know more about Micro-Frontends and how runtime and build-time dependencies interact with Micro-Frontends let's now rethink our approach to building applications.

COMPOSABILITY

Recently Gartner has started pushing the concept of “composable commerce”. The concept has both technical and organizational aspects. On the technical side the concept is fairly simple, instead of thinking about an eCommerce applications in terms of complete experiences (e.g. Home, Search, Cart, Checkout, etc.) you instead think in terms of *capabilities* and *experience elements* that can be composed into complete customer facing experiences.

For example you can take a product display element, an “add to cart” element and a product carousel element and compose them into a complete custom experience for a new product. And each of those elements can be maintained by a separate team, deployed as Micro-Frontends (either as build-time or runtime dependencies) and then composed together in a myriad of ways.

Micro-Frontends and Module Federation fit perfectly into this composable model. And this model of composability doesn’t need to be constrained just to commerce. You can apply this to any type of application, and in fact the composability model ends up blending application types. For example, a commerce experience could include elements from a Content Management System (CMS), or vice versa.

The key point here is that you need to look at experiences at a level below the granularity of a “page” or a “route” and instead into the reusability of elements within each “page”.

CAVEAT EMPTOR (BUYER BEWARE)

To reinforce one last time; Module Federation is not the only way to accomplish a Micro-Frontend architecture. Micro-Frontends can be developed using safer and less complex build-time techniques, such as NPM library sharing, Monorepos and products like Bit. Unless your requirements specifically require runtime sharing it is far easier to use a buildtime approach.

Additionally, Module Federation is very complex, and should only be attempted by experienced web developers with a strong background in debugging though issues such as CORS, library versioning and singletons.

This book will try and steer you around the complex runtime issues associated with Module Federation. However, we strongly recommend that you take a step-by-step approach to using Module Federation in your environment. Never continue on to the next layer of complexity without first ensuring that the code works as-is. And it's strongly recommended that at each point where you achieve a working system that you commit to source code control before moving onto the next change. That will ensure that you have a safe position to fall back on when you encounter problems.

Make no mistake; **you will run into issues implementing Module Federation.** Having strong debugging skills and debugging complex issues is a prerequisite for using federated modules successfully. This is not a core stability issue with Module Federation itself, which is battle tested, these are issues of library compatibility, deployment issues and more that are layered on top of Module Federation.

Please feel free to ask for your money back on this purchase if you feel like Module Federation is not the right fit for you, your current skill level or skill set.

WHAT'S NEXT

Now that you have a decent understanding of what Module Federation is and how it changes the landscape of code sharing in Javascript. The chapters of the book that follow walk through setting up Module Federation, sharing different types of code with practical examples that you can follow on your existing projects.

1.2

GETTING STARTED

Code for this chapter is [available on GitHub](#).

The easiest way to start learning Module Federation is to try it out. To do that we will create two applications; `host` and `nav`. The home page application will consume and render a `Header` React component that is exposed by the `nav` application.

However, before we can do that we need to create a simple monorepo.

THE BOOTER MONOREPO

I can hear you screaming; “I thought I didn’t have to have a monorepo for Module Federation!” You don’t, and this isn’t a monorepo in the traditional way of binding applications and library dependencies together. It doesn’t do that. What it does do is get all the applications to start (boot) at the same time easily.

We use the same “booted monorepo” in each case. So we start by creating the getting-started directory and a sub-directory named named packages.

```
% mkdir getting-started
% cd getting-started
% mkdir packages
```

Then within the getting-started directory we add this package.json:

```
{
  "name": "packages",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "private": true,
  "scripts": {
    "start": "concurrently \"wsrun --parallel start\""
  },
  "workspaces": [
    "packages/*"
  ],
  "devDependencies": {
    "concurrently": "^5.2.0",
    "wsrun": "^5.2.0"
  }
}
```

All this does is run the start script on any application located in the packages directory.




So now that we have that set up we need to create our applications.

CREATING THE APPLICATIONS

To create the host application within the packages directory we will use `npx create-mf-app` like so:

```
% npx create-mf-app  
? Pick the name of your app: host  
? Project Type: Application  
? Port number: 3000  
? Framework: react  
? Language: javascript  
? CSS: Tailwind  
Your 'host' project is ready to go.
```

Next steps:

```
 cd host  
 npm install  
 npm start
```

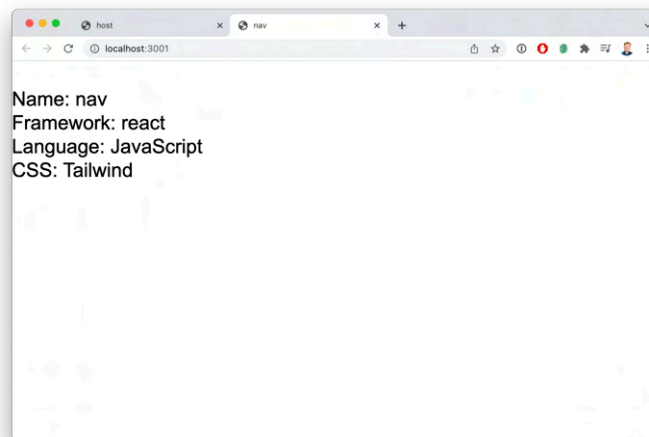
We do exactly the same thing for the nav application where the only difference is in the port number, which we will set to 3001.

The `create-mf-app` is a little application that is designed to build lightweight applications pre-configured for Module Federation in one of ten (as of this writing) different view frameworks. The list of frameworks is pretty comprehensive, but it leaves out frameworks like Angular which are compatible with Module Federation but which are more complex to set up.

With these two applications built we can run start in the parent getting-started directory.

```
% cd getting-started  
% yarn  
% yarn start
```

That will fire up both applications simultaneously and you'll get something that looks like this.



So let's have a look around at the application to see how this is all put together. The first place to look is in `src/App.jsx` in the host application.

```

import React from "react";
import ReactDOM from "react-dom";

import "./index.scss";

const App = () => (
  <div className="mt-10 text-3xl mx-auto max-w-6xl">
    <div>Name: host</div>
    <div>Framework: react</div>
    <div>Language: JavaScript</div>
    <div>CSS: Tailwind</div>
  </div>
);
ReactDOM.render(<App />, document.getElementById("app"));

```

In this simple React application we create new component called App and render it on the page. This React application is then rendered into the HTML page contained in `src/index.html`.

To get all this running we are using a combination of `babel` to transpile the code and `Webpack` to bundle the code and to run the development server. If we look into the `package.json` we can see the dependencies and these are described below.

Dependency	Description
@babel/runtime	Babel runtime functions to support ES6 code.
react	The React view library.
react-dom	The React DOM library which handles rendering React trees into DOM elements.
@babel/core	The engine of the Babel code transpiling system.
@babel/plugin-transform-runtime	Runtime helpers so that we can use ES6 syntax.

Dependency	Description
@babel/preset-react	Convenient babel presets for react.
@babel/preset-env	A present to bring import environment variables into the babel transpiling process.
babel-loader	A webpack loader the runs babel on Javascript code before it's bundled.
css-loader	A webpack loader that allows the import of CSS files into Javascript files.
html-webpack-plugin	A plugin for webpack that automatically inserts a script tag that references the bundle into the HTML template.
postcss	CSS pre-processor for Tailwind.
postcss-loader	Webpack loader to run postcss.
style-loader	A plugin that works in conjunction with CSS loader to import CSS into webpack bundles.
webpack	The webpack engine
webpack-cli	The webpack command line interface
webpack-dev-server	The webpack development server we use to run the application.
tailwindcss	The Tailwind CSS library.

So far there is really nothing new to see here. These are all standard dependencies that should be familiar to anyone who has written Webpacked React applications. The important element here is that all of these dependencies are at the latest version, and in particular that Webpack has a version of 5.57.1 or higher.

The fun begins in the `webpack.config.js` file in the host application, shown below:

```
const HtmlWebPackPlugin = require("html-webpack-plugin");
const ModuleFederationPlugin = require("webpack/lib/container/
ModuleFederationPlugin");

const deps = require("./package.json").dependencies;
module.exports = {
  output: {
    publicPath: "http://localhost:3000/",
  },
```

The first thing we do is define the public URL base path for the application.

```
  resolve: {
    extensions: [".tsx", ".ts", ".jsx", ".js", ".json"],
  },
```

Next we specify and code file extensions that we want to look for if we leave extensions off the import name.

```
  devServer: {
    port: 3000,
    historyApiFallback: true,
  },
```

This is where we specify the development server port number, and we tell the dev server to serve `index.html` for any unknown routes.

The next thing we need to do is to set up the modules that will process all the files.

```
module: {
  rules: [
    {
      test: /\.m?js/,
      type: "javascript/auto",
      resolve: {
        fullySpecified: false,
      },
    },
    {
      test: /\.((css|s[ac]ss)$/i,
      use: ["style-loader", "css-loader", "postcss-loader"],
    },
    {
      test: /\.((ts|tsx|js|jsx)$/i,
      exclude: /node_modules/,
      use: {
        loader: "babel-loader",
      },
    },
  ],
},
```

This makes sure that we properly process all the JavaScript, TypeScript, CSS or ESM files that are referenced in the application.

Finally we need to specify the plugins, including, most importantly, the `ModuleFederationPlugin`.

```
plugins: [  
  new ModuleFederationPlugin({  
    name: "host",  
    filename: "remoteEntry.js",  
    remotes: {},  
    exposes: {},  
    shared: {  
      ...deps,  
      react: {  
        singleton: true,  
        requiredVersion: deps.react,  
      },  
      "react-dom": {  
        singleton: true,  
        requiredVersion: deps["react-dom"],  
      },  
    },  
  }),  
  new HtmlWebpackPlugin({  
    template: "./src/index.html",  
  }),  
],  
};
```

Here we add two plugins, the `ModuleFederationPlugin` that will share our consume shared code, and the `HtmWebpackPlugin` that will process the `index.html` template.

Now we get to the interesting part where we are set up our `ModuleFederationPlugin`. We will get into a lot more detail as we go but the fields that configure the module federation plugin are described in the table below.

Field	Description
name	The name of this application. Never have conflicting names, always make sure every federated app has a unique name
filename	The filename to use for the remote entry file
remotes	The remotes this application will consume.
exports	The files this application will expose as remotes to other applications.
shared	The libraries the application will share with other applications.

The critical concepts here are *remotes*, *exposes* and *shared*.

- **Remotes** - These are the names of the other federated module applications that this application will consume code from. For example, if this home application consumes code from an application called `nav` that will have the header, then `nav` is a “remote” and should be listed here.
- **Exposes** - These are the files that this application will export as remotes to other applications.
- **Shared** - A list of all the libraries that this application will share with other applications in support of files listed in the `exposes` section. For example, if you export a React component you will want to list `react` in the `shared` section because it’s required to run your code.

Now that we understand the project and how it works it's time to start making some modifications.

CREATING THE HEADER

Now that we have our two applications running lets create a `Header` component in the `nav` application in `packages/nav/src/Header.jsx`. Something like this:

```
import React from "react";

const Header = () => (
  <header className="text-5xl font-bold p-5 bg-blue-500 text-white">
    I'm the header!
  </header>
);

export default Header;
```

Nothing too interesting outside of the Tailwind classes in the `className` attribute.

To make sure it works let's add it to the App component by replacing its contents with:

```
import Header from "./Header";

import "./index.scss";

const App = () => (
  <div className="mt-10 text-3xl mx-auto max-w-6xl">
    <Header />
    <div className="mt-10">Nav project</div>
  </div>
);
```

If that works then it's time to expose that Header component so that it can be consumed by the host application.

To do that we modify the `webpack.config.js` in `packages/nav` to look like this:

```
new ModuleFederationPlugin({
  name: "nav",
  filename: "remoteEntry.js",
  remotes: {},
  exposes: {
    "./Header": "./src/Header",
  },
  shared: { ... },
});
```

That's all it takes to expose the Header. We just name it create a key with the name `"./Header"` and we the value that is the path to the source file.

The next thing we need to do is to stop the `yarn start` we ran in the top level directory and re-run the `yarn start` so that we pick up the changes to the Webpack configuration.

Now we are ready to consume the `Header` component in the host application.

CONSUMING THE HEADER

Let's jump over to the `webpack.config.js` in the `packages/host` directory and make a small modification.

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    "my-nav": "nav@http://localhost:3001/remoteEntry.js",
  },
  exposes: {},
  shared: { ... },
}),
```

Once we restart the servers we can access our `Header`. But how? We'll have defined that there is a new "remote" named `my-nav` at the name and URL specified in the value. The name is `nav` because that's what is specified in the Webpack configuration of the `nav` application. And the `remoteEntry.js` file is also named in that configuration file.

Now let's go to the `App.jsx` file and consume the `Header` component.

```
const Header = React.lazy(() => import("my-nav/Header"));

import "./index.scss";

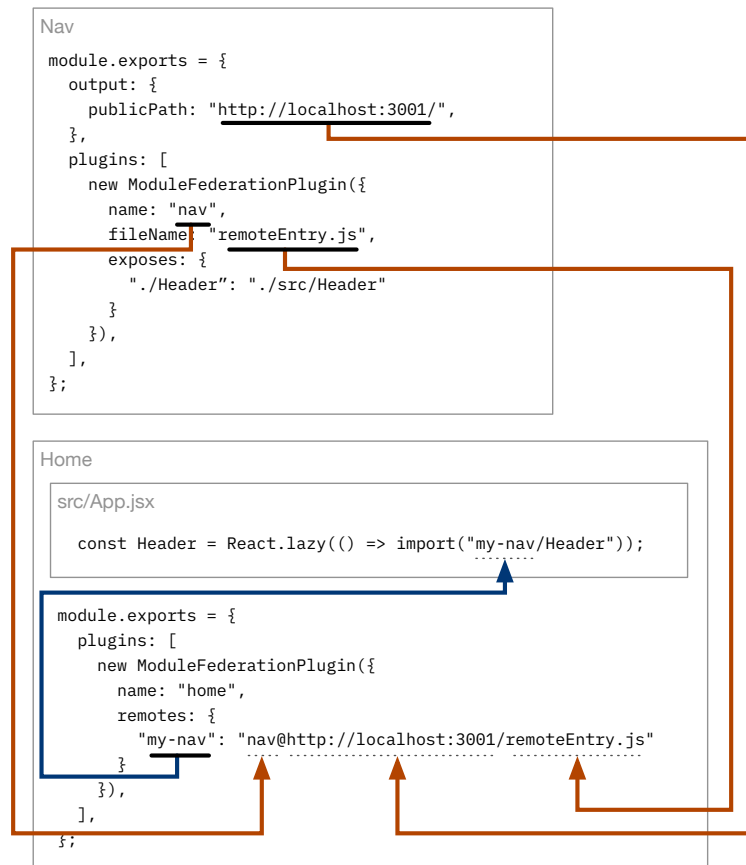
const App = () => (
  <div className="mt-10 text-3xl mx-auto max-w-6xl">
    <React.Suspense fallback={<div />}>
      <Header />
    </React.Suspense>
    <div className="mt-10">Home page</div>
  </div>
);
```

This loads the `Header` lazily using an asynchronous import and then renders it within a `Suspense`. This allows React to only render the component when the code is downloaded and ready to go.

And that's all it takes to share a component between these two applications. In fact, you can try it for yourself by modifying the code in the `Header` component and see the changes immediately after refreshing the host page.

GETTING NAMING RIGHT

It's important to know what parts of the web pack configuration of one application need to match in the corresponding parts of the consuming applications web pack config. In the figure below we show the key parts of the Nav applications web pack config and how they map into the host application's Webpack config.



The `publicPath` is the base of the remote URL which ends with the `fileName`. And then before that is the remote name defined using `name` in the Module Federation Plugin configuration. You can also change the name of the import by changing the key to whatever you like, for example in this case `my-nav`.

PROPERLY SHARING REACT

Throughout this book we will define the shared key in the `ModuleFederationPlugin` configuration object like this:

```
shared: { ... }, // Shared React dependencies
```

This is just shorthand for this full configuration:

```
const deps = require("./package.json").dependencies;
...
shared: {
  ...deps,
  react: {
    singleton: true,
    requiredVersion: deps.react,
  },
  "react-dom": {
    singleton: true,
    requiredVersion: deps["react-dom"],
  },
},
```

It's nothing sinister, this configuration is long and it would have taken up a lot of space on the page. But it is worth talking about why this shared configuration is set up this way. What we are doing is pulling the dependencies, as an object, from the `package.json` file. We are then setting the shared object to match the object from `package.json`, but we are overriding `react` and `react-dom` to set them to singleton mode and pin them to their specific current version.

They are marked as singletons because those two libraries; `react` and `react-dom` have internal state and you can't have multiple copies of them on the same page. That qualifies them as “singletons” and telling Webpack this ensures that it will never load more than one copy of `react`, or `react-dom` on the page at one time.

The value of the spread of the `deps` object (i.e. `...deps`) is that all of the other runtime dependencies for the application are added to the list of shared libraries automatically. So if you bring in `antd`, or `lodash`, those will be automatically shared.

It's worth noting that `react` and `react-dom` aren't the only singletons out there. For example, most CSS-in-JS libraries are also singletons. And if you find that you are seeing troubles sharing something `@emotion/core` then you should consider treating that exactly the same way as `react`, like so:

```
shared: { ...
  "@emotion/core": {
    singleton: true,
    requiredVersion: deps["@emotion/core"],
  },
},
```

Any time you have globally shared state within a module, which is very easy to do, then your library becomes a singleton.

The different modes for the `shared` key are explained in more in the appendices at the end of the book. But if you are using React then we strongly recommend adopting this standard pattern.

MIGRATING EXISTING APPLICATIONS

So what do you do if you have an existing Webpack 4 application and you want to expose parts of it to other applications, or use module federation to consume parts from other applications. The most direct route is to upgrade from Webpack 4 to Webpack 5 and then add the `ModuleFederationPlugin` and customize it using the techniques we've discussed so far. Thankfully the Webpack documentation has [excellent upgrading documentation](#) you can follow.

If you don't want to upgrade you might consider creating a sidecar Webpack 5 project in a subdirectory of the application. This sidecar application would include Webpack 5 along with any dependencies from the original project. The `webpack.config.js` file for the sidecar would then use the `ModuleFederationPlugin` to expose the files from the host application using relative references to the source. This sidecar technique is described and demonstrated in chapter on Alternative Deployment Options.

WHAT'S NEXT

The `create-mf-app` command is one way to build applications that use Module Federation in the next chapter we will show how to use Module Federation in applications created using Create React App.

1.3

GETTING STARTED WITH CREATE REACT APP

Code for this chapter is [available on GitHub](#).

Create React App has updated to the latest Webpack which makes it much easier to use Module Federation. Module Federation support isn't baked into Create React App but that doesn't mean you have to eject to be able to use it. You can use [react-app-rewired](#) or [craco](#), both of which have been standard ways to extend Create React App applications for several versions now. In the case of craco there is even a [Module Federation plugin](#) that is available to make it even easier and we will use that in this chapter as well.

In this chapter there are four applications that you can use as starting points.

- **Host** - This is a Create React App (CRA) that uses react-app-rewired to modify the Webpack configuration and it consumes components from the other applications.
- **Search** - This is a CRA application that uses craco to modify the Webpack configuration and has a function that reads all the files from a given directory and exposes them. This functionality can be used in any context.
- **Checkout** - This is a CRA app that uses craco and the craco-module-federation plugin to configure the application to share a component.
- **Carousel** - This is a create-mf-app based application that is shown here to demonstrate interoperability with the CRA applications.

Let's go through each of the applications one by one.

HOST APP USING REWIRED

We build the host application using standard create react app in the packages directory:

```
% cd packages  
% yarn create react-app host
```

After that we add react-app-rewired using yarn:

```
% yarn add react-app-rewired -D
```

Then we alter the package.json scripts like so:

```
"scripts": {  
  "start": "PORT=3000 react-app-rewired start",  
  "build": "react-app-rewired build",  
  "test": "react-app-rewired test",  
  "eject": "react-app-rewired eject"  
},
```

That will not only allow us to override parts of the Create React App configuration but also allow us to tweak the Webpack configuration. It also sets the port to 3000 so that its reliably at that port number.

The final step is to create a new `config-overrides.js` file, like so:

```
const ModuleFederationPlugin = require("webpack/lib/container/
ModuleFederationPlugin");

const deps = require("./package.json").dependencies;
module.exports = function override(config) {
  config.output.publicPath = "auto";

  if (!config.plugins) {
    config.plugins = [];
  }

  config.plugins.unshift(
    new ModuleFederationPlugin({
      name: "home",
      ...
    })
  );
  return config;
};
```

The critical pieces here are to set the `publicPath` appropriately and to add the `ModuleFederationPlugin` with your desired settings. We use `unshift` here to make sure that the plugin is the first along the line of plugins (which is an extensive list).

In the [example code](#) we configure the `ModuleFederationPlugin` to remote in all of the other applications and to export a `Container` component as well as a `Header` and a `Footer`.

This set of applications use styled-components to expose components that manage their own CSS. A very common question around Module Federation (an Micro-Frontends more broadly) is how to manage CSS. You cannot federated CSS files directly. You can only federate JavaScript. So a CSS-in-JS solution, like styled-components, is an excellent way to share components that have self-contained styling.

Don't Forget To Bootstrap

As with any application that uses Module Federation you have to bootstrap the application, and we have do that to all of the applications in the example code. If you don't do this your application will fail to boot and complain in the console about not being able to eagerly load a dependency.

The fix for this is straightforward. Copy the contents of `src/index.js` to a new file named `src/realIndex.js` (or whatever you choose). Then replace `src/index.js` with:

```
import("./realIndex");
```

It's that easy. What this does is give Webpack some “breathing room” to be able to asynchronously load any dependencies required by Module Federation.

SEARCH APP USING CRACO

Another way to modify a CRA application to do Module Federation is to use craco. We start by first create a fresh CRA application named search. Then we add @craco/craco as a development dependency.

From there we need to alter the package.json to run craco instead of react-scripts like so:

```
"scripts": {  
  "start": "PORT=3001 craco start",  
  "build": "craco build",  
  "test": "craco test",  
  "eject": "craco eject"  
},
```

This runs the application, with craco overrides, on port 3001.

We specify the overrides in a craco.config.js file, shown below:

```
const ModuleFederationPlugin = require("webpack/lib/container/
ModuleFederationPlugin");
...
const exposeDirectory = (dirName) =>
  fs.readdirSync(dirName).reduce((exposes, file) => {
    exposes[`./${file.replace(/[\.]*/, "")}`] = `${dirName}/${file}`;
    return exposes;
  }, {});
module.exports = {
  webpack: {
    configure: (config) => {
      config.output.publicPath = "auto";

      if (!config.plugins) {
        config.plugins = [];
      }

      config.plugins.unshift(
        new ModuleFederationPlugin({
          ...
          exposes: exposeDirectory("./src/federated"),
          ...
        })
      );

      return config;
    },
  },
};
```

With craco you return an object and within that object you provide a key for webpack where, among other things, you can provide a configure function like we have here.

In that configuration function we added the `ModuleFederationPlugin`, just like we did with `react-app-rewired`. Though in this case we used a cool `exposeDirectory` function to expose all of the modules located within a specific directory. In this case that's `src/federated`. This means that any file located in `src/federated` will automatically be exposed.

The straight forward `exposeDirectory` function answers another common Module Federation question; “How do I expose an entire directory of modules?” As you can see it's pretty easy to do, but it's not a feature that is built into Webpack.

CHECKOUT USING CRACO-MODULE-FEDERATION

Another option to use Module Federation with CRA is to use `craco` and the `craco-module-federation` plugin. The checkout application in the example code was built using this method. We created the application using Create React App and added `craco` as above. But we also added `craco-module-federation` as a development dependency.

After altering the scripts in `package.json` to run `craco` instead of `react-scripts`, we create the `craco.config.js` file that contains:

```
module.exports = {
  plugins: [
    {
      plugin: require("craco-module-federation"),
    },
  ],
};
```

This just registers the plugin with `craco`, so now we have to configure it in `modulefederation.config.js`, like so:

```
const deps = require("./package.json").dependencies;

module.exports = {
  name: "checkout",
  ...
};
```

What the plugin does for us is automatically register the `ModuleFederationPlugin`. The configuration of that plugin is left to us to build out using this configuration file.

WHAT'S NEXT

Now that you've got your first experience with Module Federation it is time to dig into the details of how to share code between applications safely. And that starts with a deep dive into the world of sharing React components.

14

HOW MODULE FEDERATION WORKS

Now that we understand where Module Federation fits into the world, and we've tried it out on a very basic application, let's increase our depth of knowledge by exploring how Module Federation works.

Let's first look at how Module Federation works at build time.

BUILD TIME

The `ModuleFederationPlugin` we've been using thus far is really just syntactic sugar over three interrelated plugins - it serves as a convenience plugin that passes options to the separated parts that perform the heaving heavy lifting. And you can use each of those individually and customize how you see fit so that you have ultimate control over the output.

These three plugins are:

- **ContainerPlugin** - This plugin manages the export the remote modules that you define in the `exposes` key of the configuration. It also creates the remote entry file that acts as a manifest for the application, which is called the "scope" internally. If your application only exposes remote modules then this is the only plugin you need. This is what a remote uses.

- **ContainerReferencePlugin** - This plugin manages the remotes in the configuration. This is what a host uses.
- **SharePlugin** - This plugin manages the shared portion of the configuration. It handles all of the versioning requirements for the shared packages. Shared is also referred to as “overrides” both internally and externally. This use used by both remote and host.

To get a deeper look at what’s built by Webpack a good way to do that is run the build in development mode and to look at the output of the `dist` directory, like this:

```
webpack --mode development
```

In an example project that exposes one remote module, `Header`, the example output is shown below:

Asset	Size		
index.html	171 bytes	[compared for emit]	
main.js	2.93 KiB	[emitted]	[name: main]
remoteEntry.js	12.1 KiB	[emitted]	[name: nav]
src_header.js.js	2.3 KiB	[emitted]	
vendors-node_modules_react_index_js.js	73.7 KiB	[compared for emit]	[id hint: vendors]
Entrypoint main = main.js			
Entrypoint nav = remoteEntry.js			

Each of these files and its role is individually described below.

- **index.html** - This is the compiled HTML file that includes the `main.js` file which runs the application.
- **main.js** - The compiled code for the main entry point for the application.

- **remoteEntry.js** - The manifest Javascript **and specialized runtime** for the exported remote modules and any shared packages.
- **src_header_js.js** - The compiled Javascript for the Header component. This is referenced in the `remoteEntry.js` file.
- **vendors-node_modules_react_index_js.js** - The compiled react package that supports the Header component which is shared.

It's important to understand that these files are a fusion of the Javascript bundles required to run the application itself as well as the Javascript bundles required for the remote modules. And often times those two overlap. For example, the same vendor bundles that are referenced by the remote modules are used by the application itself.

Now that the Javascript is all compiled, let's discuss how it actually works at runtime.

MODULE FEDERATION AT RUNTIME

When the `remoteEntry.js` is loaded by the browser it registers a global variable with the name specified in the `library` key in the `ModuleFederationPlugin` configuration. This variable has two things in it, a `get` function that returns any of the remote modules, and an `override` function that manages all of the shared packages.

For example you have an Application with the name of `nav` that exposes a runtime module named `Header`. After loading the code you would be able to open up the console and inspect `window.nav` and see that it has two functions, `get` and `override`.

With the `get` function we can get the `Header`, like so:

```
window.nav.get('Header')
```

This returns a promise, which when resolved gives you a factory. We can invoke that like this:

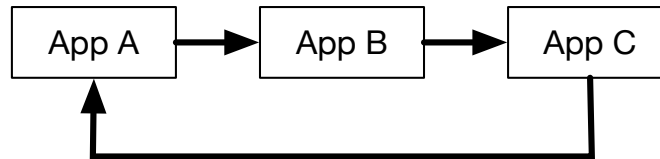
```
window.nav.get('Header').then(factory => console.log(factory()));
```

This will output through `console.log` the module as defined in the `Header` implementation. If `Header` exports a default the the value returned from `factory()` will be an object with a key named `default`. If the module has other named exports those will also be attached to the object.

A side effect of running the factory is also to load any shared packages required by the remote module. Webpack is good about only loading the shared packages required to support each module. For example, if component A imports `react` and `lodash`, but component B only imports `react`, running the factory on B would only bring in `react`.

In addition, Module Federation is smart enough to resolve shared packages between remotes. For example, a host application named `host` consumes two remotes: `nav` and `search`. Both `nav` and `search` require `lodash`, but `host` does not. Whichever remote is loaded first will load its version of `lodash`, and if the other remote module is loaded, it will use the already loaded `lodash` as long as its version parameters are satisfied.

Important Note: Circular imports and nested remotes are supported. For example App A imports a module from App B, App B imports an exported module from App C, which imports an expose module from App A. This is pictured in the diagram below.



Webpack will resolve these circular imports correctly and efficiently.

WHAT'S NEXT

Now that we've dug more deeply into Module Federation and how it works we will jump back into the world of practical implementation and cover how to safely deploy our federated modules.

1.5

DEPLOYING FEDERATED MODULES

Code for this chapter is [available on GitHub](#).

A very common question around Module Federation is; “This is really cool, but how do I deploy it?” Let’s answer that question in this chapter.

BUILDING FOR PRODUCTION

The first thing you need to know when you are going to production is the base URL from where you are going to serve the JavaScript bundles. This is important because Webpack loads your federated modules at runtime using URLs that are constructed relative to the `publicPath` defined in the configuration. Where we set the `publicPath` to `http://localhost:3000/` in development mode we might set it to `https://assets.mycompany.com/carousel` if this is the `carousel` project (just an example.)

Once we know where we are deploying to now we just need to change the webpack configuration when we are building for production. But how do build for production? We would do something like:

```
webpack --mode production
```

And that will send the production argument to the Webpack configuration.

Now that we know we are in production mode how do we switch the configuration based on that?

We can turn the `webpack.config.js` from an object into a function like this:

```
const HtmlWebPackPlugin = require("html-webpack-plugin");
const ModuleFederationPlugin = require("webpack/lib/container/
ModuleFederationPlugin");

const deps = require("./package.json").dependencies;
module.exports = (env, argv) => ({
  output: {
    publicPath:
      argv.mode === "production"
        ? "https://assets.mycompany.com/carousel"
        : "http://localhost:3000/",
  },
  ...
});
```

Then we can use `argv.mode` to point the `publicPath` (which is the value that we set to let Webpack know where it's deploying) based on whether the mode is `production` or not.

You can also do this automatically by specifying `publicPath` as `auto`, like so:

```
output: {
  publicPath: "auto",
},
```

The remotes section is not so easy, however. We can't just use auto. So we can use the same logic over in the remotes section to define where the remote code is located, like so:

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    carousel: `${
      argv.mode === "production" ?
        "https://assets.mycompany.com/carousel" :
        "http://localhost:3000"
    }/remoteEntry.js`,
  },
  exposes: {},
  ...
```

This code switches the remote based on whether we are building the host application in either production or development mode.

This does raise an interesting question though about what to do when the code is being run in development mode. Maybe you want to test your development code against the production remotes? If so you'll probably want some additional logic that looks at other arguments, for example `argv.remotes` where you can set that to `production` as well.

There is a [sample project in the production directory](#) of this chapter's GitHub code that is set up to demonstrate switching between development and production modes. First install the dependencies by running `yarn` in the root directory. To run in development mode run `yarn start` in the top level directory. To build for production mode run

`yarn build`, and `yarn serve` to run in production mode. With this sample project the production code is located on port 8081, where the development code is on 3001.

WHERE SHOULD FEDERATED MODULES GO?

Federated modules are just JavaScript files, and as such should be deployed to an asset service like Amazon's S3 service. Serving the files directly from an Express server, or any other kind of server is a waste of server processing power and money. If you have access to a Content Distribution Network (CDN) like Akamai Cloudfront you can use that to further optimize the delivery of the JavaScript assets.

EXTERNAL REMOTES

It's not unusual to have three (or more) different environments. For example; development, test, and production. Where the test environment has a complete copy of the production environment with test data. In some companies it's required that exactly the same built code runs in both test and production. This can be a problem for Module Federation because the remote URLs are stored in the built bundle. So if you build for a test deployment URL you would need to rebuild for production.

To get around this you can use the `external-remotes-plugin` which allows you to replace portions of the remote URLs at runtime. To see this in action there is an [example project](#) in the chapter's code on GitHub. In this project there are two Micro-Frontends; MFE1 and MFE2, but both are built with the same federation name "mfe" so they can be used interchangeably by just pointing at either the `remoteEntry.js` file on port 3000 or 3001 depending on if you want MFE1 or MFE2 respectively.

The host application has the `external-remotes-plugin` installed and adds it to the list of plugins in the Webpack configuration.

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    mfe: "mfe@[mfeUrl]/remoteEntry.js",
  },
  exposes: {},
})
```

The `[mfeUrl]` in the brackets will be replaced by a value named `mfeUrl` on the window object at load time. To make use of this feature the main `index.js` file gets a little more complicated:

```
fetch("/public/config.json")
  .then((res) => res.json())
  .then((config) => {
    window.mfeUrl = config.mfeUrl;
    import("./App");
  });
```

We first fetch some data from a `config.json` file located in and served from the public directory.

The config file looks like this:

```
{  
  "mfeUrl": "http://localhost:3001"  
}
```

So in this case it sets the `mfeUrl` on window to be port 3001 on localhost, which is MFE2. So now when the application loads the MFE it will load MFE2. You can change this to 3000 and see it load from MFE1 **without rebuilding the bundle**.

Of course, in your environment you'll probably want to get the value for your external remotes using either a service or by inspecting the current URL.

WHAT'S NEXT

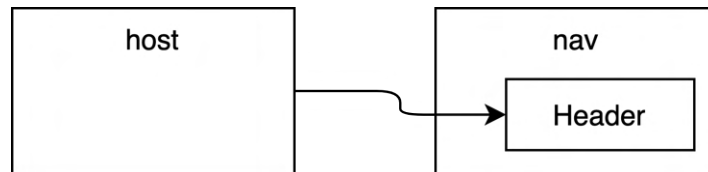
Now that we've taken module federation to production, let's talk more about how to make shared code and components safer in production.

1.6

RESILIENT SHARING OF REACT COMPONENTS

Code for this chapter is [available on GitHub](#).

Let's visualize the connection between the host app and the nav app we created in the last chapter. The host app has a runtime connection to the Header.



But any runtime connection can be broken. So let's try that out by simply starting the host server *without* running the nav app. The result is that the page is broken. No content is rendered at all.

And in the console we see that `nav` is undefined:

```
Uncaught ReferenceError: nav is not defined  
while loading "Header" from webpack/container/reference/nav
```

Which is clearly not a great customer experience.

Let's have a look at how reference the header in the code. We use a `React.lazy` import and we've wrapped it in a `Suspense`:

```
<React.Suspense fallback={<div />}>  
  <Header />  
</React.Suspense>
```

Apparently a suspense is not enough.

What we need is an error boundary to make this system more resilient. To create an “error boundary” we follow the [documentation on the React site](#), by adding a component to host application’s `App.jsx` file.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // logErrorToMyService(error, errorInfo);
  }

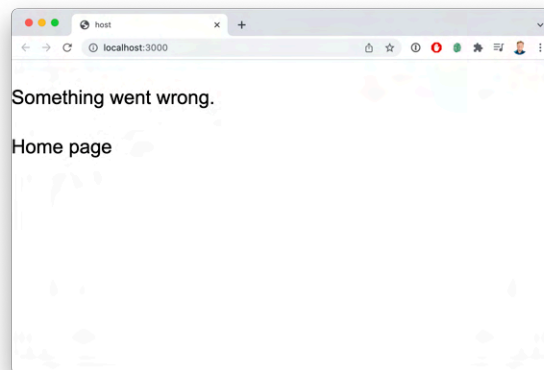
  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

And then we wrap the Suspense in this new `ErrorBoundary` component.

```
const App = () => (  
  <div>  
    <ErrorBoundary>  
      <React.Suspense fallback={<div />}>  
        <Header />  
      </React.Suspense>  
    </ErrorBoundary>  
    <div>Hi there, I'm React from React.</div>  
  </div>  
)
```

It's sad that we can't use a hook for this, but unfortunately React does not currently support hooks for error boundaries. The good news is that the page renders and we get our error message instead.



Which is certainly an improvement. But we could do better, and it starts by looking at the `ErrorBoundary` code. The important method is `getDerivedStateFromError` which is what creates the boundary. So let's rewrite this `ErrorBoundary` component into a `FederatedWrapper` component that handles both the delayed execution case

covered by `React.Suspense` and the error case handled by `getDerivedStateFromError`.

Shown below is the code for `FederatedWrapper` that handles both a slow load and an error during component execution.

```
class FederatedWrapper extends React.Component {
  ...// Same as in the ErrorBoundary class

  render() {
    if (this.state.hasError) {
      return this.props.error || <div>Something went wrong.</div>;
    }

    return (
      <React.Suspense fallback={this.props.delayed || <div />}>
        {this.props.children}
      </React.Suspense>
    );
  }
}
```

And to use this wrapper component all you need to do is wrap the `Header` component in the `FederatedWrapper` with messaging alternatives for the error and delay properties.

```
<FederatedWrapper
  error={<div>Temporary Header</div>}
  delayed={<div>Loading header...</div>}
>
  <Header />
</FederatedWrapper>
```

Now this works and is resilient to multiple types of errors.

WHAT HAPPENS WHEN THE APP GOES DOWN?

A question we get asked a lot is “what happens when the apps go down”? This is a by-product of the fact that we most often demonstrate Module Federation by using two running applications. All of the assets, including the federated modules produced by the Module Federation plugin are shared by the Express server.

It’s really important to realize that federated modules are just JavaScript files. They have no connection to the application server. So whether the server is running or not should not effect their availability. In fact, they should be deployed as you would any other JavaScript file, to an asset store like Amazon’s S3 service.

A good way to conceptually think about federated modules, from the both the deployment and security angles, is as *fancy bundle splitting*. You deploy federated modules the same way you deploy your JavaScript bundles. And the security footprint of federate modules is the same as other JavaScript bundles.

HIGH ORDER ERROR HANDLING

Building on the FederatedWrapper we can create a high order component (HOC) that takes a lazy component and returns a component with the same API

```
const wrapComponent = (Component) => ({ error, delayed, ...props }) => (  
  <FederatedWrapper error={error} delayed={delayed}>  
    <Component {...props} />  
  </FederatedWrapper>  
);
```

This new wrapComponent function takes a React component as an argument and returns a new component that takes the properties for error and delayed and sends any additional properties onto the wrapped component.

This means that we can then create the Header component using this wrapper:

```
const Header = wrapComponent(React.lazy(() => import("nav/Header")));
```

And use it in our application just as if we imported it directly.

```
const App = () => (  
  <div>  
    <Header />  
    <div>Hi there, I'm React from React.</div>  
  </div>  
);
```

This creates an API for our React components that we import via Module Federation that's both easy to use, and resilient to errors.

BOOTSTRAPPING APPLICATIONS

Another important safety tip is to “bootstrap” your application. Bootstrapping means that your main entry point should be to a file whose job is to asynchronously load the main application. For example, you would have an `index.js` file, which is the main entry point for Webpack, and its contents would be:

```
import("./App");
```

And the local `App.jsx` file would have the application, like so:

```
import ReactDOM from "react-dom";  
ReactDOM.render(<div>Hello world</div>, document.getElementById("app"));
```

This “bootstrapping” gives Webpack the opportunity to process the rest of the imports before executing the app and will avoid potential race conditions on importing all the code.

A non-bootstrapped version of this application would have an `index.js` file with the contents:

```
import ReactDOM from "react-dom";
ReactDOM.render(<div>Hello world</div>, document.getElementById("app"));
```

And since `index.js` is the main entry point of the application (which is the default for Webpack) then this code will execute immediately and not give Webpack any time to load the required remotes before execution.

If this non-bootstrapped `index.js` were to contain a remote component, like so:

```
import ReactDOM from "react-dom";
import Header from "nav/Header";
ReactDOM.render(<Header />, document.getElementById("app"));
```

Then Webpack would not have the opportunity to load the `nav` remote before that code executes, and it would result in a runtime error.

All of the examples in this book are bootstrapped.

WHAT'S NEXT

In the next chapter we discuss the different state sharing options available to you in federated React applications.

1.7

REACT STATE SHARING OPTIONS

Code for this chapter is [available on GitHub](#).

Once you have a way to share code between applications, whether that be through NPM modules or through Module Federation the inevitable next conversation that arises is how to share state between the host and the remote modules, or even between remotes. This chapter explores some options into how to share state.

GETTING THE SETUP RIGHT

Let's continue using the two example applications; host and nav, that we've been using thus far. And let's experiment a little by removing the contents of the shared array in the ModuleFederationPlugin configuration in `webpack.config.js`. So that it would look like this:

```
new ModuleFederationPlugin({
  ...
  shared: {},
}),
```

Then restart the applications using `yarn start` and you'll notice that it still works. Which is odd. We can even see the `react` library being imported along with the header in the browser's network tab.

But now let's break this application by making one small change to the `Header` component.

```
import React from "react";

const Header = () => {
  const [stateIDontUse] = React.useState(null);

  return (
    <header className="text-5xl font-bold p-5 bg-blue-500 text-white">
      I'm the header!
    </header>
  );
};

export default Header;
```

The original `Header` was stateless, but now lets add some state by using a `useState` hook in the `Header`.

Looks good, but here is what we see in the console when it runs:

```
Uncaught Error: Invalid hook call. Hooks can only be called inside of the
body of a function component. This could happen for one of the following
reasons:
```

1. You might have mismatching versions of React and the renderer (such as React DOM)
2. You might be breaking the Rules of Hooks
3. You might have more than one copy of React in the same app

This error is telling us what we already know; there are two `react` library instances on the page. But we didn't get the error until we tried to use hooks, and that's because hooks are internally stored as global state within the `react` library. So everything was fine until we used even the most basic hook, and boom, we got an error.

It's important to understand this behavior, and the importance of sharing all of your runtime dependencies for this reason; any state sharing library is likely to have global state, or use features of the framework that require global state. You always need to share these libraries.

To fix this we use this configuration when we share `react` and `react-dom`.

```
const deps = require("./package.json").dependencies;
...
shared: {
  ...deps,
  react: {
    singleton: true,
    requiredVersion: deps.react,
  },
  "react-dom": {
    singleton: true,
    requiredVersion: deps["react-dom"],
  },
},
```

This defines `react` and `react-dom` as singletons which tells Webpack to make sure that there is never more than one copy of `react` or `react-dom` on the page at a time.

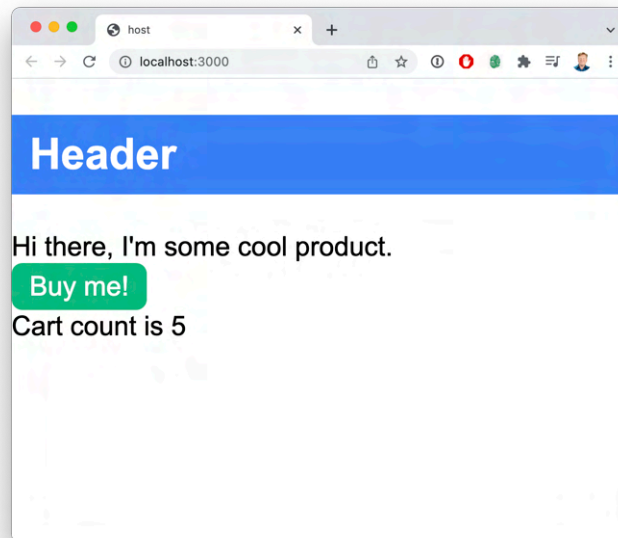
With some understanding of how these libraries can go right and wrong, it's time to dig into some specific state sharing options.

The simplest and cleanest way to share state is to use `useState`. So let's start with that and change the host app so that you can add items to a cart and the Header in nav so that it can display the count of items in the cart.

```
const App = () => {
  const [itemCount, setItemCount] = useState(0);
  const onAddToCart = () => {
    setItemCount(itemCount + 1);
  };
  return (
    <div className="mt-10 text-3xl mx-auto max-w-6xl">
      <React.Suspense fallback={<div />}>
        <Header />
      </React.Suspense>
      <div className="mt-10">Hi there, I'm some cool product.</div>
      <button
        className="px-5 py-2 bg-green-500 text-white rounded-xl"
        onClick={onAddToCart}>
        >Buy me!</button>
      <div>Cart count is {itemCount}</div>
    </div>
  );
};
```

This new version of the App component has its own state that contains a cart count.

After pressing the “Buy me!” button five times the result in the browser looks like the figure below:



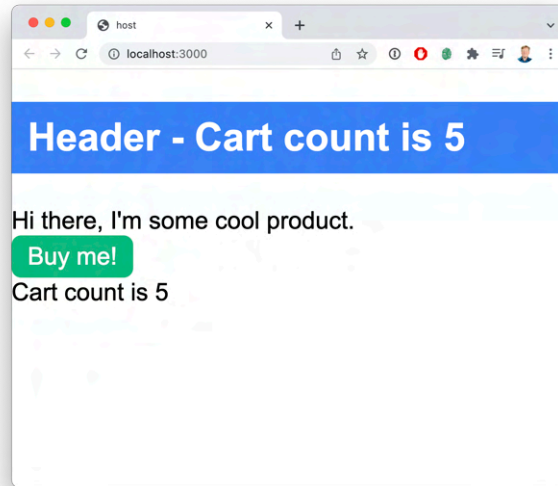
And from here we can send this state onto the Header by adding a property:

```
<Header count={itemCount} />
```

And we can change the Header to display that value.

```
const Header = ({ count }) => {  
  return (  
    <header className="text-5xl font-bold p-5 bg-blue-500 text-white flex">  
      <div className="flex-grow">Header - Cart count is {count}</div>  
    </header>  
  );  
};
```

And the result is that we now have a `Header` from the `nav` application that tracks with the cart count state that's stored in the host application.



And now to make it bi-directional let's pass through an `onClear` callback to enable the cart clearing button in the `Header`. So in the host application we change the `Header` component reference to:

```
<Header count={itemCount} onClear={() => itemCountSet(0)} />
```

Now we are passing the `onClear` to the header.

So we can change the Header implementation changes to:

```
const Header = ({ count, onClear }) => (  
  <header style={{ fontSize: "xx-large " }}>  
    <span>Header - Cart count is {count}</span>  
    <button onClick={onClear}>Clear</button>  
  </header>  
)
```

And that completes the round circuit where have state store in the application that's read and updated from a React component in a Federated Module.

It's not surprising this worked. This is the simplest and most straightforward way to share state in React. But it does provide a solid baseline to compare with some other approaches.

USE-CONTEXT

Obviously prop-drilling is only going to get you so far, and it's certainly no surprise that it works. But what about using a React context? That works as well and it provides an excellent setup for looking at other state managers like Redux, Mobx, Zustand, etc. since they use context as well.

So let's start with the `use-state` example code that we have to start and migrate that to a `use-context` example that uses React context to share state.

The first thing to do is to create a shared store file in `packages/host/src/store.js` like so:

```
import React, { createContext, useContext, useState } from "react";

const CountContext = createContext(0);

export const CountProvider = ({ children }) => (
  <CountContext.Provider value={useState(0)}>
    {children}
  </CountContext.Provider>
);

export const useCount = () => useContext(CountContext);
```

This is a pretty standard, if simplistic, context setup. We have two exported values, `CountProvider` which is a component that you use at the top of your tree to share the state down the tree. And `useCount` which is a custom hook that gives you the count and the count setter.

The next thing we need to do is augment the Webpack configuration in `packages/host/webpack.config.js` to both add the host (ourselves) as a remote and also share the store.

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    "my-nav": "nav@http://localhost:3001/remoteEntry.js",
    host: "host@http://localhost:3000/remoteEntry.js",
  },
  exposes: {
    "./store": "./src/store",
  },
  shared: { ... },
}),
```

Why are we adding ourselves as a remote? Because we want both the `host` and `nav` applications to look at the same instance of the `store` module, and importing both through the remote ensures that we will.

Now we alter the packages/host/src/App.jsx file to both provide and use the context.

```
import React, { useState } from "react";
import ReactDOM from "react-dom";

import Header from "my-nav/Header";
import { CountProvider, useCount } from "host/store";

import "./index.scss";

const App = () => {
  const [itemCount, setItemCount] = useCount();
  const onAddToCart = () => {
    setItemCount(itemCount + 1);
  };
  return (
    <div className="mt-10 text-3xl mx-auto max-w-6xl">
      <Header />
      ...
    </div>
  );
};

ReactDOM.render(
  <CountProvider>
    <App />
  </CountProvider>,
  document.getElementById("app")
);
```

We start at the bottom by providing the context down the tree. Then in the App component we use the new useCount custom hook to get the current state value and state setter. And we can also remove the drilled properties from the header.

Now over in the nav project we need to update the Webpack configuration, located in `pacakges/nav/webpack.config.js`, to add the host application as a remote:

```
new ModuleFederationPlugin({
  name: "nav",
  filename: "remoteEntry.js",
  remotes: {
    host: "host@http://localhost:3000/remoteEntry.js",
  },
  exposes: {
    "./Header": "./src/Header",
  },
  shared: { ... },
}),
```

Now that we have access to the host as a remote we can update the Header component.

We navigate over to `packages/nav/src/Header.jsx` and alter the code to use the `count` context:

```
import React from "react";

import { useCount } from "host/store";

const Header = () => {
  const [count, setCount] = useCount();
  return (
    <header className="text-5xl font-bold p-5 bg-blue-500 text-white flex">
      <div className="flex-grow">Header - Cart count is {count}</div>
      <button onClick={() => setCount(0)}>Clear</button>
    </header>
  );
};

export default Header;
```

This works, but now the `Header` embedded in the `App` component of the `nav` application no longer works because we don't have the context.

So let's add the `CountProvider` to the `App` component as well:

```
import { CountProvider } from "host/store";

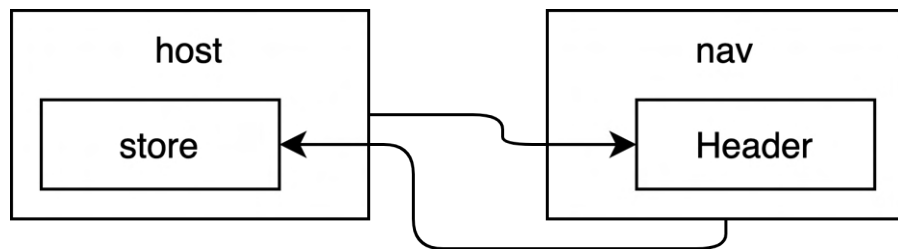
const App = () => (
  <CountProvider>
    <div className="mt-10 text-3xl mx-auto max-w-6xl">
      <Header />
      <div className="mt-10">Nav project</div>
    </div>
  </CountProvider>
);
```

And now both applications work just fine and communicate through the shared context. Cool, right?

Now that we know we can share context this way we can start looking at using more complex state managers. But before we get into that, let's talk about some alternatives when it comes to sharing the `store` module because the current model we will use works fine for a single application, but probably won't scale up cleanly to multiple host applications.

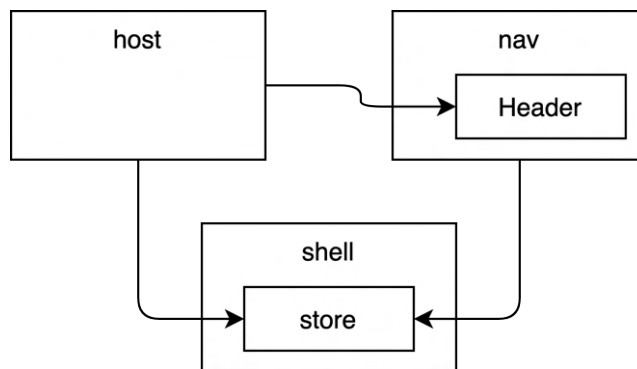
SHARING STORES

As we saw in the `use-context` example the `host` application exposed a `store`, and the `nav` application consumed it.



So why is that important? In the case of our `use-context` implementation you have to share the same *instance* of the module because it's the instance that holds the global context for the state.

Of course, scaling this particular sharing architecture up could be problematic if we had multiple applications that could provide the store. So we could go with something like this:



Where we have `shell` federated module that holds the `store` and is referenced by any application that needs it or wants to provide it.

For simplicity sake we will be using having the host application provide the store in the different scenarios that follow. But you should pick the state sharing setup that is most appropriate to your architecture.

REDUX

If you've done any React you've heard of Redux, it's easily the most commonly used state manager. To see how Redux works in the Module Federation context we start with simple project we finished in the Getting Started chapter. From there we add the Redux Toolkit `@reduxjs/toolkit` and `react-redux` to host and nav applications.

```
% cd packages/host
% yarn add @reduxjs/toolkit react-redux
% cd ../nav
% yarn add @reduxjs/toolkit react-redux
```

Now we have Redux installed. And we can build our store.

That means creating a `packages/host/store.js` file that looks like this:

```
import { configureStore, createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: {
    count: 0,
  },
  reducers: {
    increment: (state) => {
      state.count += 1;
    },
    clear: (state) => {
      state.count = 0;
    },
  },
});

export const { increment, clear } = counterSlice.actions;

export const store = configureStore({
  reducer: {
    counter: counterSlice.reducer,
  },
});
```

This very simple Redux store has an initial state with a count of zero. And there are two actions, `increment`, will add one to the count, and `clear` will set the count to zero.

The next thing we want to do is to share the store externally (more about this later).

Let's add the store to the Webpack config like so:

```
new ModuleFederationPlugin({
  name: "host",
  ...
  exposes: {
    "./store": "./src/store",
  },

```

Our next step is to import and add a `Provider` to the application.

```
import { Provider, connect } from "react-redux";

import { store, increment } from "host/store";
```

And we then change the `ReactDOM.render` invocation to wrap our `App` in a `Provider` with the store .

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("app")
);
```

This is a critical step because this `Provider` is how we are going to get the state to any of the consumers. The big question is whether it will work from the host application through Module Federation into the nav app.

Before we get there we need to adjust the App React component to use this store.

```
const App = () => {
  const count = useSelector((state) => state.counter.count);
  const dispatch = useDispatch();

  return (
    <div className="mt-10 text-3xl mx-auto max-w-6xl">
      <React.Suspense fallback={<div />}>
        <Header />
      </React.Suspense>
      <div className="mt-10">Hi there, I'm some cool product.</div>
      <button
        className="px-5 py-2 bg-green-500 text-white rounded-xl"
        onClick={() => dispatch(increment())}
      >
        Buy me!
      </button>
      <div>Cart count is {count}</div>
    </div>
  );
};
```

Modern Redux uses the `useSelector` hook to get the state values we are interested in, and the `useDispatch` hook to get a dispatch function that we can use to dispatch actions.

The only thing left to do is update the Header so that it connects to the store properly. That starts with allowing the nav app to see the store by adjusting its Webpack configuration.

```
new ModuleFederationPlugin({
  name: "nav",
  filename: "remoteEntry.js",
  remotes: {
    host: "host@http://localhost:3000/remoteEntry.js",
  },
  exposes: {
    "./Header": "./src/Header",
  },
})
```

Now we can access the store in the host application. This means that at this point host depends on nav for the Header component and nav depends on host for the Redux store. That's fine. That's a bi-directional connection and Module Federation supports that just fine.

So let's head over to the nav application and alter the `App.jsx` file to look like this:

```
import React from "react";
import { useSelector, useDispatch } from "react-redux";

import { clear } from "host/store";

const Header = () => {
  const count = useSelector((state) => state.counter.count);
  const dispatch = useDispatch();

  return (
    <header className="text-5xl font-bold p-5 bg-blue-500 text-white flex">
      <div className="flex-grow">Header - Cart count is {count}</div>
      <button onClick={() => dispatch(clear())}>Clear</button>
    </header>
  );
};

export default Header;
```

And this works just fine. Turns out that you can access your Redux store seamlessly in React components integrated using Module Federation.

Zustand is a popular alternative Redux that has a similar level of structure to a Redux store but has far less boilerplate. Let's try it out by starting with our use-context implementation and adding in the zustand library to the host application.

```
% cd packages/host
% yarn add zustand
```

In this case we don't need to add zustand to the nav app because as long as we have the custom hook that Zustand creates we are good to go.

The next thing we need to do is change the packages/host/src/store.js file to read:

```
import create from "zustand";

export const useCount = create((set) => ({
  count: 0,
  increment: () => set((state) => ({ count: state.count + 1 })),
  clear: () => set((state) => ({ count: 0 })),
}));
```

I'm joking, right? This is a store? Not joking, no. And yes, this is a Zustand store. You have the values at the top, in this case count, then the methods you want. And the methods just call set to set the state. How cool is that! No wonder folks love Zustand.

Now the next thing we need to do is use that over in the packages/host/src/App.jsx file:

```
import React, { useState } from "react";
import ReactDOM from "react-dom";

import Header from "my-nav/Header";
import { useCount } from "host/store";

import "./index.scss";

const App = () => {
  const { count, increment } = useCount();
  return (
    <div className="mt-10 text-3xl mx-auto max-w-6xl">
      ...
    </div>
  );
};
ReactDOM.render(<App />, document.getElementById("app"));
```

We also get to take the context out because all you need is the new useCount custom hook that was created by Zustand.

To complete the migration we need to go over to the `packages/nav/Header.jsx` file and update the code over there.

```
import React from "react";

import { useCount } from "host/store";

const Header = () => {
  const { count, clear } = useCount();
  return (
    <header className="text-5xl font-bold p-5 bg-blue-500 text-white flex">
      <div className="flex-grow">Header - Cart count is {count}</div>
      <button onClick={clear}>Clear</button>
    </header>
  );
};

export default Header;
```

You can check out the complete setup for yourself in the [example code associated with this chapter](#). I think you'll come away impressed with just how easy Zustand is to use. You can use Zustand both within the React context and outside of the React context as well. It also works very well with asynchronous workflows.

Mobx is another popular state manager for React applications. It works an observables model. To try out Mobx in Module Federation we first need to install it in both the host and nav applications.

```
% cd packages/host
% yarn add mobx mobx-react
% cd ../nav
% yarn add mobx mobx-react
```

Then we create the store, which is as easy as adding this code to a new packages/host/store.js file:

```
import { observable } from "mobx";

const store = observable({
  count: 0,
});

export default store;
```

We then pass the store to the App component by re-writing the ReactDOM.render invocation.

```
ReactDOM.render(<App store={cartStore} />, document.getElementById("app"));
```

And then to finish the implementation of the App we wrap it an observer.

```
import { observer } from "mobx-react";

...

const App = observer(() => {
  return (
    <div className="mt-10 text-3xl mx-auto max-w-6xl">
      <React.Suspense fallback={<div />}>
        <Header store={store} />
      </React.Suspense>
      <div className="mt-10">Hi there, I'm some cool product.</div>
      <button
        className="px-5 py-2 bg-green-500 text-white rounded-xl"
        onClick={() => (store.count += 1)}
      >
        Buy me!
      </button>
      <div>Cart count is {store.count}</div>
    </div>
  );
});
```

We've removed the state from App and now we just use the count as a regular Javascript object. To set it all we need to do is just set its value and the observable handles all the rest. We also need to send the store to Header as a property.

Then over in the Header implementation in the nav application we also wrap that in an observer.

```
import React from "react";
import { observer } from "mobx-react";

const Header = () => {
  return (
    <header className="text-5xl font-bold p-5 bg-blue-500 text-white flex">
      <div className="flex-grow">Header - Cart count is {store.count}</div>
      <button onClick={() => (store.count = 0)}>Clear</button>
    </header>
  );
};

export default observer(Header);
```

Once again, we display the data simply by showing the value. And we reset the value just by setting it to zero. And this all works seamlessly across these components brought together through Module Federation.

The reason all this works is because we are sharing the same instance of the observable from the store exposed by the host application.

RECOIL

Recoil is a novel state management system from FaceBook that is designed to extend the existing `useState` and `useContext` mechanisms by providing state that is “orthogonal to the rendering tree”. Meaning that `useState` and `useContext` relate directly to where they are used in the React tree. When `useState` or `useContext`

change anything below where they are defined in the React tree will be re-rendered. Recoil on the other hand stores its state as “atoms” that are held apart from the tree. So when an atom changes state then only those components that are using it will re-render.

Let’s try it out and you can see more of what I mean. The first step is to install `recoil` in both the `host` and `nav` applications.

```
% cd packages/host
% yarn add recoil
% cd ../nav
% yarn add recoil
```

With that done we can launch the servers with `yarn start`. We then create a new file in the `src` directory of the home app called `atoms.js` with these contents.

```
import { atom } from "recoil";

export const cartCount = atom({
  key: "cartCount",
  default: 0,
});
```

This creates a new “atom” that holds the cart count value. In the recoil model state is stored as atoms and components subscribe to those atoms. If an atoms value changes then any component that subscribes to it will be updated. You can have as many atoms as you like. And you can use another mechanism called a “selector” to create composite atoms from a collection of other atoms.

To use the `cartCount` atom we return to the `App.jsx` file and import both `recoil` and the atoms:

```
import { useRecoilState, RecoilRoot } from "recoil";  
import { cartCount } from "../atoms";
```

We then wrap the `App` component in a `RecoilRoot` as part of our `ReactDOM.render`.

```
ReactDOM.render(  
  <RecoilRoot>  
    <App />  
  </RecoilRoot>,  
  document.getElementById("app")  
);
```

Next we change `useState` to `useRecoilState` and point it at the `cartCount` atom.

```
import { cartCount } from "host/atoms";

const App = () => {
  const [itemCount, setItemCount] = useRecoilState(cartCount);
  const onAddToCart = () => {
    setItemCount(itemCount + 1);
  };
  return (
    <div className="mt-10 text-3xl mx-auto max-w-6xl">
      <React.Suspense fallback={<div />}>
        <Header />
      </React.Suspense>
      <div className="mt-10">Hi there, I'm some cool product.</div>
      <button
        className="px-5 py-2 bg-green-500 text-white rounded-xl"
        onClick={onAddToCart}
      >
        Buy me!
      </button>
      <div>Cart count is {itemCount}</div>
    </div>
  );
};
```

One of the great things about Recoil is how it's almost a plug and play replacement for `useState`. What do you do if more than one component need the state? Make an atom and point both components at it by just replacing `useState` with `useRecoilState`. It's very clean.

Notice that the Header now no longer has any properties. So to make that work we first need to export the atoms from home by altering the `webpack.config.js`.

```
new ModuleFederationPlugin({
  ...
  exposes: {
    './atoms': './src/atoms',
  },
  ...
}),
```

Then over in the nav application new need to alter the `webpack.config.js` to add host as a remote.

```
new ModuleFederationPlugin({
  ...
  remotes: {
    Host: "host@http://localhost:3000/remoteEntry.js",
  },
  ...
}),
```

This is creating a bi-directional linkage between host and nav so that we can share the same instance of the atoms. Just as we have shared the same instance of the store in previous examples.

To finish the example we use the `useRecoilState` hook to both get and set the value of the `cartCount` atom:

```
import React from "react";
import { useRecoilState } from "recoil";

import { cartCount } from "host/atoms";

const Header = () => {
  const [itemCount, setItemCount] = useRecoilState(cartCount);
  return (
    <header className="text-5xl font-bold p-5 bg-blue-500 text-white flex">
      <div className="flex-grow">Header - Cart count is {itemCount}</div>
      <button onClick={() => setItemCount(0)}>Clear</button>
    </header>
  );
};

export default Header;
```

And it's as easy as that. When the user clicks on the button the atom is updated and all the consumers of the atom are updated automatically.

RxJS is a very popular event bus library. It's not often used as a state mechanism, but there is no reason why it can't be done.

Let's try it out and you can see it in action. The first step is to install `rxjs` in the host application.

```
% cd packages/host  
% yarn add rxjs
```

The next thing to do is to import `Subject` from `rxjs` and to create a `Subject` that will hold the cart count.

```
import { Subject } from "rxjs";  
const count = new Subject(0);
```

Subjects in RxJS are event busses. And in this case we are starting the topic off with the value of zero. You can post a new value by using the `next` method on the object. And you can subscribe to the bus by calling to `subscribe` method.

To integrate it with the App component in the App.jsx file in home we will first create a new custom React hook called useSubject. It's defined like this:

```
const useSubject = (subject, initialValue) => {
  const [value, valueSet] = useState(initialValue);

  useEffect(() => {
    const sub = subject.subscribe(valueSet);
    return () => {
      sub.unsubscribe();
    };
  }, [subject]);

  return [
    value,
    {
      increment: () => count.next(value + 1),
      clear: () => count.next(0),
    },
  ];
};

export const useCount = () => useSubject(count, 0);
```

This hook takes the subject and its initial value and uses a standard useState to hook to track its state. It then subscribes to that subject and updates the state any time it changes.

Passing in the initialValue is a hack because we didn't use a BehaviorSubject that can return the most recently posted value at any time. There are several different types of subjects each with different semantics. I chose Subject because it's the simplest type and I wanted to show that it works with the simplest possible contract.

So, now that we have our hook we can use it in the App component.

```
const App = () => {  
  const [count, { increment }] = useCount();  
  return (  
    <div className="mt-10 text-3xl mx-auto max-w-6xl">  
      <React.Suspense fallback={<div />}>  
        <Header />  
      </React.Suspense>  
      <div className="mt-10">Hi there, I'm some cool product.</div>  
      <button  
        className="px-5 py-2 bg-green-500 text-white rounded-xl"  
        onClick={increment}  
      >  
        Buy me!  
      </button>  
      <div>Cart count is {count}</div>  
    </div>  
  );  
};
```

We use the useCount hook we just created to get the current value of items in the cart. And then we use the next method on the count subject to increase its value, or set it to zero in the case of the clear.

Now we could just send `itemCount` to the `Header`, but instead let's send the subject so that it too can subscribe to it. To do that we change `Header.js` in the `nav` project. The result is shown in the code fragment below:

```
import React from "react";

import { useCount } from "host/store";

const Header = () => {
  const [count, { clear }] = useCount();
  return (
    <header className="text-5xl font-bold p-5 bg-blue-500 text-white flex">
      <div className="flex-grow">Header - Cart count is {count}</div>
      <button onClick={clear}>Clear</button>
    </header>
  );
};

export default Header;
```

And we reuse the same `useCount` code from `host`, which is critical because we have to share the same instance of the subject to get the state sharing to work.

Using RxJS as an Analytics Bus

Another, probably more appropriate use of RxJS would be for something like an analytics bus. Where any component could post events to the bus that would then be sent to an analytics service.

To set that up we first create a new file in `host` named `analytics.js` with these contents:

```
import { Subject } from "rxjs";
const analyticsBus = new Subject();
export default analyticsBus;
```

The next step is to expose that file (module) in the `webpack.config.js`:

```
exposes: {
  "./analytics": "./src/analytics",
},
```

That allows us to import it using the name `host`, like this in `App.jsx`.

```
import analyticsBus from "home/analytics";
```

It's important that we import it by referring to `home` to ensure that both the `App` component and the `Header` component get exactly the same `analytics` subject. Now that we have it imported we can attach a console function to it using `subscribe`.

```
analyticsBus.subscribe((evt) => {
  console.log(`analytics: ${JSON.stringify(evt)}`);
});
```

Next we can start to instrument our code, like the `increment` and `clear` function from `useSubject` which now sends out analytics messages:

```
return [  
  value,  
  {  
    increment: () => {  
      analyticsBus.next({ type: "addToCart", value: value + 1 });  
      count.next(value + 1);  
    },  
    clear: () => {  
      analyticsBus.next({ type: "onClear" });  
      count.next(0);  
    },  
  },  
];
```

Using RxJS for an analytics event bus in a way that's more in-line with its original purpose. It also opens up the opportunity to put loggers, filters, mutations and much more into the bus to work the analytics events in whatever way you please. RxJS provides a lot of tooling for that.

HOW TO CHOOSE

The choice of a state manager is highly dependent on the type of application you are building. But here are some recommendations.

For applications that will be using Module Federation to bring in code to extend the existing interface, like a customer extensible dashboard, then using a conventional `useState` approach with property drilling and well defined callbacks. This provides the safest and most well-defined interface without additional package requirements.

For an internal dashboard, where the modules will pick and choose between a variety of data then recoil seems like an interesting solution because it would limit the re-renders to just those components subscribed to the specific atom that is being updated.

An e-Commerce application, or any application that has very limited data sharing requirements (user identity, cart contents, etc.) between federated modules using either a few Recoil atoms or a small Redux store would work well.

There are many state managers in the React ecosystem and this chapter has only covered a few. If you are starting with a new project you should develop a set of requirements and then evaluate and run proof of concepts of several solutions so you can evaluate how well they match your requirements and how the developer experience feels.

WHAT'S NEXT

Module Federation is about sharing more than components and state, any type of Javascript can be shared. In the next chapter we cover multiple techniques to add resilience to any type of function or data that's shared by Module Federation.

1.8

RESILIENT FUNCTION AND STATIC DATA SHARING

Code for this chapter is [available on GitHub](#).

Important Note: In the previous chapters we have followed a walkthrough style of teaching Module Federation concepts. From this chapter on we will be walking through the completed applications available to you in the [GitHub repository](#). You are strongly encouraged to use these projects as a starting point for your own work where you can experiment and see how to suit the code to meet your own needs.

One of the most unique and important aspects of Module Federation is that you can share anything that can be represented as a Javascript module. This includes:

- **Primitives** - Numbers, strings, Dates, etc.
- **Arrays**
- **Objects** - From regular objects to JSON that's transpiled by Babel
- **Functions** - And not just React component functions
- **Classes** - Which are really just functions, but need to be invoked with new

To emphasize this point we'll walk through some examples and show how they are exported and consumed, and also show different ways to manage them safely.

THE LOGIC PROJECT

To experiment with different types of modules to federate we've created an application named `logic` that exposes modules with several different types of signatures. The `webpack.config.js` exposes all of these relevant files.

```
new ModuleFederationPlugin({
  name: "logic",
  filename: "remoteEntry.js",
  remotes: {},
  exposes: {
    "./analyticsFunc": "./src/analyticsFunc",
    "./arrayValue": "./src/arrayValue",
    "./classExport": "./src/classExport",
    "./objectValue": "./src/objectValue",
    "./singleValue": "./src/singleValue",
  },
  shared: {},
}),
```

This is then consumed by a home project that works with the different types of data.

PRIMITIVE VALUES

Let's start with the simplest case, the single value. Looking at the `singleValue.js` file we see it implemented as just a default return of a string.

```
export default "single value";
```

But that single value could be any primitive, a number, a boolean, whatever.

In the host application we then consume it using an asynchronous `import` within a `useEffect` hook that runs only once.

```
const SingleValue = () => {
  const [singleValue, singleValueSet] = React.useState(null);

  React.useEffect(() => {
    import("logic/singleValue")
      .then(({ default: value }) => singleValueSet(value))
      .catch((err) => console.error(`Error getting single value: ${err}`));
  }, []);

  return <div>Single value: {singleValue}</div>;
};
```

This code creates some state to hold the value using a `useState` hook. It then uses the `useEffect` hook to start the `import`. The `import` can either return successfully, in which case we get back an object with the `default` key that we map to `value` and then use to set the `singleValue`. In case of an error we send that error to the console.

And at the end we render the result into a `<div>` tag in our application.

So what we are seeing is that importing values, of any type, just requires an asynchronous `import` to bring in safely. You can use synchronous imports in some cases, but to build a resilient code base you should import the code asynchronously and handle the case when you are unable to get the code you need.

Arrays and objects work exactly the same as the single value case and you can have a look at the book code to see this for yourself.

FUNCTIONS

It starts to get more interesting as we look at functions. Let's take an example of an analytics sending function. The example code defines it like this:

```
export default (msg) => console.log(`Analytics msg: ${msg}`);
```

This is exported as `nav/analyticsFunc`. And we want to be able to run it at any time. The simplest alternative is to write a small wrapper function like this:

```
const analyticsFunc = import("logic/analyticsFunc");
const sendAnalytics = (msg) => {
  analyticsFunc
    .then(({ default: analyticsFunc }) => analyticsFunc(msg))
    .catch((err) => console.log(`Error sending analytics value: ${msg}`));
};
```

Here is the interesting thing about promises, they will resolve immediately if the promise has already been resolved. So in this case you can send as many calls as you like to this function and once it resolves the first time it will unlock the rest and send everything out.

If you want something more generic you can create a functor that takes an `import` with a function as the default and returns a function that you can call in the same way as `sendAnalytics` above.

```
const createAsyncFunc = (promise) => (...args) =>
  promise
    .then(({ default: func }) => func(...args))
    .catch((err) =>
      console.log(`Error sending value: ${JSON.stringify(args)}`)
    );

const sendAnalytics = createAsyncFunc(import("logic/analyticsFunc"));
```

This high order `createAsyncFunction` takes an asynchronous `import` promise and returns a function that takes whatever arguments you have and sends them to the function once it has resolved.

If you want to queue up messages to the function until it resolves we can create a different high order function that maintains a queue of messages to go to the function before it resolves.

```
const queuedFunction = (funcPromise) => {
  let queueFunc = null;
  const queue = [];
  let pending = false;

  return (msg) => {
    if (queueFunc) {
      queueFunc(msg);
    } else {
      queue.push(msg);

      if (!pending) {
        pending = true;
        funcPromise
          .then((func) => {
            queueFunc = func;
            queue.forEach(queueFunc);
            queue = [];
          })
          .catch((err) => console.log(`Error getting queued function`));
      }
    }
  };
};
```

This function takes an asynchronous `import` promise, just like the one above. But until it resolves it maintains a queue of messages that it then sends to the function using the `forEach`.

To create the `sendAnalytics` function using this high order `queueFunction` functor you call it this way:

```
const sendAnalytics = queuedFunction(  
  import("logic/analyticsFunc").then(({ default: func }) => func)  
);
```

CLASSES

The last thing to try out is a non-React class. The test class in the logic application is defined this way.

```
class MyClass {  
  constructor(value) {  
    this.value = value;  
  }  
  
  logString() {  
    console.log(`Logging ${this.value}`);  
  }  
}  
  
export default MyClass;
```

To bring this into the home app we create a function called `newClassObject` that takes arguments and sends them to the constructor once we have it.

```
const classExport = import("logic/classExport");
const newClassObject = (...args) =>
  classExport
    .then(({ default: classRef }) => {
      return new classRef(...args);
    })
    .catch((err) => console.log(`Error getting class: ${err}`));
```

This code starts the `import` and gets a promise back. It then creates a function that uses that promise and either throws an error if it catches or, if it works, returns a new object with the values it's given as arguments.

You invoke it and monitor it as a promise, as in the code below.

```
newClassObject("initial value").then((theObject) => {
  theObject.logString();
});
```

And the result is that you see in the console is:

```
Logging initial value
```

Exciting stuff! But it does show that you can safely consume and create objects from classes exported from remote modules.

The fact that Module Federation can share any type of Javascript code opens up a world of possibilities for your projects. Here are some ideas for you:

- **A/B testing code** - Code for different test variants that you can load on the fly - using asynchronous imports
- **Feature flags** - Javascript objects containing feature flags or settings
- **i18n strings** - The localization strings for different languages, lazily loaded
- **Persistent state management** - Javascript code to manage and store persistent state in local storage on the client

WHAT'S NEXT

Now that we know how Module Federation works and how to export and import any kind of code safely, the next chapter will cover how to load Federated Modules dynamically.

1.9

DYNAMICALLY LOADING FEDERATED MODULES

Code for this chapter is [available on GitHub](#).

So far the relationship between our host applications and our remotes has been fixed. For example the `host` page includes a reference to the `nav` app where it gets the remote `Header` module. But you don't have to do this. You can dynamically load the `remoteEntry.js` onto the page and then execute your own version of `import` to run the code.

The book code for this chapter has three directories;

- **widget** - This is a single React component that is shared as a remote module in the scope of `widget` and under the name `Widget`.
- **host-wp5** - This is a host application, running on Webpack 5, that dynamically loads and runs `widget`.
- **host-wp4** - This is a host application, running on Webpack 4, that dynamically loads and runs `widget`.

We'll start with having a look at the widget directory and the `webpack.config.js` within that directory.

```
new ModuleFederationPlugin({
  name: "widget",
  filename: "remoteEntry.js",
  remotes: {},
  exposes: {
    "./Widget": "./src/Widget",
  },
  shared: {...},
}),
```

The important elements here are that we export the `Widget` component and that the name of this application is `widget`. It's also good to see that we are running and exporting this remote module off of port 8080.

```
output: {
  publicPath: "http://localhost:8080/",
},
devServer: {
  port: 8080,
},
```

The React widget itself is super simple as shown in the code below.

```
import React from "react";

export default () => <div>I'm a cool widget</div>;
```

You can spruce it up if you like. But for now it's time to run `yarn start` in the `widget` directory and get the Webpack development server up and running.

Over in the `host-wp5` directory we have the host for the Widget remote module. If we have a look at its `webpack.config.js` we can see it only uses `ModuleFederationPlugin` to share `react`, other than that this is just a standard Webpack 5 configuration.

The real action happens over in `App.jsx` where we define a new React component called `System` that loads the `remoteEntry.js` script, gets the code in the module and then renders it.

```
function System(props) {
  const { ready, failed } = useDynamicScript({
    url: props.system && props.system.url,
  });

  if (!props.system) {
    return <h2>Not system specified</h2>;
  }

  if (!ready) {
    return <h2>Loading dynamic script: {props.system.url}</h2>;
  }

  if (failed) {
    return <h2>Failed to load dynamic script: {props.system.url}</h2>;
  }

  const Component = React.lazy(
    loadComponent(props.system.scope, props.system.module)
  );

  return (
    <React.Suspense fallback="Loading System">
      <Component />
    </React.Suspense>
  );
}
```

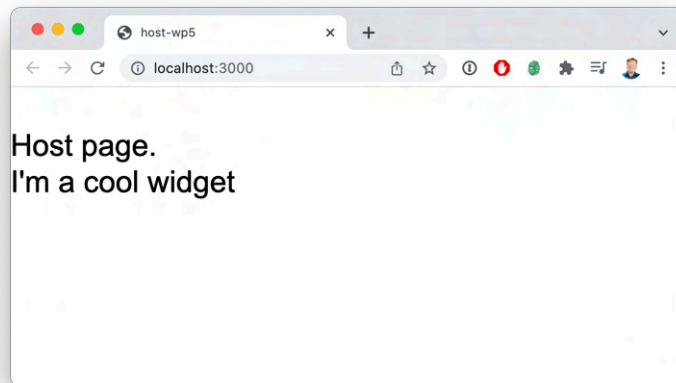
The `useDynamicScript` hook is a custom React hook that, given a URL, in this case for the `remoteEntry.js`, adds a script tag to the page and then sets `ready` to `true` when the script is loaded (or failed to `true` if that failed).

After that, we use `React.lazy` and `React.Suspense` as we have in many other examples in this book. However, where we would normally use an asynchronous `import()` statement, we are instead using `loadComponent`.

This function is doing what `import` does. It's using the `scope (widget)` and `module (Widget)` to get the module factory from the globally mounted object. Then it invokes that factory and returns the module. This is just what `React.lazy` wants to see.

The reason that `react` is called out in particular is that React is a singleton that requires that there is only a single instance on the page at a time. So this code initializes the `scope` with the host pages React library instance so that the `widget` module avoids loading its' own version.

The result when we run `yarn start` in the `home-wp5` directory (while the server is also running in widget) looks like this.



It might not look like much, but when combined with all of the potential types of code and visual components that we can use with Module Federation, the potential for what we can do if we load code from anywhere is astonishing.

However, what happens if you aren't on Webpack 5. Can you still do this?

CONSUMING FEDERATED MODULES FROM WEBPACK 4

It's great that we can consume federated modules on Webpack 5, but Webpack is complex and sometimes migration can be an involved process. Can we use federated modules today even in Webpack 4? Absolutely!

Looking at the `host-wp4` directory and the `package.json` within that directory you can see that the Webpack version is currently 4.

```
"devDependencies": {  
  ...  
  "webpack": "^4.43.0",  
  ...  
},
```

And the `webpack.config.js` file for this project has no mention of module federation as it was not available in version 4.

In the `src/App.jsx` file we can see the implementation for the host React component. It has the same `useDynamicScript` hook that we used in the Webpack 5 version just to get the code onto the page.

The difference is in the `DynamicWidget` component that uses `useDynamicScript` to get the code and then uses the factory functionality to give `React.lazy` the `Widget` module to run.

```
const DynamicWidget = ({ url, scope, module, ...props }) => {
  const { ready, failed } = useDynamicScript(url);

  ...

  window[scope].override(
    Object.assign(
      {
        react: () => Promise.resolve().then(() => () => require("react")),
      },
      global.__webpack_require__ ? global.__webpack_require__.o : {}
    )
  );

  const Component = React.lazy(() =>
    window[scope].get(module).then((factory) => {
      const Module = factory();
      return Module;
    })
  );

  return (
    <React.Suspense fallback="Loading System">
      <Component {...props} />
    </React.Suspense>
  );
};
```

The `React.lazy` and `React.Suspense` are basically the same as in the Webpack 5 example. The big difference is that we are manually setting the overrides for the

scope. We are telling with `widget` remote module that we already have `react` and that it should use our version instead of downloading its own version. This part is critical because otherwise there would be two copies of `React` on the page and we would have issues as are described in detail in the `React State Sharing` chapter. All of your shared packages need to be listed in this `override` call.

Once that's all set up we can bring in the `Widget` from the `widget` application just like any other `React` component.

```
const App = () => (  
  <div>  
    <DynamicWidget  
      url={"http://localhost:8082/remoteEntry.js"}  
      scope={"widget"}  
      module={"Widget"}  
    />  
    <div>Hi there, I'm React from React in Webpack 4.</div>  
  </div>  
);
```

It's a testament to the stability of the `Webpack` architecture that we can use cutting edge features from one version reliably in the previous version.

WHAT'S NEXT

In the next chapter we dig into how to share packages properly across federated modules.

1.10

SHARED LIBRARIES AND VERSIONING

If you take a look at any Javascript file in your application it's probably going to start with at least a few, if not a lot, of `import` statements, many using node modules or packages. That's just the nature of the Javascript ecosystem. But it does make for a challenge when we share code because the piece of code that brings in those imports needs those packages, and needs them at the right versions.

This is why getting the shared key in the `ModuleFederationPlugin` right is so critical.

- **Duplication** - Without sharing the packages you get duplication and this will slow down the experience.
- **Version mismatch** - If we don't get the sharing setup correctly then different remote modules could end up consuming incorrect versions which could lead to crashes.
- **Singletons and internal state** - Many libraries, particularly frontend libraries, have internal state that is required in order to run properly. That means there can only be one instance of that library running at a time. Which we call a "singleton".

This chapter walks through the different ways to configure the shared key with recommendations on when to use each of the different options.

SPECIFYING VERSIONS

There are three ways to specify shared versions. The first is the array syntax we've been using thus far with the names of each of the libraries to share. The second is an object syntax with the name of the library as the key, and the version (using [semantic versioning format](#)) as the value. It looks like this:

```
"shared": {  
  "react": "^16.12.0",  
  "react-dom": "^16.12.0"  
}
```

If that looks familiar that's because it looks exactly the same as it does in `package.json` and it works exactly the same way. In fact, because `webpack.config.js` is just a Javascript script you could do:

```
"shared": require('./package.json').dependencies
```

That is perfectly valid and potentially even preferable since it automatically shares out any runtime dependencies with their versions.

However, you can be even more precise about the control by using the third form where you hint Webpack in exactly how to manage the shared dependency.

FALLBACK

Webpack has a fallback behavior if the package provided as shared doesn't match the requirements specified in the `ModuleFederationPlugin` or `SharePlugin` configuration.

Here is an example, both applications A and B consume `lodash` but at different versions. And both have copies of `loads` available to them as potential fallbacks. A loads first and bring in its own version (1.0) of `lodash`. B is starting to load and looking for `lodash` but requires a later version (2.0). With the right version hinting to Webpack the B remote module will “fall back” on its version of `lodash` because of that version mismatch.

This fallback feature enables versioning deployments at scale because not all applications will be required to push new versions at the same. time.

WEBPACK SHARING HINTS

There is another variant of the object syntax where the key is the name of the package and the value is another object containing different hints to modify the behavior of Webpack's `SharePlugin`.

For example, this shared configuration.

```
"shared": {  
  "react": {  
    singleton: true  
  },  
  "react-dom": "^16.12.0"  
}
```

Will inform Webpack that react should be treated as a singleton, meaning that under no loading circumstances should there ever be more than one copy of react running at a time.

The react-dom package however is packaged as normal using semantic versioning (semver) rules.

The sections that follow detail the different hinting options you can send to Webpack's SharePlugin through ModuleFederationPlugin, or directly, if you prefer that level of control.

import

This is the import name of the package. Here is an example:

```
"shared": {  
  "react": {  
    import: "react"  
  }  
}
```

This will import the react package from the module reference "react".

shareKey

This is the name that will be used to identify the package within the scope. Here is an example:

```
"shared": {  
  "react": {  
    shareKey: "react"  
  }  
}
```

This will share the package the "default" scope under the key "react".

shareScope

This is the scope that will be used to store the shared package. Here is an example:

```
"shared": {  
  "react": {  
    shareScope: "default"  
  }  
}
```

This will share the package the "default" scope under the key "react".

singleton

Toggles singleton status for this shared package. If singleton is truthy then only one instance of this package should be running at any time. Here is an example:

```
"shared": {  
  "react": {  
    singleton: true  
  }  
}
```

This enables the management of the react package as a singleton.

strictVersion

The `strictVersion` flag tells Webpack to use the fallback if there is no version match. If the `singleton` flag is enabled and there is no fallback then Webpack will throw an error . Here is an example:

```
"shared": {  
  "react": {  
    singleton: true,  
    strictVersion: true,  
    version: "^16.12.0"  
  }  
}
```

This configuration tells Webpack that react is a singleton and it must be in the specified semantic version of `"^16.12.0"`, and if nothing is available then fallback or throw.

version

The version of the package. Here is an example:

```
"shared": {  
  "react": {  
    version: "^16.12.0"  
  }  
}
```

This configuration tells Webpack that react should be used at the specified semantic version.

requiredVersion

The version of the package that is required otherwise Webpack should use the fallback. Here is an example:

```
"shared": {  
  "react": {  
    requiredVersion: "^16.12.0"  
  }  
}
```

This configuration tells Webpack that react must be used at the specified semantic version.

As you can see a bunch of thought has gone into making a reliable versioning mechanism for shared packages in Module Federation.

WHAT'S NEXT

Before we wrap on the basics of Module Federation lets take a look at how we can use Module Federation to power experiences using frameworks other than React.

1.11

NON-REACT VIEW FRAMEWORKS

Code for this chapter is [available on GitHub](#).

Module Federation is strongly associated with React and Micro-Frontends, but it's just a mechanism for sharing JavaScript code, any type of JavaScript code including code for other frameworks.

At the time of this writing we tested these frameworks and they were all compatible with Module Federation: [Angular](#) (all versions), [Inferno](#), [Lit](#), [Mithril](#), [Preact](#), [Solid-JS](#), [Vue](#) (both 2 and 3). In addition of course you can federated Vanilla JS applications. In general most frameworks that use or are compatible with Webpack 5 can use Module Federation. Some static site generation systems, like [Astro](#), are incompatible with it because they generate HTML and not JavaScript.

In the [example code associated with this chapter](#) there are example projects in Vue and Solid-JS. Both were generated using `npx create-mf-app` which gives you an option to generate applications using most of the frameworks listed above.

THE VUE EXAMPLE

Vue is a very popular VDOM framework similar in architecture to React. In this example we have host and remote applications. The remote application share out one

component, named `Widget`, that we specify in `packages/remote/src/Widget.vue` like so:

```
<template>
  <div class="p-5 text-white border-dashed ...">
    My Cool Remote widget
  </div>
</template>
```

We then expose that widget in the remote application's Webpack configuration:

```
new ModuleFederationPlugin({
  name: "remote",
  filename: "remoteEntry.js",
  remotes: {},
  exposes: {
    "./Widget": "./src/Widget.vue",
  },
  shared: require("./package.json").dependencies,
}),
```

This exports the component and creates the necessary `remoteEntry.js` file.

We then reference that `remote` in the `remote` key within the host Webpack configuration.

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    remote: "remote@http://localhost:3001/remoteEntry.js",
  },
  exposes: {},
  shared: require("./package.json").dependencies,
}),
```

You can name the `remote` application whatever you want of course, and have as many of them as you want.

Finally we import the Widget into the host application and use it.

```
<template>
  <div class="mt-10 text-3xl mx-auto max-w-6xl">
    <div>Name: host</div>
    <Widget />
  </div>
</template>

<script>
import Widget from "remote/Widget";

export default {
  components: {
    Widget,
  }
}
</script>
```

It's just that easy! I have to say this is just as easy, or maybe even easier because of the elegant simplicity of the Vue developer experience, when compared to sharing React components between applications.

THE SOLID-JS EXAMPLE

Solid-JS is a very lightweight framework that uses “fine grained updates” to create wildly fast applications, nearing the speed of hand written Vanilla JS applications. Which is one reason why it's so popular.

We've created two applications in the example source code project using `create-mf-app`, they are named `host` and `remote` and the `remote` application shares a single

Widget component from `packages/remote/src/Widget.jsx`. The Widget code looks like this:

```
export default () => (  
  <div class="p-5 text-white border-dashed ...">  
    My Cool Remote widget  
  </div>  
)
```

Looks a lot like React, right? We then expose Widget using the remote application's Webpack configuration like so:

```
new ModuleFederationPlugin({  
  name: "remote",  
  filename: "remoteEntry.js",  
  remotes: {},  
  exposes: {  
    "./Widget": "./src/Widget",  
  },  
  shared: {...},  
})
```

This creates a `remoteEntry.js` manifest file.

We can point to in the host application's remote configuration value located in `packages/host/webpack.config.js`.

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    remote: "remote@http://localhost:3001/remoteEntry.js",
  },
  exposes: {},
  shared: {...},
}),
```

And then we can consume it just as easily as we can any other Solid-JS component in the `packages/host/src/App.jsx` file:

```
import Widget from "remote/Widget";

const App = () => (
  <div class="mt-10 text-3xl mx-auto max-w-6xl">
    <div>Name: host</div>
    <Widget />
  </div>
);
```

Now that is pretty slick, you have to admit!

WHAT'S NEXT

We've learned in this chapter that it's just as easy to federate code within non-React frameworks, in this case Vue and Solid-JS, as it to federate code between React applications.

We strongly encourage you to try this out for yourself. The `create-mf-app` application builder is a fantastic way to just try out new frameworks, even if you aren't playing around with Module Federation in the process.

Now that we have an understanding of the basics of Module Federation lets get on with seeing how it can be practically used in Part 2!

PART TWO

PRACTICAL USE CASES

2.1

RESILIENT HEADER/FOOTER

Code for this chapter is [available on GitHub](#). There is also a [Blue Collar Coder video](#) associated with this practical use case.

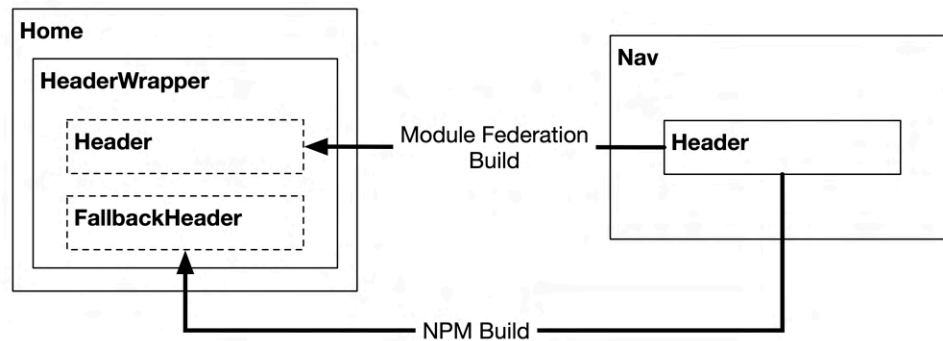
A very common architectural problem occurs in the move from a monolithic architecture where we have one big app with one huge frontend codebase, to a micro-site architecture where we have a bunch of smaller NodeJS applications. We gain the ability to release quickly and independently, but we lose code sharing. NPM is great but the process of getting a new NPM module rolled out and consumed is a huge source of friction. But some things absolutely must be shared, and chief among those is the shared Header and Footer.

Ideally we want a shared Header and Footer system with these attributes:

- **Live sharing** - We want all the micro-sites to update the header at the same time. Otherwise we get an inconsistent user experience, or worse.
- **Sharing without requiring version bumps** - The ideal system would not require the micro-sites to re-deploy with every update.
- **Safety and resilience** - The header should be absolutely safe and never take down the applications that consume it.

The great news is that Module Federation allows us to come up with a sharing architecture that fills all of these requirements.

In this example we will build a `React Header` project that exports both an NPM module and a Module Federation build. We then have a Home page application that consumes both of these builds.



The frontend code then uses the Module Federation build first, then if an error occurs, it falls back onto the NPM code which it loads lazily. The `Header` from Module Federation is exactly the same as the `Header` from NPM, which is aliased to `FallbackHeader` in the `HeaderWrapper`.

The only difference is that the `Header` from Module Federation and the one from NPM is that the Module Federation version is guaranteed to be up to date with the most recently released version (i.e. live sharing). Where the NPM version will be whatever version was imported the last time the Home page was built.

A fun way to think about this approach is like pitons in rock climbing. Module Federation is the climbing up the rock, but the NPM build is like the piton that keeps your application from falling too far if your grip slips.

Now that we know what we are building let's take a look at the [code for this chapter on Github](#).

The first place to look is in the `package.json` of the `nav` project located in `packages/nav` directory. This is the standard `package.json` we've used throughout this book, with small modifications in the package scripts and the addition of a `main` key.

```
"scripts": {  
  "build": "npm run build:mf && npm run build:npm",  
  "build:mf": "webpack --mode production",  
  "build:npm": "babel src --out-dir build",  
  ...  
},  
"main": "./build/index.js",
```

Now the build creates both a `dist` directory that has the contents of the Module Federation build, but also a `build` directory that has the NPM build. The `main` key then points the NPM consumer at the babel transpiled `index.js` file.

The `index.js` file looks like this:

```
import Header from "./Header";  
export { Header };
```

And the Header implementation file is a very simple React component.

```
import React from "react";

const Header = () => (
  <div className="bg-green-800 text-white font-bold text-2xl p-4">
    New Header
  </div>
);

export default Header;
```

In the [video associated with this code](#) we use the background color to differentiate between an older NPM build of the nav project and the current live Module Federation code.

Now that we understand the nav project and the Header component we can look to the consuming home application located in packages/home. The first thing to understand is that the overall project here, located in the root package.json, is a workspaces project. So it's managing the NPM linking between the home and nav projects.

You can see this in the `package.json` of the home app which references the nav at version 1.0.0.

```
"dependencies": {  
  "nav": "1.0.0",  
  "react": "^17.0.2",  
  "react-dom": "^17.0.2"  
}
```

To import the Module Federation version of the nav project the `webpack.config.js` in the home project has this `ModuleFederationPlugin` configuration.

```
new ModuleFederationPlugin({  
  name: "home",  
  filename: "remoteEntry.js",  
  remotes: {  
    "mf-nav": "nav@http://localhost:8081/remoteEntry.js",  
  },  
  exposes: {},  
  shared: { ... }, // Shared React dependencies  
}),
```

The interesting part here is that we are importing the nav remote but we are aliasing it to `mf-nav` within this project. That way in the code we can actively decide whether we are referencing the NPM version of the `Header` or the Module Federation version of the `Header`.

In fact you can see that right away in the `src/App.jsx` file where we lazily import both versions of the `Header`, like so:

```
const FallbackHeader = React.lazy(() => import("nav/build/Header"));
const Header = React.lazy(() => import("mf-nav/Header"));
```

We lazily import both versions, but for different reasons. The `FallbackHeader`, which is sourced from the NPM library, is lazily loaded because we don't want to bring in the NPM header unless the Module Federation version fails. The Module Federation `Header` is lazily loaded because a failure to do so would bring down the application if the import failed.

The `HeaderWrapper` is a React class because a React “Error Boundary” cannot be a functional components. It has two jobs, first is to catch any errors, which is handled by this code:

```
class HeaderWrapper extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError() {
    return { hasError: true };
  }
  componentDidCatch() {}
  ...
}
```


This render function renders the two different versions based on whether or not there has been an error.

```
render() {
  if (this.state.hasError) {
    return (
      <React.Suspense fallback={<div>Loading fallback header</div>}>
        <FallbackHeader />
      </React.Suspense>
    );
  }
  return (
    <React.Suspense fallback={<div>Header loading</div>}>
      <Header />
    </React.Suspense>
  );
}
```

In the case of an error we lazily render the `FallbackHeader`, and otherwise we render the Module Federation Header (which may cause an error and trigger the rendering of the `FallbackHeader`).

To try this all out first run `yarn` and `yarn build` at the top level of the project. From there you can run `yarn start` in each of the packages directories; `home` and `nav`. To check resilience simply stop the `yarn start` process on the `nav` project to see the home application fall back onto the NPM version.

Module Federation also provides additional levels of fallbacks for the Module Federation code by offering an array of externals. If the first one fails the second will be tried and so on.

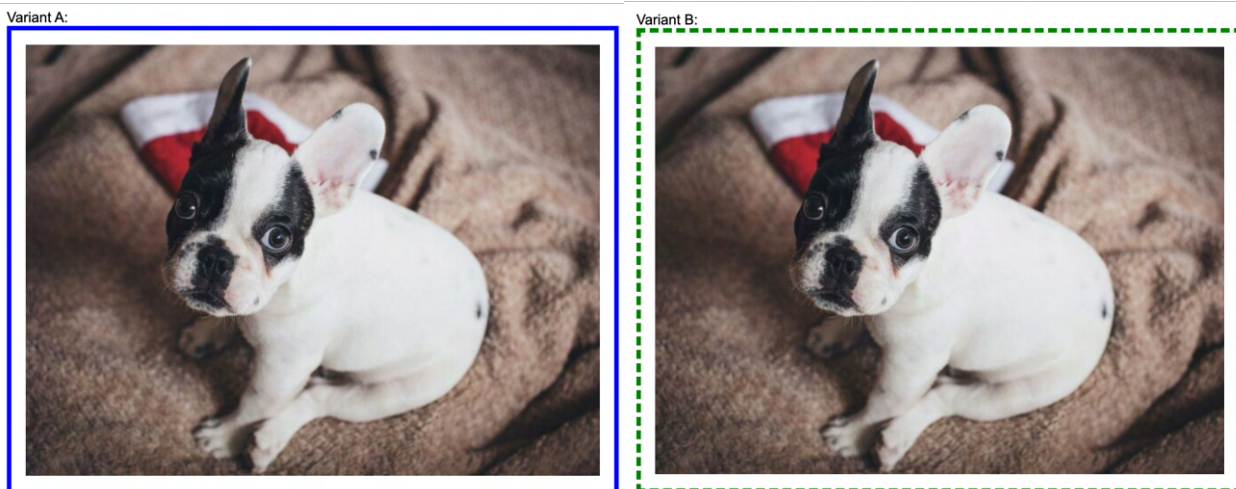
2.2

A/B TESTS

Code for this chapter is [available on GitHub](#). There is also a [Blue Collar Coder video](#) associated with this practical use case.

There are plenty of places to start on your journey towards implementing Module Federation across your org, and one of those is using it to implement A/B tests, and a shared A/B test manager.

A/B tests are a way of seeing true customer preferences through data. You take two variations on the same piece of interface. In the case of the example code we are comparing two different image borders:



Variant A has a blue border, and variation B has a green dashed border. In production this choice would be linked to some meaningful action, for example, clicking an “Adoption” button to start the adoption process.

There are three parts to this system:

- **A/B Test Code** - The code that renders the different variations
- **Test Manager** - Some code, or a service, that picks which option the customer will be presented to the customer
- **Results Manager** - This is the system that receives the analytics messages indicating which choice the customer choose, tabulates the results and chooses a “winner” after a statistically significant number of results have been gathered.

In this example we will be focused on the first two of these, the A/B test code and the test manager. In reality however, the test and results manager roles are most often filled by third party systems as services.

We start off the example by looking at the `packages/home` directory which contains a React application. Within that project in the `src` directory are the `FrameA` and `FrameB` components, both of which look like this:

```
import React from "react";
export default ({ src, style, ...props }) => (
  <>
    <div>Variant A:</div>
    <img
      src={src} style={{
        ...style,
        padding: "1em",
        border: "5px solid blue",
      }}
      {...props}
    ></img>
  </>
);
```

The `FrameA` and `FrameB` components only vary by the contents of the initial `<div>` tag and the styling around the image.

In the host `App.jsx` component we first lazily load these components.

```
const FrameA = React.lazy(() => import("host/FrameA"));
const FrameB = React.lazy(() => import("host/FrameB"));
```

We lazily load them because we don't want to load the variation unless it's going to be displayed. This isn't a huge concern in this case since the components are pretty small. But in real world experiments the code sizes can be larger.

We also load the `VariantChooser` from the `ab-manager` remote and invoke it like this:

```
import VariantChooser from "ab-manager/VariantChooser";

<VariantChooser
  test="test1"
  variations={{
    a: FrameA,
    b: FrameB,
  }}
  src="https://placedog.net/640/480?id=53"
  style={{ width: 640, height: 480 }}
/>
```

It's the job of the `VariantChooser` to roll the dice to see which variation we get, then to render the correct variation from the `variations` property. While also sending along the additional props (i.e. `src` and `style`). The `test` property defines the name of the test we are running, and what the possible values are. In this case the name is `test1` and the possible values are `"a"` and `"b"`.

Which brings us to the other project in this monorepo and that's the `ab-manager` project which holds the list of available tests, and the `VariantChooser` implementation.

In the `variants.js` file within `packages/ab-manager` we see the list of all of the possible tests and their corresponding variations:

```
export default {  
  test1: ["a", "b"],  
};
```

We then import that set of tests into the `VariantChooser` implementation:

```
import React from "react";  
import variants from "ab-manager/variants";  
  
const chooseVariant = (test) =>  
  variants[test][Math.floor(Math.random() * variants[test].length)];  
  
const VariantChooser = ({ test, variant, variations, ...props }) => {  
  const [testVariant] = React.useState(variant || chooseVariant(test));  
  const Component = variations[testVariant];  
  return (  
    <React.Suspense fallback={<div>Loading variant</div>}>  
      <Component {...props} />  
    </React.Suspense>  
  );  
};  
export default VariantChooser;
```

At the top we bring in the `variants`. We then define a `chooseVariant` helper function which just picks a random item from the test supplied. The component uses that to choose the variant to hold it in the state. It then renders the variant within a suspense.

The import of the `variants` is interesting. We could have just imported the `variants` from `'./variants'`, since it's located in the same directory. But by using the `'ab-`

manager/variants' version we are forcing the import to go through Module Federation. This ensures that variants.js gets split out from the VariantChooser.jsx code. And that makes variants.js independently deployable. So it would be possible to push a new version of the ab-manager with only the tests changed.

To get all this working we set up Module Federation this way in the ab-manager:

```
new ModuleFederationPlugin({
  name: "ab_mgr",
  filename: "remoteEntry.js",
  remotes: {
    "ab-manager": "ab_mgr@http://localhost:8080/remoteEntry.js",
  },
  exposes: {
    "./VariantChooser": "./src/VariantChooser",
    "./variants": "./src/variants",
  },
  shared: { ... }, // Shared React dependencies
}),
```

It's interesting to notice that the federated modules package is defined as ab_mgr but we then alias the remote to ab-manager. This is because the name "ab_mgr" is actually used as Javascript variable and a minus is not valid within a Javascript variable. So we use the underscore instead.

We also expose the VariantChooser and variants using the exposes key.

If we look at the `webpack.config.js` file in the `packages/home` directory where the home application is located we see the import of the `ab-manager`:

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    "ab-manager": "ab_mgr@http://localhost:8081/remoteEntry.js",
    "host": "host@http://localhost:8080/remoteEntry.js",
  }, exposes: {
    "./FrameA": "./src/FrameA",
    "./FrameB": "./src/FrameB",
  },
  shared: { ... }, // Shared React dependencies
}),
```

As well as the exposing of both of the `Frame` variations.

Specifying `host`, which is the current project, as a remote also allows us to bring in the `Frame` components using standard Module Federation imports:

```
const FrameA = React.lazy(() => import("host/FrameA"));
const FrameB = React.lazy(() => import("host/FrameB"));
```

To run this example run `yarn` and then `yarn start` in the root directory. That will launch both the `home` and `ab-manager` applications. You should then have a look at the network tab to see that the Webpack is only bringing in the required variation.

In the [Blue Collar Coder video](#) that accompanies this chapter we also change the values in the `variants.js` file to demonstrate that you can push just the list of variations without having to update the code in order to specify a winning variant.

2.3

LIVE PREVIEW BETWEEN APPLICATIONS

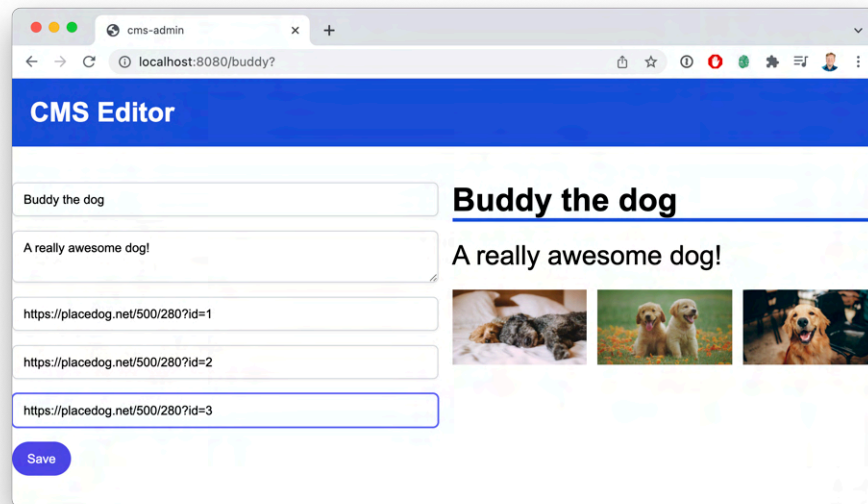


Code for this chapter is [available on GitHub](#). There is also a [Blue Collar Coder video](#) associated with this practical use case.

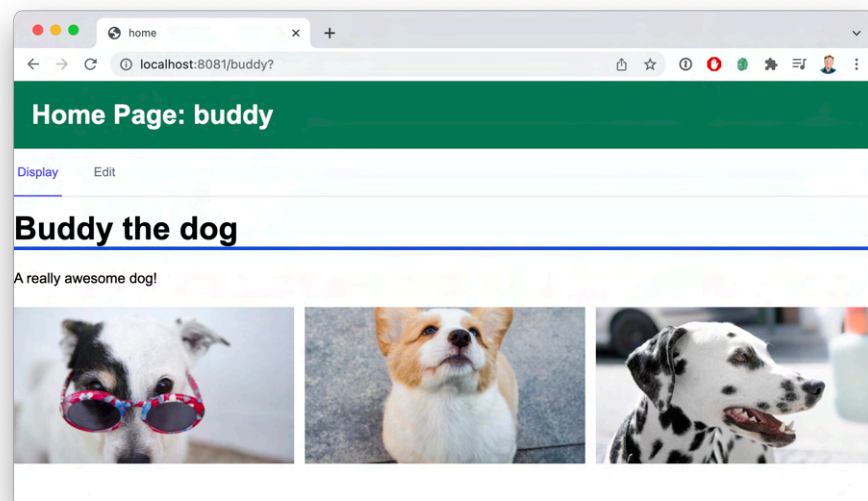
In large enterprise settings its not uncommon at all that you will get a division between the customer facing portion of the site and the administration facing side of the site. And while that's maybe fine from an organizational standpoint, it's not great for code sharing, particular code that really should be shared. A decent example of that is when you have the customer facing portion of the site rendering some data that is administered by the admin team.

When the administration pages want to preview the data as it would appear to the customer they can either iframe in the production site and patch the data through in a “preview mode”, or they can just maintain a second copy of the rendering code. But Module Federation provides a “third way” by making it easy for the customer facing team to share rendering code with the administration team, or vice versa. And to do that sharing live so that absolute fidelity is preserved between the customer facing site and the administration site. A win for everyone!

In our example system the CMS editor looks like the site pictured below:



And the corresponding page on the customer facing site looks like this:



Buddy is such a cute dog! Honestly, if you are looking for good placeholder images you should check out placedog.net.

There are two applications in the example code for this; `cms-admin` and `home`. In this case the `cms-admin` application has both a shared content editor component, named `EmbedPage`, and a shared content viewer component, called `EmbedEditor`.

The CMS application does not use the `EmbedPage` and `EmbedEditor` components directly. Those are both wrapper components that connect to the REST API endpoint using CORS so that the requests can go to a different port.

THE CMS APPLICATION

Both the `cms-admin` and `home` applications are derived from the getting started project. But the `cms-admin` project significantly alters the `webpack.config.js` to extend the `dev-server` with a REST API to support reading and writing JSON payloads that represent the different “pages” in the Content Management System (CMS).

The `ModuleFederationPlugin` configuration should be familiar by now though;

```
new ModuleFederationPlugin({
  name: "cms",
  filename: "remoteEntry.js",
  remotes: {},
  exposes: {
    "./EmbedPage": "./src/EmbedPage",
    "./EmbedEditor": "./src/EmbedEditor",
  },
  shared: { ... }, // Shared React dependencies
}),
```

We can take a quick look at `EmbedPage` to see how it retrieves content from the server and then sends it on to the `Page` component for display.

```
import Page from "./Page";
import { fetchPage } from "./api";

const EmbedPage = ({ page }) => {
  const { data } = useQuery(["getPage", { page }], () =>
    fetchPage("http://localhost:8080")(page)
  );
  return data ? <Page {...data} /> : null;
};
```

This `EmbedPage` component uses the excellent `react-query` library to connect to the CMS service defined in the Webpack configuration using an absolute URL.

And the underlying `Page` component is a straightforward content formatter:

```

const Page = ({ title, text, img1, img2, img3 }) => (
  <div>
    <h1 className="border-b-4 border-blue-700 text-4xl font-bold">
      {title}
    </h1>
    <p className="mt-5">{text}</p>
    <div className="grid grid-cols-3 gap-3 mt-5">
      {[img1, img2, img3].filter(Boolean).map((img) => (
        <img
          key={img}
          src={img}
          className="w-full h-full object-center ..."
        >
      ))}
    </div>
  </div>
);

```

You could also share out `Page` from `cms-admin` for clients specifically interested in rendering without the data fetching. In the case of this example the `EmbedPage` makes it easier on the client with a simple contract; given a page name the component will make the necessary requests to show the page.

The rest of the `cms-admin` applications follows this pattern with an `Editor` component wrapped by an `EmbedEditor` that also connects to the service.

This then leads us to the home application that consumes these components.

THE HOME APPLICATION

The home page starts up its use of these components by configuring just the `remotes` section of the `ModuleFederationPlugin` in the Webpack configuration.

```
new ModuleFederationPlugin({
  name: "home",
  filename: "remoteEntry.js",
  remotes: {
    cms: "cms@http://localhost:8080/remoteEntry.js",
  },
  exposes: {},
  shared: { ... }, // Shared React dependencies
}),
```

In the `App.jsx` for home we then bring in the components using `React.lazy` imports and wrap them in local `React.Suspense` wrappers that provide some nice feedback as the component loads.

```
const EmbedPage = React.lazy(() => import("cms/EmbedPage"));
const Page = () => {
  const { page } = useParams();
  return (
    <React.Suspense fallback={<div>Loading</div>}>
      <EmbedPage page={page} />
    </React.Suspense>
  );
};
```

There is another similar wrapper for `EmbedEditor` and both of these are used in the main page layout that uses tabs to swap between the static and editable views of the page content.

This live preview example isn't exploring any new technical boundaries in our understanding of Module Federation. It's a fairly simple system, and that's the point. There is a huge business value in the reuse and synchronization of code between these two systems, and potentially even more value as the number of clients expands. And it doesn't take any huge technical leaps to get there given the functionality provided by Module Federation.

An ideal extension to this example would be to use the resiliency pattern as shown in Resilient Header Footer chapter in the Use Cases part. In this pattern we export the components using both federated modules and NPM, so that should the live sharing federated modules fail, you have a recent version as a fallback that you can render.

PART THREE

ADVANCED MODULE FEDERATION

3.1

FULLY FEDERATED SITES

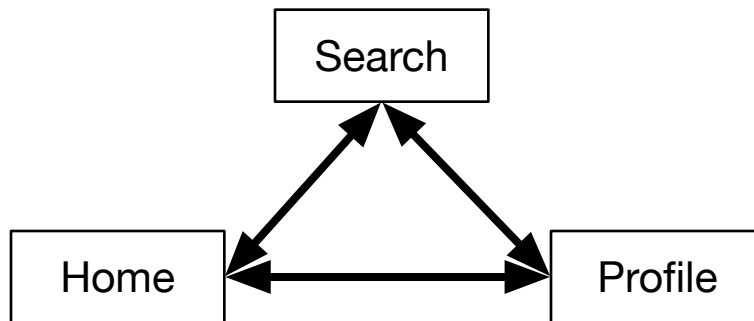


Code for this chapter is [available on GitHub](#). There is also a [Blue Collar Coder video](#) associated with this practical use case.

Most large sites run on what's labeled the “micro site” architecture where a single domain (e.g. [amazon.com](#)) isn't served by a single application, but instead different routes will go to different applications. All of these applications work together through shared session state, or shared backend APIs to create what appears to be a single coherent site for the customer.

The advantage of the micro site architecture is that each of these applications can deploy independently, because the site as a whole doesn't need to get deployed every time one route changes. One downside, which we've been addressing throughout this book, is the complexity of sharing code between these different applications. But another downside is that you lose the ability to create a Single Page Application easily in this methodology. It also make it difficult to test an end-to-end flow because you'd have to inject the experience you are testing into the middle of the entire production end to end flow.

But what if you could share whole routes between the different applications? So that each application would be able to render all the others? For example, if we had three applications; home, search and profile. Could we bi-directionally share between them as in the figure below.



So if you are working on the search application, then you federate in the home and profile applications. And as you navigate around the interface you are still technically running in the search app, but you get the entire site experience.

This is what we call a Fully Federated Site and it has some big advantages.

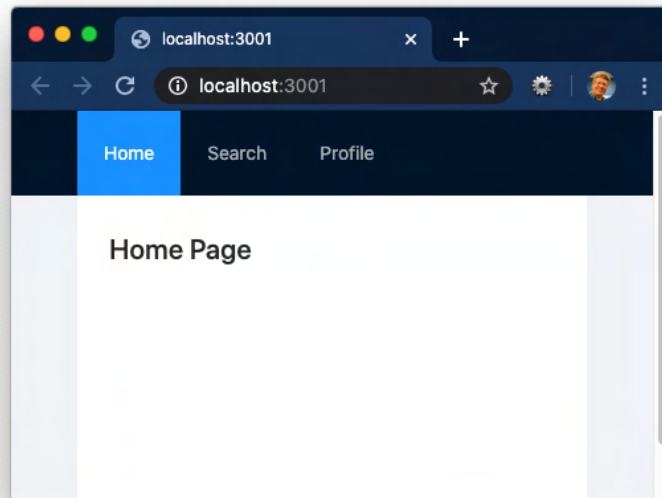
- It can be deployed as the site itself, giving your customers a Single Page Application experience but still allowing the individual teams to deploy independently.
- It provides a better developer experience where the contributor can see how their feature works through the entire experience.
- It means that you have the ability to completely end to end test your entire site even across multiple projects.

AN EXAMPLE APPLICATION

Setting up an entire fully federated application would be too much code to work through in the context of a book. So we've opted to provide the book code on Github and look at key sections of the code to demonstrate how it works.

As with our other example applications if you run `yarn start` at the top of [this directory](#) you will launch all of the servers. The home page runs on port 3001, profile on 3002 and search on 3003. For simplicities sake we've put the port numbers in the same order as the alphabetical order of the project names.

If you navigate to <http://localhost:3001/> you'll see the home application hosted experience.



In this version the Home tab is the only one with code actually in the application. The Search and Profile tabs are running on code provided, through module federation, by the profile and search applications on ports 3002, and 3003 respectively.

To get a quick taste of the power of a fully federated site simply make a change to `packages/home/src/Home.jsx` and refresh your browser on <http://localhost:3001/>. Of course you would expect to see a change there, but when you navigate to <http://localhost:3002/>, the profile application, and <http://localhost:3003/>, the search application, you will see the changes as well.

The really impressive part about all this is that, given what we've learned thus far through this book, this is not anything we haven't already seen. This is just the culmination of our existing work in understanding Module Federation.

Let's look at the `webpack.config.js` in the `packages/home` directory. In there the `ModuleFederationPlugin` is configured like so:

```
new ModuleFederationPlugin({
  name: "home",
  filename: "remoteEntry.js",
  remotes: {
    search: "search",
    profile: "profile",
  },
  exposes: { "./Home": "./src/Home" },
  shared: { ... }, // Shared dependencies
}),
```

The home application specifies the search and profile applications as remotes, and exposes its own Home page React component. It also shares out its dependencies on react, react-dom and the antd component library.

In the App.jsx file we import the different components using a combination of a straight import for the home page and lazily loaded components for search and profile.

```
import Home from "./Home";  
const Profile = React.lazy(() => import("profile/Profile"));  
const Search = React.lazy(() => import("search/Search"));
```

In the React component code for the application we use the `react-router-dom` components to select between the different routes for display:

```
<Routes>
  <Route
    path="/"
    element={
      <React.Suspense fallback={<div>Loading home</div>}>
        <Home />
      </React.Suspense>
    }
  />
  <Route
    path="/search"
    element={
      <React.Suspense fallback={<div>Loading search</div>}>
        <Search />
      </React.Suspense>
    }
  />
```

The Home component we use as-is, but for the lazily loaded Search and Profile pages we use a suspense.

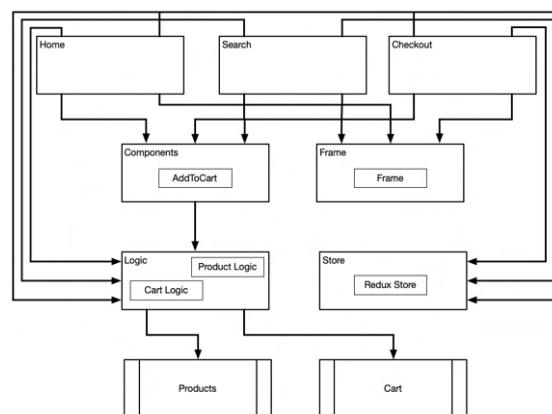
If you want some more fun just check out the network tab to see just how little extra code is loaded as you navigate between the tabs.

This is certainly not the only possible way that you could construct a fully federated site. You can imagine a very small booter application that only hosts the content frame and manages navigation between the routes that are supplied via federation. By default the application would bring in the routes from production, but by changing a flag or

setting an environment variable an engineer could switch one of the routes to their own localhost running copy of the part of the interface they are working with.

LARGER SCALE EXAMPLE

Let's look at a much larger example system. In the sample code is an entire three application micro-site with associated node modules and micro-services located in the large-example directory. Shown below is a block diagram for that system.



We have the applications at the top, Home, Search and Checkout, and they connect to some React components shared using NPM (AddToCart and Frame). And they also have a shared codebase for the redux store which holds the cart and some shared API wrappers in the logic NPM module. And all of this sits on top of two micro-services.

And this is the kind of thing we see a lot of nowadays; the "micro-site" architecture. The most common architecture before that was the "monolith", a single, usually Java application, where all the frontend code was in one application.

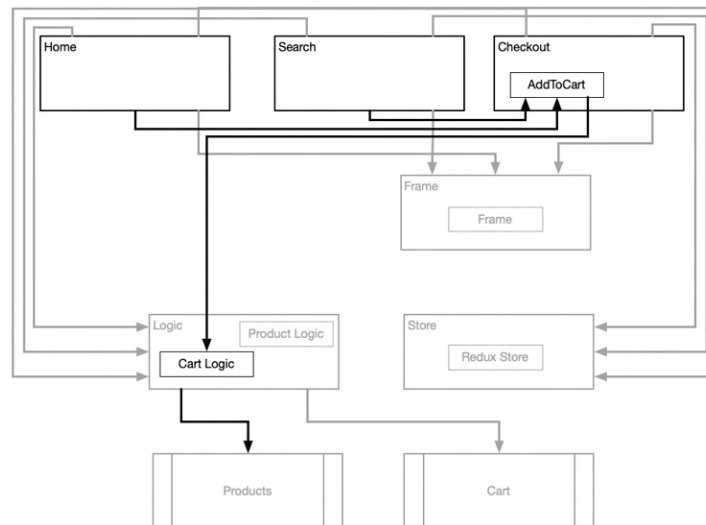
The advantage of the micro-site architecture is that each application can deploy independently. But there are some disadvantages:

- **Code sharing** - Code sharing is really difficult - Having shared code in NPM modules creates friction to sharing code since it has to be extracted from the original application, which means jumping from repo to repo, doing version bumping, etc.
- **Single Page Applications (SPA)** - SPAs are difficult to pull off with micro-sites, but being on a SPA is a huge performance win for the customer. There are great options like SingleSPA, but those have a learning curve.
- **End-to-end testing** - E2E testing is very difficult. How do you know when you release a new version of checkout it's going to be compatible with the other micro-sites and not break existing flows? That's what end-to-end testing is for, but staging the entire site using existing tools is very tough.

We are going to step-by-step improve this site using Module Federation until we end up with a system that runs as a Single Page Application, has easy code sharing, and is E2E testable. In the accompanying video you can follow along with the [step by step code changes](#). In this book we will cover the high level architectural moves.

Relocating the UI Components

First step is to move the AddToCart React component from components NPM library to the Checkout application.

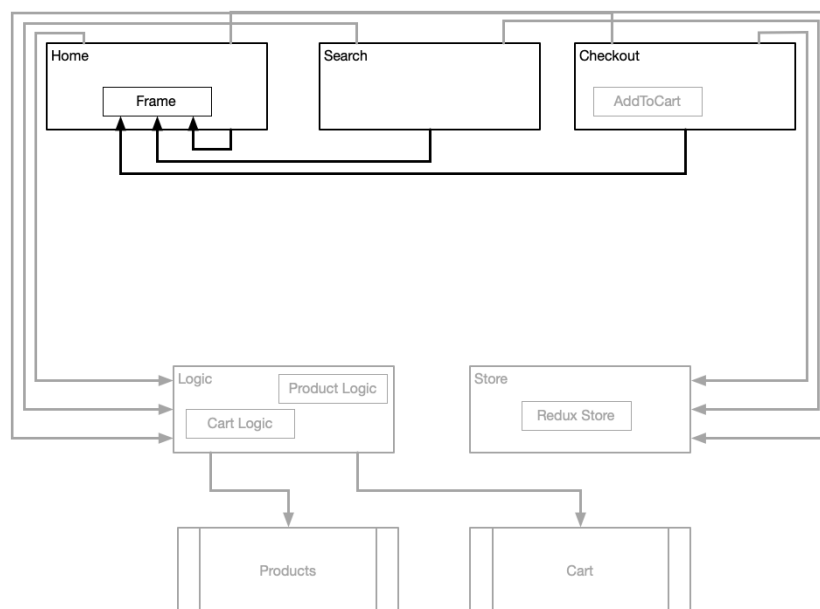


This is the natural place for that code because it's functionality that fits within the purview of the Checkout team. Then we change the Home and Search applications to remote in the Checkout application and import AddToCart from there.

This means that not only are we reducing the number of NPM libraries we need to version by one, but we also get live code sharing so that when Checkout updates the look or the functionality of the “Add To Cart” button the Home and Search sites will update immediately.

Moving The Frame (Header)

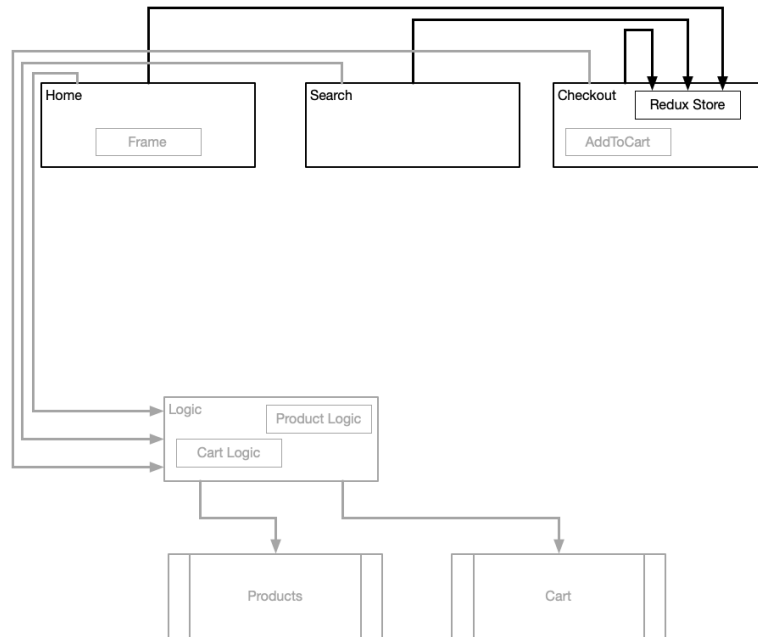
Our next stop on the train towards full site federation is to move the header (or Frame) component over into the Home application and then to expose it out to all the applications.



Right now Frame just has anchor tag links to the other applications. In a later step, because all of the content from all of the applications is exposed as remotes, we will later Frame to give us full SPA functionality. But for now this is a huge win because the Home page team can update the Frame header and all the applications will see those changes immediately after Home page deploys.

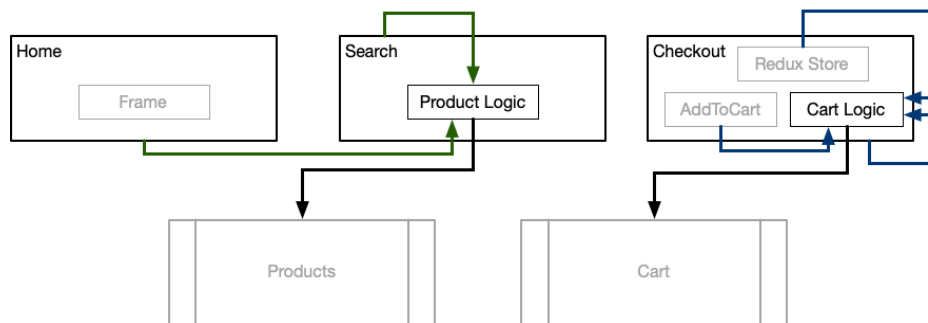
Moving The Business Logic

The next thing to move is the Redux store into the Checkout application (because it only stores the cart state).



All of the applications now use the remote of Checkout to get the store redux code. This is nothing new, but when we move to a Single Page Application at the end, because the user never navigates away from the first page we will be able to retain the store in memory through the entire customer journey.

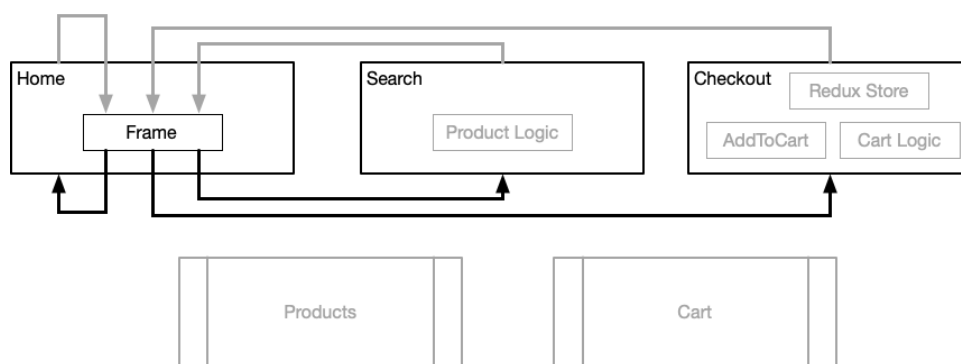
And then there is one more move to split the business logic code between both the Checkout and Search applications.



Checkout manages the Cart Logic because it relates to their role in the system. And the Search team manages the product related API wrappers which are also shared out to any other applications that need product data or search capabilities.

Moving To A Single Page Application (SPA)

The final step is to move to a full SPA by exposing all the body content components for each of the applications And then linking those to the already shared Frame component using a SPA router such as `react-router-dom`.



Because any site can vend the content from any other site a user can start on the search page application and never actually leave the original search page as they go perhaps to the home page, then back to search, and then into checkout. And that means your can

use an End-to-End (E2E) testing system to test the entirety of the site from initial customer visit all the way through ordering an item, with your local code changes run within that test.

This new system is conceptually easier to understand as there are less moving parts and shared components and business logic are sharing out of the applications where they are most ideally located. But, it can be hard to understand how dramatic of a change full site federation is without trying it yourself. I strongly encourage you to watch the associated video and to walk through the example before trying to migrate your site to a fully federated model.

WHAT'S NEXT

It's not just about sharing modules out of running applications with Module Federation. There are other ways to share code as well. In the next chapter we cover these other methods in detail.

3.2

ALTERNATIVE DEPLOYMENT OPTIONS



Code for this chapter is [available on GitHub](#).

Thus far we've been deploying federated modules from applications for consumption by other applications, but there are alternatives to that model.

MICRO-FRONTEND MODEL

In the Micro-Frontend model we use Webpack to create a `dist` directory that we then statically deploy to a static hosting service. Amazon's S3 service would be a good example target for this approach.

Applications would then consume the `remoteEntry.js` just as they would from a remote application, but in this case there is no running application, just the federated modules.

Let's start by looking at the widget directory within the micro-fe directory of the book code for this chapter. In that directory we find the `webpack.config.js` file that just packages up the Widget using standard a standard `ModuleFederationPlugin` configuration.

```
module.exports = {
  output: {
    publicPath: "http://localhost:3002/",
  },

  ...

  plugins: [
    new ModuleFederationPlugin({
      name: "widget",
      filename: "remoteEntry.js",
      remotes: {},
      exposes: {
        "./Widget": "./src/Widget",
      },
      shared: { ... }, // Shared React dependencies
    }),
  ],
};
```

And in the `package.json` we have a `build` script runs the webpack build, and a `start` script that runs a static server that imitates something like S3.

```
"scripts": {  
  "build": "webpack --mode production",  
  "start": "cd dist && PORT=3002 npx server"  
},
```

To run this widget server first run `yarn build` then `yarn start`.

And to test it out run `yarn start` in the home directory that is a peer to the widget directory.

The home application itself should be very familiar by now. The `ModuleFederationPlugin` in the `webpack.config.js` sets up `widget` as a remote. The `index.html` file includes a script tag to the `remoteEntry.js` file on our static server. And the `App.jsx` file lazily imports the `Widget` component and hosts it within a `Suspense` as show in the code below:

```
import React from "react";

...

const Widget = React.lazy(() => import("widget/Widget"));

const App = () => (
  <div className="mt-10 text-3xl mx-auto max-w-6xl">
    <React.Suspense fallback={<div>Loading</div>}>
      <Widget />
    </React.Suspense>
    <div>Home App</div>
  </div>
);
```

The import thing to get from this exercise is that you don't necessarily have to have the federated module code deployed with the application. Which leads us to a second possible build and deployment option; the sidecar.

SIDECAR DEPLOYMENTS FOR WEBPACK 4

So you have a Webpack 4 application and it's going to take some time to port to Webpack 5, but you want to export some of its modules today. What to do? Well, you could do a sidecar deployment as show in the `sidecar/wp4-project` directory.

This project is, as you might guess from the name, a React application that is packaged using Webpack 4. But we want to expose the `src/Carousel.jsx` file as a federated module. How do we do that? We create a Webpack 5 project within this project in a directory called `wp5-sidecar` (though you can call it whatever you want.)

In the sidecar's `webpack.config.js` there are two items worth noting:

```
module.exports = {
  module: {
    rules: [
      {
        test: /\. (js|jsx)$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader",
          options: {
            presets: ["@babel/preset-react"],
          },
        },
      },
    ],
  },
  plugins: [
    new ModuleFederationPlugin({
      name: "widget",
      filename: "remoteEntry.js",
      remotes: {},
      exposes: {
        "./Carousel": "../src/Carousel",
      },
      shared: { ... }, // Shared React dependencies
    }),
  ],
};
```

The first thing to notice is that we are setting up the babel options in the webpack configuration because babel will not find them with the relative pathing that we use to get to the `Carousel` module.

Other than that it's a simple `yarn build` to create a `dist` directory that could be deployed to a static hosting service along with the Webpack 4 bundled code.

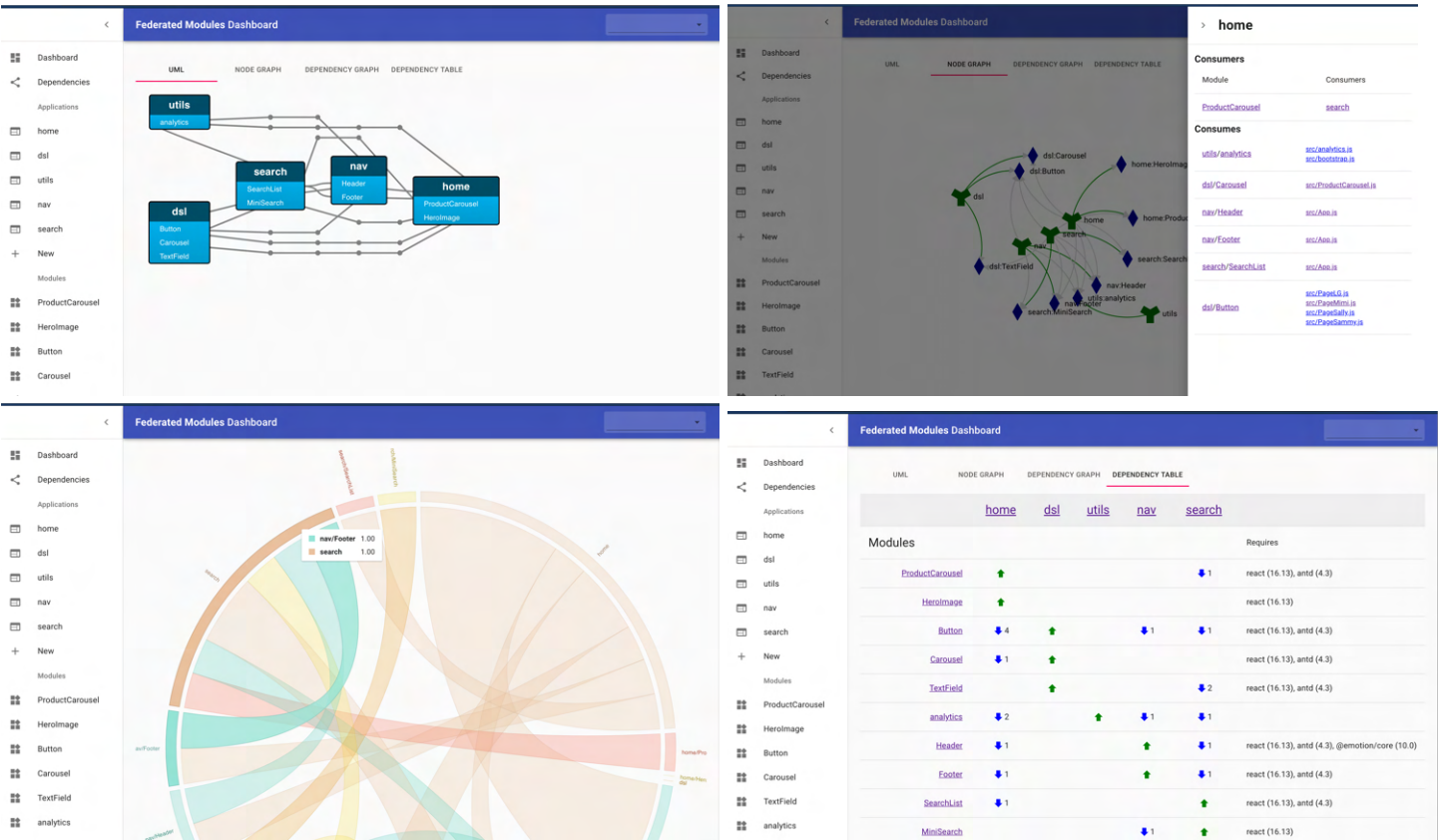
It's true that the Webpack 4 code won't know about the federated module Javascript or use it. But this is a method for exporting code from Webpack 4 applications today if there is going to be a lengthy migration process to Webpack 5.

3.3

MEDUSA



With Module Federation making it very easy to share code and components between applications it became readily apparent that we needed a way to visualize everything being shared between applications. So we created a Module Federation dashboard named Medusa and it’s available on Docker today. Here are some screenshots from a populated dashboard.



The dashboard is currently in alpha release status as we are rapidly iterating on features, adding new functionality, and changing the data model. If you want to try it

out you will need to use Docker to run the dashboard as shown in the shell command below:

```
> mkdir dashboard-data
> docker run -p 3000:3000 \
  --mount type=bind,source="$(pwd)"/dashboard-data,target=/data \
  -t scriptedalchemy/mf-dashboard:latest
```

This will create a local directory named `dashboard-data` that will hold the data files so that there is no data loss when the Docker image is shutdown.

Once this is running you can go to <http://localhost:3000/> to see the dashboard. When no data is loaded you will get a home page that explains the dashboard, what it's useful for and how to integrate with it.

To send data to the Dashboard we've developed a plugin that integrates right into your Webpack configuration. To get started first install the plugin in your project: Then configure the plugin to point at the dashboard server.

```
yarn add @module-federation/dashboard-plugin -D
const DashboardPlugin = require("@module-federation/dashboard-plugin");
...
new DashboardPlugin({
  dashboardURL: "http://localhost:3000/api/update",
}),
```

The next time you start or build your application the dashboard plugin will automatically update the dashboard with the latest information from the build.

There is also additional metadata that you can add to the DashboardPlugin configuration that is [documented on the NPM page for the plugin](#).

We are really excited about the potential for the Dashboard. It's a level of detail and insight that isn't available from NPM. As we continue to add new features to enhance the experience and stabilize the application

3.4

FREQUENTLY ASKED QUESTIONS

It's hard to cover every potential use case for a technology as fundamental as Module Federation. In this chapter, we cover a variety of related topics and questions that often arise as we roll this out in projects and organizations.

What is the `remoteEntry.js` file?

The `remoteEntry.js` file contains references to the exposed modules of an application. It doesn't contain the code for those modules, but it knows where to go get the code. That's important because it allows those modules to be lazily loaded via `import()` if that is how you choose to load your code.

When you have one of the examples in this book up and running it's worth having a look at the `remoteEntry.js` file to see what is in it so that you can get a better understanding how Module Federation works at the code level. By default the URL for `remoteEntry.js` is <http://localhost:8080/remoteEntry.js> in the getting started projects.

You can name the `remoteEntry.js` file whatever you choose. The one truly important detail is that the `publicPath` defined in the `webpack.config.js` matches where the code is deployed. Because the `remoteEntry.js` contains references to the code, and not the code itself, Webpack uses the `publicPath` to find the modules and

load them asynchronously. So that `publicPath` must match the deployment location for the system to work.

Can I use Module Federation with SingleSPA?

Absolutely! In fact Module Federation is one of the two code loading styles that is covered on the [SingleSPA documentation website](#). The advantage of using SingleSPA in tandem with Module Federation is that SingleSPA extends the capabilities of the solution by allowing you to host view code from multiple view frameworks (e.g. React can run on a Vue page, or vice versa.) And Module Federation allows you to share other types of code as well (i.e. business logic, etc.)

What about A/B Testing?

A/B test code is an ideal use for Module Federation. You can expose both of the variants and use asynchronous imports (or `React.lazy`) to import which ever variant your A/B provider tells you to. In this solution you will ever only import one variation of the code.

When should I still use NPM?

This question is really about the architecture of your solution and as such you should create your own criteria for that decision. That being said, here is some high level guidance on when to choose NPM over Module Federation.

- **Externally shared code** - If you are writing a wrapper for your API then - developers will expect to see that on NPM.

- **Slow moving business logic** - If it's not going to change that often, and the upgrade procedures are very specific, then it's probably best controlled and versioned as an NPM module.
- **When contextually appropriate** - If similar functionality is already packaged as NPM and that functionality is not migrating to Module Federation, then you should consider sticking with NPM.

Module Federation is a new architectural paradigm and it will take time for the industry to adapt. It's worth spending the time before rolling out Module Federation at scale to develop guidance on how to decide on packaging code as NPM packages or as federated modules.

It's worth noting that any code shared with NPM can always be imported and either shared using the shared module system or exports to other applications using exposes.

3.5

TROUBLESHOOTING

We see some common errors in Module Federation. So here's what's going on if you see one of them.

UNCAUGHT ERROR: SHARED MODULE IS NOT AVAILABLE FOR EAGER CONSUMPTION

This is an easy one. You are eagerly executing an app which is operating in omnidirectional mode. Meaning that application A is importing from B which is in turn importing from A. You have a few 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 wise to provide it only at one point of your app, e. g. the shell.

```
shared: {
  ...deps,
  react: {
    eager: true,
    singleton: true,
    requiredVersion: deps.react,
  },
  "react-dom": {
    eager: true,
    singleton: true,
    requiredVersion: deps["react-dom"],
  },
}
```

Where we define deps as:

```
const deps = require("./package.json").dependencies;
```

If you do not want to set dependencies as `eager` then you can take advantage of `bundle-loader` in your webpack configuration.

To do that add this loader:

```
module: {
  rules: [
    {
      test: /bootstrap\.js$/,
      loader: "bundle-loader",
      options: {
        lazy: true,
      },
    },
  ],
}
```

Then change your entry point to look like this.

```
//index.js
import bootstrap from "./bootstrap";
bootstrap();
```

Create a `bootstrap.js` file, and move the contents of your entry point code into that file:

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

What this does is create an opportunity for Webpack to coordinate with other remotes and decide who will vend what, before beginning to execute the application. This

approach will increase Round-Trip Time (RTT) as Webpack is unable to async load everything in one roundtrip.

The recommended solution to eager imports

Methods mentioned above work, but can have some limits or drawbacks.

With Webpack we strongly recommend a dynamic import of a bootstrap file. Doing so will not create any additional Round Trips, it's also more performant in general as initialization code is split out of a larger chunk.

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 beta.16 to beta.18 within ModuleFederationPlugin.

Change the exposes from:

```
exposes: {  
  "Button": "./src/Button",  
},
```

To this:

```
exposes: {  
  "./Button": "./src/Button",  
},
```

The change is done so that we can follow the ESM syntax inside Node 14.

UNCAUGHT TYPEERROR: FN IS NOT A FUNCTION

You likely are missing the `remoteEntry.js`, make sure its added in your page template. If you have the remote entry loaded for the remote you are trying to consume, but still, see this error. Add the host's `remoteEntry.js` file to the HTML as well.

Depending on how your application exposes, shares, and is architected. An omnidirectional host can leverage **its own remote** when coordination is taking place. This has no performance drawbacks. Webpack is just leveraging itself in an omnidirectional manner. Since its both a host and remote at the same time. It may use its own remote. This might seem strange, but this architecture is extraordinarily powerful, with omnidirectional hosts allowing dead simple ways to introduce A/B tests without ever writing a feature flag or needing to remove one.

APPENDIX A

REFERENCE

The sections below cover the different plugins that make up the Module Federation ecosystem within Webpack 5 and their various options.

ModuleFederationPlugin

Name

This is the name of the module being exported.

Library (optional)

This specifies how the module is to be formatted for use in the browser context.

You should use the `var` type and match the name with the module name.

Examples

```
{ library: { type: "assign", name: "app1" } }
```

Sets the webpack library target type to `var` under the name `app1`. There are many library target types, we recommend `var` as it will be assigned to a global variable exposed under that name and be easily referenced.

Filename (optional)

The file name for the remote entry. This is optional and defaults to `remoteEntry.js`.

Remotes (optional)

An object that specifies the remotes this application consumes.

Examples

```
{ remotes: {  
  nav: "nav",  
} }
```

Specifies that the incoming `nav` federated module is a known remote and that it can be imported using the name `nav`.

```
{ remotes: {  
  nav: "checkoutNav",  
} }
```

Specifies that the incoming `checkoutNav` federated module is a known remote and that it can be imported using the name `nav`.

```
{ remotes: {  
  nav: {  
    external: "nav@http://localhost:3001/remoteEntry.js"  
  }  
}
```

This automatically fetches the remote named `nav` from the specified URL without the requirement to add a script tag to the host page.

remoteType (optional)

Specifies Webpack library type for the remotes.

Examples

```
{ remotes: {  
  nav: "nav",  
}, remoteType: "var" }
```

Specifies that the remotes are packages as vars.

Exposes (optional)

The list of internal modules to exposed as federated modules.

Examples

```
exposes: {  
  "./Header": "./src/Header",  
},
```

Exposes the source file `./src/Header` as the named export `Header`.

```
exposes: {  
  Header: {  
    import: "./src/Header",  
    // optional metadata  
  }  
},
```

Exposes the source file `./src/Header` as the named export `Header` with the additional metadata provided in the object.

Shared (optional)

The list of packages that this application will share with other applications that consume its exposed modules, and share with federated modules it consumes as remotes.

Examples

```
{ shared: ["lodash"], }
```

Shares loads without version information, or eager consumption, or as a singleton.

```
shared: {  
  "react": "^16.12.0"  
},
```

Shares react with the NPM semver mechanics starting at 16.12.

```
{ shared: require("./package.json").dependencies },
```

Shares all the packages listed in the package.json dependencies.

```
{ shared: {  
  react: {  
    singleton: true  
  }  
}  
}
```

Shares react but marks it as a singleton meaning that only one copy can be in use in the application at one time.

```
{shared: {  
  react: {  
    version: "^16.12.0",  
    import: "react",  
    shareKey: "react",  
    shareScope: "default",  
    singleton: "true",  
  }  
},
```

Shares react at 16.12 as a singleton with the import name as react and in the default scope under the name react.

ShareScope (optional)

The scope name to use for the shares..

Examples

```
{ shared: ["lodash"], shareScope: "default" }
```

This sets the name of the scope to use for the shared packages to “default”.

ContainerPlugin

This is the plugin that exposes modules for module federation. It accepts the name, library, filename and exposes keywords with the same options as `ModuleFederationPlugin`.

ContainerReferencePlugin

This is the plugin manages remotes for module federation. It accepts the `remoteType` and `remotes` keywords with the same options as `ModuleFederationPlugin`.

SharePlugin

This is the plugin manages shares for module federation. It accepts the `shares` and `shareScope` keywords with the same options as `ModuleFederationPlugin`.

APPENDIX B

GLOSSARY

Below is a glossary of the different terms used in this book and what they mean.

Term	Description
Exposed	A given module should be exported by the application for federation.
Host	The currently running application that is hosting federated remote modules.
Module	The smallest unit of shareable code. In the Webpack system a module can be a single file.
Override	The Webpack internal name for a shared package
Package	An NPM package.
Remote	A reference to an external federate module.
Remote Module	Any file exported through Module Federation.
Scope	A scope is synonymous with an application. Every application is exported as one named scope.
Shared	A module that is shared from an application to a remote module.
Shell	A host application, which may provide additional APIs to the MFEs it hosts.
Singleton	A constraint that only a single version of this module or package should be in use within that application at any given time.

REVISION HISTORY

v1.1.0 - 6/18/2020: Updated to beta 18

- Updated references from beta 17 to beta 18
- Fixed small inconsistencies in the code examples

v1.2.0 - 6/26/2020: Content Upgrade

- Split into three parts
- Added chapter 9: Header, chapter 10: A/B tests and chapter 11: Live Previews
- Substantive content added to the Full Site Federation chapter

v1.3.0 - 7/23/2020: Content Upgrade

- Updated all the code to v22 of Webpack 5
- Modified all the examples to use the new remote syntax
- Numerous context fixes and enhancements

v1.4.0 - 10/11/2020: Content Upgrade

- Updated all the code to the Webpack 5.0.0 release
- Updated the code samples on GitHub
- Updated the dynamic loading code

v2.0.0 - 12/26/2021: Massive content overhaul and upgrade

- Updated all the code for all chapters
- Completely rewrote the introductory chapter
- Added a new chapter on using Create React App
- Added a new chapter on deployment
- Added a new chapter on non-React frameworks
- Added more state managers to the state management chapter
- Rewrote the code and copy for the CMS chapter
- Rewrote the code and copy for the Full Site Federation chapter
- Migrated all the examples to `create-mf-app`
- Cleaned up many inconsistencies across the chapters
- Re-structured the book to be more easily updatable

ABOUT THE AUTHORS



Jack Herrington is a Principal Full Stack Software Engineer. He has written seven books on a variety of topics from Podcasting to PHP and now to Module Federation. He is also the host of the Blue Collar Coder channel on YouTube which covers mainly frontend topics from beginner to advanced architecture levels. He lives in Portland, Oregon with his wife and his daughter who is attending Oregon State University.

Zack Jackson is the Principal Javascript Architect at Lululemon and the creator of Module Federation. Zack strives to prove the impossible to be possible. He has designed distributed application architecture for over 10 years and has architected stacks consisting over 150+ independent Micro-frontends. Before Module Federation, he co-authored the code- split SSR technology that's still used today by all React tools which offer the capability.