Section 1: The Basics of Microfrontends

Topic 4. Application Overview

Instructor: Let's take a look

at the first application we're going to build to

get a better idea of how micro frontends work.

So here's a little mock-up.

We're gonna make a very simple

very straightforward application

that's going to use just about

no freight marks inside of it.

We're gonna make a kind of e-commerce store.

On this page, we're gonna list out a section where

we have some different products for sale,

and then underneath it we'll have

another section where we list out

the number of items that a user has in a cart.

One thing I wanna make sure is clear right away

is that we're working with

a hundred percent fake generated data here.

We're not gonna make use of any API

or anything like that.

In addition, we're not gonna have

any kind of interaction

so there's no actual adding items

to a cart or anything like that.

We're really just trying to get some plain text

to appear on the screen.

And when I look at this application,

I really see two distinct features.

First off, we've got a listing

of different products

that are available for sale.

And we also have a display of the cart

that's gonna list out the number of

items that a user has in a cart.

And remember, that number right there.

So when it says one item

that's really just a randomly generated number

that we're gonna stick inside there.

So when we notice right away that we've got two

distinct features, we might decide

if we were making some kind of

monolithic style application

to do something like this.

We might have some kind

of app component that displays

a product list component

and a cart component.

So again, this is what we would do

if we were making a classic

or traditional monolithic style app.

But how would we approach this

if we were making use of micro frontends?

Well, we might decide to take each major feature

of our product and put it into a

different micro frontend application.

So we could have micro frontend number one,

that contains all the code related

to our product listing feature.

So essentially all the code needed

to get this list of products

to appear on the screen.

We might also create a second micro frontend app

that contains all the code needed to

get this cart to be displayed on the screen.

You'll notice right away that

if we only have these two micro frontends,

we might have a little bit of a challenge

in actually getting these things to be displayed

on the screen.

In other words, how do we know

that the micro frontend number one needs to

be displayed up here at the top

and micro frontend number two needs to

be displayed down here at the bottom?

To coordinate where to show each

of these different micro frontends

on the screen and when to show each of them,

we very frequently end up

creating a third micro frontend app,

that we usually refer to as the container.

The container is what decides when and

where to show all the different

micro frontends that we have.

So to build up this little fake e-commerce store,

we're gonna end up making three small projects.

We're gonna make this container app.

We're gonna make micro frontend number one

to show a listing of products

and micro frontend number two to show the cart.

We're going to decide when and where to show

each of these micro frontends

by adding in a little bit of logic

to this container application.

So let's get started

on putting this all together in the next video.

Topic 5. Understanding Build-Time Integration

Instructor: Let's take a look

at the first application we're going to build to

get a better idea of how micro frontends work.

So here's a little mock-up.

We're gonna make a very simple

very straightforward application

that's going to use just about

no freight marks inside of it.

We're gonna make a kind of e-commerce store.

On this page, we're gonna list out a section where

we have some different products for sale,

and then underneath it we'll have

another section where we list out

the number of items that a user has in a cart.

One thing I wanna make sure is clear right away

is that we're working with

a hundred percent fake generated data here.

We're not gonna make use of any API

or anything like that.

In addition, we're not gonna have

any kind of interaction

so there's no actual adding items

to a cart or anything like that.

We're really just trying to get some plain text

to appear on the screen.

And when I look at this application,

I really see two distinct features.

First off, we've got a listing

of different products

that are available for sale.

And we also have a display of the cart

that's gonna list out the number of

items that a user has in a cart.

And remember, that number right there.

So when it says one item

that's really just a randomly generated number

that we're gonna stick inside there.

So when we notice right away that we've got two

distinct features, we might decide

if we were making some kind of

monolithic style application

to do something like this.

We might have some kind

of app component that displays

a product list component

and a cart component.

So again, this is what we would do

if we were making a classic

or traditional monolithic style app.

But how would we approach this

if we were making use of micro frontends?

Well, we might decide to take each major feature

of our product and put it into a

different micro frontend application.

So we could have micro frontend number one,

that contains all the code related

to our product listing feature.

So essentially all the code needed

to get this list of products

to appear on the screen.

We might also create a second micro frontend app

that contains all the code needed to

get this cart to be displayed on the screen.

You'll notice right away that

if we only have these two micro frontends,

we might have a little bit of a challenge

in actually getting these things to be displayed

on the screen.

In other words, how do we know

that the micro frontend number one needs to

be displayed up here at the top

and micro frontend number two needs to

be displayed down here at the bottom?

To coordinate where to show each

of these different micro frontends

on the screen and when to show each of them,

we very frequently end up

creating a third micro frontend app,

that we usually refer to as the container.

The container is what decides when and

where to show all the different

micro frontends that we have.

So to build up this little fake e-commerce store,

we're gonna end up making three small projects.

We're gonna make this container app.

We're gonna make micro frontend number one

to show a listing of products

and micro frontend number two to show the cart.

We're going to decide when and where to show

each of these micro frontends

by adding in a little bit of logic

to this container application.

So let's get started

on putting this all together in the next video.

Topic 6. A Run-Time Integration  
Instructor: Let's now take a look

at what a Run-Time Integration would look like.

So here's an example

of one possible way of implementing a Run-Time Integration.

Once again, we might have an engineering team

that develops the ProductsList application.

They would eventually decide that it's time to

deploy their application, and at that point

rather than deploying their project to say NPM

or something like that, they would bundle up their project

and deploy it to some static URL.

Something like, say my-app.com/ProductsList.js.

So this JavaScript file right here has all the source code

for the ProductsList application.

So then whenever a user navigates to say, my-app.com,

we would load up the Container app

and at that point in time

the Container app would then fetch that

ProductsList.JavaScript file or .js file and execute it.

So notice that in this approach

the container only gets access

to the source code of ProductsList

after the container has been loaded into the browser.

Now the, this approach has upsides and downsides as well.

So at the Run-Time Integration

the big upside is that we can independently deploy

the ProductList application at any time

without having to redeploy the container.

That's definitely a big upside.

Another big upside is

that we can easily have different live versions

of the ProductsList application.

So we might be doing some kind of

AB testing where we have two versions

of the ProductsList, and it can be

up to the container to decide which

of those two versions to use.

The downside to this approach is that the tooling

and the setup for it is way, way more complicated

and there's really a lot more going on here

that you have to understand as an engineer.

Now, having said all this

and having kind of compared a Run-Time integration

versus a Build-Time integration,

in this course you and I are gonna be focused

a hundred percent on a Run-Time Integration

and we're gonna implement it using

Webpack Module Federation.

The reason that we're covering this inside this course

is that it is definitely the hardest thing

to setup and to understand, and so in my opinion

it makes it way more sensible to cover this

inside of a course where I can essentially

walk you through everything, piece by piece

and help you really understand what is going on.

Now, let me be clear,

you might be saying, "Hey Steven I don't want

to do the hardest thing here as I'm just getting started.'

Don't worry, when I say, "Hardest to setup,"

I just kind of mean relatively.

It's still, it's not gonna be ridiculously hard,

I just want you to be aware

that we're going to be doing this approach

because it just makes a little bit more sense

to cover it in a course where you can have a lot

of, kind of guided instruction as you go through.

The real reason that we're using this approach

is that it's definitely the most flexible

and performant solution for doing integration right now.

One of the big downsides, however

is that we're gonna have to spend a lot of time

around Webpackk and really understanding

what Webpack is doing for us and how it works.

If you don't know anything about Webpack right now,

that's totally fine.

I'll give you a really good overview in the coming videos.

Okay, so now we've got a better idea of what's going on.

Let's start to tackle our e-commerce application

in the next video.

Topic 7. Project Setup

Narrator: Let's take a look at how

we're going to implement our e-commerce project

using a run-time integration.

To get started, we're going to create a separate folder

for each of our different micro frontends.

So one for our container, one for cart,

and one for products.

Each of these different folders

is going to have all the code needed

to implement that project

and get it running in both isolation,

in other words, by itself,

and as a part of the overall application.

Every one of these three different folders for this project

is going to have the same different files inside of it.

So we'll have an index.js file with some implementation.

We'll have an HTML file, a package.json file

to list out all of this sub-projects dependencies,

and then finally, a webpack.config file as well.

Each of our different sub-projects are going to be built

without using any JavaScript framework.

So for this first go, no React, no Angular, no Vue,

just plain JavaScript.

And again, we want to make sure that we can run each

of these in isolation and through the container app as well.

We're going to first focus on the implementation

of the products micro frontend.

So just products by itself.

We're not going to worry about the container

or anything like that.

8. Starter Project Files  
**Starter Project Files**

Attached to this lecture is a zip file containing a starter project with all of the specific versions of dependencies needed. This is **required** for the code we will be writing in sections 1 through 3.

First, make sure that you have installed the **LTS** version of Node onto your system:

<https://nodejs.org/en>

Once you have installed Node, please download and unzip the starter files attached to this lecture into your development environment.

After unzipping, change into the **ecomm/container** directory using your terminal:

cd ecomm/container

Install the dependencies:

npm install

Then, change into the **ecomm/products** directory:

cd ../products

Once again, install the dependencies:

npm install

We will use these files in the next lecture.

Resources for this lecture

Formularbeginn

Topic 9. Generating Products

Instructor: All my dependencies are installed.

So I'm now gonna open up my code editor inside

of this products directory

and we're gonna start to

put together some basic implementation for the product list.

To get started, I'm gonna create a SRC directory

inside of here,

and then inside of that folder

I'll make a new file called index dot js.

So eventually inside this file we're going to

want to generate a list of products

and show them on the screen to our users.

But right now, let's just generate a list

of products and console log them instead.

We're not gonna worry about trying to render them

onto any screen or anything like that.

To generate a fake list of products.

I'm going to make use of the faker module

that we installed just a moment ago.

The Faker Library is used to generate fake data.

That's all it does, nothing else.

So we can use Faker to generate some fake product names

and stuff like that.

To make these fake product names

I'm going to create a new variable

with a let keyword, that I'll call a products.

I'm going to assign that an empty string.

I'm then going to create a Four Loop

and iterate from zero to I, Lesson three.

Inside of here,

we'll generate a product name using the Faker library.

So I'll do a faker dot commerce dot product name.

Double check your spelling on commerce right there

just make sure you've got the correct spelling.

That's gonna give us a fake product name

which will work for the purposes of our application.

And then going to join that into this product string.

I'm going to join it in essentially as a little bit of html.

Again, for right now

we're just gonna console log this out, for right now

but eventually we will render it onto the screen.

We're gonna do a products plus equals

and then a set of back ticks,

and inside of a div,

I'm gonna print out name like so.

And then finally, at the bottom, after the for loop

I'll do a console log of products.

Okay, so just a very simple,

very basic implementation for right now.

So if we wanted to run this as it stands right now

we would get a lot of errors

if we tried to execute this code inside the browser.

And that's because a lot of browsers do not have support

for import statements right now.

So rather than trying to execute this code directly

we're going to instead use

that Webpac dependency that we installed just a moment ago.

So remember I mentioned that

in this course we need to have some reasonable knowledge

of Webpac and how it works.

So let's take a pause right here.

When we come back to the next video

we're going to use Webpac to transform or process this file.

And in the process we'll get a good review or overview

depending on how familiar you are

with it around Webpac.

Topic 10. Some Background on Webpack

Formularende

Instructor: Let's set up Webpack so it processes

this index.js file and like I said

we're gonna get a better idea

of how Webpack works in general.

So to get started inside of my root project directory,

so in products right now,

I'm gonna make a new file called webpack.config.js.

Then inside of here I'm gonna do a module.exports

of an empty object and the only thing

we're gonna write inside of here is mode development.

Like so.

Then after that we're going to open our package.json file

and I'm going to find the scripts section inside of here,

I'm going to replace the test script with start

and we're going to have the start script

just run Webpack for right now.

I'm gonna make sure I save both these files.

I'll then go back over to my terminal

and I'm going to run that start script.

So back at my terminal inside of the products directory

I'll do an NPM run start.

We're then going to very quickly see a ton of output

scroll across the screen.

It looks like something was built successfully.

Okay, that's good.

Let's really investigate what was created

and what just happened behind the scenes with Webpack.

So back inside my editor, I can find

a new directory inside of here called dist,

short for distribution and inside there

is a main.js file.

I'm gonna open that file up.

Now right away, you're gonna see

a tremendous amount of content inside of here,

and the vast majority of it

is probably gonna be very hard to read and understand.

That's totally okay.

I just wanna point out a couple of different qualities

about this file, but first

let's take a look at a diagram very quickly.

Okay, so we just wrote out an index.js file.

Inside that file we wrote out an import statement

to get access to the faker module.

The faker module is not just one single file.

Instead, faker is composed

of many different files internally.

When we wrote out that import statement

we were essentially saying

that we wanted to get a collection

of different files from that faker module and make use

of those different files inside of our index.js file.

So we can really picture the code that we just wrote out

as being like the left hand side of this diagram right here.

We created an index.js file

and we imported into it a couple of different dependencies.

We can see all the different files that constitute

the faker module if we take a look back at our terminal.

So you'll notice that

all these different lines are referring

to different files within the faker module.

So I've got something around faker, some more faker.

These are all files around faker, so we can see very easily

there's a lot of different files that are being

included just so we can make use of faker.

Now, of course, when we start to load up all of our code

inside the browser, in general, as a rule of thumb,

we do not want to load up many different files.

Instead, it usually makes a little bit more sense

to load up as few files as possible.

Now there's a lot of different variations

on that rule and it's not a hard and fast rule,

but in general we don't want to try to load up

hundreds of different files into the browser

and that's where Webpack comes into play.

The goal of Webpack is to take

many different JavaScript files

which we have right now, even with one single dependency,

and combine them all down to one single output file

which in our case was named by default main.js.

That's the file we just saw inside of that dist directory.

That main.js file is the output of Webpack.

You will frequently see that file also named

as bundle as well.

So that one single file contains not only the code

from our index.js file

but also all those different files from faker as well.

If we go back over to our editor

and take a look at the main.js file,

we can scroll through here and even though

a lot of this code looks really confusing

you can see that there are references to faker/index.js.

So that's the index.js file inside the faker module.

We can see some stuff around addresses

and commerce and company.

These are all different files.

All the different code inside these files

have been extracted and thrown into

these different eval statements.

So you'll notice this eval function

if you scroll all the way to the right hand side

this is a really, really long line of code.

This one line contains all the code

out of that one file inside the faker module.

So we can see very easily that Webpack has

taken all the code outta these different files

and essentially place them into this one main.js file.

We can find the code out of our index.js file

if we go down to the very bottom of this file

and then scroll back up just a little bit

and you'll see right here a comment for src index.js.

So that is our file.

So all the code out of our index.js file

is contained inside this eval line right here.

You will notice that there's a lot of extra code

inside of here as well just due to how

Webpack processes our code.

If you scroll over to the right far enough

you'll eventually see some stuff

around creating that product string.

And then our for loop, the body of the for loop

and then eventually

the console log of products right there.

So at this point in time

all I really want you to understand around Webpack

is that we throw in some collection of files.

It somehow combines all those files together

into one single file and we get this output named

either Bundle or Main or any other

another other couple of names.

We can really name that file anything we want,

but usually by convention it's called either

main js or bundle js.

Okay, so now that we've seen a little bit around Webpack

let's start to expand a little bit more in the next video.

Topic 11. A Touch More on Webpack

Instructor: We've now got our main.js file created,

but just creating this file

doesn't really do us a whole lot.

We now want to somehow take this file and execute it

inside the browser, and make sure that eventually

the code inside of here generates some HTML

that gets displayed on the screen.

So we need to load up main.js into the browser.

To do so, we're gonna first set up

a little module called Webpack Dev Server.

Webpack Dev Server is gonna take the output

from our Webpack process and make it available

in the browser.

To set up Webpack Dev Server,

We already installed it just a moment ago.

All we have to do is make a change

to our Webpack config file.

So, back inside of our editor,

I'm gonna find the Webpack config.js file.

I'm gonna add a new section inside of here of devServer.

Make sure that server has a capital S.

Then I'll put in a port of 8081, and it's really important

that you use a port of 8081.

Later on this port number is gonna become very critical

and we need to make sure that we've got

absolutely identical port numbers.

Otherwise, you're gonna have to figure out

some configuration steps on your own.

I'm gonna save that file and then to make sure

that I run Webpack with dev server enabled

I will also go into my package.json file.

I'm going to find the start script again,

and I'll update the command to "webpack serve", like so.

I'll then go back over my terminal

and let's run, npm run start, once again.

After another little pause we'll see a message right there,

very briefly, that says that our project

is running at localhost:8081.

Now we can access the main.js file that was created

by opening up our web browser and navigating to

localhost:8081/main.js.

And there is the contents of our main.js file.

You'll notice that we are seeing the contents

of the file, which is not really useful to us.

We don't wanna see the contents.

We want to run this file.

To run the file, we have to create an HTML document

and then reference this main.js file using a script tag.

So, let's go through that process as well, right away.

Back inside of my editor, I'm going to create

a new directory that I will call "public."

Then inside that public folder I'm gonna make a new file

called "index.html."

At the very top, I'll put in a DOCTYPE.

I will put in an HTML tag.

A head,

A body.

And that's all I'm gonna put in for right now.

You'll notice that we don't have

any script tag inside of here at all.

Well, it turns out that when we are making use of Webpack

we are not going to try to manually reference

that main.js file.

Instead, we're going to have Webpack figure out how

to add the script inside of here for us.

Lemme show you a diagram just to help you

understand why we're going to do that.

Okay, so eventually we want to load up an HTML document

inside of our browser that's gonna have a script tag

that tries to load up that main.js file.

So we should see some kind of script tag

that says something like, source main.js,

or something like that.

We could definitely add that in manually.

But as you're gonna see very quickly, and I can actually

kinda modify our index.js file to do this very easily,

We might eventually have files coming

out of Webpack that do not have predictable names.

Right now our file that comes out,

once it's been joined together,

is always going to be called main.js.

But it's very easy to accidentally make some changes

to your project, and you're going to eventually

want to make these changes, and that's going to have Webpack

spit out some file names that have dramatically

different names that are absolutely unpredictable.

So, it might have some kind of identifier like this

right at the start of the file name.

And these different identifiers

are gonna be really hard for you to guess.

So, we can't really just have a static script

inside of our HTML document because it would

be really challenging for us to guess

what those identifiers are always going to be.

So, instead, we're going to make use of a little plugin

called Webpack html plugin.

Actually I think it's called HTML Webpack plugin.

This plugin is gonna take a look at the different files

that are coming out of Webpack.

It's gonna take a look at those file names,

and then automatically update that HTML document

we just put together to add in some script tags

that specifically reference these

very specially named files.

So, this plugin is essentially going to create for us

and add into the HTML document,

three script tags with the name of 1k4j43.bundle,

and 46j3ji.bundle, and the last one right there as well.

So, this plugin will make sure that we always load up

the correct output files that are coming out of Webpack.

So, let's set up that plugin and tell it to modify

that HTML document we just put together.

To do so, we'll go back over to our Webpack config file.

At the very top, we will import that plugin

with HTML Webpack plugin.

And then at the bottom of this object,

I'm gonna put in a plugins array.

Inside there I'll add in new HTML web pack plugin.

We're gonna pass it an object, give it a template property,

and tell it to use the HTML file

inside of public index.html.

So, this plugin is gonna take a look at whatever files

are coming out of our Webpack process.

It's going to find those file names, and then

add the appropriate script tags automatically

behind the scenes to this HTML snippet right here.

Okay, so let's save this.

We'll go back over to our terminal

and we're going to restart Webpack.

We're gonna do a Control + C,

and then another npm run start.

And now if we go back over to our browser and go

to localhost:8081, without any file names

or anything like that, and open up our console,

we will see our console log from our code.

So, it looks like the contents of our index.js file

is now correctly being executed.

We can also make changes to that file at will.

So, how about back inside of our index.js file,

let's now do everything from zero up to five.

We're gonna do an i < 5 right there.

That should automatically be updated by Webpack.

So, back in our terminal, it looks like Webpack

saw the update we just made to that file.

And if we now go back over to our browser, it looks like

the console log has automatically been updated as well,

and we now see five products being generated.

Okay, so that is the basics of Webpack.

We've now gone through the process of having Webpack

concatenate together-

Not really concatenate, but combined together

a bunch of different files, produce one output file,

and we're now loading that file up inside of our browser.

Now, if all this stuff is just total review for you,

I apologize for the review on Webpack,

but it's really important for you to understand these basics

because as we go through this idea of module federation

inside of Webpack, it becomes super critical for you

to understand what is going on

behind the scenes with Webpack.

So, let's take a pause right here

and continue in just a moment.

Topic 12. Finishing the Product List

Instructor: To finish off our products microfrontend,

we need to make sure

that our list of products we've now generated

somehow gets displayed on the screen to the user.

To do so, I'm gonna open up my index.html file.

Inside of that body element,

I'm gonna add in a new div with an ID of dev-products,

and close the div off.

I'm gonna save that HTML file,

I'll then go back over to my index.js file,

I'm gonna delete the console log of products,

and I'm going to replace it

with a querySelector for that element we just created,

and to show our list of products on the screen,

we're gonna set that elements innerHTML property

to products.

You notice that innerHTML,

the HTML right there has all capitals.

Let's now save this file and go back over the browser

and just make sure that our list of products

is being displayed.

Yeah, there we go.

Well, believe it or not,

that is kind of it for our products microfrontend.

We are now generating a list of products

and showing them to the user,

but that's not really the end

of what we need to do inside this project.

Right now, we have developed a standalone application,

this thing works by itself,

but there's really nothing inside of it

to ensure that we can somehow integrate this

with our container application,

so let's have another quick pause right now,

when we come back,

we'll figure out how we can somehow

plan around integrating products with the container app

once we build it.

Topic 13. Scaffolding the Container

Instructor: The products micro frontend

is now in a pretty good state,

so now we're gonna start to work on our container

and then figure out how to integrate

these two micro frontends together,

so let's start working on container.

Inside my code editor,

I can currently only see the products directory.

I'm gonna reopen my code editor

so I can work on both the products

and the container projects at the same time.

To do so, I'll go to file and then open.

I'm then going to select my e-comm directory and open that.

Now if you're in Windows,

the process will be just slightly different,

but ultimately, all we want to do

is open up the e-comm project inside of our editor,

so we should see both our container

and our products folders.

Okay, so now we're gonna do a little bit more setup

on our container.

The setup inside of here is gonna revolve

around the exact same files we had created for products,

but some of the files are gonna have

some different configuration and code inside them.

At the end of the day, though,

we're gonna have the pretty much same files,

an index.js, an HTML file, and a Webpack config.

So I'll first begin by creating that index.html file.

I'm gonna create a new directory called public,

and then an index.html inside there,

I'll place my DOCTYPE and HTML tag.

There's my head, my body,

and we're gonna leave it just like that for right now,

so one thing I wanna point out here is that,

inside of our container HTML document,

we've got an empty body,

and inside of our products HTML file, we've got a div

where we can render our actual products project,

so that list of projects are going to be inserted

into that div right there,

so I just wanna point out really quickly

that these two HTML files are just slightly different.

Next up, inside of our container,

we're going to make another folder of SRC.

Inside there, I'll make a new file called index.js,

and for right now, I'll do a very simple console log,

and I'll just say container.

I'll then save that file and close it.

And then finally, we'll make one more file,

the webpack.config.js file.

So for right now, this file is going to look kinda similar

to what we just did over inside of products,

but we're gonna very quickly make some big changes to it.

So inside of here,

we still want to have that HTML Webpack plugin,

and I'm going to require that from HTML Webpack plugin.

We'll then do another module exports.

We'll set our mode to development.

We'll set our dev server,

and this time around, for our port, we're going to say 8080.

So at this point in time, remember,

our products project is on port 8081,

and our container is gonna be on 8080,

and then finally, we'll set up our plugins array.

We'll have a new HTML Webpack plugin.

We'll set the template to be public/index.html.

Okay, just one last piece of setup.

Inside of our package.json file,

we'll make sure that we add in our start script,

so I'm gonna replace inside of package.json

inside the container directory, test, put start,

and whenever we run that,

we're going to actually execute our Webpack serve, like so.

Okay, so we've now got products, and we've got container.

These are two separate projects.

There's no link between them, there's no tie,

there's no integration at all.

Just to make sure the container is working by itself,

let's go over to our command line,

make sure we can launch container,

and then just try to visit it inside of our browser.

So back at my terminal,

it looks like my NPM install is all complete.

I'll make sure I'm inside my container directory,

and I'll do an NPM start or NPM run start.

Either one is totally fine.

Okay, so looks like everything has started up.

I don't see any errors.

I might see a warning or two. Totally fine.

Now, to make sure that our container is working correctly,

I will open up my browser once again,

and then I will navigate manually to localhost:8080,

and I should see inside my console something

that says container, like so.

Okay, so we've now got two projects created,

and now we need to think about how we're going

to integrate them together.

So as you could guess,

that's gonna be a big topic for our next video,

so quick pause right here, and we'll dive right in.

Section 2: The Basics of Module Federation

Instructor: We've now got our container and products

projects in a pretty good state, but now we need to make use

of some code from products inside of our container.

We need to set up our integration process

and that's exactly what we're gonna

start to do inside this video.

So lemme show you a diagram that's gonna walk

through the exact series of steps we're going to go through.

Okay? So we're gonna go through this one by one.

We're gonna go through all these steps rather quickly.

And in the coming videos we're gonna go into great detail

and I'm going to explain what

all these different steps are about.

So to get started, we're going to first

designate one application as a host and one as a remote.

Our host is going to be the application that is

trying to make use of code from another project.

So for you and I, our host is the container.

Your remote is a project that is making code available

to other projects.

So for you and I, our remote is products.

Next up, inside of our remote, which again is products

we're going to decide which modules

or files really is what we call 'em you want to

make available to other projects.

So for you and I, right now

this is a really easy decision to make.

Inside of our products project

there's only one source code file available

and that is our SRC index dot js file.

This is the only file that we can somehow

make available to other projects.

So we want to make index dot js available

to other projects inside of our overall project.

Okay, onto step number three.

So this is where things are gonna start to get

really interesting.

To somehow integrate these

different projects together,

we're going to use a plugin from webpack

called the Module Federation plugin.

Like I said, we're gonna go into great detail

about what all these things are about very shortly.

But for right now, let's just set up this plugin.

To do so, I'm gonna find the webpack

and fig file inside of my products directory.

Then at the very top I will require in

module federation plugin

from webpack slash

lib slash container slash module federation plugin.

Once I've got that required statement, I'll then

find my plugins array.

I'll do a new module federation plugin

and we're going to provide a couple of options

to this thing.

First, we're gonna put in a name of products,

a file name of remote entry dot js,

and then finally exposes.

This is gonna be an object with a key of dot slash

products index and a value of dot slash src index.

Now I'm sure all these different options right here

right now look a little bit mysterious.

Like I said, don't worry, we're gonna come back

and understand what this is all about.

So I'm gonna save this file and then close it.

Now we'll move on to our next step;

inside of our host, which is our container,

we're then going to decide which files

we want to get from the remote.

So in our case, once again, really easy decision

there's only one file available.

The only file that we can get access to inside

of our host is the index dot js file inside of products.

So we want to get access to that file.

Once we've decided upon that, we're going to set

up the module federation plugin inside of our host, as well.

So inside of our container webpack config dot js

we're going to again require in module federation plugin

from webpack lib container module federation plugin.

I'll then go to my plugins array.

We're gonna set up in a very familiar or very similar way.

So new module federation plugin

and then we'll pass it a couple of options.

So in this case we're gonna add

in a name of container

and then a remotes.

That's gonna be an object.

We'll put in products and we'll put in a string here.

And the string is gonna look a little bit confusing

so please double check your spelling and all your typos

inside of here.

We're gonna put in products at http colon slash slash

local host 80 81 remote entry dot js.

Okay, there's our setup inside the host.

Now this next step, once again

little bit confusing, don't sweat it.

Inside of our host we're gonna do

a little bit of a refactor.

So all we have to do inside of our container directory

we're gonna find the index dot js file.

I'm going to rename that file to bootstrap.

Now we've got bootstrap dot js.

And then inside of src I'm gonna make

a new file also called index dot js.

And inside of here I'm gonna put in an import function call.

So notice we don't have an import statement

it's an import function call and we are going to

write in there dot slash bootstrap like so.

And then finally inside of our host,

we're going to import whatever files

that we actually need from the remote.

So for that, we're going to try to figure out,

"Hey where do we actually want to make use

of our source code that we're importing from products?"

Well, we probably want to get access to all that source code

inside this new bootstrap dot js file.

And I say probably, but that's probably not really

clear why I say probably just yet.

Don't sweat it right now, just inside a bootstrap js

at the very top, we'll put in an import of products

slash products index like so.

And that should be it.

Okay, for the fifth time in this video, I just wanna mention

I know a lot of this stuff is probably

really mysterious, really confusing.

Don't sweat it.

We're gonna go over it step by step.

I apologize for repeating that,

but just wanna make sure it's super clear.

Now to test everything out, we're gonna go back over

to our terminal and restart our webpack processes.

So back at my terminal,

I have two terminal windows open.

I've got one right here for my container or for my products

and then the other for container.

So I'm gonna stop both these with the control C.

So here's products folder and the container folder.

And inside of both I'm going to do another npm run start.

Okay, so now we're ready to test everything

out inside the browser and I want you to know

that when we run this in the browser, we might end

up seeing an error message, but that's totally okay.

We're gonna fix it up really quickly.

So back inside my browser

I'm going to navigate to local host colon 80 80.

And then once there I'm gonna open

up my console and I might see some kind of error like this

something that says cannot set property innerHTML of null.

So we're seeing this error right now

because we are trying to run some code

from our products application.

If we go back to products and find index dot js

remember that when we load all this code and execute it

we try to find some element on the screen with an ID

of dev dash products and then we try to set its innerHTML.

So right now we are running this code

inside of our container project

and inside of our container project

we don't have an HTML element with that selector.

So to get our code to work, we're just gonna very quickly go

into our public index dot HTML file inside of container

and inside that empty body I'm gonna add

in a div with an ID of dev dash products.

I'll then save the file,

pull it back over,

and sure enough

at local host 80 80

we now see our list of products on the screen.

Remember local list 80 80 is our container application.

We can also see a console log of container right here.

So I think at this point in time,

it's pretty clear that we are inside of our kind

of container application environment, but we have loaded up

and executed some code coming from our products application.

So this is definitely good, but a lot of mysterious code.

So let's take a pause right here and then start to figure

out what is going on in the next video.

Topic 15. Understanding Module Federation

Narrator: We've now set up some integration

between our products project and our container.

Well let's now take a look at just our configuration

around products, and really understand what the

module federation plugin,

inside the products webpack config file is doing for us.

So I just want to focus on this module federation plugin

inside of products for right now.

Okay, we're gonna take a look

at a couple of different diagrams.

But first, I want to go back over to my browser.

I'm still at localhost 8080.

Once here, I'm gonna open up my console.

I'm gonna go to the network tab.

I'm gonna filter request by JavaScript files,

so now we're only gonna see requests

that are trying to load some JavaScript.

And then I'm going to refresh the page.

I'll then see a couple of different files are being loaded.

Now remember, we are at localhost 8080 right now,

which is the port for our development server

for our container application.

On this list of files,

I'm gonna right-click any one of the headers,

and then enable the URL column.

Once I get that column up,

I can take a look at where all these different requests

are being made to.

You'll notice that three different files are being made

to localhost 8081,

which is where our products development server is running.

So when we go to our container application right now,

it looks like we are loading up three different files

from products.

One file is called 'remoteEntry',

another is called 'src\_index\_js',

and another has a really long name that seems

to be related to our faker dependency.

So now that we've identified these three different files

that we are loading up from our products project,

let's take a look at a diagram and understand

what they're all about.

Okay, so here is a diagram of what's going on

inside of just our products project for right now.

So we still have products,

we still have an index.js file inside there

that makes use of the faker module.

When we start up webpack with this

module federation plugin, we are now emitting

two different sets of files.

We are still bundling up that index.js file as usual,

and we are spitting out a main.js file that includes

everything from index, along with all the dependencies

that that thing requires.

I only mention this because that means that we can still

run our products project in isolation,

without having to worry about running a container

or any other application.

So we can still develop our entire products application,

in a kind of standalone environment.

By adding in the module federation plugin, however,

we are now emitting a second set of files.

The module federation plugin is causing us to

output another set of files,

which are the remote entry file, src\_index,

and something around faker as well.

These are the three files that we just saw

inside of our network request log.

The first file, remoteEntry.js, is a JavaScript file

that contains kind of a manifest of sorts.

It lists all the other files that have been emitted

by the module federation plugin, and it also has

directions on how to make use of them.

The remoteEntry.js file can be thought of as being, kind of,

a set of directions for other projects like our container.

This set of directions tells the container how to load up

some source code from the products project.

So the remoteEntry.js file will tell our container

that if the container needs that src-index.js file,

well, here's some directions on how to go

and fetch that file.

And if it needs something around the faker module,

here's some directions on how to fetch this file as well.

The src\_index.js file and the faker.js files,

once again we saw these inside of our network request log

just a moment ago, are essentially identical to the

index.js file we've got inside of our products project,

and the actual faker dependency.

But they've been processed by webpack.

So rather than just having the exact code that we have

inside of our code editor, such as this right here,

webpack will take all this code and convert it,

so that we can make use of modules

like import statements and stuff like that,

inside the browser.

Long story short, the version of this src\_index.js file,

and this faker stuff, that gets output from the

module federation plugin, can be safely loaded and executed

in the browser.

Let's go take a look at the source code for src\_index.js

inside of our browser right now,

just to see what it looks like once it's been processed

by the module federation plugin.

So back over here, I'm at my network request tab once again,

here is src\_index.js, and I can see that it's been loaded

from our products project.

If I click on that,

and then go to the either preview tab or response,

we're going to see some source code inside of here

that essentially says,

"Here is the source code from the actual src\_index.js file".

All the source code from it is inside this eval string,

and inside there we can see that there's some direction

saying, "Hey, well we need to load up faker,

and eventually once we load faker, we're going to create

that product string, do our four loop, build up the list

of product names inside of a div, and then eventually

render that output to some element that's inside the dom."

So you can see very easily that this file right here has

all the same code from our real src\_index.js file,

but it also has some stuff injected into it

by webpack that says, "Here's all the other dependencies

that this file requires to run safely."

Okay, so at this point in time, we've really established

what's going on with the module federation plugin when it

is running in a remote mode.

Or, essentially, when we are telling

the module federation plugin that we want to take

some code out of our current project,

and make it available to other projects.

The module federation plugin spits out this set of files

that includes a set of directions on how other projects

can get access to the source code of products.

So now that we understand just that piece,

lets take a pause here, and then understand

what this same plugin does over inside of our

container application.

Formularbeginn

Formularende

Topic 16. More on Module Federation

-: Let's now take a look

what the Module Federation plugin does

for our container application.

So, we're gonna take a look at a very similar diagram.

The first thing I want to address

is the fact that we had created a second file

called bootstrap.js

when we did the integration process.

So, back inside of our editor,

here's my container application.

We now have a bootstrap.js file

which contains our actual application code

and an index.js file.

And inside of index.js,

all we are really doing here is importing the bootstrap file

and nothing else.

And what's more,

we're using some different syntax for the import.

So, what is this index.js file all about

and why do we add it in?

Well, this really gives Webpack the opportunity to realize

that before executing that bootstrap.js file,

we have to go and fetch some code from the products project.

That's what the index.js file is for.

It is just to allow Webpack that opportunity

to go and get some additional JavaScript

and make sure that we've got the project code

or something the products code ready

along with the faker code ready as well

before we try to actually do something with it.

When we actually execute the contents

of the bootstrap.js file,

the expectation is that we will have access to this module

and all the different dependencies that it requires.

So, if we tried to go directly into our bootstrap.js file,

in other words if we tried to execute this first,

we would end up with an error.

We would end up with some kind of message that says,

"Sorry, but we don't have any code for products.

We don't know where to get that code.

We don't have it available."

So, by adding in the index.js file,

we are just giving Webpack an opportunity inside the browser

to go and fetch some dependencies from products

before actually executing the code of Bootstrap.

We can kind of see this rather easily inside the browser.

So, if we go back over to our browser,

open up that network request tab again,

we could take a look at that main.js file.

So, if you click on it

and go to either a preview or response,

you can then go down to the very, very bottom of the file

and then scroll back up just a little bit

and you might see an entry right here

for source index.js.

This is the index.js file inside of our container.

We once again have this string with all the code from it.

You'll see that the code has been transformed a little bit.

And so now, rather than just having something that says,

"import bootstrap",

we've now got something that says,

"require in bootstrap js or source bootstrap js."

And then after that, try to execute the code inside of it.

So long story short, once again,

this is really just giving us

or really giving Webpack the opportunity

to go and fetch some code from products

before running our index.js file.

Okay. So, back over to this diagram.

So again, inside of our container,

we've got the index.js file.

That's really generating main.js

and then our Bootstrap is really being output

as a separate file.

We first run main.js.

That's going to try to fetch some additional code

from products

and then execute the code inside of bootstrap js.

Now, let's start to put our container

and our product stuff together

inside of one diagram

and understand the real flow of what's going on.

Okay.

So, in the browser, we've got a series of steps that go on.

We've then got whatever is coming out

of our Webpack Dev Server for container

and on the right, we've got products.

When we first go to localhost:8080,

the first thing you do

is download and execute the main.js file

from our container.

Once we execute that file,

Webpack then sees a required statement saying that,

Oh, we need to import and somehow execute bootstrap.js.

So, Webpack is gonna start to download that file.

As it starts to download that file,

Webpack also realizes

that it needs something from the products project

in order to execute Bootstrap.

So simultaneously,

as it starts to download bootstrap.js,

Webpack inside the browser is going to fetch remoteEntry

from our products application.

This remoteEntry.js file says that

in order to run bootstrap.js,

we have to download source index.js

and faker from products as well.

So, Webpack is gonna download source index.js

and everything for faker.

Once it has completed the download for Bootstrap,

for source index and for faker,

now it finally has all the code that is needed

to actually execute Bootstrap.

So, it runs the code inside of Bootstrap.

We've got everything from products

including the source index.js file

and faker over there as well.

Another way that we can really verify this entire process

is by taking a look

at the sequence of requests that are being executed.

So, back over here inside of our network request log,

we can take a look at the waterfall chart.

So, inside of our waterfall chart,

you'll see that the first thing happens

is we download and execute the main.js file.

Once we execute that file,

we then realize, okay, we also need something from products

and we need that bootstrap file.

So simultaneously,

Webpack goes to download the bootstrap file

and it also knows

that it needs to download the remote entry file

to figure out how to get some code

from the products project.

Once it gets the remote entry file,

it then sets up another two requests

to get the faker JS stuff and the index.js stuff

from products.

So, it's only when these two requests right here

are finished

and we get the ability to execute the code from products,

that essentially all three of these different files are ran.

The bootstrap, faker and index.js.

That's what's going on behind the scenes.

Okay. So, all the stuff I know is pretty intense.

There's a lot of stuff going on here

with this Module Federation stuff

and it actually even goes deeper than what I presented.

It turns out

that there's a lot of extra corner cases

around handling dependencies

for stuff like faker.

We're going to explore some of that

as soon as we start to add in our final micro front end,

which is the cart application.

So, let's take a pause right here.

We're gonna come back in the next video

and make one or two changes to our products project.

I wanna point out one or two things around it

that is not quite great right now.

In some ways, we could probably improve

some of the structure of our code.

So, quick pause,

and we'll do some improvements in just a moment.

Formularbeginn

Quiz 2: Module Federation Terminology

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Topic 17. Understanding Configuration Options

Instructor: At the end of the last video,

I mentioned that we were gonna change some code

inside of our products project.

But before we do so,

I want to explain some of the options we provided

to our webpack configuration really quickly.

So specifically the options we provided

to the module federation plugin.

We're gonna first take a look

at the options we provided to our container.

So inside our container, webpack.config.js.

All right, here we go.

So the first option we provided was the name.

The name is not being used

whenever we are creating a host module.

Remember, our container is a host.

It is trying to use some code from some other projects.

A name is only required

when we are creating a remote.

Remember, a remote is something

that is trying to share code with the outside world.

Even though this name property is not used

because we are inside of a host,

it is usually convention to add it in anyways.

Next up, we have a remote section.

The remotes controls how webpack is going to try to decide

whether or not to load up that remote entry .js file.

Rather than taking a look at all the notes on here,

let me just give you a very practical example

of what this remote section is all about.

Okay. So inside of our container project,

I'm gonna find the src bootstrap.js file.

You'll notice that inside of here,

we've got an import statement that begins

with the word products.

So when we build our project

and it's compiled or processed by webpack,

webpack is gonna see that we are trying

to import a module that begins with the word products.

If webpack cannot find a products dependency

inside of our node modules directory,

then it's going to take a look

at the module federation plugin,

specifically the remote section.

Inside this object, it's gonna see

if there are any keys inside of here

that have the word products

and sure enough, there it is right there.

So this word products matches up

with this key products right here.

This tells webpack that to try

to implement this import or to try to actually get access

to this code, webpack needs to load up the remote entry file

that is listed right here.

So if we ever wanted to change how we refer

to our products project inside of our container,

we would change the word products right here

and this key right here.

Let me show you an example of that.

Maybe we want to change this to productsApp

or something like that.

Maybe that's how we want to import this thing.

To do so, we would have to also go into our webpack config,

find the word products right there

and change it to productsApp.

I can then save both these files and then go

and restart my webpack process really quickly.

If I then go back over to my browser,

I can refresh the page

and everything still works as expected.

So you only have to change this key right here

if you want to change how you write out the import statement

for this outside project.

I'm gonna very quickly change this back

to products and products like so.

Okay, so that is the purpose of the key of that object

and the value is a string.

On the right hand side of that @ sign is a url.

And as you guess that is where we should go

to try to get the remote entry file

to figure out how to load in some modules

from this outside project.

The word on the left hand side of the @

which says products is related

to some configuration that we put into our webpack config

corresponding to our products webpack config file.

The word on the left hand side, the @ right there,

products, is a tricky little thing.

I'm gonna show you a little bit more

around that in just a moment.

So we're just gonna forget the word products right there

for just a moment.

Okay, so that's it for our container webpack config file.

Now let's take a look at our webpack config

that we placed inside of products.

So here's a diagram for it.

I'm gonna also open that up

inside of my editor very quickly.

Okay, so now here we've got a name of products.

That products right there.

That string has to be identical

to that products string right there.

That's what needs to match up.

So if that string is not identical

to that string inside the remote config,

nothing is gonna work as expected.

Let me show you this in practice very quickly.

So I'm gonna open up my products webpack config.

I'm gonna find that name property

and I'll change it to, how about just like,

myproducts like so.

I'll then save this

and I'm gonna restart my webpack processes once again.

I'm then gonna go back over to my browser

and now I'm gonna see this nasty error here.

It says sorry, but something went wrong.

So again, the rule of thumb here is

that name of products right there has to match up

with wherever we are trying

to make use of this module from,

has to be identical to that string right there.

That's pretty much it.

Now, beyond that, we've also got a file name

that controls the name of the remote entry.js file.

There's just about never a very good reason

to rename this file.

I recommend that you just always place it

as remoteEntry.js.

And then finally we've got exposes.

So the exposes object right here controls

which modules or which files inside of our products project

that we are gonna expose to the outside world.

So if we had many different files inside of products,

we could choose to export all of them

or maybe just one of them

or as many in between as you want.

So in this case, we are saying that we are going

to expose a module called ProductsIndex.

And if anyone tries

to import something called ProductsIndex,

we want to give them the src/index file.

So let's take a look at that as well.

Back inside of our bootstrap.js file,

remember we had said products/ProductsIndex.

So webpack is going to get that remote entry file.

Inside there, it's gonna try

to find something called ProductsIndex.

Inside of our webpack config for products,

anytime someone asks for ProductsIndex,

we're going to instead give them src/index

which is the index file inside of our products project.

Now, this line right here is probably gonna be one

of the most confusing things for you.

You might be saying,

"Wait, we're essentially renaming something."

That's what's going on here.

It's an alias.

And you might be saying, "Wait, why would we ever do this?"

What's the point of it?

Why don't we just expose everything inside of products

and say, "Hey, that's it."

You can just reach through

and just take out any file you want

and it can have any arbitrary name.

The reason that we have this kind

of aliasing ability right here is

because there might be some scenarios

where the name of a file inside

of our products project might be kind of

only meaningful in the context of products.

Let me give you a really quick example of that.

So quick diagram.

Here we go.

So let's imagine that rather

than making some kind of products thing,

maybe we're making some kind of charting library.

A library that generates charts

and shows them on the screen or something like that.

Maybe inside of our chart project,

we've got some different files organized like so.

Maybe we've got a barchart directory

and inside there is an index.js file

and that file has all the code required

to create a bar chart.

Maybe we've got another one for making a line chart,

a radar chart and so on.

And so all these different files are named something

like barchart/index, linechart/index and so on.

This is a really small thing,

but we might not want to write out

inside of our container project something like

import bar or really be charts/barchart/index.

Adding on the index on there

at the very end is just not super necessary.

We don't really gain anything by putting on the word index.

And so maybe it would be just ever so slightly

more convenient

if we could just write out something like charts/barchart

and be given that barchart/index.js file.

So that's all this kind of aliasing thing is doing for us.

It's just making it so it's slightly more convenient

to choose exactly the names and paths

for all these exported files.

That's all.

At the end of the day, once again,

all this really means is

that if someone asks for the ProductsIndex file,

give them the src/index file.

And so again, back inside of bootstrap.js,

we ask for ProductsIndex

and we got back src/index.js from products.

Okay. So again, if that part

about little aliasing part right there is confusing,

I apologize for that, but it's just reality.

It is what it is.

Okay, so that's a deeper dive into alias configuration.

I apologize for the last video being really long

and this one being really long

but hopefully you've now got a better idea

of what's going on with module federation.

So let's take a pause right here

and continue in just a moment.

Topic 18. Scaffolding the Cart

-: Let's get started on our cart Micro Frontend.

As a reminder, this little cart feature is just

gonna show the very simple text right here

that says something like you have one item

or two items or whatever in your cart.

The actual number is gonna be randomly generated

so we don't have to worry

about having any actual adding functionality

or anything like that.

The code for the cart is gonna be extremely similar

to the products list,

but as soon as we add in the cart

we're gonna start to run into a couple

of difficulties and we're gonna have to think

about some different ways to resolve them.

So let's get to it.

Back inside of my code editor.

I'm gonna first begin

by creating a new folder called Cart.

Then to save just a little bit of time

I'm gonna copy the package dot json file from products.

So here's a copy and I'm gonna paste it into cart.

Then inside the cart package dot json file

make sure that you've got the cart one open,

we'll just change the name of the project to cart.

This name is not actually used anywhere

it's just to for record keeping.

So let's now go over to our terminal.

We're gonna open up

a third terminal window that's gonna be dedicated

to running our cart project

and we'll install dependencies into it.

So I'm gonna open up my third terminal window.

I'll change into the cart folder

and then do an npm install inside there.

Once I start that up

I'll go back over to my editor and we're going to

essentially create the exact same files inside of cart

as what we have inside of products.

I would really encourage you to write these all out by hand

as opposed to just doing some copy paste

because it just builds a little bit of muscle memory.

So let's get to it.

I'm gonna first create a new folder of public

and I'll make a new file inside there of index dot html.

So once again, do a doc type

html

a head

a body

and I'll place a div with an id of dev dash cart.

So that's gonna be eventually where we tried to

render our cart application into.

After that, I'll make another new directory

inside of cart

of src.

Inside there, I'll make a new file of index dot js

and inside of here we'll have some code

that's going to just require in or import in faker.

We're then going to generate our fake text to say

You've got so many products,

and once again just show it into that

div element that we just created a moment ago.

So at the very top, I'm gonna import in faker.

I'll then put together a variable

we'll call it, How about, message?

How about, cart text you can call it anything.

doesn't really make a big difference.

I'm gonna make this a template string.

We'll put it in a div and I'll say you have

and then dollar sign, curly braces, baker, random number

and then items in your cart.

Let me zoom out just for a second

so you can see that entire line.

So then after that, we're gonna try to take that text

and render it

into that div right there.

We'll do a document query selector

cart dash dev

set inner html to cart text like so.

Okay, so that's it.

Now our last thing inside of cart

I'll make a new web pack config dot js file.

So again, this is gonna look very similar

to one we just put together

for our products application.

At the very top I'm gonna require in HTML

web pack plugin

I'll get my module federation plugin.

Then my module dot exports

set our mode to the

development.

Start up a dev server

And then remember

we've already got a development server running

on port 8080.

That's our container.

We've got one on 8081 that's products.

So we will put our cart on 8082.

We'll then set up our plugins.

So for plugins

first do our new module federation plugin.

Remember, we need a name,

a name of cart

a file name of remote entry dot js.

Again, really no great reason to use any file name

besides that.

And then exposes, and we'll do a string

of dot slash cart show.

And anytime someone asks for that cart show file

we're going to actually give them

source index

or just index like so.

And then finally after that plugin

we can put in our new HTML web pack plugin

and there's our template of public index html.

Okay, so that's it.

We're gonna make sure I save all these files.

And then let's try to run our cart

in isolation just to get started.

So I'm gonna go back over to my terminal.

Looks like my npm install process is complete.

So inside of my cart directory, I'll do an npm.

Oh, you know what I just realized

We forgot to put in our start script.

Let's go back over to our editor

open up that package dot json on file.

Oh wait, no, we copied it so it's good to go.

My mistake.

So yes, we can go over

to our terminal and do an npm run start.

Now we should be able to open

up our browser and go to local host 8082.

So I'm gonna go to local host colon 8082

and it looks like I probably put together

that ID for that element incorrectly.

That's my guess here.

So I'm gonna just go check that out really quick.

Here's cart dash dev.

Oh, dev dash cart.

Yeah, that's the reason.

So one was dev dash cart, the other was cart dash dev.

Just gonna fix that up really quickly.

Now if I go back over and refresh.

There we go.

So I've got 23,000 items in my cart.

That's a lot of items, but whatever.

It works for the purposes of our app.

Okay, so we've now got cart running in isolation.

Now a quick pause right here

and we're gonna start the integration process

into our container in the next video.

19. Cart Integration

Instructor: Let's start to integrate our cart project

into our container.

To get started, I'm gonna open up the container.

Webpac config.js file.

Inside of here, I'm gonna find the remote section.

I'm going to add in another property of cart.

And then I'll give it a string of cart at

http colon slash slash local host 8082,

remote entry dot js.

Then inside of my container, SRC bootstrap dot js file

I'm going to import cart slash cart show.

So lemme just give you a quick reminder

on how this import statement works.

First Webpac is gonna see the word cart.

The word cart is gonna match up

with the cart that we just added

in to our container Webpac config file.

So it's gonna see cart right here

and Webpac is going to understand

that it needs to load up this remote entry dot js file

in order to access our cart code.

Once that file has been loaded up

we're then asking for specifically the cart show module.

Our cart project doesn't actually have a module

called cart show.

so you might recall it inside the cart, Webpac config file

we had set up this alias inside of here.

So we had said that if anyone ever asks for CartShow

we're gonna actually give them source slash index.

So that's how we are ultimately

getting Cart Source index dot js

into the context of our container.

Just a little note on that, a leasing feature

at this point in time, you can start to kind of understand

that if we had an import statement of something

like products slash source index and cart source index

it's kinda hard to understand what's going in here.

Going on here with these imports.

Source Index doesn't really tell us

what this file is all about.

But if we have file names like Products Index

and CartShow a lot more clarity,

I could look at this name right here,

and I've got reasonable suspicion

that this is probably gonna show a list

of products on the screen.

And CartShow is probably gonna show some details

about some particular cart on the screen.

Okay, so this looks good.

Very last thing we have to do is make sure that when

we import in CartShow and run all the code inside there

we need to make sure there's some actual dom element

that our cart is gonna be able to render into.

So inside of my container, public index dot html

I'm gonna copy that div right there,

and put down another div, with an ID,

and I forgot what we ended up calling it.

Okay, it's cart dash dev.

So cart dash dev,

and notice that they're opposite in here.

No big deal.

We're actually gonna come back and change these

divs in just a moment.

Okay, so I'm gonna save all these files and let's

go and restart our container.

So I'm gonna find my container running at the terminal.

I'm gonna stop with a control C

and then do another NPM run start.

So now we should be able to go to

local Host 8080,

and see both our product list

and our cart up here on the screen.

Awesome. So that means that we are loading

up code from those two sub-projects into our container.

Now this is definitely really, really good, but

there are some other considerations I wanna throw out there.

Some stuff that's a little bit awkward

about this project,

that I wanna go into in the next video.

Topic 20. The Development Process

Presenter: We've now integrated

all three projects together.

So at this point in time

I wanna take a very quick pause

and just make sure that it's crystal clear

how you would develop an application

like this in the real world.

All right, so quick diagram.

So we're going to imagine

that maybe we have one engineering team

in charge of each sub-project.

So for team number one, they might be in charge of products.

Team number two might be in charge of cart

and team number three might be

in charge of the container.

Teams, number one and two

can develop their entire application in isolation.

In other words, team number one

can develop products by itself

and they don't really have to care

about what any other team is doing.

They can have all their own dependencies

all their own tooling.

The only requirement is that they make use

of Webpac and they make use of the module federation plugin.

That's pretty much it.

So for team number one, in order to develop their project

all they have to do is change into the products directory

run NPM start inside there.

They can then navigate to local host:8081

and see their application appear on the screen.

They can then develop their products application

in isolation.

Same thing for team number two.

The only difference is

that they're gonna go into the cart directory

and their application will be running at local host:8082.

Team number three is the only one that's gonna

have any amount of crossover.

So team number three would need to make sure

that all three sub-projects are running.

So they would need to make sure they run 'NPM Start' inside

of cart products and container.

Then they would navigate to local host:8080

and they would be making changes only

to the container application, just to make sure

that they are showing the correct sub app

in the correct part of the screen.

Make sure that navigation works and stuff like that.

We'll go into more

on what the container's goal really is over time.

Right now remember we're just saying that's

showing the other two applications.

Now there is one other thing I wanna make sure

that's really, really clear

because you might be a little bit confused

around the existence of these HTML files.

Okay, same diagram.

I'm just a little bit lower

and I added on these red boxes.

So just to make sure it's really clear, our products

and cart applications have index html files inside them.

Those HTML files are only used

during the development of these sub-projects.

They're only used to load up products

and cart and get them visible on the screen of the browser.

So inside of any kind of production environment

we do not make use of the HTML files

inside of products or cart.

Instead, we load up the JavaScript only

from products and cart into our container.

And so ultimately it is the HTML file inside

of container that will be used during production.

So again, we only really worry

about the HTML files in product and cart during development.

And for that reason those HTML files are usually

gonna be very, very simple and very sparse.

We're not gonna really have a lot of markup in them.

In general, we're just gonna have some kind

of root element where we can render our application into

and then we just make sure that our app works as expected.

Okay, so hopefully this entire development process

is just a little bit more clear now

and there is just one or two more changes

and other topics I wanna go into in this application.

So quick pause and just a little bit more.

Section 3: Sharing Dependencies Between Apps

Topic 21. Using Shared Modules

Instructor: In this video, we're gonna take a look

at sharing dependencies between different sub projects.

So as a quick reminder, both our products

and our cart projects make use of the Faker module.

Let's take a look back inside of our browser

and just see how Faker currently is being loaded up.

So back inside my browser at local host 8080,

I'm going to open up my network request tab,

go to JavaScript or filter by JavaScript requests,

and then do a refresh.

Once I do so, you'll notice that there are two files

being loaded up called something like

vendors node modules faker.

So at present, because both products and cart

are making use of this dependency,

they are both loading their own copy of it.

This is definitely not very good,

especially because the Faker module itself

is very, very large.

You'll notice it's about 2.9 megabytes large.

That's extremely large,

and we really want to make sure that we do not load up

two copies of it if we can somehow avoid that.

So if we could somehow figure out some way to get both

Faker and Cart, or excuse me, both cart and products

to use the same copy of Faker,

that would really be fantastic.

And it turns out we can do this very easily

by using the Module Federation plugin.

Let's first begin by taking a look at a diagram

that's just going to explain the series of actions

that are going to occur once we make a little

configuration change.

So when we start up our application the container's gonna

try to find the remote entry file from products

and the remote entry file for cart.

These remote entry files are going to mention

that they both require the Faker module.

So at that point in time, the container application

understands that we need to get a copy of Faker,

and both of these different sub apps of cart and products

need that copy.

So the container can do,

can decide to only load up a single copy of Faker,

and then share that single copy with both cart and products.

So you'll notice that we're getting some really crazy kind

of coordination between these different sub applications

through the Module Federation plugin.

It is the Module Federation plugin that is injecting

or adding in additional code to our project to help

instruct the container on how to make this

series of decisions.

You and I do not have to write out any code

that makes this happen.

We just have to change a couple of configuration options

to make this process occur.

So let's do that right away.

Back inside my editor,

I'm gonna find my products webpack config.js file.

I'll find my Module Federation plugin, and then I'm going to

add in a new option to this thing of shared,

and I'll put inside of an array a string of faker.

I'll then save this file,

and then go over to my cart webpack config.js file.

So here's the cart, and over here we'll put in

shared Faker as well.

So all we're doing here is saying that if we ever try to

require in or import the Faker module,

let's try to share it.

Let's see if some other application inside of our container

has already loaded up the Faker.

If we have, then we're gonna get a preloaded copy

and we're not going to try to fetch our own

independent copy.

You'll notice that we did not have to make

any actual changes to container.

The container is going to automatically see that

we're trying to share the Faker module once it loads up

the remote entry file for cart and products.

Before we test this out, we do have to restart webpack.

So let's do that very quickly.

We only have to restart the products

and the cart applications.

So I'll make sure that I restart my

products webpack process,

and my cart webpack process as well.

Then after that, I'll go back to local host 8080,

refresh the page,

and you'll see that now we are only loading up

one single copy of Faker.

Awesome.

This definitely works,

but unfortunately we just broke something else.

So you can open up a second browser tab,

and go to local host 8081.

Remember that's gonna try to load up our product application

in isolation.

And when we do that now, if you open up your console,

you'll notice that nothing appears on the screen

and we get this nasty error message

that you're probably gonna see very frequently.

So I want you to remember this error message.

It says,

"Shared module not available for eager consumption."

So what's going on here?

Well, remember, when we load up products in isolation,

the first file that really gets executed

is our source index.js file.

And inside of here we have some code that says

get access to Faker.

Right away.

Like we instantaneously want Faker available

inside this file because we're gonna call some functions

on it right away.

Unfortunately, when we mark Faker as a shared module,

that causes it to be loaded up by default asynchronously.

So when we load up our index.js file,

we do not yet have Faker available.

And this is not an issue when we load up products

through our container,

because when we load up products through container,

we are first loading up the remote entry file for products

and a remote entry file has some code and configuration

inside of it that says,

"Hey, we need to get access to index js,

"and to run that file we need Faker."

So when we load up our application through remote entry,

we don't run into any issues,

because webpack can very easily have the time and say,

"Hey, we need to get both index js and the Faker module."

But when we load up products by itself,

we are getting index right away.

We're trying to use Faker right away

and we end up with the error message that says,

"Sorry, but we don't have this module yet."

Because we are trying to use it instantaneously,

but we haven't loaded it.

So that's why we are seeing everything work

over here through our container,

but not when we try to run products through isolation.

So obviously we need to fix this.

Because we still want to run products in isolation

even though we're trying to use a shared module.

So let's take a look at how we fix this error message

in the next video.

Topic 22. Async Script Loading

Instructor: So how are we gonna fix up this air message?

Well, it's really simple.

We're gonna use the exact same pattern that we used inside

of our container project.

Remember inside of container we have an index

dot js file that has an import for Bootstrap.

This import statement is called as a function

rather than being a kind of standalone import.

Like so.

When we use import in this kind of function form,

it's gonna load up the Bootstrap file asynchronously.

So we're gonna do the same kind

of thing inside of products as well.

By doing that we're going to give Webpac the opportunity

to take a look at what files this code requires

to run successfully.

And when we give Webpac that pro-

or that ability to see what we need to run this code

Webpack is gonna have the ability

to load up faker before we try to execute any of this.

So it's all thanks

to just putting in that little import statement.

Okay, so inside of our SRC directory inside of products

I'm gonna make a new file called bootstrap dot js.

I'm then going to open up the index file.

I'm gonna cut everything out of here

and paste it into Bootstrap and save the file.

Then back inside of index js,

I will import dot slash bootstrap.

Now I know this might seem like a really funny,

really pointless little step.

Again, all this does is give Webpack the ability

to asynchronously load up the bootstrap JS file.

And because we are loading up that code asynchronously

Webpack has the opportunity in the browser just to realize

that before we run this code, we also have to fetch faker.

So let's save both these files,

then go back over the browser.

I'll then refresh this page and now everything works.

And if we go to our network request tab,

we can see exactly why that is.

So we can see that we are now attempting to get

our main dot js file and inside of our dot main js file

we've got a ton of stuff inside of here.

But down towards the very bottom down here,

we'll see our source index dot js.

So here's the long line of code.

So we've got this import boot strap js,

and it's that import statement right there

that gives Webpac the ability

to take a look at the Bootstrap file and understand what

the Bootstrap file needs to run.

So now Webpac inside the browser understands to

get that file we first need to get a copy of faker.

So here's the load for faker right here.

And then we can actually go ahead and download

and run the bootstrap js file.

Now if we go back over to local host 80 80

and refresh the page, you'll notice that everything pretty

much works identical over here as well.

We do have an extra request for Bootstrap.

So we are now loading up an additional file

inside this container environment, but when we eventually go

over to production, we're not gonna have quite

so many requests going on.

Okay, so this definitely looks good.

We do have to repeat the same process for our cart as well.

If you try to go to local host 80 82,

you'll currently see the same air message

'cause once again we're trying to make use

of Faker before it's actually been loaded into the browser.

So back inside of cart, we're gonna do the same thing inside

of src I'll make a bootstrap dot js file.

I'm gonna cut everything

from index js and paste it into Bootstrap.

And then inside of index js, we'll do an import bootstrap.

We then save that.

We can now go back over and at local host 80 82.

Once again, our cart application works.

I wanna repeat just for the second time.

I know this seems like a kind of pointless little step.

It's just about introducing some asynchronicity

into loading up our JavaScript code

and that gives Webpac the opportunity to figure out what all

of our different files actually need to run successfully.

Okay, so now that we've got this all fixed up,

I wanted to have a little bit more discussion

about how sharing modules

between our different sub-projects works.

So let's go a little bit deeper in the next video.

23. Shared Module Versioning

Instructor: Right now, cart in products are

using the same exact copy of faker when we load them both

up through our container.

So once again, we can see that very easily

by going to local host 80 80, refreshing the page,

and we're seeing only one copy

of the faker module being loaded here.

But there's a really quick question that starts to come

up once you consider this dependency being shared

between both products and cart.

Remember, these two projects might be developed

by totally different engineering teams

and they could very well decide that they want

to use different versions of the faker module

or any other module for that matter.

So what would happen if one project tried to

use a very specific version of faker and then the other

tried to use a completely different version of faker?

Let's kind of simulate that process and see what happens.

So over inside of my terminal

I'm gonna find the terminal window that is

running my cart project right now.

I'm gonna stop Webpack.

I'm then going to do an npm install faker@4.1.0.

Just so you know, our products application is

using version 5.1.0 of faker.

So we now have two very different versions

of faker being used across our different sub-projects.

Let's now start our cart back up.

And then refresh the page at local host 80 80.

And when we do so, we'll now see that we are loading

up once again two separate copies of faker.

This is desirable, this is what we want to have happen.

We do not want to use our different

or want our different projects to try to use the same copy

of faker if they both expect to have different

and very specific versions.

If we were trying to make use of version 4.1.0 inside

of our cart, there might be a very good reason for that.

We might be relying upon some feature that might break

inside a version or even be removed

inside a version 5.1.0.

So if we have different sub-projects that require

different versions, completely different versions,

we want to make sure we load up different copies

and that is what is occurring here.

So this is definitely good and expected behavior.

Okay, so you might be curious,

how does the versioning system actually work?

Well, the module federation plugin is gonna take a look

at the versions of these different modules

that you specify inside of your package.json file.

So inside of products, we've got version 5.1.0 right here

and inside of cart we've got version 4.1.0.

The module federation plugin is only going to decide to load

up different versions of these modules

if your version specified right here

are significantly different and do not match.

Just so you know, this little carrot right here

means that we can use version 4.1.0

or any other minor version or patch version.

So 4.1.0 with a carrot in front of it is equivalent

to say, 4.6.0 or 4.6.4 or whatever else.

The only thing that can't change and we've

got the carrot right there is major number.

So that's the four at very start.

So let's try tweaking these versions

and see when we can get module federation plugin to

once again share these two versions, even

if the exact number right there is slightly different.

So for that, I'm gonna go back over to my terminal window.

I'm gonna stop cart once again

and do an npm install faker@5.0.0.

So now cart and products have the same major version.

They're both trying to use version 5,

products is trying to use version 5.1.0.

But because we've got the carrot in front of our version

that means version 5.1.0 is okay to use with version 5.0.0.

So I'll now do another npm start.

I'm gonna go back over to local host 80 80, refresh,

and then you'll notice once again we are

using the same version, or excused me, the same copy.

Technically, we like have or we, we have the same copy here

even though they want to use slightly different versions

but because of the little matchers we put on there,

that little carrot in particular, we're saying

that version 5.0.0 right here is compatible

or can be used with the products 5.1.0.

So the little carrots there or you might also

see the squigglies which refer to the mutable patch version.

That's where these things start to

get a little bit more important.

Okay, so now that we can see that we can work

with different versions between these different modules,

definitely a fantastic feature.

There's one other thing around jury dependencies

I wanna show you in just a moment.

Topic 24. Singleton Loading

Instructor: There's one more thing

I wanna show you around sharing dependencies.

So I'm back inside of my cart, webpack config file,

looking at this shared option right here.

Our faker module can be loaded up several

different times inside the browser

without any problem whatsoever.

In other words, we can load faker

and it's still going to work

even if we load it into the browser multiple times.

There are other modules however and the most important

of which is React, where we cannot load them up

multiple times inside the browser.

If you try to load up multiple copies of React,

you're going to eventually get an error message

from React that says, hey, you've got multiple copies

of React running.

Don't do that.

So there might be some scenarios where we want

to have different versions of a library,

which would imply that we want to load up different copies.

But we might want to actually prevent that behavior

'cause if we do load up different copies

we would end up with some kind of error.

So let's simulate that and just see what happens.

And how we can make sure that we only restrict

and get just one single version

or at least get some kind of notice saying,

hey, you probably have something wrong going on here.

So to get started I'm gonna find shared right here

and we're going to define the shared options using

a slightly different syntax.

Rather than putting an array of strings,

I'm gonna put an object with a key of faker

and then we're gonna put in here singleton true.

Singleton true means that we only want to load up one

single copy of faker no matter what, only one copy.

So I'm gonna save this.

I'll then go over to products webpack config

and I'm gonna make the exact same change.

Once again, we're going to restart our webpack processes.

So here's cart, I'll do an NPM start.

Here's products, I'll do an NPM run start.

And now if I go back with my browser and refresh,

once again we're gonna see that

we are only loading one single copy.

So now let's try to get webpack

to try to load multiple copies

and see it kind of conflict with that singleton rule.

Remember, webpack will attempt

to load up multiple copies of a module if we try

to specify multiple different incompatible versions.

So let's go back to cart.

We're gonna reinstall version 4.1.0 of faker

and we're gonna see how the versioning stuff plays out

with the singleton rule and just see what webpack does.

So back at my terminal, I'll find my cart terminal window.

We'll do an NPM install faker at 4.1.0 again.

So now we have incompatible versions

and the module federation plugin is going

to want to load two separate copies of faker.

But like I said, we also put in that singleton rule

which says don't try to load up single copies.

So let's just see what occurs.

Okay, so I'm going to refresh.

We'll see that we still get one copy right here.

But if you open up your console, you'll see a big warning

message right here.

So the warning message says that we are trying

to get version 5.1 0,

or we have version 5.1.0, but we want version 4.1.0.

So this is the sign from webpack that it was not able

to load up the dependencies as you expected.

You said only get one copy, but also get a copy

of version five and a copy of version four,

conflicting rules.

And that gets manifested with a warning message like this.

So whenever you see a warning message like this

it means that you have different sub-projects

that are trying to make use of different versions

but you also said only load one copy.

So this message would be assigned to you that

you would have to go do some debugging and figure out,

okay do we really only want to have one version or one copy?

If you really only want one copy,

can you increment one version or can you decrement

the other version to make these things compatible?

So again, this is the sign to you

that you need to do a little bit more digging and figure

out how to make these two sub-projects compatible.

Adding on that singleton flag doesn't change anything

about running these projects in isolation.

Everything still works as expected.

It's only once we start to load up these different

sub applications together then we start

to see this warning message.

Okay, so I just wanted to show you that air message just

so you understand what happens when we try to load

up one copy, but also multiple copies.

So now we're going to undo that change we just made.

So I'm gonna revert this back to being just faker

inside of products.

Revert it back to being just faker inside of cart.

And then once again, restart my web pack processes.

There's cart and here's products.

They are using different versions right now

so if I go and refresh, I will no longer see a warning

but I will be loading up two copies of faker once again.

Okay, so again, just want to give you that information.

Let's take a pause right here and we're gonna move

on to one last really big topic around this project.

Topic 25. Sub-App Execution Context

Narrator: There's one last major thing

I wanna cover around our project.

It's gonna involve a little bit

of a refactor to all three different projects.

To remember, a major absolute requirement

of micro-fronted applications is

that we should be able to develop each sub-project

in isolation and we should be able to run each project

in the context of the container.

Now, one thing to recall around this

entire process, is that depending upon whether

whether we are running products or cart

in isolation vers a container, we're going to

be executing our projects with a different HTML document.

So if we have products in isolation

we're gonna use the product's HTML file.

That's only during isolated development.

If we are ever developing the container,

or running the container in production,

then our product's application is gonna be

using the container's HTML file.

So it turns out that there's some really

interesting concerns around this.

Let's go back over to our code editor.

Inside of here, you're gonna find

my product's source bootstrap dot JS file.

So inside of here, I just wanna point out

that we have some code that is going to attempt to

generate some HTML and then always

and instantly try to show that HTML inside of

some very particular DOM element.

Now right now we are making a very big assumption.

We are assuming that there is some element inside

of our HTML document with an ID of dev dash products.

We can make that assumption about our product's HTML file

because the team who is working on this sub-project

has total control over this file and they can

make this ID right here anything they want it to be.

However, the products team is not the container team.

The container team is a separate engineering team in theory.

And the container team might decide to have

an HTML file that does not have a div inside

of here with an ID of dev dash products.

They might have instead some kind of ID of products.

Or how about my dash products?

Let's do that.

So my dash products here.

So the products team cannot really control the

structure of the container HTML file.

They can't 100% ensure or guarantee that

some div exists, or some element I should say

with an ID of dev dash products.

So the code that we have inside of our products,

bootstrap dot JS file specifically, that segment

right there is not too great.

It's not very good because it assumes that some element

exists when the element might not exist at all.

And our products team can't really guarantee that

that element will exist.

With that in mind, we're gonna refactor this

bootstrap dot JS file and better take into account

the fact that this element might not exist at all.

So I wanna first begin by writing out a couple of

comments just to help you understand

the refactor we're going to do.

We're gonna make two major changes, or really 3 1/2,

or 2 1/2 maybe, major changes to this file.

So the first comment I'm gonna put in here,

I wanna think about one way in which we are going

to execute this bootstrap dot JS file.

The first way, and I might call this

like context or situation or whatever number one,

that we're gonna execute this file in is

when we are

running this file

in development

in isolation.

So, in that scenario, we are using our

local index HTML file,

which definitely

has an element with an ID of dev dash products.

So, in this context, or this scenario right here,

situation number one we probably want to

immediately render our app into that element.

There's no reason to delay.

We're actively developing this project.

We want to see this HTML appear

on the screen because we're actively developing it

and trying to tweak how this application works.

So, that is context, or situation number one.

So, let's now think about situation number two.

So, context, or situation number two,

whatever you wanna call it.

In this scenario,

we are

running this file

in development

or production

through the container app

and there is no guarantee

that an

element with an ID of

dev products

exists.

So in this scenario, we do not want to

try to immediately render the app.

We don't wanna do that

because that element

that we're looking for might not exist.

And if we try to render something

into an element that doesn't exist,

we're just gonna end up with an

error message very quickly.

So, if I could get the spelling of

'immediately' good enough.

So, we need to write out some code

to handle both of these different situations.

We need something to handle context or situation number one

and something for situation number two.

So. let's kind of think about how we can

write out some code to meet or match

both these situations and let's write out

some implementation for these in the next video.

Topic 26. Refactoring Products

Instructor: Let's write out some code

to satisfy Situation #1 and #2.

To get started, we're gonna refactor some of the code

at the top of this file first.

I'm going to define a new function

that I'm going to call mount.

There's nothing special about the name mount whatsoever.

It is just an arbitrarily decided upon name.

The goal of the mount function

is to take in a reference to an HTML element.

So we're gonna receive some argument right here

that we'll refer to as el.

So, again, this is an HTML element.

Then inside the mount function,

we're gonna do everything required to start up our app

and then, eventually, produce some HTML

and render it or display it

inside of that element right there.

So to make that happen inside of our case,

I'm going to take everything from products

down to about line 10 right there.

I'm gonna cut all that

and paste it inside of mount.

Then rather than rendering our application into this element

that we're trying to find on the fly,

we're going to instead render our application

into that element right there.

I'll do el.innerHTML, like so.

So now we've got a function that we can call

in either situation, #1 or #2.

And just so you know,

the approach of using this mount function right here

can be used we many other frameworks as well.

For example, if you're making use of, say, React,

rather than doing this last line right here,

we could do something like, say, ReactDOM.render

and then maybe some kind of App,

and we want to render it into that element

that is being provided.

So the idea of some mount function like this

is going to ultimately work

with many different frameworks we might decide to use.

Now how are we going to use this with Situation #1

and Situation #2.

Well, in Situation #2,

there's something I wanna point out here.

Remember, we don't have any guarantee

that some element exists on the screen

or inside of our HTML page

that we can definitely render into.

In addition, we probably do not want

to try to immediately render the app.

If we are showing our container,

it's really up to the container

to decide how, when, and where our products application

should be displayed on the screen.

The container might only want to display

our list of products

if the user goes to some very particular URL

or if they click on some button.

So, in context, for Situation #2,

rather than trying to call the mount function immediately

or anything like that,

we are going to instead export the mount function.

By exporting the mount function,

now our container can import the mount function

and make use of it whenever it wants to.

So now it's up to the container

to decide when to show our products application

on the screen.

Now we do have to go over to the container

and make sure we now import the mount function

and eventually call it.

But before we do,

let's go ahead and take care of Context or Situation #1,

the situation in which we are running this application

in isolation.

The first thing we have to do, in this case,

is decide whether or not we are in isolation.

For that, there are two checks

that we're going to implement.

The first check we're gonna make

is ensure that we are running this project in develop mode.

So for that, we're gonna add in an if statement,

and we're gonna say

if process.env.NODE\_ENV === 'development'.

This is an environment variable that gets set automatically

by Webpack.

That environment variable gets set,

because inside of our webpack.config.js file,

we had added in a mode of development.

So because we put in that option right there,

webpack is going to automatically replace this

with development.

So if we are running our application in development mode,

well, we're gonna go into this if statement.

Then, once we are inside of here,

we need to do one further check.

Remember, we can be in Situation #2

and still be in development mode.

So, in other words, if we are running

our products application through our container

in development mode,

we're still gonna satisfy this check.

So now we need a second check inside of here to decide

whether or not we are running products in isolation.

This check is a little bit harder to implement,

because, well, how do you decide

whether or not you're running products by itself?

One very easy way we can do this

is just put in some kind of very unique HTML element

into to products html document.

And that's why we put in this div right here

with an id of dev-products.

That thought here is that only our products HTML file

is gonna have a div with that id.

So if there's some element inside of our HTML document

with that id right there,

we are just going to assume

that we are running products in isolation.

In other words, we are assuming this HTML file

has that element,

and we are assuming that our public HTML file does not.

Now that's not a very great assumption,

because, of course, our container,

or the engineering team in charge of the container,

they could very easily decide

that they want to have some kind of element inside of here

with an id of dev-products,

so it's not a great assumption,

but I'm sure you can probably think of some way

to make a very unique identifier

that is probably not gonna show up

inside the container app anytime soon.

So you could change this from, say, dev-products

to dev-products-dev-only or something like that,

something to just guarantee

that we're only gonna have an element like this

inside of our products HTML file

and probably never inside of our container.

Okay, so I'm gonna stick with dev-products.

So inside this if statement,

I'm gonna do a el = document.querySelector

for #dev-products.

And I'll say if we found an element

then let's call our mount function

and provide that element, try to render our app into it.

So I'm gonna put inside of here, or right above,

just a comment and say, "Assuming our container

doesn't have an element with id 'dev-products'...."

That's, again, a really big assumption,

but, honestly, well,

we're probably not gonna have an id like that

inside our container.

So if we get into this if statement right here,

that means we are probably running in isolation,

so just running products by itself,

and we're executing products through our container app.

Okay, so we are ready to test out products by itself

in isolation and make sure

that our application still appears on the screen.

So let's try that really quickly.

I have my browser, I'll go to localhost:8081.

And yep, looks like our app still appears.

So now we just need to make a quick change

to our container app.

We need to make sure that the container

now tries to import this mount function

and eventually calls that mount function

with wherever the container wants to show the product list

on the screen.

So let's take care of that refactor in the next video.

Topic 27. Consuming Remote Modules

-: Our products application is now exporting this

mount function.

So now inside of our container

inside the container's bootstrap.js file

we need to make sure that we import the mount function

and then eventually call it.

So inside of the container source bootstrap.js

we're gonna update the products import statement

to be { mount } from 'products/ProductsIndex'.

So now that we've got this mount function

we can choose to call it

at any point in time that we want

and we can display our product list app

at anywhere on the screen.

So we could only decide to show the product list

when the user is at some particular URL

or maybe when a user clicks

on a button or any other kind of scenario.

So for right now, well we're not really

in any of those kind of scenarios.

We really just want to show the product list no matter what.

So we are going to immediately call mount

and we're going to make sure that we provide a reference

to some HTML element that exists inside of the container

so that our product list knows where to display itself.

So inside of our container index HTML file

remember I change that ID right there just a moment ago

to my-products.

So I'm gonna do a selector and get that element

and pass it off to the mount function.

Like so.

Now before we test this out

there's one little gotcha inside of here.

Remember we are trying to import

products/ProductsIndex.

So that is gonna map

up to our containers, webpack.config file.

We asked for something from the products module,

so that's gonna cause our webpack

module Federation plugin to go

and try to load up that remote entry file.

That's gonna take us

into our products webpack.config file.

Inside of here, we had said if anyone

tries to import

the products index,

then instead give them the index file.

Now just one little issue,

our index.js file is not what is

actually exporting that mount function.

It's actually the bootstrap.js file.

So now inside of our products webpack.config file,

rather than giving someone source index,

if they ask for products index,

we actually want to give them the bootstrap

file instead,

which does export the mount function.

So just a little gotcha there.

Okay, so I'm gonna make that change and save this.

We then need to very quickly

restart our products application

because just made a change to its

webpack.config file.

So here's products.

I'm just gonna restart it very quickly.

And now I think we are ready for a test.

So, back inside my browser, I should be able to go

to localhost 8081 and still confirm that everything works

as expected.

Yep, looks good.

And I should be able to go to localhost 8080

refresh the page and still see my product list appear

on the screen.

Awesome.

Okay, so the pattern that we now have here

specifically inside of our products bootstrap.js file,

this is how we are going to export stuff

from our application in general.

We're gonna have some kind of

central point inside of our project

that's gonna define a mount function.

The mount function is gonna be the true startup location

of our project.

If we call mount

then we're going to create our React application

or our Angular app or our View app or whatever else

and we're gonna try to render our application

into that element.

Then lower on down the file

we're gonna have these different set of checks.

We're gonna see if we are running this project in isolation.

If we are,

we're going to attempt to render our app right away

into some element.

That probably only exists inside

of our local project's HTML file.

Otherwise, if we are trying to run our project

through some other kind of means,

in other words through our container,

we're just gonna rely upon the export.

Now just so you know

the export always happens no matter what

whether we are in development or whatever

but this just means we're going to make use

of this export statement.

Okay, So this looks good.

Let's take a quick pause right here.

We're gonna repeat this exact same process

for our cart application as well.

That'll be really quick

since we now know exactly what we need to do.

Formularbeginn

Topic 28. Refactoring Cart

-: All right, let's repeat these exact same refactor inside

of our cart application.

So to get started, I'm gonna find my cart.

I'm gonna find the SRC directory, bootstrap.js.

At the top right underneath our import statement

I'll add in const mount.

We're gonna receive some kind of

HTML element as an argument.

I'll then cut and paste everything down there

into the mount function and then rather try then

trying to render our application into some cart-dev element.

We're going to instead render it into that provided element.

Then after that we'll put

in a check to see if we are running this project

in development and in isolation

we'll do a process env, node env

of development.

Then a el will try to find some element inside

of our local HTML file with an ID of dev-cart.

And you know what?

I think we actually call this one cart-dev.

Yeah. Cart-dev.

This one is actually cart-dev.

So then if that element does exist, let's call mount.

Okay. After that we will export our mount function

and that should be it.

So again, the export is gonna be used

by our container application.

Before we go over to our container

we do have to update our webpack config file.

We need to make sure

that anytime someone tries to get at cartshow

we're not gonna give them the index.js file.

Instead we're going to give them our bootstrap js file.

So bootstrap.

After saving that web pack file, we'll then go

and restart our web pack process for the cart very quickly.

Once again.

So back at my terminal, here's my terminal window

that is running my cart.

I'll restart webpack

and then back inside of our container.

I'm gonna first go into the containers index.html file.

I'm gonna make sure that this html file does not

have a cart-dev element anymore.

I'll change this to my cart cuz remember if it

does have a dev-cart or cart-dev, whatever it was

then the cart is gonna try to render itself immediately.

And we don't want that.

We want the container to control when the cart renders.

Then finally, inside of our container SRC bootstrap file

we can import mount from cartshow

and call mount

with document

query selector

my cart.

And that should be it.

So I'm gonna save this, go back over to my browser.

I'll do a refresh.

Duplicate variable names here.

My mistake, little change we need to make.

We've got two functions called mount now,

all we need to do is just rename one of them

or both them on the fly.

So I'm going to rename the products index mount as

products mount like so

and as cart mount.

Then the first one will be products mount instead.

All we're doing is renaming that import

so that they are not colliding and cart mount much better.

Okay, so now if we test once again

we see our application appear

and I should also be able to go

to local host 8082 and see my cart in isolation as well.

Perfect.

So now we can develop all these different applications

in isolation if we want.

Here's local host 8082.

Here's 8081, which is products, and here's the container

which is loading up both applications.

So let's say this looks pretty good.

All right, well that's pretty much it

for this first application, but there's still a lot

for us to learn about micro front ends.

So let's take a pause right here.

Gonna do a wrap up on this application and

continue on to our next one and just a little bit.

Topic 29. [Optional] A Funny Gotcha

-: This is an optional video.

I just wanna show you

a little bit of behind the scenes stuff

around the Module Federation plugin.

In particular, I wanna show you one bug

that's a total gotcha.

So, if you are not interested,

if you just wanna move on to the next application,

pause this video right now and continue on.

Otherwise, stick around.

So, I'm gonna first get started

inside of my cart webpack.config.js file.

You may recall that we briefly discussed

this name property right here

which we had listed as cart.

And when we discussed it,

I had told you that this cart right here

has to match up with our container,

webpack config file,

specifically cart at part of that string right there.

Cart right here has to match up with cart right there.

I just wanna tell you a little bit more

about why that is

and go into one very small bug

that you might run into

that's really, really challenging to troubleshoot.

Okay.

So, first off, back inside my browser,

I'm gonna go to localhost:8080.

And once here,

I'm gonna again open up my network request tab

and I'm gonna take a look at the remote entry file

that is coming in from localhost:8082,

which is, you'll recall, our cart application.

So, here's localhost:8082 remoteEntry.js.

I'm gonna click on that.

And if I take a look at either preview or response,

I want you to notice something here

on around line nine or so.

You'll notice it says var cart.

So, it declares a variable

and then it assigns a value to that variable.

And the value it assigns

is basically the result of all of this Webpack stuff.

So, the issue here is what is this var cart thing?

Well, it turns out that when Webpack inside the browser

loads up this remoteEntry.js file,

it needs some way of accessing the code

inside of remoteEntry.

It needs some way of reaching in there and saying,

Aha, this is the code or the configuration

that I am being provided.

This cart variable is how that communication is done.

Whenever our browser

or Webpack running inside the browser

loads up that remote entry file,

it's going to look for a cart variable inside there

and it's going to try to access that variable

to get all the information contained inside

of this remote entry file.

So, this cart variable

is really being accessed and set

on both sides of the equation.

First off, whenever our container webpack.config file,

whenever our container is trying to access

that remote entry file,

it's going to try to look inside there

for a variable called cart

and that variable name is set by that value right there.

If we change that cart right there to anything else

like, let's say, how about "my cart"

and then save this

and then restart our container Webpack process.

I'm gonna restart the container very quickly here.

If I then go back over to my browser

and refresh the application,

we're gonna very quickly see an error.

You'll notice inside of here

we're seeing some errors.

Load script failure, blah blah blah, all this stuff,

because now our container app

which is running inside the browser

is trying to look inside that remote entry file

and find a variable called my cart instead of just cart.

But the remote entry file that is being served up

from our cart application

still has a variable inside there of cart.

So, if you wanted to fix this,

we could obviously revert the change we just made.

Or alternatively, we could go into our cart project,

find the Webpack config inside there

and then change that name to 'mycart' as well.

So now, if I say this

and also restart my cart application.

We're gonna restart cart.

Now, both sides of the equation

are kind of matched up, so to speak.

The container is looking for a variable called mycart

and the cart is providing a variable called mycart.

So, now if I refresh this, everything works as expected.

And now inside that remote entry file,

I can once again see my cart.

Now, I mentioned a bug around all this stuff.

Where's the bug?

What's the kind of issue here?

Well, let's make a very small change.

First off, I'm gonna revert the two my cart things

really quickly.

So, inside of cart webpack config,

I'll change the name back to cart and save the file

and back inside of container webpack config,

I'm gonna find that string right there,

change mycart back to cart as well.

And then I'll restart once again

both those webpack processes.

So, here's container

and here's cart.

I'm gonna very quickly just verify

everything is still working as expected.

Okay. So, where's the bug?

Well, let's go back over to our editor.

Inside of my container,

I'm gonna open up the index.html file

and you'll recall that we put in a div right here

with an id of my-cart.

I'm gonna very quickly just change that to cart by itself.

Very important that we put just cart.

I'm gonna save that file.

I'm also gonna go to my bootstrap.js file.

You'll recall that is where

we are trying to select that element.

And I'll just update that selector to pound cart,

just to say, Hey look,

we are trying to select the correct element.

So, in theory, everything should work just fine.

All we did was change the ID of an element

but we did go and update the selector as well.

So, I honestly would fully expect

everything to continue working as expected.

But as you guess, well, it doesn't.

If we flip back over to localhost:8080,

we now get a nasty error message

"fn is not a function."

So, what in the world is going on here?

Why would changing an ID on an HTML element

somehow break our application?

Well, the answer is pretty straightforward.

First, I'm going to undo that ID change very quickly.

So, I'll undo the change in bootstrap

and inside of index html.

So, I'm back to, in theory, everything working.

So, everything works again in the browser.

And now inside my console,

I'm gonna try to print out the cart variable.

This cart variable is being created

by our remoteEntry file

coming from our cart application.

That same remote entry file we were just looking at.

This right here is being declared as a global variable.

So, we can very easily print it out at our console.

If we print it out and then open this thing up,

you'll see that it's an object that has a GIT function

and in a knit function.

This is essentially how we interface

with a code coming from that remote entry file

of the cart application.

So, essentially we have an object here

that allows us to get access

to all the code we're looking to load into our container.

So, now that we've established

that there's a global variable, called cart.

Let's now go back and change that ID once again.

So, I'm gonna go back to just cart

and I'll change it inside of bootstrap.js as well.

Save both those files.

Now, here's the error message again.

And now this time, let's print out cart again

and see what happens.

Now when we try to print out that global variable,

we instead get a reference to that div.

So, here is the bug. The big bug.

I shouldn't even say bug for say,

but something that's just really, really misleading.

If you ever assign an ID to an element,

your browser is gonna try to create a new global variable

with the same exact name as that id.

So, because we gave this div an ID of cart,

your browser is gonna try to create

a global variable called cart.

That global variable is going to override

the cart global variable

that gets defined inside of this remote entry file.

I'm not looking at products, but you get the same idea.

It's gonna try to override the global variable

that was defined inside of our cart remote entry file.

So eventually, when Webpack inside the browser

tries to access cart instead of getting our code

coming from the cart application,

it instead gets an HTML element.

And that's why we end up with this error right here,

fn is not a function.

So, that's a pretty weird bug.

You kinda have to connect the dots here.

Understand that the ID is going to be assigned

as a global variable.

And you also have to understand that the remote entry file

coming from our cart application

is gonna try to declare a global variable as well

and the two are gonna conflict.

So, to fix this, all we have to do

is make sure that we don't have some element

inside of our DOM

with an ID equal to whatever is coming across

inside of that remote entry file.

So, in other words, we don't wanna have an ID

with a same name as that thing right there.

Now, that might sound like not that great.

A very common class name

that you might see in a lot of different projects

is about card.

That's a very common class name.

So, if you ever have a sub project that is named card,

you might want to put a name of card right here.

And then if you end up with say,

rather than a class name and ID of card

on some HTML element,

well, all of a sudden, everything is gonna break.

Don't worry, we're gonna fix this whole situation up

in a little bit

and ensure that we probably have some module names

that are not gonna conflict with any elements

that we are rendering or trying to create

inside of our HTML document.

Okay. So, I apologize for the length of this video.

I just thought this was a very interesting little bug

to run into.

So, really quickly,

I'm gonna go back to my bootstrap JS file

inside of container.

I'm gonna restore the ID of my-cart right there

and inside of my HTML document,

I'll change that back to my-cart as well.

And now, if I load up inside the browser,

once again, everything works as expected.

Okay.

So, let's pause right here

and move on to our next application

in just a moment.

Section 4: Linking Multiple Apps Together

Topic 30. Application Overview

Steven: Our first application turned out really well

but there's still a tremendous amount for us to learn.

For example, we need to handle, say, navigation.

We need to be able to work

with multiple different frameworks,

such as React or View and so on.

We need to also think very deeply

about stuff like deployment as well.

So in this video, we're gonna start

to kick off our next application,

which is going to be way, way more complex

than the one we just put together.

Let's first begin by taking a look at a couple of mockups

of what we're going to try to build.

Okay, so we're going to imagine that we are building

a kind of startup

or a kind of software-as-a-service application.

So we're going to be building a landing page.

We're gonna be building stuff like some authentication

and then eventually some kind of internal feature

that a user can only get to after they sign up.

Now, I wanna be extremely clear from the get go right now,

this application doesn't actually do anything.

There's no actual database.

There's no actual remote server or API.

There's no real authentication system.

Just about all the elements that you're gonna see here

in these different mockups are gonna be like dummy elements.

A user can click on them and obviously see them

but they're not really gonna do anything per se

or present any real data.

We are much more interested in just how we make use

of all these different frameworks,

how we wire them together and stuff like that.

That's what we are focused on for this application.

Okay, so the first thing that we're going to put together

for a user to see is a landing page of sorts.

So a user is gonna come to our application at say, app.com

and they'll see some kind of marketing landing page

where we will tell them, hey, welcome,

come and take a look at our pricing

and eventually sign up for our application.

If a user clicks on this see pricing button,

we'll then take them to a pricing page

where a user can take a look

at all the different prices that we offer.

Now, again, there is no actual payments

or anything like that.

This is all just absolutely fake data.

But the point is there's gonna be

some kind of pricing page a user can go to.

Now we'll show this to a user whenever they go to /pricing.

Eventually, in theory, a user is going to either

try to sign up or maybe they're gonna try to log in

using that button inside the header.

So whenever user signs up or logs in,

we're gonna take them to some kind of authentication page.

So a page to sign in or alternatively sign up.

Then whenever a user clicks on this

sign in or sign up button, we're then gonna take them

to the primary feature of our application,

which is gonna be some kind of admin dashboard of sorts.

So on this page, we're going to eventually show

a bunch of different pretty charts and tables

and stuff like that.

Again, all the data is gonna be 100% fake.

Not gonna interact with it in any way,

but we are gonna show some just pretty stuff

to our users and whatnot.

Okay, so that's what we're gonna be building.

Now, the first thing I want to think about

is what we might call each of these different pages.

I wanna think about what the goal of each page is

and just assign a name to them,

so that you and I can easily refer to them in the future.

So as I start to look at these different pages,

I think it's pretty easy to come up

with some different names for each of them.

First off, this main landing page,

we could call it say, landing page or homepage.

Either one works totally fine.

We definitely have some kind of pricing page right here.

This could also be a plans page or something like that.

Either one's totally fine.

We then have a sign in and a sign up,

and then finally some kind of dashboard page over here.

Now, once we assign names to each of these things,

I want you to start to think about how you would group

each of these different pages together

in terms of functionality and what they do

and their overall goal and purpose.

So outta these five different pages,

how might you group these together?

If you had to say,

hey, this page is really similar to this page.

Well, I think that a pretty obvious way to go about this

might be something like this right here.

I think the homepage and the pricing page are really

about marketing or user sign up or stuff like that,

trying to get people to sign up to our application.

The actual sign in and sign up pages

are really all about authentication

and those are clearly very tightly coupled pages,

they really do just about the same exact thing.

And then finally, this dashboard page down here

is really its own standalone thing.

We could call that easily dashboard, or maybe,

if we knew exactly what kind of data we were showing,

we could think of a better name for it than just dashboard.

But I think for right now, just a overall category

of calling it dashboard will work just fine.

So overall, I think that we have five different pages

and I think that we have

three distinct, different areas of functionality.

So we might decide to create

three different engineering teams.

Maybe one team will be in charge

of the marketing aspects of our product.

Maybe another will be in charge of authentication

and then another in charge of the actual dashboard

or real functionality of our product.

Now, when I say that, you might say,

Steven, we don't need a whole team of engineers

in charge of just authentication.

Okay, totally understand.

Nonetheless, we're just gonna pretend

that we need a full team of engineers

to knock out this whole authentication piece.

Okay, so at this point,

I think that we've laid out the overall project,

we have an understanding

that we're gonna be building five different pages

and I think that there's three distinct areas

of functionality inside of our product

that we need to think about

and we might want to have a dedicated team

to each of those three distinct areas.

So with this in mind,

let's do just a little bit more planning in the next video.

Topic 31. Tech Stack

Instructor: We've now established that we're gonna have

three major areas of our application,

marketing, authentication, and the dashboard.

So we're gonna create three different micro front ends,

one for marketing, one for authentication,

and one for dashboard.

In addition to these three different micro front ends,

we're still going to have a container application

that is going to coordinate these together

and decide which micro front end to show on the screen

at any given time.

For each of these rather than writing them out

with plain JavaScript code

as we did on our first application,

we're going to instead use some front end frameworks.

So here are the different frameworks we're going to use.

For the container marketing and authentication,

we're going to be making use of React.

And for the dashboard, we're going to make use of Vue.

Now, as soon as I show you this,

you might say, hey Steven, wait a minute.

I want to use Angular or I wanna use Svelte

or WebComponents or something else.

Well, I gotta be totally honest with you.

We are going to spend not a lot of time trying

to understand how to get Micro Frontend Frameworks to work

with these different front end libraries like React or VUE.

Instead, we're going to focus on understanding

how to create different integration techniques

where we don't have to worry about

the implementation details

of React or VUE or WebComponents or anything like that.

That's the real goal here.

We want to create Mirco fronted apps that can be substituted

in for each other without having to worry

about having some really precise coordination

between say the container and authentication.

In other words, when we try to show authentication

inside the container,

even though they are both React applications,

we are not gonna show authentication

as a plain component inside of container.

Instead, we're gonna have a very generic coupling method

which guarantees that we could eventually substitute

out this React version of authentication

with some other framework in the future.

Like maybe we want to convert it to be VUE

or Angular or whatever else,

and we would want to make sure

that we could do that conversion

without having to do a major rewrite

of the container or anything like that.

So again, we're gonna focus on having

very generic interface points.

Okay, so now that we understand that,

I just wanna give you one more very brief discussion

in the next video,

and just lay out some really big requirements of our project

and really understand what we are trying to accomplish here.

Topic 32. Requirements That Drive Architecture Choices

Steven: This video is gonna be a little bit long

so I apologize in advance,

but it is a very important video,

lemme tell you why.

Okay, so as we start to build out this application

we're gonna make some very distinct architectural decisions

and some of the code and techniques I'm gonna show you

are gonna be a little bit different

than what some different blog posts, articles,

and so on out there are going to tell you to do

when it comes to microfrontends.

Now that's not to say

that I'm telling you to do stuff

that is just wildly different than accepted practices.

Instead, the real situation here,

the reality of what's going on around microfrontends

is that there is a wide spectrum,

or wide variety of different techniques you can use

and some blog posts and some articles

are gonna be just in direct disagreement

with what the best practices are.

So for this application I sat down and said,

here are the hard requirements.

Here is how this application must behave.

And I used those requirements to come up

with a set of architectural decisions.

So in this video I'm gonna tell you

about some of the requirements of our app

and you're gonna very quickly understand

how those requirements are driving the exact technique

of how we are putting everything together.

The one thing I wanna do, just to be really clear here,

is tell you exactly what some techniques are

that we're gonna do

versus what some other people are gonna say.

So some things you might see

in some different blog posts and whatnot,

you might see stuff that says,

"Oh yeah, you should always share state

"between applications using Redux."

You might see stuff that says,

"The container app must be built with web components,"

and some other stuff like that.

Now none of these things are necessarily wrong, per se,

they just have different sets of requirements

than what our requirements are.

Once I show you our requirements

I think you're going to agree

that we have a very reasonable set of requirements

that very closely mesh with the overall goals

of microfrontends.

So even though some people will be

in a little bit disagreement

with some of the techniques we are using,

I personally think this is a very solid foundation,

a very solid architecture,

that we're gonna build out here.

Okay, so here are our requirements.

So we've got about five of them

and each of them I've labeled as being inflexible.

So that means we must, under all scenarios,

satisfy these requirements,

and again, these different requirements

are gonna drive our overall architecture.

So our first big requirement is that we must have

zero coupling between child projects.

So that means that we cannot do any imports of, say,

functions, objects, classes, anything

between our different child projects.

We can't have any shared state.

No Redux stores, no reducers, no context.

Nothing like that can be shared between child projects.

The one exception are shared libraries

through the module federation system.

And the reason for that

is that it's absolutely okay if we share a library

because if some other project falls away

or decides not to use that library anymore

our first project that might use that library

is still going to get access to it

through the module federation system.

So why is this? Why do we have this requirement at all?

Well, let me lay out a real quick scenario for you here.

So let's imagine we've got our container,

marketing and authentication

and maybe marketing and authentication

are both built using React.

Now let's imagine, for just a moment,

that we build these two sub-projects

with a very high degree of coupling.

Maybe they both make use of Redux

and they try to share the same store.

Maybe they expect to receive actions from each other.

Maybe they try to share React components or whatever it is.

At the end of the day,

let's imagine we have very tight coupling

between these two projects.

Now let's kind of think forward like five or 10 years.

Let's say that in five or 10 years we are still working

on the same overall project

and maybe in five or 10 years we decide

that instead of using React for marketing

maybe we decide to completely rewrite our marketing project

and we're gonna use some brand new framework

that just came out called Flubber for some reason.

Maybe it's really great for marketing stuff.

Well, in five or 10 years

if we decide to go and replace marketing,

that would kind of imply that we would have to make

some serious updates to our authentication system as well,

'cause like I just said,

there's very tight coupling between these two projects.

Maybe authentication just doesn't work correctly

unless marketing is loaded up inside the browser.

So if we have any coupling like that

and we try to change marketing,

we might be easily breaking, 100%,

our authentication application.

So the less we couple these things together

the less we have to worry about making changes

to one project and breaking another by mistake.

Now this entire requirement might not make a ton of sense.

You might say, "Hey Steven, surely we can share

"React components or something like that.

"We could always just like copy paste

"those shared components from marketing

"over to authentication."

Well, if you've never worked at a large company

where you are working on software systems

that were written many years ago,

let me tell you I very much have,

and something that you really need to understand

if you have not been in that scenario,

is that at some point in time

we are all gonna forget how React works.

At some point in time, in some number of years,

React is gonna be an old ancient framework

that nobody knows how to use

or maybe nobody wants to use in general.

So the more that you can isolate your different applications

the better off you're gonna be in some number of years

'cause it means that you're going to be able to replace

or make updates to different part of your application

without worrying about breaking some other part

that you might not want to touch at all

or might not even know how to touch.

You might not know how to even start up a React project

in five or 10 years, and trust me,

that is entirely possible.

Okay, so that is hard requirement number one.

Zero coupling. Just none, no sharing of anything whatsoever.

So, hard requirement number two.

Here we go, we're gonna have almost zero coupling

between our container and child applications.

So our container is gonna be able to have

some amount of communication with the child app.

Having said that, the container will not assume

that the child is using any particular framework.

So even though we're starting off with the container

using React and two of our sub-projects using React,

we're not gonna communicate between them

in any necessarily React specific way.

Instead, we're gonna try to make that interface

as generic as possible.

Any communication that we do have to do

is not gonna use any fancy framework,

like Redux or anything like that,

instead we're gonna use very simple and basic systems,

like callbacks or very simple event structures.

That's it.

That means that we can have some communication

between our container and let's say,

the marketing application or the authentication application.

These two things can communicate just fine

but we don't wanna have communication

between marketing and authentication.

So why the divide here?

Well, it really comes down to just practicality.

At some point in time, the authentication system

for our application absolutely just does have to communicate

up to the container in some way.

Lemme tell you why that is.

If we think back to our mock up over here,

here's our sign in and sign up,

we've got a header on every single page.

This header is gonna be rendered by our container.

Eventually a user is gonna sign in

and whenever they do we're gonna wanna flip

that login button right there to say log out instead.

So we need to make sure that whenever a user

successfully logs in through this sign in system,

or the authentication system,

we need to somehow tell the container

that the user is now considered to be authenticated

and that we should change the text of that button.

So we are going to allow some level of communication

between a child app and the parent.

We technically can have ways where we even avoid

this level of communication,

and I'll tell you what those are,

but we're gonna say that we kind of assume,

yeah we need to have at least a tiny bit of communication

between a child and a parent.

Okay, requirement number three.

CSS from one project should never affect another.

What this really means is that all of our styling,

all of our CSS, should be a hundred percent scoped.

So any styling that we add to, say, the marketing page

should never under any scenario

affect something on, say, the dashboard.

The reason for this is very simple.

If we start changing the CSS on the marketing page

I don't want to just magically find out a week later

that we accidentally broke the display of the dashboard

at some point in time.

This goes back to the entire same idea

of making changes to one project shouldn't break another.

So very simple, we don't want to have any CSS

kinda shared or accidentally affecting

some other part of our app.

We're gonna have to figure out some techniques

and styles to scope all of our CSS.

Okay, number four.

So with number four, we are going to eventually make use

of version control as we start to deploy our application.

I don't want our style of version control to have any impact

on how we develop our project

or how we deploy it or anything like that.

Obviously, if we use, say, a monorepo

where we're gonna have all of our projects

inside of a single repository

versus separate repositories for each individual project,

obviously we are gonna have

some level of difference between the two.

It kinda comes down to having to do a single git checkout

versus, say, 10 git checkouts, but beyond that,

I don't want to have a major impact

on how we develop our project,

how we deploy it or anything like that.

Some people out there want to use monorepos.

If you do, totally fine.

Other people want to have a separate repo

for each individual project

because they say, "Hey these are microfrontends,

"they should be standalone."

Either way, it should just plain not make a difference.

For our project, we are going to use a monorepo,

but at the end of the day,

if you want to use separate repos

everything will work 100% okay if you want to do so,

it just requires a lot of additional setup

because you now have to create four separate repositories

on GitHub, which is what we're gonna use,

as opposed to one repository.

Okay, very last thing.

This one is a little bit challenging to understand

but we're gonna spend a lot of time

to really understand this requirement

and why it exists at all.

This really comes down to

the production deployment of our project.

So that's where this requirement

is really gonna come into play.

So our container should have the ability

to decide to use either the latest version

of a microfrontend

or a specific version of a microfrontend.

So in other words, at some point in time,

you and I are going to, say, make some changes to marketing

and we might eventually deploy it

and then realize, oh we need to add in some new section

or something like that, we'll make some changes

and redeploy the project.

Every time we do a deploy we're kind of creating

a different version of our marketing project

and I want to allow our container

to decide either to always use

a very specific version of marketing

or just use the latest version.

Now, whether you would decide to use one or another

really comes down to your requirements as a company.

Some companies say, "I want to be able to pin a version

"of something like marketing

"and always use that same version

"and only do updates at very specific points in time."

That's pretty much where this requirement is coming from.

That's our last big requirement, and again,

we're only gonna really see this start to come up

as we start to think about deployment

and some stuff like that.

Okay, so that is it.

I apologize again for the long video, but again,

these requirements are 100% driving our architecture choices

along every single step of the way.

We're gonna follow each of these requirements

as best as we can, the only exception

is for requirement number two right here.

On requirement number two,

we're gonna bend this one just a little bit

when it comes to handling navigation

and I'll tell you exactly how we are bending this rule,

it's gonna be very obvious,

I'm gonna very clearly outline it

and I'm gonna tell you how we can not bend it

if you want to make sure that you have true separation

between the container and the child applications.

Okay, so that's it.

Let's pause right here and get started

on our application in the next video.

Topic 33. Dependency Files

**Dependency Files**

Hi!

Our project requires a lot of dependencies across several different projects. To speed up the install process and to make sure we're using the same version of these dependencies, you'll find a file attached to this lecture called 'mfp.zip'.

**Please download this file and extract its contents - we will use the contents in the next video**

**Important** - Please make sure that you are using only the **current Node LTS**. The newest version of Node and older versions are not supported.

***Note****- We have provided an updated version of the completed application that uses React 19, React Router 6, and MUI 5 at the very end of the course:*

<https://www.udemy.com/course/microfrontend-course/learn/lecture/51373675>

Resources for this lecture

Topic 34. Dependency Installation

Instructor: In the last video,

you should have found a zip file attached to a lecture,

called something like mfp.zip.

Inside of here is a collection

of different package.json files.

Each of them list outs

some the different dependencies

that we're going to need

for each of our different sub projects.

We're going to extract the zip file,

that's going to create a new folder called mfp,

and then inside there,

we're gonna find, once again,

each of these different project directories

with a package.json file inside them.

So we're going to open up our terminal

and go into each of these different project folders

and run npm install

to get all the different listed dependencies.

So at my terminal, I'll change into this mfp folder,

inside there is a directory called packages,

so I'll change into there,

and then once again,

we're going to go into each of these different directories

and do an npm install.

So I will first go into auth, and there we go.

All right, so we're gonna take a pause right here,

I'm gonna let you go through the three other sub-projects

and run npm install inside of each of them.

After you do that,

we'll come back in the next video

and go through a little bit more setup.

Topic 35. Initial Webpack Config

Instructor: Hopefully you had

some success installing dependencies

for all four different sub-projects.

Between the last video and this one,

I installed all dependencies as well.

And then I opened up this MFP directory inside

my code editor.

So I can now see that there is a packages folder,

and inside there are all four different sub-projects.

Inside of each of these directories,

you should see a package JSON file,

a package lock and a node modules directory,

if you installed dependencies for each of them correctly.

Now we're going to start to wire up

all these different projects from scratch.

So we're gonna put together

some very low level configuration,

for Webpack and stuff like that.

You might be a little bit curious,

why we are setting everything up from scratch

as opposed to using something like say

Create-React-App or Vue CLI to generate these projects.

Well, there's two reasons,

that we are not making use of Create-React-App or Vue CLI.

The first reason is that Create React App

and Vue CLI both currently are shipping

with a version of Webpack

that doesn't have support for the module federation plugin.

Now by the time you watch this video,

that might change,

Create-React-App might have support for a version of Webpack

that will or can make use of the module federation plugin

but that's just not what reality is right now.

The other reason we are not making use

of either these project generators,

is that we are going to be making use

of incredible number of different dependencies.

And all these different dependencies are changing over time.

Chances are by the time you watch this video,

at some point in time in the future,

there might be different versions,

different major change versions

of all these different dependencies.

And so if you just use the latest version,

of all these different things,

well you might get some unexpected behavior,

stuff is probably gonna break.

So I really gotta ask you,

please don't make use of Create-React-App

or Vue CLI in place of any of the projects,

or the starter package.JSON files I just gave you.

If you use these package JSON files that I gave you,

for all the different dependencies,

everything is gonna work just fine.

But if you start replacing stuff,

well chances are things are going to break.

Okay, so enough said around that.

So what are we going to work on first?

Well, we're going to go through our marketing project first,

and we're gonna start to set up

some Webpack configuration for it.

For our marketing project,

we're going to try to get it running in isolation.

So again, just by itself,

without worrying about the container

or anything like that.

As we start to set up

our Webpack configuration for marketing,

we're going to create

a couple of different files for Webpack,

as opposed to the one single file

that we had created previously.

Remember, this project we are going to eventually deploy it

in production on a live website.

So we need to run Webpack eventually in two different modes.

We need to run Webpack in development mode

when we are running or developing on our local machine.

And we also need to run Webpack in production mode,

to build a production version of our application.

That is going to require two separate files,

one file for all the Webpack configuration,

to build our application for development

and one for production.

We're gonna create a third file,

that is gonna have some configuration that is common

for both development and production.

So whenever we run Webpack in development,

we're really going to take

all the configuration in both of these files,

and mush them together

or merge them together is probably a better term.

Whenever we run Webpack for production,

we're gonna take these two files

and merge them together.

So that is the general idea.

We're going to repeat the same pattern,

on all four different projects.

So let's get started on this right away,

inside of our marketing project.

Okay, so inside of marketing

I'm gonna create a new folder called Config.

Inside there I'm gonna make three different files.

The first one I will call webpack.common js

then webpack.dev.js

and webpack.prod.js.

So I bet you could guess what the purpose of each file is.

Got Common for common configuration,

dev for development,

prod for production.

Okay, we're going to write out a little bit

of configuration inside of the webpack common.js file

to get started.

So inside of here,

we're going to assign an object

to module.exports.

Inside of that,

we're then going to define a module property,

and assign it an object,

a rules property and assign it an object

not an object, but an array.

And then inside of this rules array,

we're going to define what is called a loader.

The goal of a loader,

is to tell Webpack to process

some different files as we start

to import them into our project.

The first loader that we're going to wire up is Babel.

Babel is going to be in charge

of processing all of our code,

from the ES 2015, '16, '17, '18, '19, '20 and so on,

and turn it into regular ES five code

that can be easily executed inside of a typical browser.

Making use of babel in any modern project

is extremely typical, very, very common.

So we're going to add in an object to this array.

We will give it a test property

and put in /\.m?js$/.

So this essentially says,

that whenever we import in a file

that ends with an extension of either

mjs or just js,

we want it to be processed by bable.

That's all test does.

Next up,

we will add in an exclude property.

We'll put in /node\_modules.

So that means,

do not try to run this bable thing on any file

out of our node modules directory.

We'll then add in use,

assign an object to it,

put in loader babel-loader.

Immediately after that,

we'll put in another key of options,

inside there we will write out presets.

That's gonna be an array.

We'll give it a string of,

@babel/preset-env.

We're also going to,

right before that I put in the

second string first,

my mistake.

So right before that string,

we're gonna put in another string

of @babel/preset-react.

Then after this presets option,

we'll add in plugins.

This is going to be an array with one string inside of it,

put in @babel/plugin transform runtime.

Like so.

Okay, so that is it for some initial setup.

So the preset of React right here,

means that Babel is gonna process

all the different jsx tags,

so we add into our application.

So that's a little bit of react related code.

If you're not familiar with jsx,

don't sweat it.

Preset, right here is going to transform our code

in a variety of different ways.

So take all the kind of ES 2015, 16, 17

and so on syntax,

and convert it down to ES five.

And then finally,

plug in transform runtime,

is going to add in a little bit of additional code

just to enable some different features for our project,

inside the browser.

Such as, async await syntax

and some other related things.

Okay, so this is a good place to get started.

Let's take a pause right here,

and continue in just a moment.

Topic 36. Creating and Merging Development Config

Formularende

Formularende

-: Let's now move on to our development webpack file.

So webpack dot dev dot js

inside of here, we're gonna first require in

something called merge from a library

named webpack dash merge.

Merge is a function that we can use to merge together

two different webpack config objects.

So this merge function is what's going to

allow us to take all the config that we just wrote out

inside that common file and merge it together

with a configuration that we're about to write

inside this development file.

That's all the merge function does.

After that, we'll then get a very familiar plugin

called the HTML web pack plugin.

Remember, this is what is going to take some kind

of HTML file inside of our project

and inject a couple of different script tags inside of it.

After that, we'll then define our configuration.

So I'm going to say dev config.

Notice that we are not assigning anything to

module dot experts here just yet.

Inside this object, we'll set up a

mode property of development.

We'll assign a dev server

that's going to have a port of eighty eighty one.

We're also gonna give this thing an additional

option this time called History API fallback.

Do double check your capitalization

make sure you've got capital A,

lowercase P and I, and a capital F.

I will give it a index property of index dot html.

We're gonna come back and understand

what this property does in a little bit

once we start to talk about navigation inside of our app.

Then after the dev server section, we'll put in plugins.

That's gonna be NRA as well.

And just as we did before,

we're gonna add in the new HTML web pack plugin.

Give it an object with a template property of SRC.

Excuse me, we want public index HTML, that's better.

Okay, so this is our development specific configuration.

This is configuration we only want

to provide to webpack when we are running it

inside of a development mode.

So now we need to take our dev config and merge it together

with a configuration that we

wrote out inside the common file.

To do so, we're going to require

in the common configuration, at the top, we'll say const

common config is required dot slash webpack dot common.

And then at the very bottom of this file,

we're going to do our typical module dot exports,

and we will assign it the result

of merging together our common config

and the dev config by listing out the dev config second,

that means that the dev config is going to override

or kind of take priority over any other similar

options that we might have assigned to common config.

Okay, so that should be it for our initial webpack setup.

We've now got everything running,

hopefully, inside of a development environment.

Now the last thing we need to do is create some kind

of starter index dot JS file

and a starter HTML file as well.

So, let's take care of some initial

code setup in the next video.

Topic 37. Running Marketing in Isolation  
Instructor: Inside of our marketing directory,

we're gonna create two new folders,

one is gonna be an src directory,

and the other will be a public directory.

These two folders are gonna do

the same thing they did on our last project,

src will contain all of our source code,

and public will contain an HTML file.

So inside of marketing,

I will make one new folder called public,

and then a new file inside there called index.html.

Inside that, I'm going to immediately

add in some very basic HTML elements, so a DOCTYPE,

we'll put in an HTML element, a head, and a body,

and for now we will leave the body element empty,

I'll then save that file and close it.

Next up, once again, inside the marketing directory,

we'll make a new folder of src,

inside that, I'll make a new file of index.js,

and for right now,

we'll just do a very simple console log of, hi there.

Okay, so now all we have to do

is add in a script to start up our project,

and then I think we are ready to test everything out.

So I'm gonna open up my package.json file

inside the marketing directory,

I will find the script section,

I'm gonna delete the test script that is inside there

and replace it with a script that I'll call start.

So whenever we run start,

we're going to execute webpack serve,

just as we did on our last project,

but we're also gonna tell webpack

which of our different config files it should use.

So we're gonna add in a --config

and then go into the config directory, so config/,

and use the webpack.dev.js file

as a source of all of our configuration.

Okay, I'm gonna save this,

we'll then go back over to our terminal

and try starting our project up.

So at my terminal, I'm at my mfp overall project directory.

I'm gonna first change into the packages folder,

which is where all of our different sub-projects live.

We'll then change into marketing

and try an npm start,

or npm run start, either one will work just fine.

Okay, looks pretty good,

so I don't see any errors or anything like that.

I do see what looks like a breaking change right here,

that's totally fine,

you might have some deprecation warning,

totally fine,

as long as you don't see a distinct red error message,

we're totally okay.

So now we can test out our application inside the browser.

To remember, this app is running on port 8081,

so we will open up our browser

and go to localhost:8081.

Then once there, we should be able to open up our console

and see hi there appear.

Perfect.

Okay, so it looks like we are off to a good start.

Let's start to implement some actual interesting stuff

around our marketing app in the next video.

Topic 38. Wiring Up React

-: I think we're now ready to start doing some

serious development on our marketing project.

Before we start to write any code, however,

I just want to give you a very quick

reminder about something that happened

on our e-commerce project just a little bit ago.

So I've got my code editor

from the e-commerce project right here.

You might recall that on our

cart and products applications,

we eventually try to share

a dependency called Faker,

and you might remember that

as soon as we started to share that dependency,

we then started to get an error

whenever we were loading up our different

sub-applications of products or cart in isolation.

That error was related to the fact

that we were trying to make use of Faker

before it had actually been loaded into the browser.

To get around that error,

we created two different files to start up our project.

So inside of products, and this happened in cart as well,

we had created inside of our SRC directory

an index.js file that would import in the bootstrap file.

So the only goal of index.js

was to add in this import function,

and all that did was give Webpack

inside the browser sometime to load up Faker

before it actually executed the bootstrap.js file.

Because as soon as Bootstrap gets executed,

we are going to try to immediately import Faker.

And if we don't have access to Faker,

well, we're going to very quickly get an error message.

So we're going to take the same kind of pattern

inside of all of our different sub-projects

inside of MFP as well.

We're going to create an index.js file,

which will immediately import

something called simply bootstrap.

Okay, so back inside of marketing,

I'll find the SRC directory.

Inside there, I'm going to make a new file

called bootstrap.js.

Then inside of index.js,

I'm going to remove the console log we just added,

and I'm going to import./bootstrap.

Okay, that's it.

We don't have to touch the index.js file again.

Now, inside of bootstrap

is where we're going to add in our main startup code.

Thinking back to our e-comm project,

or the e-commerce project once again,

you might recall that in bootstrap,

we had to find a mount function to start up the app.

We had then defined two different cases to handle.

We had said if we are in development

and in isolation,

call mount immediately.

Otherwise, and technically not otherwise,

'cause we didn't really have an else case

or anything like that,

we would assume that we are running through container

and we should export the mount function,

so that the container can decide

where to mount our marketing app.

So we're going to use this exact same approach

on marketing as well,

and dashboard and off for that matter.

So let's first get started by putting together

a little bit of basic code,

just what we need to get a very simple

react application started up.

Now of course you might not have any familiarity

with react whatsoever.

If you don't, that's totally okay.

I'm going to explain exactly what we're doing

as we go through this.

So at the very top of this file,

we'll first import React from react,

we'll import another library called react-DOM

from react-dom.

Then, to define our mount function,

we'll define a function called Mount,

and we're going to assume that this will be called

with some kind of HTML element.

Then you might recall, on our previous projects,

we just generated some html,

and displayed it inside of this element.

Now that we are using React,

that process is going to be slightly,

slightly different.

So to get our application to start up,

or to get some content to appear on the screen,

we're going to instead call a function called

ReactDOM.render.

We're going to call it,

I'm going to split this into multiple lines,

just to make it a little bit easier to read,

and then as the first argument to it,

I'm going to put in what looks like html.

So it's going to be an h1 element that just says:

Hi there!

I'll then put in a comma,

and as the second argument, we'll put in where

we want to render this html looking stuff too.

So in this case, we're going to render it

to the element that was passed into this function.

We'll provide el, like so.

And that is it for our mount function.

Believe it or not, it is just that simple!

Now we're going to add in our two other checks.

They're going to look identical to the checks

we had added in on our previous projects.

So we first need to decide whether

or not we are running this application in development.

For that, we can take a look at process,

always get that auto complete.

Process.env.NODE\_ENV,

and we're going to check to see

if that is equal to the string development.

If it is, then we're going to try to find

some element inside of our html file

that we're going to expect to

only be present inside of our marketing html file.

So let's go add in some element right away,

and we're going to give this thing an ID or some kind

of selector that's probably not going to be duplicated

inside of our container app.

So I'm going to add in a div here,

with an id, and we'll give this thing an id of

how about \_feed-dev-root,

very unlike,

excuse me, not feed, but marketing:

marketing-dev-root.

So very unlikely that our container html file

is ever going to have an id like that.

So then back over inside of bootstrap.js,

we're going to try to select that element.

I'll call it devRoot this time,

last time I think we called it el.

Doesn't really matter what we call it,

and we're going to try to select that using

a document.querySelector,

and our selector will be #,

and then what we just wrote out over there,

which was marketing-dev-root with the underscore

at the start.

Then if we find that element,

if it actually exists,

we will call our mount function,

and pass in devRoot like so.

Okay. So that's it.

Then finally we'll handle the case in which we are running

our application through the container.

So in that case,

we'll once again just do an export of mount.

Okay, so let's save this,

go back where the browser, and see how we are doing.

So back inside the browser at localhost.8081,

I should see the content hi there up here on the screen,

which means we are now successfully

making use of React to display some content.

So I'd say this is a pretty good start.

We've got our marketing application

running react in isolation.

So let's take a pause right here,

and start to add in some more interesting content

on the screen in just a moment.

Topic 39. Marketing Components

**Marketing Components**

Hi!

**Attached to this lecture is a file called 'marketing-components.zip'. Please download it - we will use it in the next video.**

Resources for this lecture

Topic 40. Adding the Pricing and Landing Pages

Instructor: In the last lecture,

you should have found a zip file

called marketing components.

Inside of here are the two react components

that our marketing app needs to show,

specifically the landing page and the pricing page.

Let's extract the zip file,

inside there is a components directory,

we're gonna take that components folder

and drag and drop it

into the src folder of our marketing project.

So I'll do a very simple drag and drop like so,

we're going to make sure we copy the folder

as opposed to adding it as a workspace.

So now inside of the src directory,

I should see components, and then landing, and pricing.

So why are we not writing out

all these components from scratch?

Well, it's very simple,

inside of each of them

is really just essentially a lot of markup.

If you open up either file

you're gonna see a ton of JSX elements inside of here,

so it's essentially just a lot of styling stuff,

stuff that's not really relevant for us to go through

inside the context of this course.

So rather than sitting down

and spending probably about 20 minutes

writing out the contents of these files,

I just wanted to save a little bit of time

and give you the contents directly.

All right, now that we've got these files,

inside of here,

we need to actually wire them up to our project,

so let's take a very quick pause right here

and then wire them up to our marketing app

in the next video.

Formularbeginn

Formularende

Section 5: Generic Ties Between Projects

Topic 41. Assembling the App Component

Speaker: Let's wire up those two new components

to the rest of our application.

And we could definitely add in some code to do that directly

to our bootstrap.js file.

But then our bootstrap file would start to get really large

and hard to work with.

So instead, we're gonna make a new file

inside of our src directory, called app.js.

Inside of here, we're gonna create a new React component

that's going to show those two different Landing

and Pricing components.

We'll then make use of it inside of our bootstrap.js file.

So inside of app.js,

we're going to add in a couple of imports to get started.

We'll get React from react.

We are gonna get Switch Route,

and BrowserRouter from react-router-dom.

We're going to add in StylesProvider

from @material-ui/core/styles.

I'm then going to import Landing and Pricing.

And then finally, I will create a brand new React component

and export it at the bottom.

So I will do an export default.

Inside of here, I'm going to return a div.

And inside there, we're gonna place

a couple of different elements.

And you might notice that I also added an import here

for something called StylesProvider, Switch, Route,

and BrowserRouter.

And you might be really curious about

where these things are all coming from

and why we are including them at all.

These are actually gonna be some very large topics,

and understanding why we are making use

of a Router here, and the Styles provider,

are going to eventually kind of unearth

some really big issues inside of our micro-frontend app.

As of right now,

we're just gonna add in a little bit of markup

and then we're going to eventually come back to this file

and get a better understanding of what these things

are all about.

Okay, so inside this div,

I'm gonna first add in StylesProvider, as a JSX element.

After that, we will put in BrowserRouter.

Then inside there, I'm gonna add in a Switch, a Route,

with properties of exact path with /pricing,

component of Pricing.

I'm gonna close off the route

and I'm going to add in one other Route.

So Route, the path, no exact in this case, a / and

a component of Landing.

Okay, so again, a lot of markup here.

Might not be really clear what the StylesProvider is,

what the BrowserRouter is all about.

Don't worry.

Like I said, we're gonna come back to this file

because these different things are gonna start

to expose some really big issues inside of our app.

Right now, let's save the file.

We're then going to go back over to our bootstrap.js file

and make use of this App component that we have created.

So inside of bootstrap.js, at the very top,

we will import the App component that we just created,

and then we are going to render that to our DOM

instead of a very simple h1.

So I'm gonna replace the h1 right there with App, like so.

Okay, so that should be it.

Let's save all these different files we've now changed

and then go back over to our browser and just make sure

that we can see some content on the screen.

Specifically a Landing page and a Pricing page.

So back inside my browser, at localhost:8081,

we can now see our Landing page.

You'll notice a ton of styling inside of here.

So this is pretty much all that markup that you just saw

inside of that landing.js file.

Again, we could have written

this out together inside the course,

but it really just would've taken a whole bunch of time

and that's why I just gave you the file directly.

Now, as a reminder, just about all the different things

that we're gonna see on these different pages is fake data,

and there's not really gonna be a whole lot

of interaction here.

So you will see some number of cards here.

They're gonna have some random images on them.

They have some view and edit buttons.

Those buttons are not gonna do anything.

They're just here to say,

Hey look we've got some content on the screen.

The buttons that do actually have some effect,

are the PRICING buttons right here.

So we should be able to click on PRICING,

navigate over to localhost:8081/pricing

and see our Pricing page up here on the screen as well.

So here's the Pricing page.

We've got some different pricing options,

Free, Pro, Enterprise.

Again, there's no real actual Signup,

or billing or anything like that.

We should be able to click on some of these different links

and it'll take us to a route of auth/signup,

but we still see our original Home Page on here.

So I think it's kinda clear that,

not everything is quite working just yet,

but we do at least have the Landing page

and the Pricing page on the screen.

Okay, so this looks good, but like I said,

there's still some kind of outstanding topics

inside the app.js file that we need to think about.

We also need to make sure

that we start to build up our Container application

so that we can start to wire up the Marketing micro-frontend

to the overall Container as well.

Right now, we have no implementation

for the Container whatsoever.

So let's take a pause right here

and continue on in just a moment.

41. Assembling the App Component

Speaker: Let's wire up those two new components

to the rest of our application.

And we could definitely add in some code to do that directly

to our bootstrap.js file.

But then our bootstrap file would start to get really large

and hard to work with.

So instead, we're gonna make a new file

inside of our src directory, called app.js.

Inside of here, we're gonna create a new React component

that's going to show those two different Landing

and Pricing components.

We'll then make use of it inside of our bootstrap.js file.

So inside of app.js,

we're going to add in a couple of imports to get started.

We'll get React from react.

We are gonna get Switch Route,

and BrowserRouter from react-router-dom.

We're going to add in StylesProvider

from @material-ui/core/styles.

I'm then going to import Landing and Pricing.

And then finally, I will create a brand new React component

and export it at the bottom.

So I will do an export default.

Inside of here, I'm going to return a div.

And inside there, we're gonna place

a couple of different elements.

And you might notice that I also added an import here

for something called StylesProvider, Switch, Route,

and BrowserRouter.

And you might be really curious about

where these things are all coming from

and why we are including them at all.

These are actually gonna be some very large topics,

and understanding why we are making use

of a Router here, and the Styles provider,

are going to eventually kind of unearth

some really big issues inside of our micro-frontend app.

As of right now,

we're just gonna add in a little bit of markup

and then we're going to eventually come back to this file

and get a better understanding of what these things

are all about.

Okay, so inside this div,

I'm gonna first add in StylesProvider, as a JSX element.

After that, we will put in BrowserRouter.

Then inside there, I'm gonna add in a Switch, a Route,

with properties of exact path with /pricing,

component of Pricing.

I'm gonna close off the route

and I'm going to add in one other Route.

So Route, the path, no exact in this case, a / and

a component of Landing.

Okay, so again, a lot of markup here.

Might not be really clear what the StylesProvider is,

what the BrowserRouter is all about.

Don't worry.

Like I said, we're gonna come back to this file

because these different things are gonna start

to expose some really big issues inside of our app.

Right now, let's save the file.

We're then going to go back over to our bootstrap.js file

and make use of this App component that we have created.

So inside of bootstrap.js, at the very top,

we will import the App component that we just created,

and then we are going to render that to our DOM

instead of a very simple h1.

So I'm gonna replace the h1 right there with App, like so.

Okay, so that should be it.

Let's save all these different files we've now changed

and then go back over to our browser and just make sure

that we can see some content on the screen.

Specifically a Landing page and a Pricing page.

So back inside my browser, at localhost:8081,

we can now see our Landing page.

You'll notice a ton of styling inside of here.

So this is pretty much all that markup that you just saw

inside of that landing.js file.

Again, we could have written

this out together inside the course,

but it really just would've taken a whole bunch of time

and that's why I just gave you the file directly.

Now, as a reminder, just about all the different things

that we're gonna see on these different pages is fake data,

and there's not really gonna be a whole lot

of interaction here.

So you will see some number of cards here.

They're gonna have some random images on them.

They have some view and edit buttons.

Those buttons are not gonna do anything.

They're just here to say,

Hey look we've got some content on the screen.

The buttons that do actually have some effect,

are the PRICING buttons right here.

So we should be able to click on PRICING,

navigate over to localhost:8081/pricing

and see our Pricing page up here on the screen as well.

So here's the Pricing page.

We've got some different pricing options,

Free, Pro, Enterprise.

Again, there's no real actual Signup,

or billing or anything like that.

We should be able to click on some of these different links

and it'll take us to a route of auth/signup,

but we still see our original Home Page on here.

So I think it's kinda clear that,

not everything is quite working just yet,

but we do at least have the Landing page

and the Pricing page on the screen.

Okay, so this looks good, but like I said,

there's still some kind of outstanding topics

inside the app.js file that we need to think about.

We also need to make sure

that we start to build up our Container application

so that we can start to wire up the Marketing micro-frontend

to the overall Container as well.

Right now, we have no implementation

for the Container whatsoever.

So let's take a pause right here

and continue on in just a moment.

Topic 42. Assembling the Container

Instructor: I mentioned several times in the last video

that we're gonna spend a lot of time to talk about

what react router is doing here,

and what the StylesProvider is doing as well.

However, it's not gonna make a lot of sense to dive into

a discussion around these things and how they really fit

into a microphone and app until we also have

our container set up.

In other words, right now our marketing application

in isolation works just fine.

But as soon as we start to run it through our container,

we will see some really big issues start to come up.

And in some other cases, such as with this StylesProvider,

we're only gonna see some issues start to arise

once we actually deploy our application to production.

Like I said, we are gonna come back to this stuff.

If it's all mysterious, don't sweat it.

But we need to do a little bit more set up first

before some of these different things are gonna make

a lot of sense and why we would care about them at all.

So in this video,

we're gonna temporarily bench our marketing project,

and we're gonna start to do a little bit of setup

on our container.

Inside the container, we're gonna add in a lot of very

similar configuration to what we just did

inside of marketing.

We're gonna essentially make a very basic react application.

We'll then wire these two things together,

the container and marketing,

and then we can start to investigate some other really

interesting issues around micro frontends.

So let's get to it.

The first thing we're going to do is take our

marketing config directory,

which you will recall has some different files all

around setting up webpac,

and we're gonna copy that entire folder over to container.

All of this initial webpac config is gonna be identical

between marketing and container,

so we might as well save ourselves a little bit

of time and just do it direct copy.

I'm gonna copy just the config folder

and paste it into container.

Okay, there we go.

Next up, I'm gonna open up my package.json file

inside of container.

We're gonna add in our start script.

So I'm gonna replace "test" with

"start": "webpack serve -- config,

and we want to use the config webpack.dev.js file.

Okay, simple enough.

Next up, inside of our container,

we will create two directories.

First, a public folder

and an src folder.

So here's public and here's src.

Inside of public, I'll make my index.html file.

So as usual, just the same as our last big project,

this HTML file inside of our container is gonna be

the main overall HTML file that gets used whenever we

load up our application through the container

or whenever we load it up inside a production.

So inside of here we'll put down just a little bit

of very plain HTML, just very similar to what we've had

previously, both run by <!DOCTYPE html>,

a head, and a body for right now.

And you know what?

Just to save ourselves a little bit of time

I'm also gonna put in a div id="root"

which is, is where we're going to eventually

show our container application inside of, cuz remember

our container app is gonna be a React app itself.

So let's just save ourselves a little bit of time with that.

Okay, so that's it for our index.html file.

Next up I'm gonna create two files inside

of our SRC directory.

I'll make one file called index.js

and another that I will call bootstrap.js.

So as usual

for all the same exact reasons inside of index.js

all we're gonna add in is a ./bootstrap.

Like so.

And again, we've spoken about this several times

this just gives Webpack when it's running

inside the browser a little bit

of time to make sure that we can load

up all the appropriate dependencies to actually

run our bootstrap.js file.

So then inside of bootstrap.js is where

we're going to actually show some content

on the screen inside of here.

For right now, I think all we really have to do is set

up a little React application.

So at the top I'm going to import React from 'react'

I will import ReactDom from 'react-dom'.

I'm then going to assume

that we're going to eventually have some kind

of app component that we want to show on the screen.

So I'm gonna add in an import App from App.

We don't have a app file inside of our container just yet

but we will create one in just a little bit.

Then after that we're gonna take that app and just render it

and display it on the screen using ReactDom.

So I will add in a ReactDOM.render.

I'm gonna place in app like so

and then we will add a document query selector

or #root select that div

that we just wrote out inside of our HTML file.

And you might be curious, why are we not doing

that same kind of check for development?

Why are we not exporting some kind of mount function

or anything like that?

Remember, our container doesn't need a mount function

whenever the container application is gonna be shown

inside the browser.

We always, in all scenarios, want the container to

show itself immediately.

It's only our sub-projects that need to export some kind

of mount function or render themselves conditionally

depending upon whether or not we are in development.

So that's a big difference between the bootstrap.js file

we had put together inside a marketing and the one inside

of container.

Okay, so now very last thing.

Let's create an app component inside my src directory.

I'll make a new file of app.js inside of here.

We will import React, and then I will export default.

And as usual, for right now, we'll just do a very

simple h1 say hi there!

Okay, so I think we're just about ready to test this out.

The very last thing we need to do is make sure that we start

up our development web server or our container project.

There is one very small change we need to make inside of our

-: Config directory.

We're gonna find the webpac dev.js file.

So we do not want to run our container

on port 8081 as well.

Instead, let's run our container on port 8080.

Now we're gonna have our container

on 8080 and marketing on 8081.

So I'll save that file and we'll go back over

to our terminal and open up another terminal window

and start running our container inside of it.

So here's my terminal window where I'm still

running our marketing application.

I'm gonna open up a second terminal window.

I will change into our container directory

and do an npm run start.

Okay, so looks like it's all started up.

So now we should be able to open up our browser, open

up another tab, go to local host 8080 and see hi there.

Appear on the screen.

And you'll notice that I do not have any errors

or anything like that inside my console.

So if you had any problems with that initial startup process

or if you have any errors inside of your console

chances are you've got a small typo somewhere.

So you'll want to hunt that

down. Okay, so that's it

for some initial setup around our container.

Now the last thing we really have to do here

is do some web pack config setup

around the module federation plugin and make sure

that we can show marketing inside of our container.

So let's get started on that in just a moment.

Topic 43. Integration of the Container and Marketing

Instructor: Now that we've got our marketing

and container apps put together,

we need to somehow join them together,

or remember, the term we use is integration.

For that, we're going to once again

make use of the module federation plugin.

So we have to go to our marketing project

and do a little bit of setup around webpack

with a module federation plugin,

and same thing inside the container as well,

so let's get to it.

Inside of my marketing directory I'll find the config folder

and inside there we'll open up our webpack dev js file.

Now why are we adding our module federation plugin

to the dev file?

Because remember we're still gonna use this

module federation stuff in production as well,

the reason for that

is we're gonna have some slightly different configuration

around our module federation stuff,

depending upon whether or not

we are in our development environment,

or our production environment,

so we are gonna make use of our module federation plugin

two times in just about every one of these sub-projects,

once inside the webpackdev.js file,

and once inside the prod file.

So inside my webpackdev.js file,

we're gonna go through the exact same setup

that we went through on our last application,

we're going to require in the plugin at the top

and then add it into our plugins array.

So I am going to require in module federation plugin

and we're going to get this

from that really long import string,

it is webpack/lib/container/ModuleFederationPlugin.

Once you've got that in,

we'll then go down to our plugins array

and add in a new module federation plugin,

and I think my editor might have,

ah, yep, right there,

my editor added in a require statement automatically,

which I don't want,

so I'm gonna delete that really quickly.

Now inside of here, we're gonna use,

once again, very similar setup to what we had previously,

we're gonna put in a name property,

this will be the name of our sub-project,

which in this case is marketing,

and remember in that optional video

which you may or may not have watched,

what that marketing name really means,

marketing is going to be used to declare

some kind of global variable

when our script loads up inside the container,

so again, that's where the name actually gets used.

We'll then specify a file name once again,

remoteEntry.js,

not really a great reason

to use just about anything else there.

We then need to designate exactly what file

we want to make available to the outside world.

So remember inside of our marketing project,

we've got the bootstrap.js file,

in this file is where we define that mount function

and export it down here.

So I think that this is the file

that we want to make available,

so we're going to define that exposes object,

we'll put inside of here some file name,

and say whenever someone asks for this,

we're going to give them bootstrap.

In this case, let's say that whenever someone

tries to ask for, how about, marketing,

I guess that's probably good enough,

let's give them the ./src bootstrap file,

and we will eventually come back

and add in some shared modules inside of here as well,

but for right now, we're just gonna kind of

duplicate all of our dependencies

and we'll worry about reducing them down

and setting up some different shared dependencies later on.

Okay, so I think this is a good enough place to get started.

Now we need to go over to our container

and make sure that the container

tries to load up this shared module,

so inside of our container directory,

we'll find our webpackdev.js file.

Inside of here, we're gonna once again

add in that same exact required statement at the top,

so module federation plugin from webpack,

and what is it?

Lib container module federation plugin.

Once we've got that in, we'll then, once again,

go down to our plugins array,

add in new module federation plugin.

Now in this case,

we are setting up our container,

as remember the term that we use here?

Our host.

The host wants to make use of some modules

coming from other parts of our application,

so over here all we have to do

is designate our different remote options,

and we can optionally add in a name here as well.

Again, the name for our host

never actually gets used for anything,

but you will see a lot of documentation and whatnot

that says, oh yeah, even for your host, set up a name.

Again, not strictly required,

but we'll just follow convention anyways.

Then inside of here we'll set up our remote section.

Remember the syntax for remotes,

we're gonna put in a collection of key value pairs.

The keys are the names of the different modules

that we're going to try to require in or import in

to our container project,

and then the value will be where the remote entry file is

for that module.

So in our case, we wanna say that

if we ever tried to import something called marketing,

then we want to go to marketing@http://localhost:8081,

and it's specifically port 8081 here,

'cause remember we set up our marketing application

to run on port 8081.

And from there,

we want to load up the remoteEntry.js file.

So one more quick reminder,

the word marketing right here,

right before the at symbol,

matches up with the name marketing

that we just wrote inside

of the marketing webpack dev file.

And then the marketing key right here

means that if we ever write out an import statement

inside of our container

and ask for something called marketing,

we're going to load up that remote entry file

and look inside there for it.

Okay, so let's save this,

and now finally,

we need to actually try to import marketing

and make use of it somewhere

inside of our container project,

so we're going to eventually make use of it

inside of our app.

Inside of my app.js file, for right now,

I'm gonna add in import mount from marketing,

so marketing means go and get that remoteEntry.js file,

and then out of there we need to ask for some distinct file,

we need to ask for something

out of the marketing remote entry file.

The actual thing that we exposed here

we can find from the marketing, config, webpack.dev.js file,

so this is what is available to us.

From the marketing module,

we can ask for a file called marketing.

Now this is kind of a bad name,

I'll be totally honest with you,

calling the thing marketing,

maybe not like the best thing in the world,

maybe we could think of some better name for this,

maybe something like just home,

but I don't really wanna call it home,

'cause that kind of implies the homepage,

and we also have really a pricing page coming here.

I'll tell you what, let's instead call this marketing app,

or maybe just like app,

maybe that's a little bit better.

Nah 'cause if we do app,

that kind implies that we're getting the app.js file,

let's just do marketing app,

just to make this thing something

other than just plain marketing,

which can end up being a little bit confusing.

So inside of our webpack.dev.js file,

inside the marketing directory,

we're gonna call that marketing app,

which means now inside of our container app.js file,

we're going to import mount from marketing/MarketingApp,

that's a little bit better.

Now let's not make use of the mount function just yet,

instead we'll do a very simple console log

of the mount function

and just make sure that it's been successfully loaded up

and we have access to it.

So now to test all this out

we have to restart webpack very quickly

for both the container and marketing.

At my terminal,

I'm going to restart both them,

so here's a restart of marketing

and a restart of my container.

And once those are done,

I should be able to refresh local host 8080,

so that's our container,

and there's the console log of mount right there,

and I should still be able to run our marketing application

in isolation as well.

So at local host 8081, I should be able to refresh,

and still see either the pricing page,

if I'm at /pricing,

or if I go back to the root route of just local host 8081,

I should be able to see the homepage.

Okay, so it looks like

we have joined these two things together,

we at least have access to the mount function,

but now we need to actually make use of it in some way.

That's gonna be the next big challenge we run into,

how do we actually make use of this mount function

in a kind of react environment?

Topic 44. Why Import the Mount Function?

Instructor: We've now set up some limited integration

between container and marketing.

Now we've got access to the mount function inside

of container and we need to make use of the mount function

to somehow get some content on the screen.

Remember, the mount function is a function

that takes in a reference to an HTML element

and then displays some content inside there.

So we cannot use this mount function

directly inside of a React component.

In other words, we can't put in something like mount.

That doesn't make any sense.

This mount function is not a React component.

It is a simple function that takes

in a reference to an HTML element.

So we need to figure out some way of making use

of the mount function from inside

of a React component because hey,

we can't just use it as a React element

like I just showed you.

And you might be kind of curious,

why are we making this challenging

or difficult on ourselves?

Well, lemme show you what I mean by that.

I mean, why is that difficult at all?

Well, remember our marketing project uses React

and our container project uses React.

So one way that we can make our lives way more easy

and straightforward is to just have

our marketing application export a React component

that the container can use directly.

In other words,

we could have our marketing app export something called

like marketing React component or something like that.

I'm sure we could think of a better name.

And then we could take that and just display it

as a React component from inside of our container.

We could absolutely positively do this

and this would get our marketing application to be displayed

and everything would work as expected.

So why aren't we doing that

if it's so easy and straightforward?

Well, it all goes back to one of the big requirements

around our project that we had set up originally.

It was inflexible requirement number two.

Remember, we had said that we want

to have almost zero coupling between

a container and a child application.

And we had said specifically that we do not want

our container to assume that a child

is using some particular framework such as React.

That means that we do not want to ever exchange

between our container and marketing

some very framework specific thing.

We don't want to try to export

or make use of some React component.

Instead, we wanna keep the coupling between our container

and marketing as generic and simple as possible.

And that is why we are exporting

and making use of the mount function.

It is a very generic function

and we can reuse that same kind of function signature

where we just have a function that takes in a HTML element,

even if we start to make changes

to marketing or a container.

Just to make sure it's super clear

why we're taking this approach,

let's consider a couple of different scenarios very quickly.

So let's say that at some future point in time we decide

that our container should not make use of React anymore.

Maybe in some number of years we decide to use

some new framework called Flubber or whatever else.

Well if our marketing application

is still exporting some kind of React component,

chances are this Flubber library won't know

what to do with the React component.

So as soon as we start to refactor a container,

that would force us to do a refactor to marketing as well.

And we would have marketing instead export,

I dunno maybe a mount function or something like that.

That could actually be used by this Flubber framework.

The downside here is that to make any update to container

we would have to make an update to marketing as well.

And again, we don't wanna be in that kind of scenario.

Now let's consider the reverse.

Let's consider the case in which

we decide to refactor marketing.

And maybe we want marketing to use

some new framework called Flubber.

Well, now marketing can't export a React component

'cause there is no React component.

So now we would have to have marketing export

some other kind of thing, maybe a mount function

or maybe a Flubber (mumbles gibberish)

Whatever it is.

Who knows what this Flubber framework does?

And we would have to somehow make

a change to container to make use of this Flubber thing

or mount function or whatever else.

So in both these scenarios, in order to change one side

of this equation, in order to change say the container,

we would be forced to make a change to marketing.

And if we wanted to make a change to marketing,

we would be forced to make a change to container.

That is the exact scenario that we are trying

to avoid when we make use of micro frontends.

We want all the coupling here to be as generic as possible.

So if we ever decide to swap out something down here,

we won't have to make a change to the other thing.

So that's why we are not exporting a React component,

even though it would make our lives a lot easier right now.

Okay, so in all this discussion I didn't really explain

or say anything about how we are going to make use

of the mount function from inside of our container.

So to understand how we're gonna make use

of the mount function, we need to learn a little bit more

about React 'cause it's really gonna be pretty much how

do we get React to somehow make use of this mount function.

So let's figure that out in the next video.

Topic 45. Generic Integration

Instructor: All right, let's try to make use

of this mount function from inside of container.

Now, making use of the mount function

is gonna be just a little bit challenging

cause there's not really any built in kind of structure

or function or API

or anything like that inside of React

that expects to work with a function like mount,

one that takes a reference to an HTML element.

So we're gonna have to kind of mess around

with React a little bit just to get this stuff to work.

Now don't worry, we're not gonna do anything totally wild

so it's not gonna be anything too crazy,

but we are going to need to have

just a little bit of knowledge around React

to make this happen.

So we're gonna write out the code for this.

I'm going to kind of assume

that you are at least somewhat familiar with React here.

If you're not, totally fine, just write out the code

and we'll be done with it pretty quickly.

Inside of my SRC directory, I'm gonna get started

by making a new folder called 'components'.

And then inside there I'll make a new file,

we'll call it 'MarketingApp.js'.

Inside of MarketingApp, we're going to import

the Marketing mount function.

We'll add in,

{mount} from 'marketing/MarketingApp',

We are also going to import React from 'react'.

And from React,

we are also gonna add in a secondary import as well.

We're also going to get something

called 'useRef' and 'useEffect'.

Now 'useRef' and 'useEffect' are two React hooks.

So this is where I'm kind of assuming

you've got a little bit of knowledge around React.

If you don't know what these functions are, totally fine,

let's just write the code out.

So inside of here we're going to create a React component

and export it.

We're then going to use the 'useRef' function

to create a reference to an HTML element.

This is gonna be the element that we're gonna try to

render our mount or our MarketingApplication into.

We'll say ref is 'useRef',

and I'll provide a starting value of null.

We're then going to return a div.

It's gonna be a plain div just by itself, nothing else.

I'm going to assign the reference to it like so.

So all this code right here is doing is showing a div

and we are getting a reference to the HTML element

that is being displayed on the screen.

Now we need to take this reference to that element

and provide it to the mount function,

but when we do so, we need to make sure

that we only try to call the mount function one time

when this React component first is displayed on the screen.

So for that we're gonna make use of the 'useEffect' hook

which we also imported.

The 'useEffect' hook is gonna make sure that

we can run code just one single time

when this component is first displayed.

So I will call a 'useEffect' and pass it in arrow function.

This arrow function is what's gonna be executed when

this component first shows up.

Then inside of here,

we're going to call our mount function

and we'll pass in 'ref.current'.

So that is the reference to the HTML element.

We are passing that into the mount function.

Mount is gonna take it,

it's gonna try to create an instance of our MarketingApp

and render it into that div.

Now the good thing about this approach right here

is that it is pretty much fully reusable.

We can use this exact same approach

with just about any other framework

that we ever use inside of a child application

as long as that framework can render itself

or render its app into some arbitrary HTML element.

So you should probably be able to use this kind of pattern,

with say, an Angular app or view.

And we are going to use this with view on this project

and of course you can use it with React as well

and probably many other frameworks.

So once again, even if we start to change our MarketingApp

and have it used some totally different framework

this approach inside of our container

should not need to change in any serious way.

Now just you know,

we are going to eventually come back here

and make a couple changes to the 'useEffect' function

once we start to have the ability to hide

and show different sub apps.

So we're not quite done here.

There are some other things we're gonna have to come back

and add in, but let's kind of figure out

why we'd have to add those things in first.

I just want you to know that's going to eventually happen.

Okay, so let's go back over to our App.js file

inside of our container.

We do not need to import mount inside of here anymore.

And of course we don't need to do a console log.

We're going to import the MarketingApp component

we just created.

So I will import MarketingApp from components/MarketingApp,

and then we will show an instance of MarketingApp

inside of the app component.

Now just to make sure

that we are actually inside the container

and see some different content on the screen,

rather than only showing the MarketingApp,

I'm gonna put in a div right here.

I'm gonna delete the semicolon at the end of the 'h1'.

I'll close off the div,

I'm gonna put it in the MarketingApp.

And you know what, I also put in an 'hr' right there as well

just to get a little horizontal separation.

So the only reason I'm doing this

is just to make sure that it's really clear

that we can render some other content inside of React app

and then also show inside of here at the same time

the MarketingApplication too.

So let's save this.

Let's go back over to the browser and see how we're doing.

So at localhost:8080,

I can see 'Hi there!' here on the screen.

There's the hr, the horizontal divide,

and then there's our MarketingApplication.

So we can see the homepage very easily.

And if we go back over to localhost:8081,

we are running our MarketingApp in isolation.

Of course, this page still works as well.

Okay, so I'd say this is a good start.

We can now show the container, which is React app,

and inside there we can show another React app

and we've got some super generic coupling between the two.

So if we ever swap out the framework on one side,

no problem whatsoever.

All right, so let's take a pause right here

and continue on in just a moment.

Topic 46. Reminder on Shared Modules

Instructor: Let's take a look at some

of the different requests that are being made

for JavaScript files when we load up our marketing

application through the container.

I'm gonna go to my Network request tab,

filter by JavaScript requests, and refresh the page.

All the different files that we are loading up

here can be put into two different categories.

Some of the files are being loaded up

by the container, and some are being loaded up

through our marketing application.

The main.js file is a part of our container.

The first vendors-node\_modules is a part

of the container, and then one of the bootstrap

files is from our container, as well.

The other three files, so remoteEntry,

the other node\_modules, and the second bootstrap.js

are all coming from our marketing app.

Now at this point in time, if you just look at the name

of these node\_module files, you might see the first one

says something like react-dom\_index.js.

You might notice that the second one does not

have any mention whatsoever of React,

or at least it doesn't for me.

So when you see this, you might kind of assume that

this first vendors or node\_modules file has React

inside of it, and the second one might not.

Well, that's not what is really happening.

The different file names of these node\_modules files

are somewhat randomly generated,

and the names that are being applied to them don't

necessarily indicate all the different dependencies

that are being placed inside of these JavaScript files.

So even though it might appear that we are only

loading up one copy of React right now,

that's not what is really happening.

If we click on the first node\_modules file,

you can do a search on the Response tab for React,

and you'll eventually see, okay, this is definitely

something that indicates we're loading up

ReactDOM right here.

We can then click on the second node\_modules file,

so this is the one that is coming from

our marketing application, and even though the file name

doesn't have "react" in it, if we do a search inside

of here as well for React, you will eventually

see some stuff that definitely looks like React.

You can see just tons and tons of references to React.

So I think that at this point in time, it's pretty clear,

and if you wanna see the actual React library,

'cause even the search I'm doing doesn't really pull it up,

you can, to be honest with you, just go down to the bottom

of the file, probably, scroll back up, and you'll see some

of the different module names right here.

So here's Scheduler, Tracing, Scheduler Index,

and I think this is probably, these are all a part of React.

We keep going up, there we go, there's React right there.

So the point is, very clearly, we are loading up two copies

of React, two copies of ReactDOM, probably multiple copies

of some other dependencies, as well,

even though these file names don't really make that

very evident from the get-go.

So as usual, we can make use of the module federation

system to reduce the number of duplicate dependencies

that we were loading up between the container

and our marketing application.

We already saw how to do this previously.

So you might recall that we can open up our config,

dev.js file, inside of marketing.

So here's the marketing ModuleFederationPlugin.

We can very easily add in a shared section right here,

and then list all the different dependencies

we want to share.

Maybe React, react-dom, and let's just start

with those two for right now.

We could then go over to our container config,

find the webpack.dev.js file, and then after remotes,

put in shared React and react-dom, as well.

If we then save both those files and restart our two

Webpack processes, so there's a restart of the container,

and here's a restart of marketing.

Now if we reload our application inside the browser,

we're gonna see a very different number

of files being loaded up.

Without a doubt, we are now only loading up one copy

of React, and one copy of ReactDOM, as well.

If you want to, you can always double-check the node\_modules

or some of these different other files inside of here,

and just verify that you are in fact just getting one copy.

So here's some kind of React file right here.

Okay, it looks like we're definitely loading up React.

There's another file right here

with the name of React, as well.

That one is actually ReactDOM.

We can continue taking a look at all the others,

and you're not gonna see any other copy of React.

You might see some other mentions of React

if you do a search inside them,

but that's because we are also loading up

the Material UI library, which internally will have

some references to React inside of it.

But at the end of the day, we are now only loading up

one copy of React and ReactDOM.

Okay, so obviously that is good, that's great.

But there's one thing I just wanna point out

here, really quickly.

You might eventually start to make use of a lot

of different modules inside of all your different projects,

and it might get a little bit tedious rather quickly

to have to go into all these different Webpack files,

and write out the shared modules array,

and keep those all updated as you start to add in

dependencies, change versions, and stuff like that.

So let's take a pause right here.

In the next video, I'm gonna show you an optional little

shortcut that we can use to kind of just delegate

all this module-sharing stuff off to Webpack.

So we don't really have to come back and make changes

all the time, as we start to add in additional modules.

So let's take a look at that little shortcut

in just a moment.

Topic 47. Delegating Shared Module Selection

-: Having to go into these different webpack config files

and update the shared dependencies array can

definitely get tedious.

At the same time you might want to do it.

Lemme tell you what I mean.

There might be some scenarios where you are,

want to be really specific

about all the different modules that are being shared,

in the exact versions of those modules that you are sharing.

So maybe you are working

on a project where you want to be extremely precise,

and you have no problem, whatsoever, going

into your different webpack config files

and updating that shared modules array.

But at the same time, maybe you are a little bit more lax.

Maybe you say, Hey, you know what?

I just want webpack to take care of this stuff for me.

I don't want to have to go and remember to

update the shared modules array

every single time I add in a different dependency,

or updated version, or something like that.

In this video, I'm gonna show you

a little shortcut we can use, to just say,

Hey, webpack, just figure this stuff out for us.

You might not always want to use this shortcut,

because again, you might want to be very specific

about the exact versions and settings

around these shared modules.

Just throwing that out there.

Okay, so here's the shortcut.

Inside of our container project,

I'm gonna open up the package.json file.

So, inside of here is our dependency section.

Inside of dependencies, are all the dependencies

that we're going to eventually make use of,

in production, or really, inside the browser.

That is totally different than the dev dependency section.

These are different dependencies that we only

use when we are trying to build up our

application and serve it inside the browser.

So, I do not ever expect to share any

of these dev dependencies,

but I definitely expect to share, probably,

all of the dependencies listed

inside the dependencies section.

So, to kind of automate this entire process

of sharing these modules, or at least specifying which ones

we want to share, we can actually require in

the package.json file directly to our webpack.dev.js

file, and then we can take this dependencies

object right here, and just pass the entire object off

to webpack and say, Hey, you know what?

Here's the entire list of dependencies that I just

want you to manage for me.

The benefit to this approach, is that if we ever

start to install additional dependencies,

as soon as we restart webpack, webpack will

get the updated list of dependencies

we have listed inside this section.

So we don't have to go and update that

array of strings all the time, as we start

to add or remove modules.

So, let's take a look at how this is actually done.

Inside of my container config webpack.js file,

at the very top, I'm gonna add in a require statement

for package json, and that's going to be a literal require

for our package.json file.

So this variable, right here, is pretty much

this exact object that you're looking at.

All this json gets parsed and provided to us

as a plain JavaScript object.

So package.json has, for example, a

dependencies property on it, and that

dependencies property is gonna be this exact

object you see right here.

So, we can take that object

that we now have access to, and just pass it off.

I'm gonna delete that.

That was just a quick example right there.

We can just take that entire object,

and pass it directly to this shared property.

And webpack is gonna say, Okay, this must be the list

of dependencies that you want me to manage.

Now, even though you see that we have an

array of strings right here, right now, we can

just as easily provide a object where

the keys are the different module names,

and the values are the different versions.

And the module federation plugin will take that

object just as happily and readily as an array of strings.

So I'm going to replace React, React on,

right there,

with package json.dependencies.

I'm then going to go over

to my marketing webpack dev.js file

and repeat the same process.

So at the very top, I'll get package json

from up one directory package.json,

and then down at the shared array,

I'll replace this with packageJson.dependencies.

Okay, so now, like I said

as we start to add or remove dependencies,

we don't have to worry about going into a webpack

config and starting to update this shared modules array.

So again, I just wanna repeat one more time.

You might not want to do this

if you want to be very specific

about what modules you are sharing, but it's really

up to you in the application you are building.

Let's now go back over to our terminal.

As usual, we're gonna restart both these webpack processes.

There's marketing

and there's container.

I'll then go back over to my browser,

refresh,

and now we are sharing absolutely everything.

One thing that might be kind of fun here, is to

pick out those shared options and see

how many megabytes of JavaScript you were loading up,

and then add them back in,

and see how much we are loading up now.

As a matter of fact, I'm just gonna

do that really quickly, just for fun,

so, you don't have to do this.

Just a really quick example.

We're gonna comment out both those sections.

I'm gonna take note that currently I'm loading

up 2.3 megabytes worth of JavaScript.

I'm going to undo those two changes.

So we're gonna go from 2.3 up to 3.5.

So definitely a very significant size savings,

by just deciding to share dependencies

between these two different sets of projects.

And remember, these are development sizes.

In other words, all this code is not minified right now.

When we move into production, those amounts, that amount

of JavaScript we're downloading right now,

is gonna be significantly, very significantly, reduced.

So if that seems really high right now, don't sweat it.

It's gonna go down.

Okay, So I'm gonna uncomment both the shared sections,

and then restart webpack one more time.

Again, I was just doing that out of total curiosity,

and so now I should be back down

to the 2.3 megabytes, or so.

Okay, so good improvement.

We now no longer have to worry

about making all these manual tweaks

to that shared module array, which is fantastic.

Let's continue on in the next video.

Section 6: Implementing a CI/CD Pipeline

Topic 48. Requirements Around Deployment

-: We've only got the absolute beginnings

of our container and marketing projects done,

and we've not even began working on our auth or dashboard.

Nonetheless, I want to start thinking

about deploying our application right away.

The reason for this is that we're gonna have a lot

of different problems start to come up,

that are going to impact how we develop

further parts of our application

once we see some challenges that start to arise

as our app gets deployed.

So with that in mind, even though we've only

got a little bit of work done on the project,

we're still gonna start the deployment process.

So, to get started, I first wanna lay

out a couple of high level notes

and big requirements we have around our deployment process.

So the first big hard requirement that we're gonna have,

is that we must be able to deploy

each microphone end independently,

and that also includes the container.

So remember the underlying assumption here,

is that we're gonna have multiple different teams

working at some company on some overall product.

Each team is responsible for a different sub-project.

Each of these different teams might be adding a new features

at completely different rates.

And as each team adds in a new feature,

they might want to deploy their application.

And they will want you to deploy their app

without having to worry about

some other team finishing up their features.

So we need to make sure

that as we start to change each sub-project

we can deploy each one by themselves.

The next big thing we need to be aware of,

and this is gonna seem like a very,

very specific technical detail,

but it turns out that this is actually gonna give us

a little bit of a headache.

So the location of each child app remote entry file

has to be known at build time

when we are deploying our container.

Let me make sure it's just really clear what I mean by that.

Okay, so remember, when we are running our app in production

or even in development for that matter,

we're eventually going to load up a main dot js file

coming out of our containers Webpac build process

into the browser.

Remember, we don't actually have a main JS file

inside of our code editor.

It is after our code gets processed by Webpac

that we get a main.js file,

and that's what actually gets loaded up in the browser.

When that gets loaded into the browser,

at some point in time, our application is gonna decide

that it needs to load up some code from

the marketing application.

And at that point in time,

we're gonna make a follow up request

over to our marketing application

and get the remoteEntry.js file.

And that file is gonna have some instructions on

how we can load up all the code we need

from the marketing sub-project.

So the main.js file essentially,

needs to know exactly what URL to go to,

to get that remote entry file.

And what's more that has to be known at build time

or essentially when we are taking all of the files out

of our container, processing them with Webpac

and then spitting out that main.js file.

It is when we are building our application

that we need to know exactly what the URL is gonna be

for all those different remote entry files.

When main.js is running inside the browser,

there's no outside api, there's no kind of file manifest,

there's no kind of magic way that our application

can figure out where those remote entry files are.

So they have be known when we are building up our project.

Now, just to make sure this is super clear,

we can even see this in action,

running our project in development right now

if we open up our browser.

So inside the browser at local host 8080,

we can look at our network request tab,

filter by JavaScript requests, and find the main.js file.

So again, this is coming out of our container project.

Inside of here, you can do a search with command

or control F, and look for, "remote entry."

So inside of here, you'll notice we've got the exact URL

of the remote entry file right there

for our marketing project.

That's just a comment, but...

if you kind of advance the search forward a little bit,

you're gonna see inside of the actual code right here

we have a very direct reference

to where to go to get that remote entry.js file.

So not only do we need to know that URL at build time

but we also need to somehow make that URL available

to our container config webpack config stuff.

So right now, inside of our dev file

remember that's where we specify that url.

So we essentially, when we build up our

Webpack production file, we need to substitute out

this string right here,

with wherever to find the production version

of our marketing remote entry file.

Okay, so I know this bullet point is a little bit specific

but I just wanna point out

that this might be a little bit of a challenge

or at least something we really need to think about.

All right, next big challenge.

There are many different ways

of deploying a front end application.

You can use very easy to use services such as now.sh,

you could use Heroku,

you can use... what else is there? Versel these days?

There's a lot of different options.

What a lot of these different options are

kind of assuming is that you are building up

one single project and that's it.

And you're gonna take the output of that one single project

and deploy it to some domain, and that's all there is.

That's not what we need.

We need a service that will allow us to take the output

of many different webpac build processes,

and somehow deploy all those

and deploy each of them separately.

So we need to make sure that

whatever solution we pick is gonna work

with our very specific requirements

of having multiple different projects.

All right, next one.

Now remember, in all this discussion

around micro front ends,

we're really taking kinda the idea here

that we might be working with multiple different teams,

possibly at a large company.

So, if you've never worked at a large company before,

it is pretty rare that you're ever going to

deploy your application by just running some command

on your personal laptop, and having that directly

deploy files to some production environment.

Instead, it's way more common for you to kind of

queue up a process or have some outside process

do the deployment for you.

So in this course, we're not gonna do the easy way out.

We're not just gonna deploy stuff directly off our computer

to some production environment.

Instead, I'm gonna give you a real production

workflow that you can use at a real company,

with multiple different teams working on the same code base.

So, we're not gonna take a shortcut here,

we're gonna go through a real process

and understand how we can kinda work with multiple teams

to eventually deploy our stuff in some reasonable way.

All right. And the last one, and this kind of relates

to a earlier note here around the remote entry file.

Right now, the remote entry file name is fixed.

In other words, it is always remoteEntry.js.

So we need to think about some caching issues around this.

If caching issues here doesn't make sense, don't worry,

we're gonna go into great detail on that.

In addition, I say "at present" here,

I mean "at present" in terms of

the Webpac module federation plugin.

There is work being done to make sure

that we can have a dynamic name

for the remote entry file, but that's not yet complete.

So the requirement with the current version

of the Webpac module Federation plugin,

is that we do have to have a fixed name.

So when I say, "at present" here, in other words

I don't mean to say that you and I are gonna figure out

a way to change that.

We are always gonna have a named file

of remoteEntry.js

And that might change at some point in time in the future.

There might be a way of getting around that,

but right now we have to have a file with that name.

So we need to think about some caching issues around that.

And again, we'll talk about exactly what I mean

by caching issues, if that doesn't make any sense.

All right, so those are our big requirements.

So now that we understand some really big things

we need to consider here

as we start to deploy our application;

let's talk about what the deployment process actually is

in the next video.

Topic 49. The Path to Production

-: Let's take a look at the overall end to end flow

that we're going to use to deploy our application

and access it in production.

Okay, so here's the general idea.

We currently have all of our different sub-projects

inside of one directory.

We're gonna set up a git repository in that folder.

We're gonna refer to this as a Git Monorepo because

we simply have a lot of different sub-projects

inside of one single repository.

Now, I want you to know

that you do not have to use a Monorepo if you don't want to.

You can create a separate Git repository

for each sub project.

I highly recommend, as we go through this course,

that you create a Git Monorepo.

I just mention this because you can use the exact same flow

that I'm going to show you, even if you have separate repos

for each distinct project.

In this course, we are only really using a Monorepo

so that we don't have to create and set up

a bunch of different repositories,

which would just take a lot of time,

and not really be for any great benefit.

So, we're gonna set up a Git Monorepo.

It's gonna include all four of our sub-projects.

We're then going to eventually push all of our code

up to Github.

Now, this entire pushing process, there's gonna be

a little bit of detail in there that we go into

but for right now we're pretty much just going to say

we're gonna push all of our code up to Github.

When we do so, we're going to have some script on Github

that takes a look at each sub-project.

And, when we look at each sub-project

we're gonna ask ourselves, okay in this latest push,

in this latest version of all this code that we're getting

were any changes made to the container?

Were any changes made to marketing, dashboard and auth?

If any changes were made to these projects

then we're gonna start a follow up process.

So, for example, if we made any changes to the container

then we're going to kick off a process

that will make a production version of our container app

and we're gonna build that production version using webpack.

Once we build that production version we're then going to

upload all these built files to Amazon S3.

And, we'll talk about what goes on after that step

in just a moment. First, I just want to clarify

that each of these question and build processes right here

are independent of each other.

So, we can answer yes to this question right here

and do the following build steps and still at the same time,

answer no to marketing and not do those build steps.

So, when we ask these series of questions

we're only going to try to build each project

that actually had some changes made to it.

And if we did not change some project, no problem,

we're not gonna build it, we're not gonna deploy it

or anything like that.

So, that's how we're going to make sure that we can deploy

each of these different sub-projects without requiring

a deployment of any of the others.

Okay, so like I said, we eventually end up

uploading all of our files to Amazon S3.

So, what is S3 all about?

Well don't sweat it, we're gonna get

a lot of details about S3

and exactly what we're doing with Amazon, in general.

Ultimately, Amazon S3 is gonna hold the built version

of all of our different sub-projects.

So, inside of this S3 thing, we're gonna have

all the different files from our container application,

such as the main.js file, which you will recall,

is one of the files that gets created by webpack

and our index.html file.

We'll also have our marketing with the main.js file

and all the other files that are required such as

the remote entry and the dependency files,

all that kind of stuff.

At some point in time, a user is gonna try to navigate

to our application inside their browser.

When they do so, we're not gonna serve them up files

directly from Amazon S3.

Instead, the browser is gonna make a request

to something called Amazon CloudFront.

CloudFront is a CDN or a content delivery network.

There are a lot of reasons we make use of a CDN

and just as we're gonna discuss Amazon S3,

we're gonna talk about why we are using CloudFront as well.

CloudFront is gonna see the incoming request and then

figure out which files to pull out of our Amazon S3 bucket

and serve those back up to the browser.

So, as we've discussed many times now,

what's gonna essentially happen is

the same thing that happens in the development environment.

We're gonna send over the index.html file to the browser.

That's gonna have some number of script tags inside of it.

The first script tag is probably gonna say to load up that

main.js file right there.

That main.js file is gonna tell the browser

to go get some remote entry file from marketing.

The really remoteEntry file and then that remoteEntry file

might say oh yeah, you need to load up the main.js file

from marketing that has the code you're looking for.

Though it's essentially the same process

that we have in development right now,

we're just adding in these additional parts

to actually serve these files up.

Okay, so that is the general overall flow.

Now, last thing I want to mention here very quickly,

we are making use of AWS.

So, you will need, absolutely, if you want to go through

all of this deployment stuff, an AWS account.

AWS does require you to have a credit card to sign up.

However, everything that we are doing around Amazon S3

and CloudFront is gonna fit underneath the free tier,

in other words, you should not be paying any money.

Even though you do have to have a credit card

and you have to enter it,

you should not be getting charged a dime.

However, I can't guarantee that, per say.

You might have an old AWS account

that is outside the free tier.

You might leave all this stuffing running

for a really long time and forget to turn it off.

That's technically not an issue here.

There is no like, constant fee

that you're going to be charged but

I will show you how to shut everything down.

I just mean to say that if you do get charged for something

I just can't guarantee that you won't, per say.

I'm not gonna offer that guarantee.

Okay, so that is the general idea, that's the general flow.

So, we've got a lot of work to do here.

So, let's take a quick pause and get started

on all this in the next video.

Formularbeginn

Formularende

Topic 50. Initial Git Setup

Narrator: Let's start to go through some

of the different Git related deployment steps.

A lot of these different steps only have to

be done exactly one time for some initial setup.

After we go through all these different steps

we'll be able to deploy our project

with just one or two very quick commands.

Now, you'll notice very quickly that I'm going to assume

that you are already familiar with GitHub.

I'm going to assume that you already have a GitHub account

and that you're generally somewhat okay with Git in general.

So with that in mind, let's get started.

First thing we're going to do is create a new repo

on GitHub.

So inside my browser I will go to github.com.

Once there, on the top right hand side

I'm gonna click on the plus and create a new repository.

I'm go to call this repository simply mfe.

Short for micro front ends.

You can call it absolutely anything you want to call it.

I'm going to set the repository as public,

not strictly required.

You can do private if you want to

but I'm gonna set mine as public, and I'll create the repo.

Okay, I now got my link right here

and really I want the SSH one,

so I'm gonna hold onto that for just a moment.

In the meantime we're gonna go do some local setup

on our machine,

make sure we create a git repo and do some other setup.

So back inside of our terminal,

I'm gonna open up a new terminal window,

I'm gonna change into our mfe, or sorry mfp

is what we called it, directory.

Inside of here I'm gonna do a git init,

and I'm going to create a .gitignore file.

I'm gonna do that over at my code editor

just to make the process really clear.

This .gitignore file will make sure

that we do not commit certain sets of files inside

of our project, such as say the node modules directory.

So inside of mfp, or our root project directory,

I'm gonna create a new file,

called .gitignore

and inside there I'll put in node\_modules.

I'm also gonna put in

dist because you'll notice that sometimes

when we've been been running webpac to build our project

it's been creating a dist directory.

We do not really want to commit that stuff

to GitHub, or Git really.

So I'm going to add that in there.

You can also add in stuff if you're on macOS, for example

like your .DS\_Store file, that ignoring, annoying file

that always seems to pop up everywhere.

Okay, so there's our .gitignore file.

So now we can go back over to our terminal

we can do a git status and just verify that we

have created the .gitignore file in the correct location.

We'll then do our first initial commit.

So we'll do a git add, git add . to be specific

and then a git commit -m

and we'll give it a message of initial commit.

Okay, there we go.

Now that we've done our initial commit

we need to link this repository

to the GitHub repository that we just created.

So for that I'm gonna go back over to the browser.

I'm gonna take that URL.

And then back at my terminal

we'll do a git remote add origin

and I'll paste in that URL.

And then finally, we're gonna push our code up.

So I'll do a git push origin master.

Just so you know, at the time that I'm recording this video

GitHub has started to move

towards calling the primary branch main instead.

So if you want to call it main, totally fine as well.

I just have not changed out to call this thing main.

Totally fine if you wanna change master to main

no problem whatsoever.

Okay, now we've pushed all of our code

up and we should be able to flip back over to GitHub

refresh the page, and see our code start to appear.

Very good.

Okay, so that's the first couple steps here.

We have set up the repository,

we set up our local git repo, and we did just

a very quick test to make sure we could push our code up.

Now we're going to take a quick pause, come back

and start to set up webpac with a production config.

So let's take care of that in just a moment.

Topic 51. Production Webpack Config for Container

Instructor: Let's start to set up

our webpack production config for our container application.

This config is gonna be very similar

to the development one we already put together.

But there's gonna be one or two little small considerations

around it.

So to get started, I'm gonna go back over to my editor.

I'll find the container directory.

Inside there is config and inside there we've got our

three different webpack config files.

So we're going to be adding in in a lot of configuration

to the webpack.prod.js file.

Now just a quick reminder here,

when we actually run webpack,

we're going to either be merging the common

and dev file together in our dev environment.

And in production, we'll be merging together the common

and production files.

So some of the config for production has already

really been written out for us inside that common file.

Okay, so inside of our prod file, let's get started.

We're gonna first require in the merge function

from webpack-merge.

And that's what's going to allow us to kind of merge

or smash together the common and the prod.js files.

We'll then require in our ModuleFederationPlugin.

Once again, from

webpack/lib/container/ModuleFederationPlugin.

After that, we'll get our commonConfig

from the same directory, webpack.common.

And then our package json file

coming from up one directory package.json.

And you will recall that we are requiring that in

so that we can eventually have webpack just

take care of all the shared modules for us.

Now inside of here, we'll create a new variable

called prodConfig.

Set it to an object.

The first configuration setting

that we're going to put inside of here

is a mode of production.

When we set mode to production

that's gonna cause webpack to run slightly differently.

It's gonna make sure that all the JavaScript files

that are built get somewhat optimized.

It's gonna mini file them

and do some other small optimizations.

Takes a little bit longer to run webpack in production mode.

However, it's gonna make sure that we get a much more

production-specific build coming out.

After that, we're gonna set up

another section inside of your called output.

Inside of output we're gonna put in file name.

We're then going to provide a string

and put in a square set of brackets with

[name].[contenthash].js.

So this ensures that whenever we build some files

for production, all the different files that are

built are gonna use this as a template

to figure out how to name them.

We're gonna first put down the

name of the file that was created, and then a hash

of the contents of the file.

This is done primarily for caching issues.

And you'll recall, we're gonna have a longer discussion

around caching later on.

After that, we'll set up our list of plugins.

So the only plugin we have to wire up here

is the ModuleFederationPlugin.

We already set it up once inside the dev.js file.

The reason we're setting it up here again,

is that we want to make sure that we have

a copy of ModuleFederationPlugin

with some production specific settings to it.

So for example, we need to make sure all the

remote entry file URLs are gonna point to some

actual production domain, as opposed to the local hosts

that we're pointing at inside of our development

webpack and config files.

We'll put inside of here, new ModuleFederationPlugin.

Put in a name of container.

Remember, not strictly required

because this is a host module that we're in

and you do not have to provide a name for a host module.

But a lot of documentation and whatnot, says

you should put that in, regardless.

It doesn't actually say that you should.

It just kind of shows it

and it doesn't really say why it's there,

but just, you know, not required for a host module.

After that, we'll put in our remotes.

So where we should go to to get some source code.

Right now, the only remote we have is marketing,

and remember that key right there needs to match up

to the first part of some import statement

inside of our container project.

Then our string that we're gonna put right here will be

'marketing@'.

And then remember, usually we would put in

a domain right here of where we should go

to get our remote entry file for the marketing project.

And this is where that earlier note that I mentioned,

this one right here, it starts to come into play.

So right now, you and I have not created

any infrastructure on AWS.

We do not know where our application is gonna be hosted.

We don't know its domain.

So we have no idea where to tell our container application

to go to to get the marketing remoteEntry.js file.

So we have no value to put in for the URL

right here, right now.

However, we can do a little bit of setup

that's gonna make this a little bit easier in the future,

when we eventually know what that URL is.

Let me show you what we're gonna do.

Right above prod config,

I'm gonna create a new variable called domain,

and then set its value to process.env.PRODUCTION\_DOMAIN.

Quick reminder,

make sure process right here has one C on it.

So this is gonna be an environment variable

that you and I set up.

This environment variable is going to be defined

when we build our application through our CI/CD pipeline.

The environment variable is gonna contain a string

that says exactly where our production application

is actually hosted.

That's how we're gonna make sure that we eventually

get the correct remote entry URL inside of here.

When we actually go to deploy our application

and create all of our infrastructure on AWS,

we'll know what our production domain is.

And so then we can go back over to our CI/CD pipeline.

We can set up this environment variable, and then

whenever we run our webpack production build,

that value for the domain will be automatically

put into this string.

So I'm gonna replace the single quotes we have right there

with back ticks.

And then right after the @ sign, we'll put in

some interpolation for domain, like so.

That's gonna make sure we don't have to come back

to this file and change the domain a whole bunch

when we start creating our infrastructure.

Now, there is one other thing, or two other things

we're gonna put inside this string.

Remember that our container application is eventually

gonna have a remote of marketing and dashboard,

and authentication as well.

All these different sub-projects are gonna have

a remote entry file that we have to load up.

And all these different remote entry files

are going to be hosted at the same domain.

So we need to make sure that we can figure out

which remote entry file is which.

In other words, we need to make sure that we can all

identify these different remote entry files

and not have them like, override each other

or anything like that.

Long story short,

we're gonna make a really big assumption here.

We're going to assume that the remote entry file

for our marketing project is gonna be nested inside

of a folder at this domain.

We're going to assume that it's nested

inside of a folder called, "marketing."

So eventually, once everything is set up,

we should be able to go to our

{domain}/marketing/remoteEntry.js

and that should give us our marketing file.

Now, we have not done anything to guarantee this just yet.

We are making an assumption here and we need to make sure

that as we set up all of our infrastructure

we have that URL, or really, we have that JavaScript file

eventually located at this URL.

Okay, so that's it for remotes.

Last thing we're gonna do is set up our shared dependencies.

And once again, that's just gonna come from

packageJson.dependencies.

Okay, very last thing we have to do inside this file

for right now.

Remember, we have to merge together and export

our commonConfig and the prodConfig we just created.

So at the very bottom of the file, I'll do a

module.exports of merge.

And we'll merge together commonConfig with prodConfig.

All right.

Now, before we move on, one other thing we need to do

very quickly.

You might recall that for our container application,

we want to have a index.HTML file created

for our production deployment.

Because our container needs to have a production

or an index HTML file both for development and production.

Right now we defined that HtmlWebpackPlugin

inside of our webpack.dev.js file.

So that means that if we were to build our application

for production right now, we would not get an output

of an HTML file.

We need to make sure that we get out an HTML file,

both in development and production.

So for that, we can either copy this plugin

over to our prod file, or alternatively we can just

move the plugin setup into our webpack common file.

Because any config we put inside of common will be used

during both our production and development processes.

I think the easier solution here is to just

move the plugin into common.

Probably easiest way to handle it.

So from our webpackdev.js file

I'm going to cut the HtmlWebpackPlugin require statement.

I'm gonna go over to webpack common.js

and paste it at the top.

I'll then go back to my webpack dev.js file.

I'm gonna go down to the plugins array,

and I'm going to cut new HTML webpack plugin.

Then back over to common.

I'll go down to the very bottom of this object.

We'll put in our plugins array

and paste in the HTMLWebpackPlugin.

All right, one more very small thing I wanna do.

We need to add a script to our package.json file

to make sure that we can actually build everything

for production.

So inside of my package.json file,

I will find the script section.

At the very end of the start command.

I'm gonna add in a comma,

a comma right there at the very end.

and we'll add in a new script called "build".

And whenever we run build, we will execute webpack.

We're not gonna put in serve.

We're not serving anything up here.

We're just trying to build our project.

So we're gonna put in

webpack --config config/webpack.prod.js.

And that should be it.

So in theory, we can now make a production build

of our container application.

I would like to go over to our terminal very quickly

and test out this build command,

just to make sure that we wrote the command correctly,

and make sure that we don't have any typos

inside of the webpack prod file.

So back at my terminal,

inside of my container directory,

I'll do an npm run build,

and then we just wanna make sure we don't get any errors

or anything like that.

So when I run this

I see that everything was built successfully.

If you see any errors,

chances are you have a typo somewhere.

So you'll want to hunt that down.

Once we build our project, we can then go back over

to our code editor and we should see inside of our

container directory a new dist folder.

So this folder right here contains all

of the output for our production build.

We can see our main.js file with a content hash on there.

We can see all of our different

dependency files inside of here, as well.

And there's our index.html file.

If we open up the index.html and scroll over to the right,

you'll notice that once again,

we have gotten our different script tags

automatically injected inside of here.

Okay, so looks like our production build is good to go.

I'm going to delete that dist directory,

as it was just a very quick test.

And we are all set.

Now, we're gonna repeat the same process

inside of our marketing project, but it's gonna be,

don't worry, way faster since now we have a really good idea

of what we're doing here.

So quick pause and we'll take care of marketing.

Topic 52. Production Webpack Config for Marketing

Instructor: All right, let's repeat the same process,

more or less, over inside of marketing.

Like I said, the last video, we're going to go

through this entire process much more quickly.

So inside of our marketing, config,

webpack prod dot js file,

we're going to again get our merge function

from webpack merge.

We will get our ModuleFederation plugin

from webpack lib container ModuleFederationPlugin.

Get our packageJson from up one directory

package dot json.

And then finally, our common config

from same directory, webpack common.

Okay, after that, we can create our production

or just prodConfig is what we'll call it.

Inside of here, we're going to set up

our mode as production.

We're going to do that same kind of output thing,

where we name all the different files that come out.

So we'll put in output and we'll specify file name.

Notice the lowercase, and right there

with brackets name, brackets contenthash,

all lowercase dot js.

After that, we'll set up our plugins.

We'll put in new ModuleFederation

plugin.

Inside this configuration object,

once again, we've got to throw in some stuff

like the name, all that kind of good stuff.

We'll put in our name of marketing.

And just as before, we also need to make sure

that we specify exactly what modules we want to export.

So specifically what files we should provide

whenever someone asks for some particular file.

So for that, we'll put in exposes.

And I want to make sure I use the same exact name

for exposes as what we did inside of our dev file.

So inside of webpack dev dot js.

I just want to double check the name we put right there.

Because I don't recall, we ended up with,

it was MarketingApp.

We're going to put in dot slash MarketingApp.

And whenever someone asks for that file,

we're going to actually give them source bootstrap.

Let's not forget our file name of remoteEntry dot js.

Kind of forgot that one right there.

And then finally, we can also put in

our shared dependencies, which will be

packageJson dot dependencies.

And you'll notice that in this case, we have ended up

with a ModuleFederationPlugin right here

that is pretty much identical to our webpack dev,

really the dev edition of this ModuleFederationPlugin.

So in theory, we could extract this plugin right now

into our common module.

However, there might eventually be some scenarios

in which you want to have some slightly different

configuration between how you build up

your ModuleFederationPlugin

for development and production.

We'll take a look at what some of those

different scenarios might be in the future.

But for right now, we're just going to leave

the two copies of the ModuleFederationPlugin,

one inside of dev, and one inside of prod,

even though they are technically identical right now.

Okay, so after all that,

at the very bottom, we'll make sure

that we merge together our common and prod config.

We'll do a module exports and merge together

our commonConfig and the prodConfig.

After that, let's go to our package dot json file.

Once again, got to create a build script.

I'll make sure I get the comma at the end of start.

Our build put in webpack, no serve required.

Just config, config slash webpack prod dot js.

Once again, I would like to run the build script

very quickly, just to make sure we didn't make any typos.

So back at my terminal, I'll change on over

to our marketing project

and do an npm run build.

Looks like it was all successful.

I can see the dist directory

and it looks like we've created

all the different inquired files.

Okay, so I'm going to delete dist

because we don't need that.

Just want to make sure everything worked as expected.

And I think we're all done

with our webpack production setup.

So that is pretty much it for that step.

Last thing we have to do around this initial Git and GitHub

deployment steps, we need to start to set up

this CI and CD pipeline.

That's going to be a little bit involved,

but don't worry, it won't be that bad.

So let's take a quick pause right here

and get started on that in just a moment.

Topic 53. Understanding CI:CD Pipelines

Instructor: Before we move on

to our AWS side of deployment,

we have to start to set up a CI and CD pipeline.

This is going to be a series

of steps that are going to be executed automatically,

every single time we make a change

to one of our subprojects,

and then push those changes up to GitHub.

Our ultimate goal is to make sure

that anytime we change one of our subprojects,

we have some process or some code somewhere,

that is going to automatically run the production build

of the package that changed,

or the subproject that changed,

and then automatically attempt

to deploy that change off to our AWS infrastructure.

To implement our CI/CD pipeline,

we're going to be making use

of something called GitHub Actions.

And that's what we're gonna focus on in this video.

So let's take a look at how we can use

this GitHub action feature to implement our CI/CD pipeline.

Okay, so whenever you create a GitHub repository,

you can set up some different GitHub Actions.

These are essentially little bits of code

that you can have automatically execute,

anytime some event occurs to your repo.

For example, you can configure some code

to run automatically

anytime that you commit some code on your machine,

and then push it up to GitHub.

You can run some code automatically,

anytime a pull request is created or closed,

maybe on some set schedule,

maybe anytime an issue is created or closed,

there are many different events you can watch for.

So anytime one of these different events occurs

to our GitHub repository,

we can choose to run a set of workflows.

These workflows are essentially little snippets

of code that we can set up ahead of time.

So when we talk about setting up a CI or CD pipeline,

we're essentially saying,

we want to set up some code to run automatically,

through GitHub, any time we push some changes,

or maybe some other event we care about.

So for you and I,

we're gonna be setting up a variety of workflows,

four in total, that look pretty much like this right here.

Each workflow is going to be all about one

of our individual subprojects.

So we will have one distinct workflow

for deploying our container,

one for marketing,

one for authentication,

and one for the dashboard.

Inside of each of our workflows,

we are gonna watch for this action to occur.

We're going to wait for whenever we get some code

that is pushed to our master or main branch,

depending upon how you named your primary branch,

and whenever that commit contains a change

to whatever project this workflow is responsible for.

So if we are creating a workflow for our container,

we're gonna make sure that whenever we commit a change

to the container folder, we want to run this workflow.

So this workflow is gonna be a series of different commands,

that are going to be executed on a virtual machine.

This virtual machine is automatically created, destroyed,

cleaned up, all that kind of stuff,

by GitHub for us automatically.

So we really don't have to do any kind of excess setup here.

We pretty much just have to say whenever this event occurs,

here are the series of things we want to do.

So inside of our virtual machine

that's gonna be created for us,

we are going to first change into our container folder,

we're then going to attempt

to install all the dependencies into our project.

This step right here, just to be clear,

remember, we are not committing any of our dependencies,

or essentially that node modules directory,

into our GET repository.

When we are running this workflow

on some remote virtual machine,

we don't have any dependencies inside

of our project by default.

We have just our source code

and stuff like the package.json file and so on.

So we need to make sure that

if we want to build our project,

we have to install all the dependencies specifically

for the container subproject.

We're then going to try to create a production build.

So that's gonna create that "dist" directory,

short for distribution,

along with the HTML file in there,

and all the other built JavaScript files.

So we're then going to take that "dist" directory,

we're gonna take all the results,

and we're going to upload them off to AWS S3,

which is where our hosting is really occurring.

So this is the general strategy that we're going to use

for our workflow,

and we're going to replicate this four different times,

once for each subproject.

Alright, so now that we've got a better idea

of what our goal is,

let's start to implement our container workflow

in the next video.

Topic 54. Required Change in the Container Action - Do not Skip

**Required Change in the Container Action - Do not Skip**

In the upcoming lecture, we will be writing our container workflow and using the chrislennon/action-aws-cli@v1.1 action. Unfortunately, this is now failing and appears to be no longer maintained. A community fork was created to fix the issues which we can use instead:

instead of this:

      - uses: chrislennon/action-aws-cli@v1.1

write this:

      - uses: shinyinc/action-aws-cli@v1.2

This updated action will require an **AWS\_DEFAULT\_REGION** key, so, for now, we can just add a placeholder.

1. - uses: shinyinc/action-aws-cli@v1.2
2. - run: aws s3 sync dist s3://${{ secrets.AWS\_S3\_BUCKET\_NAME }}/container/latest
3. env:
4. AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}
5. AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}
6. AWS\_DEFAULT\_REGION: ""

Also, make sure to verify that you are using the correct branch name in your workflow. GitHub now prompts you to name this **main** by default.

Topic 55. Creating the Container Action

-: Let's start to create our first workflow.

This first workflow is gonna be all

about deploying our container application.

There are two ways we can create a workflow.

We can either go to our GitHub repository

and then find the actions tab right here, click on it

and then you can use one of these pre-setup workflows

or you can try to build one yourself.

If you try to build one yourself

it's gonna take you to a little code editor right here

and you can see that they give you a couple of hints

and comments about what you might want to do inside of here.

If you do not want to use this built-in editor,

then you can just write

out a config file directly into your project.

We're gonna do that.

We're not gonna use the built-in editor.

The reason is, well, no great reason.

I personally just like using my own code editor

as opposed to the one that's built into GitHub.

Now, to create a workflow, we're gonna make a very

specially named directory inside of our project.

We're gonna make inside of our route project directory

a new folder called dot GitHub.

Inside of dot GitHub, we're gonna make a very

specially named folder called Workflows.

Then inside of there, we're gonna make our first workflow

which we will call Container dot YML.

Anytime that you create a YAML file

inside this dot GitHub slash workflows directory

and then push that file off to GitHub,

GitHub is going to assume that this is some kind

of workflow that you're going to want to execute

at some point in time.

So all the files we create

inside this directory are gonna be very special in nature.

They're going to be automatically picked up

and read by GitHub for us.

As a matter of fact, if you go back over

to that built-in code editor, you'll notice

that the file that is being created right here

is being created inside

of dot GitHub slash workflows as well.

Again, we wanna make sure

that we create this file inside that directory.

Then inside of here, we're gonna put

together a pretty good amount of configuration.

All the configuration is gonna be written

out with YAML syntax.

YAML syntax is somewhat similar nature to JSON.

It's technically, you can take YAML

and convert it into JSON

but the actual formatting of it is very different.

All the formatting is tab based.

So all the tabs you're gonna see inside this file

that I add, all the indentation has very specific

and special meaning to it.

So make sure that you add in

all the same level of tab as I do.

A lot of the config that we're gonna add inside

of here is very particular

to just GitHub and how GitHub actions work.

It's not super important that you understand a lot

of the kind of theory

behind this stuff because if you don't plan

on using GitHub for your microphone and application,

well obviously then a lot

of the specifics here aren't super important.

So for the most part,

we're going to just write out a lot of configuration.

I'll give you a brief description of each of the steps

but for the most part we're just trying to

get the config inside of here.

The first thing we will do is provide a name

for this workflow we are putting together.

I'm gonna call ours deploy dash container

'cause our goal is to build and deploy the container.

After that, we're gonna add in an on-property.

This gonna designate when we want to run this workflow.

So essentially, what events are we watching for?

We're going to watch for a push event.

We're going to watch

for pushed events only on a branch of master.

If your primary branch that you are using is main,

then you can change that to be main instead.

Where alternatively, if you just wanna watch for pushes

to either master or main, you can put in both them like so.

I think right now I'm still using a primary branch

of master, so I'm gonna put in master.

After that, we're then going to also specify

that we only want to run this workflow whenever

we make a change specifically to our container project.

So for that, we're going to add on the same level

of indentation as branches, hats,

and then a dash packages slash container

slash star star, like so.

Again, this is specifically what says only

run this workflow when we make a change

to something inside of container.

Okay? After that we'll go back

to our top level of indentation.

We will put in a default section,

run,

and then working dash directory.

And the value for that will be packages slash container.

So this is essentially going to set our kind

of execution environment to inside

of our container directory.

All the different commands that we're going to run

in a series of commands we're gonna write

out down at the bottom of this file, are all relative

or should be executed inside the container folder.

So we want to specifically run something

like say MPM run install

or something MPM install, where MPM run build

from inside the container directory.

So all this is doing is kind of changing us

into that container folder.

Then after that is where our actual work occurs.

We will put in a job section.

We're going to create a job inside of here called Build.

And as you guess build is going to build

and then deploy our project.

You can have multiple different jobs, but they run

by default all in parallel.

We could technically make one job just to build our project

and then maybe another to actually deploy it.

But in our case, we're just gonna make one single job

that's going to first build and then deploy the app.

Inside of here,

we're gonna specify runs dash on, ubuntu dash latest.

These virtual machines that are created by GitHub

you can have many different flavors of virtual machines.

So you can have a virtual machine that runs Windows.

We're just being very specific here and saying

we want a Linux-based virtual machine.

After that, we'll specify our step section.

So this is where we write out the actual commands

and whatnot that we actually want to execute.

The first thing we're going to do,

I didn't actually label on our diagram.

We have to actually check out the code from our project.

So we need to actually take our code inside

of our repository and check it out or essentially load it

into this virtual machine environment.

For that we're gonna put in

uses,

colon actions slash

checkout

at B2.

After that, we're going to run a command.

We're going to run NPM install.

I think you can guess what that does.

That's gonna download

or install all dependencies into our project.

We're then going to run NPM run build that's going to

run our webpac build process,

generate that disc directory,

and then after that we're gonna start

to do some configuration

that right now is not gonna make a lot

of sense until we go and set up some of our AWS stuff.

So after we build our project, that's where we

then want to deploy all these built files off to Amazon S3.

For that, we're going to use something called the AWS CLI,

which is essentially gives us programmatic access

to Amazon Web Services.

To get access to the CLI, we're gonna put in a dash uses

Chris Lennon

slash action dash AWS

dash CLI

at the one point one.

Again, that's gonna give us access to the AWS CLI.

We're then going to use the CLI to run a command.

We're going to run AWS.

We're gonna reach out to the S3 feature specifically

inside that CLI, and we're going to sync our distribution

directory that was created when we built our project.

Remember, whenever we run NPM run build

that creates a disc directory.

So we're gonna sync the contents of that directory

with an S3 bucket or essentially a place where

we can store files on AWS.

Let's just write out the code for this.

And again, a lot of these different steps will become

clear as soon as we start to set up AWS.

So then put in S3,

colon slash slash

and then we are going to specify the name of the bucket

that we want to sync this project into.

We're gonna write in dollar sign,

curly brace, curly brace.

Notice that two sets of curly braces,

secrets dot AWS underscore

S3 bucket name

then slash container

slash latest.

So again, I know a lot of this stuff right here

possibly very mysterious.

We are gonna come back and get a better idea

of what that's all about.

We're not just done with this command just yet.

We're also going to add in on the same level of indentation

as run without additional dash ENV,

and then AWS,

access,

key ID,

another dollar sign, curly races

secrets dot AWS

access key ID.

And then on the same level of indentation,

AWS secret access key.

And once again,

a secrets dot AWS secret

access key.

And that's it.

Okay, so as you can see

got a somewhat straightforward workflow here.

Install dependencies, build a thing

and then try to sync all these files off to AWS S3.

So at this point in time,

we've got our workflow put together

and now the really big thing we need to do here

is make sure we start to set up AWS

and create some bucket where it can actually

deploy our application too.

I'm going to assume

that you might not have used AWS too much in the past,

so we're gonna get a pretty reasonable dive on AWS

understand what S3 is all about,

and understand what all these extra parameters

in here are doing for us as well.

But right now, let's just save this container dot YAML file

and take a brief pause.

We'll come back the next video

and continue on in just a moment.

Topic 56. Testing the Pipeline

Narrator: We just put together our first git hub action.

I would like to test this action out very quickly.

Now of course, the action is going to eventually fail,

because we have not set up anything around AWS just yet.

But I just want to make sure that the action will run,

and that we can at least successfully install dependencies,

and then build our project.

So to get our action to run,

we have to commit all the changes we have made

to our project, and then just push it up to git hub.

So back over at my terminal,

I'm going to change into my root project directory of mfp.

I'm going to do a git add . once again, a git commit,

and then this time for the commit, I'll say something like,

"Created container deploy script".

We'll then push this off to git hub,

so git push origin master.

Or main, again, if that's your primary branch name.

So we'll run that.

And then we just get kicked right back over

to our command line.

So there's not really any immediate indication

that anything is actually occurring.

To actually see our workflow running,

we're gonna go back over to git hub.

So back at git hub inside my browser,

I'm gonna find the actions tab.

And then over here, we should now see our script right here.

Or, at least the commit up here.

So there's the commit message.

We are running a workflow called 'deploy-container'.

If we click on this, we're gonna then click on 'build'.

And we'll see some immediate output from the different

steps that are being executed.

Now, remember, this thing is going to eventually fail

without a doubt, because we have not set up

any of that aws stuff.

But hopefully we should at least see some of these

initial steps around installing our dependencies,

and then building the application's exceed.

So I can see that it looks like

our npm install has ran successfully.

I can also see that we have also ran

npm run build successfully as well.

We might see a, once again,

deprecation or two inside of here,

but overall it looks like it's all good.

Now the reason I point out this input in general,

or this output right here,

is that sometimes webpack will fail,

but not fail the overall build.

In other words, webpack might fail to build our application,

but tell git hub actions that everything worked as expected.

So sometimes it might seem like the entire build

worked successfully, but you might not actually see

any changes being deployed.

So just keep that in mind, if it seems like

nothing is actually occurring that you would expect.

Always double check your npm run build step,

and make sure that webpack is not throwing some kind

of error inside of here.

In this case, I see that we've got

compiled successfully in blah seconds,

so it looks like the build ran successfully.

We eventually install the aws-cli,

but then when we try to actually run our command

to sync all these files, or deploy our new code,

well, naturally, we ended up with an error message.

Totally fine, we'll come back and fix that eventually.

Okay, so it looks like we've got

our first working git hub action.

So at this point in time, we can either go and create

a second action to watch for changes that are made

to our marketing project,

or we can go off to aws and start to setup our

infrastructure over there.

I think we should probably go over to aws,

rather than just focusing on this script stuff too much.

So, let's start working on our aws infrastructure

in just a moment.

Section 7: Deployment to Amazon Web Services

Topic 57. S3 Bucket Setup

Instructor: We're now gonna start to work on creating our

infrastructure on Amazon AWS.

In this video, we're gonna focus on creating an S3 bucket.

You can think of an S3 bucket as essentially being

like a hard drive of sorts.

We can create some different files inside there,

and then we're going to eventually make those files

available for download through another service called

Amazon CloudFront.

Let's first focus, however, just on creating this S3 bucket.

So to get started, I'm gonna go over to my browser

and navigate to the AWS management console.

You can find this at aws.amazon.com.

At this point in time, as you can kind of guess,

I'm really assuming here that you have an AWS account.

So if you do not have one,

absolutely pause the video and create an AWS account.

Just a reminder, you will need a credit card

to create an account, so don't forget to have one handy.

Once you sign up or log in,

we're going to do a search for a service called S3.

We'll then click on the link and go to the S3 dashboard.

Once here, we're going to create a new bucket.

You should see a button somewhere on the page that says

something like Create Bucket.

I'm gonna click on that, and then we're going to add

in just a tiny bit of configuration here.

The first thing we have to do is add in a bucket name.

So this is an identifier for our bucket.

It can be absolutely anything you want it to be,

so long as it satisfies AWS's own rules.

For me, I'm gonna call mine MFE, how about Dashboard?

Good enough.

All the different bucket names across AWS have to be unique,

so chances are you will not be able to use MFE Dashboard,

because I'm already using it.

So you can think of any name you want to use.

Again, just feel free to use whatever you want.

After entering the bucket name,

we're then gonna take a look at the region dropdown

right here.

You can change the region if you want to,

but I would encourage you to just leave it at whatever

default is selected for you.

The only thing you need to know about this region dropdown

is that we're going to eventually need to know the

little location identifier, which is the right most part

of the dropdown.

The US-East-1 is what it is for me.

I would encourage you to write down that little identifier

or just keep it in the back of your head.

We're going to need that identifier a little bit

in the future.

After that, we're gonna go down to the very bottom

of this page and click on create bucket.

All right, so we've created the bucket.

Just creating it isn't quite enough.

We're gonna go back into the bucket now and update some

of its configuration.

So I'm gonna search for a bucket of MFE dashboard,

'cause that's what I just called mine, and then open up

the configuration page for that bucket.

Just so you know, Amazon S3, by default, is designed to

store files securely.

So by default, none of the files you add into a bucket

are supposed to be publicly available online.

However, we are using a bucket in a very different way.

We are using it to host some files that we absolutely

want to make available for public access.

We want anyone to be able to come to our website

and be able to fetch some, say,

HTML file or JavaScript files from our bucket.

To do so, we need to update some of the configuration

settings on the bucket.

You'll notice even right now it says something like

bucket and objects not public.

So that means nothing inside of here can be accessed by

any random person online.

So to allow random people to get access to these

different files we're gonna add inside of here,

we're gonna update two different settings.

First, we're gonna go to the properties tab right here.

We're then going to scroll down all the way to the bottom

and find a section that says static website hosting.

We're gonna click on edit,

on this page, we're going to update this little selector

to enable,

we're gonna make sure that we have

host a static website selected.

For index document right here,

we're gonna add in index.html.

Now, we're not going to actually have an index.html file

in the route of our bucket.

This setting right here is actually going to

be eventually overridden by some other settings

that we're gonna add in down the line.

Specifically, when we start to add in some configuration

around that other CloudFront thing

that I mentioned very briefly.

After that, we'll go down to the very bottom

and click on Save Changes.

Save changes, come on.

All right, oh, my mistake.

We have to add in an error document right here.

We're gonna use index.html.

It used to be that you didn't have to add that in.

That was actually optional,

and as a matter of fact, the documentation for S3

actually says that's optional as well.

But I guess that's not the case anymore.

Okay, now I'm gonna click on Save Changes, and there we go.

So I see the success message at the top,

but like I mentioned,

this is not the only thing we have to do.

We have to make one other small change.

So we're gonna scroll down to the list of tabs right here,

once again.

We're going to go to permissions.

On permissions,

we'll find block public access and click on the edit button.

Then on this page, we're going to uncheck

block all public access, and we should see no check marks

on here whatsoever.

Now, as soon as you do this, you're gonna start to see

a lot of warning messages appearing all over the place

as soon as you take a look at this bucket.

and all these warning messages are going to say something

like, "Hey, all these objects are publicly accessible."

That is totally okay.

This is desired behavior for us.

Again, we are putting up some JavaScript and HTML documents

that we want to a hundred percent make available to anyone

online who's trying to access our website.

The reason that you're gonna see all these kind of

security warnings and whatnot is just because Amazon really

wants to make sure you are a hundred percent aware

that the files inside this bucket are publicly accessible.

In their eyes, it's a lot better to just be very cautious

and let you know what's going on as opposed to

not saying anything at all.

So once I've confirmed that nothing inside of here

is checked, we'll go to save changes once again.

And you'll see they're very, very serious about this.

So we do have to type in confirm, and there we go.

Okay, so that is it for our bucket creation.

We've now created a bucket.

We have a location where we can start to upload all

of our built project files.

Again, we're not gonna serve necessarily

like these files directly outta this bucket.

Instead, people are going to ultimately make a request

to something else called Amazon CloudFront

and CloudFront is what is going to be responsible

for actually taking files out of our bucket

and sending them back over to the browser.

So we still have a little bit of setup to do.

Let's take care of that in just a moment.

Formularbeginn

Formularende

Topic 58. Authoring a Bucket Policy

-: Before we start working on

our cloud front setup

there's one other very small

change we have to make

to our S3 bucket.

I'm once again back at our S3 dashboard,

looking at the bucket we just created.

I'm going to again find the permissions tab

on this page.

I'm gonna scroll down past the

section we were just working with previously,

and I'll find the next section

that says Bucket policy right here.

I'm gonna click

on the edit button, and then on this page we're

going to find a button that says

Policy Generator.

When you click on that button

it's gonna open up a new browser tab.

We're gonna use this page to generate what is

called a policy.

A policy is what allows

multiple different services

on AWS to talk to each other

and access different features inside of them.

So in this case, we are creating a policy that is

going to allow our

cloud front distribution that we're

about to create to

access all these different files

that we're going to eventually

load into our S3 bucket.

Inside of step one right here

we're gonna find the dropdown that says

select type of policy

and change it to S3 bucket policy.

Then, a little bit lower,

we're gonna make sure that we leave

effect as allow.

for principle, we're gonna put in a star.

For actions, we're gonna open up that dropdown

and scroll down a pretty good amount

until you eventually find a setting that says

Get Object.

Here it is right here.

It's about halfway down.

I'm going to enable that one setting

and then close the dropdown.

Then finally, we have to enter in

an Amazon resource name.

This is the name of the Amazon or excuse me,

the S3 bucket that we want to eventually give

our cloud front distribution access to.

To get our resource name,

we're gonna go back over

to the tab we were looking at just a moment ago.

So here's the edit bucket policy window,

and we're gonna copy our bucket ARN right there.

Then back inside of our policy generator

are gonna paste it into that input.

And then very important,

at the very end of that string,

right after whatever the name of your bucket is

we will put in a forward slash and then a star.

After that, we'll click on add statement.

That's gonna take us to the very bottom

of the page.

And just verify that you've got some options

on here that look very similar to mine.

The only big difference is that the name

of your bucket should be

slightly different right there.

Okay. Finally, we'll click on Generate Policy.

We're gonna take all the text

that was just generated copy it,

take it back

over to our S3 edit bucket policy page

and I'm gonna paste it all into this

policy text editor.

Then finally, I'll go down to the very bottom

of the file and click on save changes.

All right, there we go.

So now, like I said in the last video

last big thing we have to do is do a little bit

of configuration and set up a

cloud front distribution.

So let's talk about exactly

what that means in just a moment.

Topic 59. Minor Changes in AWS CloudFront UI

**Minor Changes in AWS CloudFront UI**

In the upcoming lecture, we will be creating our CloudFront distribution. The UI for this service has changed slightly, so, here are some notes to try and avoid confusion:

1. In the video there are two choices, Web and RTMP. Since the RTMP method has been removed, you no longer have to explicitly select Web as it is the default and only delivery method.

2. Distribution settings have been renamed to Settings.

3. The SSL certificate fields look different and no longer show that the default CloudFront certificate is selected. This is ok, the default certificate is still being used and nothing needs to be done or changed.

All other AWS settings and configurations should track what is shown in the videos through sections 7 and 8.

There is an AWS cheatsheet with all steps to create an S3 bucket, CloudFront distribution, and IAM user that will be regularly updated here:

<https://www.udemy.com/course/microfrontend-course/learn/lecture/33274448#questions>

Topic 60. Cloudfront Distribution Setup

Speaker: We've now created our S3 bucket,

but remember right now there's no files

inside the bucket whatsoever.

We do eventually have to go back over

to GitHub and configure our action,

so that it can successfully upload all of our

different built project files to the bucket.

But first, before we do that, let's finish

up with AWS and create this Amazon CloudFront distribution.

That's what we're gonna create.

A distribution is essentially a set of files

we want to make available to the outside world.

To get started, I'm gonna go back over

to my browser and I'm going to open up a second browser tab.

So I still have the tab open where I'm looking at my bucket.

We're going to need some information off this page

in just a little bit, and that's why I'm gonna

open up a second tab still on the AWS management console.

Once here, I'm gonna search for CloudFront,

and head on over there.

Once here, we're gonna click on Create Distribution.

So again, a distribution is what is going to

allow us to distribute, as you'd guess,

some different files, and we are pulling these files

from the S3 bucket we just created.

Once you click on Create Distribution,

you're then going to be asked

if you want to create a web or RTMP distribution.

We definitely want a web distribution.

So we're gonna click on Get Started for web.

Then, once here, once again we're gonna have to add in

a couple of different configuration settings.

You'll notice that all the settings

in this case look a lot more intimidating

than the bucket experience that we just went through.

So do pay close attention, make sure you get

in all the same settings as I do.

The first thing we're gonna add in here is an Origin Name.

For this, we can click on that little input right there

and then we're gonna see a little popup

and we're going to see that we can select

from all the different S3 buckets you have inside

of your AWS account.

So inside of here, we're gonna scroll on through

and find the bucket that we had just created.

So here's mine: mfe-dashboard.s3.amazonaws.com.

Once I click on that, we don't have to

add in anything else inside of this section.

We're gonna scroll down a little bit

to the Default Cache Behavior Settings.

Inside of here, we're gonna find Viewer Protocol Policy.

We're going to change from HTTP and HTTPS

to Redirect HTTP and to HTTPS.

We're then going to scroll down a little bit more.

Under Distribution Settings, you're gonna see a section

that says SSL Certificate.

So eventually you might want to have HTTPS

along with a custom domain that you own.

If you want that, you're going to eventually have to

come back here and set up a custom SSL certificate.

For you and I, we're going to use

the automatically generated domain

that gets assigned to our distribution.

So it's gonna be some random identifier .cloudfront.net.

If you make use of the default domain,

then you're going to automatically get HTTPS.

It's only if you decide to purchase

some external domain name and point it

at this distribution that you have to

worry about these settings.

We're gonna keep on scrolling down a little bit more,

and eventually on the bottom right hand side,

we'll click on Create Distribution.

Now, we're not done just yet.

My table is a little bit strangely formatted here,

but after we create this, you're going to see something.

Our new line item on here and its status

will probably be something like In Progress.

So the distribution is currently being created.

It's gonna be a couple minutes

before we can actually do anything with this.

And we do actually have to update some settings inside

of here that we were not able to change when

we initially created the distribution.

So we're gonna pause this video right now,

we're gonna wait for this In Progress thing

to change to Deployed, and then once you see Deployed,

we're gonna continue on inside the next video.

So I'm gonna see you in just a minute.

Formularbeginn

Formularende

Topic 61. A Bit More Cloudfront Configuration

-: It looks like my distribution has been created.

So as I mentioned, we have to go back in there

and update one or two settings.

These settings we're gonna change

are gonna look a little bit mysterious right now

but as soon as our application is working, overall,

we're gonna do a little bit deeper dive

and understand what CloudFront is really doing for us

and why we are about to change these settings

that we're about to change.

So I'm gonna first click on that distribution,

and then on the general tab,

I'm gonna click on the edit button right there.

We'll then scroll down a little bit,

and we're gonna find a section called default root object.

Inside of here, we're gonna change that to,

slash container, slash latest,

slash index dot html.

We'll then go down to the very bottom and click on yes edit,

on the very bottom right.

After that we're gonna find another tab

that says error pages.

We'll click on create custom error response.

We're gonna change this dropdown to say 4 0 3 forbidden.

Change the radio button to say yes,

and then we're gonna put in the exact same

response page path as we just put it in

for the default root object.

So, slash container, slash latest,

slash index dot html.

Then for http response code,

we're gonna change that to 200: Okay.

Now this setting right here might look really mysterious.

We're essentially remapping an error.

We're saying, rather than ever throwing

some kind of error, just give a person an HTML file,

and change the response code to be 200.

Again, we're gonna do a deeper dive,

understand why this is.

I'm gonna click on create on the far right hand side,

and that is pretty much it.

Now last thing we're going to do is go back over

to the general tab back over here.

We're gonna find on the general tab,

something that says domain name.

That domain name setting right there is where

we can access our application at.

So we're going to eventually try to open up that URL

inside of a new browser tab,

but we're not gonna do so just yet.

The only reason for that is that we haven't

actually uploaded any of our different project files,

so there's really nothing to look at

at that domain right now.

Now, the very last thing we have to do is

kind of bridge the gap between our GitHub action and AWS.

We need to make sure

that once we have successfully built our application,

we then try to sync our project successfully over to S3.

That's the last big thing we need to take care of.

Let's tackle that in just a moment.

Topic 62. Key Creation Update + Reminder on

**Key Creation Update + Reminder on AWS\_DEFAULT\_REGION**

**Update for Generating Keys**

In the upcoming lecture, we will be creating an IAM user and then generating a key pair for deployment. There is a minor required change to this flow. Instead of being prompted to create a key pair during the IAM user creation, you must first create the IAM user, then, create a key pair associated with that user. AWS has also changed the terminology from Programmatic Access, to Command Line Interface (CLI).

Full updated instructions can be found below:

1. Search for "**IAM**"

2. Click "**Create Individual IAM Users**" and click "**Manage Users**"

3. Click "**Add User**"

4. Enter any name you’d like in the "**User Name**" field.

5. Click "**Next**"

6. Click "**Attach Policies Directly**"

7. Use the search bar to find and tick **AmazonS3FullAccess** and **CloudFrontFullAccess**

8. Click "**Next**"

9. Click "**Create user**"

10. Select the IAM user that was just created from the list of users

11. Click "**Security Credentials**"

12. Scroll down to find "**Access Keys**"

13. Click "**Create access key**"

14. Select "**Command Line Interface (CLI)**"

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

15. Scroll down and tick the "I understand..." check box and click "Next"

16. Copy and/or download the *Access Key ID* and *Secret Access Key* to use for deployment.

**Reminder on AWS\_DEFAULT\_REGION**

[A few lectures ago](https://www.udemy.com/course/microfrontend-course/learn/lecture/30888974#questions) we mentioned the need to use a different action and left a placeholder for the AWS\_DEFAULT\_REGION key.

Let's make sure this gets set correctly now.

In the AWS Dashboard use the Services search bar to find S3 and load its dashboard. Once there, copy the AWS region listed to the right of your bucket:



Then, in your container.yml, paste in the value for the AWS\_DEFAULT\_REGION like so:

1. - uses: shinyinc/action-aws-cli@v1.2
2. - run: aws s3 sync dist s3://${{ secrets.AWS\_S3\_BUCKET\_NAME }}/container/latest
3. env:
4. AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}
5. AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}
6. AWS\_DEFAULT\_REGION: us-east-2

Topic 63. Creating and Assigning Access Keys

Instructor: I'm back inside my code editor,

taking a look at our container.yml file.

Remember, this is our GitHub action workflow.

Inside of here, you might recall that we added in

a little bit of mysterious syntax.

Around this last step, that we had said try to sync

to our s3 bucket.

Let me put in this little URL looking thing right here.

Inside of it,

we had some double curly braces with a dollar sign.

We also had some a little bit lower

on the lines underneath it.

So each of these are references to environment variables,

that we need to set up inside of our GitHub repository.

We're gonna set these up as some encrypted secrets.

Whenever we try to reference one of these encrypted secrets,

or whenever we actually run our GitHub workflow.

This entire section right here will be removed and replaced

with the actual value that we set up inside of a repository.

We have to set up right now,

three different environment variables.

One for the name of the bucket

that we are trying to sync our project to.

One for an access key, and one for a secret key.

The access key and secret key are what are going to

allow us to actually access our AWS account

and change the files inside this s3 bucket.

You can essentially think of them

as being like a username and password.

We're gonna first go into AWS,

and generate a secret access key and an access key.

And then, we're going to go back over to GitHub

and set up all three of these different secrets.

The bucket name, the access key, and the secret access key.

Okay, so back inside my browser,

I'm gonna go back to my AWS console.

I'm gonna open up a third new tab now, still at AWS,

and I'm going to search for a service called IAM.

This is what we're going to use to generate an access key.

Once here, on the left hand side

I'm gonna find the user section,

and then at the top, I'll click on add user.

I'm gonna set up a username right here.

This username is really just a name

for the set of API keys that we're gonna generate.

So, for this, I'm gonna put in about something like

mfe-github-action.

So, it's kinda assigned to me that this is talking

about our MFE project, and these set of keys are being used

by the GitHub action that we are creating.

After that, at the very bottom, we're gonna find access type

and we're gonna select programmatic access.

That's what's gonna make sure

that we generate an access key and a secret access key.

I'll then click on permissions.

Over here is where we're going to decide exactly

what permissions we want to assign to this new user.

Now, there's definitely a lot of ways

that we can approach this,

and the recommendation officially on AWS,

is that you use a policy

of what's called least responsibility or least privilege.

In other words, this user that we are creating,

is only supposed to make changes

to that one s3 bucket, or essentially upload files to it.

And, we're also going to use it to make some very

small changes to our cloud front distribution eventually.

So in theory, the best way of setting all this stuff up,

would be to say we want to give this only access

to that very particular cloud front distribution,

and just to our very particular s3 bucket.

If you want to, you can set the permissions in that way,

and you can definitely do a Google search

and that will tell you exactly what to do.

But, just to save a little bit of time,

and because I really plan on deleting this new user

I'm creating here rather quickly after I record this video

and we finish up the course, I'm just going to add

in some policies that give this user direct access

to all my s3 buckets and all my cloud front distributions.

So, not the best practice,

but at this point I'm gonna kind of let you decide

whether or not you wanna go into greater detail,

because this whole kind

of deployment thing is kind of already going

off the rails of micro front ends in general.

Okay, so I'm gonna click

on attach existing policies directly.

I'll then search for s3.

I'm gonna find a setting right here

of Amazon s3 full access, and check it.

I'll then also search for CloudFront,

and select CloudFront full access.

Once I've selected both those, I'll go to next tags

at the bottom right, we don't have to add in any tags here,

so we'll go to next review, and then finally create user.

After we create the user,

we'll then be shown our secret access key,

and a button we can click on to see our secret access key.

Now, really important, do not go away

from this page just yet.

The secret key is only going to be displayed this one time,

so we need to make sure

that we make use of this key right away.

Otherwise, we'll have to regenerate the set of credentials.

So, to use these two keys, we're going to copy them over

to our GitHub repository, not inside of our source code,

but inside the settings of the repository itself.

So, inside of a fourth browser tab, I'm gonna go back

to my GitHub repository.

Once back over here,

we're going to click on this little dot dot dot,

on the far right hand side, that might not be displayed

if you're zoomed out a little bit more.

Essentially we wanna find settings over here.

Then, once we open that up,

we're gonna go on the left hand side

and find secrets down here, towards the bottom.

Then, on the far right hand side,

we're gonna create a new secret.

So again, these are going to be values that we can only

access inside the code of our GitHub action.

So, this is where we're gonna create a variety

of different secrets and eventually refer to these inside

of our action config code.

So, we need to create three different secrets right now.

Our access key, the secret key, and the bucket name.

Let's take care of the AWS\_ACCESS\_KEY\_ID first.

So, we're going to enter in a secret name here

of exactly AWS\_ACCESS\_KEY\_ID, all capitals with underscores.

I'll then go back over to my management console.

I'm gonna find the access key right there,

copy it, and then paste it in at the value field.

I'm gonna add the secret, and there's our first one.

So, we're now going to repeat that process twice over.

I'll click on new secret again.

The next one is going to be our secret key,

and I wanna make sure I get

the exactly correct name for that.

So, AWS\_SECRET\_ACCESS\_KEY, I'm gonna copy that.

And, it's value again,

is gonna come from our management console.

So, I'll click on secret access key,

and that's my secret access key.

I'm gonna copy that.

Now, just to be clear,

you do not wanna share this access key

with absolutely anyone, 'cause they will have full control

over s3 on your AWS account and cloud front.

So, by the time you watch this video, you better

believe I will have deleted this key right here so

that no one watching this video can use this key

to control my account.

I'm gonna copy that, go back over to GitHub,

make sure I've got AWS\_SECRET\_ACCESS\_KEY,

paste the value in.

There we go.

Now just one more value, our bucket name.

We'll go to new secret.

I'm gonna make sure I get the exact correct property value

AWS \_S3\_BUCKET\_NAME.

So, I'm gonna put that in for name right there.

AWS\_S3\_bBUCKET\_NAME.

And then, in this case,

we're gonna get the bucket name

from our S3 management console.

So, I still got that tab open.

Here it is right here,

and we're gonna take the exact name right there.

So, I'm gonna copy that,

and paste it in for value, and add secret.

Okay, that should be it.

So now, in theory,

we should be able to run our GitHub action,

and copy our created distribution files

or the actual production files over to our s3 bucket.

So, let's pause right here,

and then test this all out in the next video.

Topic 64. Rerunning the Build

Instructor: We've now set up our access keys

and the bucket name.

So in theory,

we should be able to rerun our GitHub action

and see all of our built project files sent over

to Amazon S3.

To test this out, I'm gonna go to my actions tab on GitHub.

I'm gonna find our script

or our workflow right here that had previously failed.

Once I've clicked on that,

I'll then find on the right hand side Re-run jobs.

I'll click on Re-run Jobs like so.

As you guess, that's just going to rerun the exact same job.

But this time, now that we've got our secrets loaded up

into this repository,

we should successfully upload everything over to AWS.

So I'm gonna take a look at the build as it runs.

This is gonna take a minute or two,

but it will be pretty quick, so I'm just gonna let it run.

You could absolutely skip forward in the video if you want.

Tell you what, I'm just gonna pause the video right here.

We'll come back in just a moment

and we'll make sure that everything worked as expected.

Topic 65. A Small Error

Speaker: It looks like my build was successful.

So in theory, I just built my project

and transferred all the files off to AWS S3.

So now we need to test this out

and see if we can actually load up our application.

Now, just so you know

we're not gonna be able to load it up successfully.

Well, halfway, but not all the way.

We still have to do a very small amount of work.

So to test out our application

I'm gonna go back to my AWS CloudFront Dashboard.

I'm taking a look at the distribution that we just created.

I'm at the general tab.

I'm gonna scroll down a little bit

and find the option right here that says domain name.

This is the domain that we can access our application at.

If you want to, you can also purchase a domain name

and point it at this distribution

and you can use your own custom domain name

but unless you do that, we're gonna use this

automatically generated domain name for now.

So I'm gonna copy that URL,

open up a new browser tab and go to that URL.

Once I do so, I see just a white page on the screen.

Now at this point in time, if you see any error text

on the screen, particularly something that says something

like Access forbidden or something like that,

it means that you have some error in your configuration

under either your bucket or your CloudFront distribution.

So you'll need to do a little bit of troubleshooting

and figure out what's going on.

Otherwise, if you see a white screen like this

believe it or not, it is actually good.

So I'm gonna open up my console really quickly

and we're gonna figure out what is going on here.

Inside my console, I'll see an error that says

unexpected token, little bracket right there.

So what's happening?

Well, this is one of many different errors

that we are going to run into once we deploy

our application to production.

Micro frontend apps unfortunately behave a little bit

differently in the development environment

compared to the production environment.

And so this is the result of some little issue

that we didn't quite think about too much

when we started to move our app into production.

This is also an error that may or may not

come up inside of a monolithic app

and that's why I wanna spend a little bit

of time on this just to understand that,

hey this is some issue that you might have to deal

with sometimes with the monolithic app

but you're almost always gonna have to deal

with, with a micro frontend application.

Okay, so why are we seeing the error?

Well, it's really simple.

I'm gonna go to my elements tab.

Here's a HTML document that we have successfully loaded up.

This HTML document hopefully looks really familiar.

It is the same HTML document that we generated

from our container project.

So it's essentially the same public index dot html file.

The only difference is that it has been processed

by that HTML webpack plugin.

Now remember, the only goal of that plugin

is to take some script tag or add a script tag

into this HTML.

So we can see that very easily right there.

There is the script tag.

You'll notice that the script tag has an SRC

of main dot something something something dot js.

This script tag doesn't have any path on it

and that essentially means that to load up this script

the browser is gonna take whatever our current URL is,

so I'm gonna copy that really quick and just give

you a quick demonstration.

The browsers gonna take our current URL,

there it is right there,

and then try to find this main dot js file

by just appending on main js, like so.

If we make a request to this URL right here,

then our cloud front distribution is gonna try

to find some file called main dot js inside

of the root folder of our S3 bucket.

So let's open up our S3 bucket very quickly

and see if there is a main dot js file inside there.

Now, just to be clear, I know that the actual file

name is main dot something something dot js,

I'm just abbreviating it right here.

All right, so back inside of our S3 bucket,

here's my bucket.

I'm gonna take a look at the objects tab.

I'm gonna go down to the bottom

and I'm going to refresh that section.

So inside of here, we are looking

at the root directory of our bucket.

Just so you know, as a quick technicality

technically buckets don't have folders.

I know it looks like there's a folder

but behind the scenes there's not any actual folders.

I only mention that because I know someone's gonna post

inside the discussion and say, Hey, Steven,

S3 buckets don't have folders.

Anyways, totally irrelevant.

We can see very easily that

while there is a container directory

inside of here and inside of there

if I click on it, there's a latest directory.

And inside of there are all of our built JavaScript files.

You'll notice that all those are nested inside

of this container slash latest folder.

So we don't have a main dot js file inside

of our route directory inside the bucket.

It is nested inside of container slash latest.

So if we wanted to successfully load

up this main dot js script, we would instead have

to make a request to container slash latest.

If we made a request to a URL like this right here

then we probably would successfully load

up our actual JavaScript file.

So we need to figure out how we can somehow make sure

that this script tag right here;

this script tag that is generated

by the HTML webpack plugin doesn't just put in main dot js.

We need it to put in slash container slash latest

or really just container slash latest.

That's what we really need to have happen.

So let's figure out how to do that

how to get our webpack HTML plugin

to generate a better looking script tag in the next video.

Topic 66. Webpacks PublicPath Setting

Instructor: So how can we tweak

how that HTML WebPAC plugin works?

How can we make sure that it generates the correct URL?

Well to do so, we're gonna go back over

to our code editor

and we're gonna find the WebPAC prod dot JS file

inside of our container project.

Inside there is the output option.

This output option is gonna change

how some of the different files

get referred to or named

after they are processed by WebPAC.

We have already set up a file name option

which is going to change

how all of our different files get named.

There's another very common option

that you're going to set

inside of a WebPAC production config file,

especially in a micro front end application.

The other option that you'll set in here very frequently

is referred to as public path.

This public path option

is gonna be used anytime you have some part of WebPAC

that tries to refer to a file

that has been built by WebPAC.

An example of this would be

whenever our HTML plugin tries to refer

to some JavaScript file that has been created.

Right now, whenever the HTML plugin tries to refer

to some file that's been built by WebPAC,

it just makes use directly at the file name.

But we can get the HTML plugin

to try to add in some path on there as well

by setting this public path option.

In other words, if we put in

for the public path right here

slash container slash latest slash,

make sure you get the last slash on there,

then whenever our HTML plugin starts

to figure out all the different script tags

it needs to add to the HTML document,

it's gonna take all the different file names

but then prepend them

with this public path right here.

So if we think about the route that will be generated

for all of our different script tags,

it will end up as slash container slash,

oop, not source but latest slash main.

Like so.

And that is exactly the URL we want.

'Cause that's gonna go into our S3 bucket.

Then we'll go into the container folder,

then the latest folder,

and find the main dot JS file inside there.

So all we should need is that public path option

and that should fix everything up.

Okay, so now we've made that change.

We now need to rebuild and redeploy our app.

To do so, all we have to do

is commit our changes and then push them up to GitHub.

Once we do so, GitHub will notice

that we've made a change

to something inside the container directory

and automatically rebuild and redeploy our project.

So let's commit this stuff and push it right away.

Back at my terminal.

I'm inside of my MFP directory.

I'll do a get status.

And just verify that I did change that file.

Do a get add dot.

I get commit

and I'll add on the message of added public path.

I'll then do a get push to origin master.

Good, after that in theory, our GitHub action

is now running.

So let's go over to our GitHub repository

and just verify that is the case.

Back inside my browser.

I'll find my GitHub repository.

I'm gonna go to the actions tab

and there is our rebuild right there.

I'll click on the build job

and once again we can wait for this build to finish.

Let's take a pause right here really quick

and come back when this is all complete.

Section 8: Microfrontend-Specific AWS Config

Topic 67. Manual Cache Invalidations

Speaker: Well, it looks like our build is complete,

which means in theory, we have now deployed

this new version of our app.

Now, just so you know you probably

are not seeing your app still work.

So if you still see an error, totally fine.

That's actually something we really want to investigate.

So this is gonna tie back to something

I mentioned many times so far, something around caching.

So I'm gonna first begin by going back to our deployed site.

So here's my CloudFront URL right here.

Once here with the elements panel open,

I'm going to again refresh the page.

When I do so you'll notice that we still have

the exact same script tag here.

Remember we just made that change

to the public path property

hoping to see something like slash container slash latest.

But it's very clear that that is not the case.

We see instead the same exact main dot js.

So what's going on here?

Well, to do a little bit of debugging

the first thing we might do is go take a look

at the actual file that exists inside of our S3 bucket.

So for that I'm gonna go back over

to my S3 management console.

I'm looking at my bucket right here.

I'm gonna go down to our file list

at the very bottom and just refresh the list.

I'll then go to container, latest, and then

inside of here I'm gonna find that index dot html file.

So we are really hoping that this index dot html file

has some updated script tag inside of it.

Once I open up this file or some information about

the index dot html file,

I'm gonna go on the right hand side

and find something inside of here called object URL.

I'm going to right click that link

and open it in a new tab.

That is gonna load this file up directly out of our bucket.

So let's take a look at what happens now.

So it looks like I did load up this page successfully

but again, I get a white screen.

If I open up my console, I will see a couple

of different errors.

But you'll notice that all these errors look very different

than what we saw over on CloudFront.

And what's more and most important,

if we take a look at the elements panel,

the script tag right here has the correct SRC.

So I see container slash latest.

We are now successfully loading up the main dot js file.

The reason we're seeing all the errors down here is

because we've got a whole bunch of references

to our marketing application.

But remember, we have not actually deployed our marketing

app just yet.

So the errors that we now see are totally fine.

We've got the correct script tag inside of here

but it looks like these changes were never actually made

to our deployed CloudFront website.

Even now, if I refresh the page, I still see the old SRC.

So what in the world is going on?

Well, just to be really clear what the problem is here

we've got the correct file on S3,

but the incorrect one over on CloudFront.

So the whole problem is related to how CloudFront works.

Whenever you create a new distribution,

CloudFront is gonna take a look at all

the different files that exist inside of your bucket.

Anytime you add any files to your bucket,

CloudFront is gonna pick up those new files as well.

However, CloudFront does not automatically take a look

at any changed files.

It doesn't say, Oh, you just changed a file,

I'm gonna pick up and serve the newest latest edition.

So right now CloudFront has picked

up our original index dot html file,

but it's not picking up, or it's at least ignoring,

all the updates we were making to that file.

One way that we can tell CloudFront

that we wanted to use the latest version

of that index dot html file is

by creating something called an invalidation.

So on my CloudFront management page,

I'm looking at my distribution that I created

just a little bit ago.

I'm gonna go over to my Invalidations tab over here.

We can manually create something called an invalidation

and this is going to cause us to automatically tell

or I shouldn't say automatically,

but it's a little bit more manual way

of telling CloudFront that it needs to make use

of the latest version of that index dot html file.

Now this is a very manual way

of making sure that we use the latest version of a file.

Another way to do this is to create a file

with just a different name

or a different path inside of our S3 bucket.

In this case, anytime we make a change

to our JavaScript files,

they will be given a different hash

or content hash on there.

So anytime that we make a change to a JavaScript file

CloudFront is gonna say, Oh, this must be a new file.

I should use the latest version.

It is only our index dot html file

that doesn't have some kind

of unique identifier in its name.

So it's really only the HTML file where we're

going to have trouble with always loading

up some kind of outdated version of it.

But all of our other JavaScript files

should be absolutely okay.

So let's fix this problem really quickly

by manually creating a invalidation.

I'll click on create invalidation.

Inside of here we're gonna enter in

slash container slash latest slash index dot html

and then invalidate.

We'll then see in progress right here.

Eventually in progress will be resolved.

And then once it gets resolved

we should be able to go back over

to our CloudFront site, refresh the page,

and see the latest content out of our HTML document.

So I'm gonna pause this video right now,

take a brief pause, and when we come back

hopefully we'll see the status resolved right here

and we should be able to refresh our cloud front print page

and see the correct SRC up here.

Topic 68. AWS Region with Automatic Invalidation

**AWS Region with Automatic Invalidation**

In the upcoming lecture, we will be adding automatic invalidation to our Container Workflow (and in a few lectures the Marketing Workflow).

If you are using the suggested shinyinc/action-aws-cli@v1.2 action, you will need to add the AWS Region to the variables associated with the create invalidation run step:

1. - run: aws cloudfront create-invalidation --distribution-id ${{ secrets.AWS\_DISTRIBUTION\_ID }} --paths "/container/latest/index.html"
2. env:
3. AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}
4. AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}
5. AWS\_DEFAULT\_REGION: us-east-2

**Important** - Remember to replace us-east-2 with whatever your actual region is.

Topic 69. Automated Invalidation

-: It looks like my invalidation is complete

on that index.html file.

So if I now go back over

to my CloudFront site and refresh the page

I will see the correct script tag on there now

and that means that we are successfully loading

up the latest version of the index.html file.

Well, that definitely worked, but as you can guess

whenever we deploy our application, we do not

want to manually have to create that invalidation thing

if we happen to make a change to our HTML file.

Your first thought might be, Well, Steven

how often does the HTML file actually change?

Well that file's gonna change anytime the content hash

on the script tag right here changes.

So for example, if we rebuild some file

that causes this main.js file to be changed

we're gonna get a different content hash on there

and we would need to make sure that we serve

up the latest version of our index.html file

which would require us to create that invalidation.

So we definitely want to make sure that we automatically

somehow create these invalidations

and we definitely don't want to have to always

remember to do it by hand or anything like that.

So how are we gonna do that?

Well, naturally we're gonna just add

in a little bit of automation, once again.

Back inside my code editor, I'm going to

find our GitHub workflows container.yaml file.

So inside of here, we are already making use

of the AWS sdk, or excuse me, the CLI right here

in order to sync our files over to S3.

It turns out we can use the same AWS CLI to

automatically create an invalidation for us

specifically on just that index.html file

which is the only file we need to invalidate.

To do so, we'll add

in an additional configuration step inside of here.

So I'm gonna add in another - run.

The command that we're gonna execute is gonna be

aws cloudfront create-invalidation--distribution-id.

We then need to add in our distribution ID right here

rather than just pasting it in directly.

We'll set up another environment variable or

another GitHub repository secret so that we

don't have to paste it directly into this file.

So I'm gonna put in a ${{secrets.AWS\_DISTRIBUTION\_ID}}

Then after that we're gonna specify exactly what files

we want to invalidate.

So we'll put in --paths.

And then inside of "/container/latest/index html".

When we run that command, we still have to specify exactly

what credentials we want to use.

So we need to make sure that we copy paste this environment

section right here

from the previous command down right underneath the run

one like so.

Make sure all the indentation lines up.

Make sure you've only got a - right there

on the run and not on the env option.

Now we have a little bit

of a conf or copied configuration section in here.

Just so you know

we could technically move this entire end section up higher

in the file, and then we would not need to duplicate it.

However, that would also mean

that we would expose these environment variables

to some other unrelated steps inside of our build.

So in other words, our npm install

and our npm and run build processes would have access

to our AWS secret key and the secret access key.

That might not be what we want.

And so I think it's a little bit better to just

be very specific

and place these credentials exactly where we need them.

Okay, so now to test this out, we're gonna save this file.

We have to go back into our GitHub repository

and add in a new secret of AWS distribution.

ID back inside my browser.

I'll find my GitHub repository.

I'm going to again, go to the repository settings

go down to secrets, add a new secret.

We're gonna set up AWS distribution id

and then the value for this is gonna be exactly

whatever our distribution ID is.

So back on.

So on our cloud front console

I'm gonna find my distribution ID right there.

I'm gonna copy it and then paste it over here as a value.

Okay, let's add our secret.

There we go.

Now the last thing we have to do is commit our

updated workflow and push it up to GitHub.

Once we push this up to GitHub, we can then

manually trigger the workflow again and just validate

make sure that the invalidation is done for us.

All right, so back at my terminal

I'm going to again do a get, Let's do a git status first

just make sure we've got the updated workflow.

We'll do a git add a commit

and we'll say added invalidation step to build

and then a git Push Origin master.

Our new workflow is now on GitHub, but as a quick reminder

remember, GitHub is only going to redeploy our code

if there's a change inside of our container directory.

We did not make a change inside of container.

We only made a change to the workflow.

So if we want to rebuild our project and test out

that invalidation step, we have to go and make a change

inside the container directory.

So let's do that very quickly.

Back inside my editor, I'm gonna find my SRC folder.

Inside of my container project, I'll open up app.js

and inside of our high there, right here, I'm just gonna add

in a couple more !!!.

So that qualifies as a change inside

of our container directory.

So let's now commit this, push it up

and we should see the rebuild then occur.

And along with that

we should see the invalidation step.

To back over at at my terminal.

I'll do another Git status and add a commit.

I'll say updated text, and then a push to origin master.

Okay, that looks good.

Let's now go over to GitHub.

We're gonna take a look at our actions tab.

Okay, there's updated text right there.

I'm gonna click on build.

And now in our build steps

or the actual individual job right here.

So here's the job we should see.

One of the commands to be executed is the create

and validation step.

Very good.

So I'm gonna pause right here.

We're gonna let the entire deploy happen

and then we're going to validate and just make sure

that an additional invalidation was created specifically

for the index.html file to make sure

that we use the latest one.

So let's take a pause right here

and check that out in just a moment.

Topic 70. Successful Invalidation

Steven: It looks like the build went off just fine.

So, I can see the automated create validation step

right here.

And if we go over to our CloudFront dashboard

and take a look at the Invalidations tab,

here's a brand new invalidation

that was created along with our deployment.

I can click on Details and it will tell us

that we invalidated our /container/latest/index.html file.

So, that is perfect.

We want to invalidate that file with every deployment.

This looks pretty good.

But at this point in time, you might be saying to yourself,

"Hey, Steven, wait a minute,

like, who cares about this invalidation stuff?

This is something that has to happen with all applications

and this is not something that is microfrontend specific."

That's what you might be thinking right now.

Well, just a quick reminder.

Remember, one of the big things I said earlier

around this entire deployment process

is that right now the remoteEntry.js file has a fixed name.

So, the name of that file will always be remoteEntry.js.

Even if we start to change the contents of it,

it's still gonna have the exact same name.

And so we're gonna run into the same issue

around caching with that remoteEntry file

through our CloudFront Distribution.

So, as we go and put together

all of our different microfrontends,

we're gonna have to remember that during our build process,

we're gonna have to create an invalidation

for each microfrontend's remoteEntry.js file,

and stick that invalidation step

into all of our different GitHub workflows.

So, it is related very much to our microfrontends.

Let's remember to do that once we start to add in

some deployment for each of them.

Speaking of which, let's take a pause right here,

and when we come back to the next video,

we're gonna go through all the same deployment stuff

for our marketing application.

But don't worry, it's gonna be much, much faster.

All we really have to do is duplicate

our workflow for the container.

So, let's get started on that in just a moment.

Topic 71. Setting Up the Marketing Deployment

Instructor: In this video, we're gonna start to

add in some deployment for our marketing application.

There's a couple of different things we need to do here,

but each step is gonna be

relatively easy and straightforward.

The first thing we need to do

is create a new workflow file inside

of our GitHub workflows directory.

Inside of here,

we're gonna create a new file called marketing.yml.

So, inside of here,

we're gonna have some steps to build

and deploy specifically our marketing application.

All the steps are going to go

into this are just about identical

to what we did inside of our container.yml file.

It's gonna be so similar, as a matter of fact,

that we're just gonna open up our container yml file.

Copy everything inside of here,

and then paste it into the marketing.yml file.

And we'll just go through

and change a couple of different names.

So, inside of our marketing.yml file,

we're gonna first change online one.

We're gonna rename this thing to be deploy-marketing.

On line eight, we'll change it to

packages/marketing/\*\*.

On 12, change it to packages/marketing,

and then we're going to scroll down

to our aws section down here.

Now, whenever we start to sync

our dist directory off to our S3 bucket.

We don't want to place all of our built marketing files

inside of some container directory.

Instead, we're gonna place all these different

files we've built inside of a marketing/latest folder.

So, inside of our S3 bucket

we're going to eventually see marketing/latest.

And inside there are gonna be

a bunch of built JavaScript files

that all represent our marketing project.

Then after that,

we're also going to update our invalidation.

So, just a quick reminder on what the

invalidation is all about.

Remember, whenever we create a

brand new file inside of our S3 bucket,

that file is going to be sent off to CloudFront.

And Cloudfront is gonna start to use that brand new file.

However, if we ever update an existing file in place

CloudFront is not going to automatically make use of it.

To get CloudFront to make use of a file

after it's been updated.

We need to create the invalidation.

And invalidation was used on our container application,

because we always had that index.html file.

And even though its contents might change

we would never change its name.

This is gonna be a concern in all of our

sub-projects as well,

because they all have exactly one file

whose name is never going to change.

And that is the remote entry.js file.

The contents of that file might change

for every single deployment we do

but the name is never going to change.

So, to get CloudFront to use

the latest version of the file,

we have to create the invalidation.

So, for that, we're gonna scroll off

to the far right hand side.

And rather than trying to invalidate something

inside the container directory,

we're going to invalidate

marketing/latest/remoteEntry.js.

Like so.

Okay.

So, that's it for this deployment file.

That definitely looks good.

Now, there are two other changes we need to make.

But, rather than having this video

take a really long time.

Let's take a pause right here

and make the other two changes in just a moment.

Topic 72. Reminder on the Production Domain

Instructor: As I mentioned, there are two other changes

we need to make to set up our marketing deployment.

The first one is something we've previously discussed

very briefly.

And we even wrote a little bit of code around this,

but we never quite finished the entire flow.

So, inside of our packages, container directory,

I'm going to find my config, prod js file.

So, here's webpack prod dot js.

You might recall that inside of here,

we set up our Module Federation Plugin,

and when we did so, add on the remote of marketing.

Inside that remote, we had said that we need to instruct

our container where to go

to get the marketing's remote entry dot js file.

And, to do that, we need to specify the domain

where that JavaScript file was going to be hosted.

So, we had set up this entire system

for making use the domain,

remember we had said that we're gonna set a

environment variable called production domain.

We just never went through that process.

We never actually set anything up

inside of our GitHub repository

to provide that production domain variable.

And so, that's what we have to do really quickly.

We need to make sure that we have

some kind of production domain secret

on our GitHub repository.

We need to also make sure that we make that secret available

to our webpack build process when our container application

is being built.

And again, the entire idea here

is so that our container application

knows exactly where to go

to get the marketing remote entry file.

Before we go and set up the environment variable,

just one little typo I made here would put down a path of

slash marketing slash remote entry dot js.

However, our actual marketing project,

we just said we're gonna place into a folder

inside of our bucket

of marketing slash latest remote entry dot js.

So, please make sure that you add on

slash latest right here.

Okay. So, let's go and get our domain

or our CloudFront deployment,

we're gonna make sure that we set up a GitHub secret for it

and eventually expose that as an environment variable

during our build process.

So, back inside my browser,

I will go to my CloudFront dashboard,

I'm gonna find my deployment.

Once again, I'm gonna take the URL right there,

I'm gonna copy that, and then, going to go back over

to my GitHub repository,

I'm gonna find the Settings tab on the far right hand side

and again, we're going to find section of here

called Secrets.

Here it is right here.

I'm gonna create a new secret.

We're going to call it production domain,

and then paste in that URL.

And, we're also gonna put in https colon slash slash

right in front of it, like so.

Okay. So, let's add the secret now.

We should see that new secret right there,

and like I mentioned, just by creating a secret,

we don't actually get access to that variable automatically

inside of our workflow.

Instead, we need to make sure that we mainly specify it

or kind of add it in as an environment variable.

So for that, we will go back over to our code editor,

we're gonna find our workflows, container dot yml file,

then, inside of here, we're gonna scroll down a little bit

and find our npm run build step.

So, when we run this command,

we want to expose that secret we just added

as an environment variable.

So, we're gonna add in an env section,

we're going to say that

we want to have an environment variable called

production underscore domain,

and the value for this would come from dollar sign,

curly braces, secrets dot production domain.

And that should be it.

Now, whenever we build our container,

we're gonna have access to that environment variable,

we're going to run our webpack build process,

domain will be set equal to that string we just set up,

and we're going to say,

"Hey, whenever we try to find marketing,

go to that CloudFront URL

slash marketing slash latest slash remote entry dot js file.

And, that's gonna tell us how to load up

all the additional code for our marketing project.

Okay. So, that looks good.

Good change.

Now there's, like I said,

two things we needed to change here,

that was the first one.

The other thing we need to change is also related

to this production domain stuff.

So, back inside of marketing, config,

webpack dot prod dot js file.

Just like we did on our container,

we also have to set up the output,

public path property as well.

Whenever we create that remote entry file,

or to be precise, whenever the Module Federation Plugin

creates remote entry file,

remember the remote entry is going to instruct

some other program

how to load up some other code from our project.

So, the remote entry file is gonna contain a list of URLs

inside of it, and it's gonna say,

"Hey, here's the exact URL to go to to load up some kind of

name dot js file, or maybe some node modules file

or something like that."

So, for the remote entry file,

to get the correct URL inside there,

we also have to set up the

public path property here, as well.

So, we're gonna set up our public path property

to say just go to the current domain slash marketing

slash latest and then a slash at the very end.

So again, pretty much the exact same thing we did

back inside of our container,

back inside the container we set this up

so that all the different script tags

inside of our html file

would refer to the correct location

of all the different JavaScript files.

In this case, we are sending it to public path,

so that our remote entry file will

point to the correct folder inside of our S3 bucket.

Okay. So, I think that is it.

Now, last thing we have to do

is attempt to actually deploy our application.

Let's take a very quick pause right here

and do the deployment in just a moment.

Topic 73. Running the DeploymentFormularbeginn

Instructor: Let's now try to deploy

our application once again.

To do so, as usual, all we have to do is commit

all of our changes and push it off to GitHub.

So I'm back at my terminal.

I'll do a git add . a git commit.

I'm gonna put on a message of something like,

added marketing deployment,

and then I'll push it off to master.

Okay, once that push is complete

we'll go back over to GitHub.

I'm gonna take a look at my Actions tab.

And on actions, we should now see two different deployments.

So one for our container and one for marketing,

which we just added.

So now we can take a look at both these.

And if we go to our build tab, as usual,

we're going to go through that exact same process.

So this deployment will take just a couple seconds.

Let's take a pause here, come back to the next video,

and then just verify, make sure that everything

was deployed successfully.

Topic 74. Verifying Deployment

Instructor: It looks like the deployment was successful

so let's now go back off to our

cloudfront deployed website, refresh the page,

and hopefully we will see some content appear on the screen.

So back over at my cloudfront site

I'm gonna refresh the page and sure enough

I can see some content. Awesome.

That's exactly what we want. So we can see

part of the container at the very top

and then here's our marketing page right here.

Let's take a look at some of the

different paths that have been generated

once we set up that public pathing

and the production domain, all that kind of stuff.

So first I'm going to take a look at the Elements tab.

Once again, we can just verify we've got script src

and then the path inside of our S3 bucket

to the main.js file for our container app.

That's what kicks off the entire process

of loading up all the JavaScript for our project.

I'll then take a look at my Network Request tab.

So inside of here we see a collection of different files.

Once again, some of these files are coming from

our container project and some

of them are coming from marketing.

The first file coming from the marketing project

will be the remoteEntry.js file

and if I hover over that, once again,

we'll see that we are, in fact, referencing

the correct path to get at the remoteEntry file.

So it's marketing/latest/remoteEntry.

If we open up that file, we can then do a search.

So I'm gonna search inside of here

and I'm going to look for /marketing.

So you'll notice right here that's where

that public path property comes into play.

That's gonna make sure that we try to find

all the marketing-related files inside of the

correct subdirectory inside of our S3 bucket.

So once again, we don't have to worry

about any collisions between similarly named files

between all of our different micro frontends.

So the remoteEntry file loads up and then it's

gonna load up some additional number of files inside

of here related to, well, some marketing-related code.

So if you want to, you can certainly

hover over each of these. You'll see

that some of them are coming from the marketing directory.

So that's definitely a marketing-related JavaScript file.

There's one right there. There's another,

another, another, another, another.

Looks like we've got a lot of different files

right here all related to marketing.

Now one thing you might be a little bit concerned about

is the number of JavaScript files that we are

currently loading. Why are there so many exactly?

In some cases, you might want to

reduce the number of separate JavaScript files.

Now that technically used to be a older thing that we

used to care about. These days making use of http too

it's actually a little bit more advantageous

to have multiple small JavaScript files

because they load up a lot faster than single large files.

So just the fact that we've got a lot

of separate JavaScript files here, even for a very

small project, shouldn't necessarily be a cause for concern.

Nonetheless, if you're curious about why we have

all these separate files, well once again,

it all comes down to our shared modules that we declared.

So back inside of our marketing prod file right here.

So marketingconfig, webpac.prod

remember we've got shared so that means

take all of our separate dependencies, listed out

inside the package.Json file in the dependency section

and we're going to try to include all those

as separate individual JavaScript files.

That's what's going to guarantee that we only try to

load up each of these dependencies just one time.

They'll get output as separate individual JavaScript files.

So if you don't like that, if you wanna have

just one big vendor bundle you can remove shared

and then deploy the application and you're

you're going to see everything essentially reduced down

to just one or two single JavaScript files

for all those different dependencies.

But then you lose out on the benefit of not

loading up all these dependencies once per subproject.

So again, even if it's kinda looks a little bit weird

to load up a lot of different JavaScript files here

at the end of the day, it really is rather advantageous.

It's not really a bad thing necessarily.

Okay, so let's say that this looks pretty good.

So now that we have everything working on production

and we've got a pretty solid workflow here

for deploying to production, I wanna very quickly

show you a pretty good workflow that you could use

in general or some serious projects

just so you can understand how you can

easily have multiple different teams

working on these different projects

and just understand, well, how do we

kind of avoid stepping on each other's toes

and control when we actually deploy our application.

So let's take a look at a interesting workflow

in the next video.

Formularende

Topic 75. [Optional] A Production-Style Workflow

Instructor: Right now,

we have our application set up

to really just do all of our development

on our main, or master, Git branch.

So, in other words, right now,

my local Git repository is on master.

So, I can make changes to some of my

different sub-packages inside of here,

commit them, push that off to GitHub,

and all my code automatically gets deployed.

Now, that definitely works for you and I,

working on this project by ourselves,

but inside of a larger company,

that might not be a flow that

really works very nicely.

There's a wide variety of reasons

that you would not want all of your engineers

to work directly off a master branch

and have them just deploy code automatically,

by pushing their code up.

The most obvious reason is

that you might want to have

some review process in there.

Whenever a team makes some change to

some one of these individual packages,

you might want to allow other engineers

to review those changes before they actually

get deployed off to production.

So, in this video, we're gonna take a look

at a workflow that's going to allow you to

essentially do that review step and

make sure that you don't just

allow each individual engineer to

deploy code as often as they wish.

So, here's the general workflow that you can use.

We're not going to use this workflow

inside this course, because it just adds

in a bunch of unnecessary steps we don't need,

but you can absolutely use this workflow

for your own professional projects.

This workflow is going to work,

not only for this micro frontends application,

but also for any other kind of general project.

The general idea here is that each team

is gonna develop all their different features

for each individual sub-application

on a local Git branch, and they're going to

not be on the master, or main, branch.

Instead, they'll be on some other named branch,

something like maybe "container-dev"

or something similar to that.

At some point in time, they're going to decide

that they have completed whatever feature

they're working on,

and they're ready to deploy it.

Rather than just deploying that

directly off to production,

we're going to instead have that engineering team

push their branch, which again,

is named "container-dev"

or something like that, off to GitHub.

Once they push that branch up,

we'll have them create a pull request.

The pull request is going to attempt to

merge some code into your master

or your main branch.

If you're not familiar with a pull request,

it's essentially a request to merge some code

into another branch.

A pull request job, for the most part,

is to really allow other engineers

to see this request and allows other engineers

to add in comments or review code,

stuff like that.

So, that would be the opportunity

for some other engineers

to review all the changes.

After the review is complete,

you'll then merge the pull request in.

That takes all the changes you have made

to this container-dev branch

and merges those changes

into your master, or main, branch.

At that point in time,

your GitHub actions workflow is

going to detect the fact that

some change has been made

to your master or main branch, and so your code

will be automatically deployed.

So, that's the general idea.

Let's go through this entire process

just really quickly one time.

Again, we're not gonna use it actively

on our project, just because it's gonna add

in a bunch of unnecessary steps for us,

but I would recommend you use something

like this if you're ever working with other engineers.

So, to get started,

we will go over to our terminal.

We're going to check out a new branch,

and we're going to imagine that

we're making some change

to maybe our container application.

So, I'll do a "git checkout -b"

to create a new branch,

and I'm going to call this "container-dev".

Now, that we are working off of a branch,

we can go over to our editor

and start making changes.

So, back inside my editor,

I'm gonna find my container project.

Inside there, I'll find the App.js file,

and I'm going to just add in

a couple more exclamations.

We're gonna say, "That's our feature."

That's the big change that we want to now deploy.

So, I'm gonna save this file.

We're now going to go back over to our terminal.

We'll do a "git status"

and just verify we've only

made changes to our container folder.

We'll do a "git add", a "commit",

and I'll give the commit a name

of something like "Finished feature #1".

I'm then going to push this up to GitHub,

and I'm going to make sure

that I push it to a branch named "container-dev".

We'll do a "git push origin container-dev".

Now, that we've pushed this off,

we can go over to GitHub

and create a pull request.

And, as a matter of fact,

the message right here even tells us

we might wanna do that.

So, back inside my browser, I'll go to GitHub.

I'm gonna find the "Pull requests" tab.

You'll notice there's a little shortcut

right here we can use.

So, you can click on that button right there.

Or, alternatively, you can click on "New pull request".

I'm gonna click on "New pull request"

just so you'll see what to do

if this window doesn't appear for you automatically.

Then, we're told to select what branch

we're trying to merge into which,

so, I want to take my container-dev branch

and attempt to merge it into my master branch.

I'll then click on "Create pull request".

If we want to, we can give

a more detailed message right here,

and maybe explain some of the changes that were made.

So, I would then create my pull request,

and then at that point, in time it would be

up to other engineers to come

and check out our code.

So, we would invite other engineers to come

and review all the changes that have been made.

They would probably come to this page,

and then probably take a look at

the "Files changed" tab.

Over here, ti's going to explain

all the changes that were made inside

of this pull request.

So, we could take a look,

we could add some comment,

say "Looks good!"

and click on "Start a review".

Once we finish reviewing all these files,

we would click on "Finish your review"

up here and "Submit review".

Now that we've got the review in place,

maybe the original engineering team can come

and address any changes or anything like that.

In this case, I would maybe mark the conversation

as resolved, just to say,

"Okay, great. I've read this thing. Sounds good."

And, eventually, after completing the review,

we then eventually want to obviously merge

in our code and deploy it.

So for that, we would click on

"Merge pull request" right here.

I'll confirm the merge,

and as soon as we merge that in,

we can then optionally delete this branch.

You don't have to.

It doesn't really make a big deal,

to be honest with you.

Some people will say it's a huge deal,

but it really comes down to

how your company treats branches on GitHub.

For me personally, I'm just gonna go ahead

and delete the branch

Because we made a change

to our master, or our main, branch,

and a change contained a change

to our container directory,

we should now be able to go over

to our "Actions" tab

and see our deployment automatically occurring.

There it is, right there.

We only see a deployment around container,

because we did not make any changes whatsoever

to our marketing sub-project.

So, I can click on that,

again, go to "build", and

we should eventually see

the entire build being completed.

So, it looks like, right now,

I already finished the build itself.

We're now installing the AWS CLI,

and then we should do the actual sync off to s3,

and then the invalidation.

I'm just gonna let those steps

run very quickly here,

so we can check it out inside of our browser,

make sure we have the correct number

of exclamation marks inside of our real project.

Okay, so it looks like that's all done.

Now, if I go back over to my CloudFront site,

I should be able to refresh the page

and see some additional number

of exclamation marks right there.

So, refresh,

and there we go. Perfect.

Cool. So, that looks good.

Now, I'm gonna go back over

to my terminal as a little bit of cleanup over here.

I'm gonna swap back over to my master branch

by doing a "git checkout master"

and now, our master branch is

ahead of our local branch.

So, the master branch on GitHub has some changes

that our local one doesn't have.

So, to make sure that we are

fully in sync with our remote,

we would do a "git pull origin master",

and that's gonna pull down the change

that we just made inside of that pull request.

So, we should be able to see

that we did, in fact, make some change

inside the App.js file of our container project.

Okay, so, that's it.

That's the entire flow.

So, just once again,

we're not gonna use this flow

in this course, just because it adds

in a lot of steps we don't need.

But, I would very highly recommend

you consider some flow like this

if you are working on a real

micro frontend application.

The other thing I wanna mention here, very quickly,

is that this entire flow works just as well

if you're not using a monorepo, as we are.

So, if you have a separate Git repository

for each individual service, totally fine.

Everything is still going to work exactly as expected.

Okay, so now we've got this all done.

Let's continue developing our app

in just a moment.

Topic 76. AWS Setup and Configuration Cheetsheet 2min

**AWS Setup and Configuration Cheetsheet**

This lecture note is not intended to be a replacement for the videos, but to serve as a cheat sheet for students who want to quickly run thru the AWS configuration steps or easily see if they missed a step. It will also help navigate through the changes to the AWS UI since the course was recorded.

**S3 Bucket Creation and Configuration**

1. Go to AWS Management Console and use the search bar to find S3
2. Click **Create Bucket**
3. Specify an AWS Region
4. Provide unique Bucket Name and click **Create Bucket**
5. Click the new Bucket you have created from the Bucket list.
6. Select **Properties**
7. Scroll down to **Static website hosting**and click **Edit**
8. Change to **Enable**
9. Enter **index.html** in the **Index document** field
10. Click **Save changes**
11. Select **Permissions**
12. Click **Edit** in **Block all public access**
13. Untick the **Block all public access** box.
14. Click **Save changes**
15. Type **confirm**in the field and click Confirm
16. Find the **Bucket Policy** and click **Edit**
17. Click **Policy generator**
18. Change Policy type to **S3 Bucket Policy**
19. Set **Principle** to **\***
20. Set **Action** to **Get Object**
21. Copy the **S3 bucket ARN** to add to the **ARN** field and add **/\*** to the end.  
    eg: **arn:aws:s3:::mfe-dashboard/\***
22. Click **Add Statement**
23. Click **Generate Policy**
24. Copy paste the generated policy text to the Policy editor
25. Click **Save changes**

**CloudFront setup**

1. Go to AWS Management Console and use the search bar to find CloudFront
2. Click **Create distribution**
3. Set **Origin domain**to your S3 bucket
4. Find the **Default cache behavior** section and change **Viewer protocol policy** to **Redirect HTTP to HTTPS**
5. Scroll down and click **Create** **Distribution**
6. After Distribution creation has finalized click the Distribution from the list, find its **Settings** and click **Edit**
7. Scroll down to find the **Default root object** field and enter **/container/latest/index.html**
8. Click **Save changes**
9. Click **Error pages**
10. Click **Create custom error response**
11. Change **HTTP error code** to **403: Forbidden**
12. Change **Customize error response** to **Yes**
13. Set **Response page path** to **/container/latest/index.html**
14. Set **HTTP Response Code** to **200: OK**

**Create IAM user**

1. Search for "**IAM**"

2. In the left sidebar, click **Users** under **Access Management**.

3. Click "**Create user**"

4. Enter any name you’d like in the "**User Name**" field.

5. Click "**Next**"

6. Click "**Attach Policies Directly**"

7. Use the search bar to find and tick **AmazonS3FullAccess** and **CloudFrontFullAccess**

8. Click "**Next**"

9. Click "**Create user**"

10. Select the IAM user that was just created from the list of users

11. Click "**Security Credentials**"

12. Scroll down to find "**Access Keys**"

13. Click "**Create access key**"

14. Select "**Command Line Interface (CLI)**"

15. Scroll down and tick the "I understand..." check box and click "Next"

16. Copy and/or download the *Access Key ID* and *Secret Access Key* to use for deployment.

Section 9: Handling CSS in Microfrontends

Topic 77. Header Component

**Header Component**

Hi!

**Attached to this lecture is a file called 'header.zip'. Please download it - we will use it in the next video.**

Resources for this lecture

Topic 78. Adding a Header

Instructor: In a lecture right before this one,

you should have found a zip file called header.zip.

Make sure you download that zip file.

Once you've downloaded it, go ahead and extract it.

Inside of there you'll find a single JavaScript file

called Header.js.

This is a header component

that we're going to show inside of our container.

Once again, I'm giving you this component

just because there's a ton of markup inside of it

and I really just wanna save a lot of time,

otherwise it would take us 10 or 15 minutes

to write out the entire contents of that file.

We're going to place this header component

inside of our container application.

So I'm gonna find container, src, components,

and I'm going to drag and drop the Header.js file

inside that component's directory.

Okay, so here's the Header.js file.

Once I've dropped that file inside there,

I'm gonna make sure that I show the Header

from inside of our App.js component.

So inside of App.js

I will import the Header

from components/Header.

Then inside this div we don't need the hr or the h1 anymore,

so I'm gonna delete those and replace it with Header.

Before we save this, we're gonna also add in

a little bit of code from react-router-dom very quickly.

At the top, I'm gonna add in an import statement

from for Browser

Router

from react-router-dom.

And then right around that div,

I will add in BrowserRouter

and a closing tag like so.

Okay, so we just made a whole bunch of changes

without really explaining what's going on.

Well, now that we've added in this Header file

let's take a quick step back.

Since we now have our application

able to be deployed to production,

we're gonna start to very quickly see

that there are some really wild differences

between running our code in development

and running our code in production.

And a lot of these differences

really come down to very distinct problems that only come up

as soon as you start working with micro-frontend apps.

So there's going to be

some very interesting stuff that occurs

the next time we deploy our application to production.

Right now, if we save all the changes we have made

and go back over to our browser,

I'm gonna go to Local Host 8080.

Of course, make sure you still have Webpack running

for your container and your marketing projects.

At Local Host 8080, we should be able to refresh the page

and now see the header appear.

Now everything on on here really looks pretty darn good.

It all looks very, very reasonable.

I see some content.

I've got a login button right there.

Without a doubt, stuff is kind of working as we expect.

However, the next time we push this off to production,

we're gonna start to see some

really, really interesting things going on.

So let's commit the changes we just made.

We're gonna push it all off to production,

and as soon as we open this up on our production site,

well, obviously stuff might not be

quite as nice as we expect.

So to commit our code,

we're gonna once again go back to our terminal

inside of our Route Project directory.

Do a quick git status.

Yep, changes look good.

I'll do a git add, a commit with a message of Added header

and then a git push origin master.

As usual, that should kick off a deployment workflow

for only our container application,

because we only changed code inside of container.

So we'll take a pause right here,

we'll wait for the deployment to finish up

and we'll go take a look at the deployed version of our app

in just a moment.

Topic 79. Issues with CSS in Microfrontends

Instructor: It looks like the deployment of our header

was successful.

So I'm now going to go back over to CloudFront

and refresh the page.

All right, so here's CloudFront.

I'll refresh.

And you're gonna notice very quickly

that we get a very unexpected result.

We do now see the header here

but the header looks very, very strange.

It looks way too tall.

Well, maybe we're too far zoomed in or zoomed out.

Well, no, unfortunately I am at 100% zoom.

So at this point it looks very clear

that we have some different looking content

on the screen between our production site

and our local development site

at Local Host 8080.

Even though the header is present in both

we have some very different looking content

between these two different pages.

So what's going on here?

Could it be another caching issue,

where we are not using the latest version of a file,

or something like that?

Well, unfortunately no.

Turns out that there's a much deeper issue going on.

This is a big issue around handling CSS in general

in micro-frontend apps.

Rather than having a discussion about

exactly what's going on here,

we're gonna have first a larger discussion

around handling CSS.

You're gonna very quickly see that there are some

big challenges with CSS inside of a micro-frontend app.

So a couple of quick diagrams.

I just wanna walk you through a very quick scenario.

So as a reminder, inside of our application

we're going to have a pricing page.

Technically, we already have that.

And we're going to eventually have a sign in page as well.

These two pages are a part of two different sub projects.

Pricing page is a part of marketing,

and sign in page is a part of off.

Remember, one of the big requirements that we have

around micro-frontends in general.

These are all going to be projects that are developed

by different teams.

And different teams might decide to take completely

different approaches to handling and writing CSS.

So we can't necessarily guarantee that we're gonna have

any strict requirements or rules or anything like that

between these different projects.

And once more, we should almost anticipate the fact

that different teams will have slightly different

looking pages.

It's just kind of a reality of developing micro-frontends.

So I want to you to imagine one possible scenario

that might happen as a user starts to navigate

between the pricing page and the sign in page.

Let's imagine that a user starts off at the pricing page.

They're here, Everything looks totally normal on the screen.

There's some heading right there that says pricing.

Then maybe the user says, "Okay, this stuff looks cool.

Let me try signing in".

So they click on the login button up there.

That's gonna navigate them over to our authentication

application, where they will see the sign in page.

As soon as a user navigates to the authentication app

we might attempt to load up some additional CSS.

There might be some CSS that the authentication

application needs to render itself correctly on the screen.

One of those possible CSS rules might be a selector of H1,

and we might say that for every H1 we display on the screen,

let's give it a color of green.

So that would result in that header right there of sign in

appearing in green.

Very simple, very straightforward.

But the thing to keep in mind is that user,

as a user navigates around our application,

this is a single page app.

So we're not actually reloading the entire page.

We are just changing the content on the screen.

So as soon as a user navigates over to the sign in page

and they load up the CSS that says 'H1 color green'

that essentially means we have a rule

and a selector that is active inside of our browser.

Now, and anytime we show any H1 on the screen whatsoever,

it will always be given a color of green.

So now let's imagine,

that a user goes back to that pricing page.

When a user goes back, that same selector

and that same rule will still be active inside the browser.

So an user goes back to the pricing page,

all of a sudden,

pricing right here is going to appear to be green,

even though it was black just a moment ago.

So the problem here is that

as a user starts to navigate around our application,

if any of our different micro-frontend projects

have different or conflicting CSS,

the CSS from one project is going to impact the CSS

of another project.

And we might end up having

some very different looking content on the screen

as a user navigates forward and back and forward and back.

So this is a really big issue

and we really need to think about

some different ways to solve this.

So let's take a look at some possible ways

of dealing with this problem in the next video.

Topic 80. CSS Scoping Techniques

-: Let's take a look at some different ways

of solving this problem.

All the different solutions we're going to take a look at

are going to revolve around scoping our CSS.

The term scoping means that we want to somehow make sure

that we can write out some CSS

that will only affect once a project,

but not affect another.

So in other words, I want to be able to scope

some CSS to only affect the auth project,

and not the marketing one.

But here are some different options

we can take a look at.

All these different options really fall into two categories.

When we are trying to scope some CSS, that is,

some custom CSS that you and I wrote

for some very specific project,

and we also need to think about scoping some CSS

that is coming in from some outside library

that we are making use of inside of our project.

So let's first consider the option here when

when we are trying to scope CSS

that you and I wrote out inside of our project.

The first big option that we might make use of

is to make use of a css-in-js library.

These libraries are where you write out some CSS

inside of your JavaScript code,

and then try to apply it to very specific elements.

The way that a lot of these different libraries work

will cause that CSS to be automatically scoped for you.

Let me show you a diagram just to make sure

it's really clear how that scoping works.

Okay, so for css-in-js, we might want to apply

some styling specifically just to our authentication

application or maybe just,

say, the sign-in page.

To do so, you would write out some JavaScript code.

That JavaScript code is eventually

going to generate some CSS.

When that CSS is generated,

you will also get as the output,

a randomly generated class name.

So kind of a randomly generated selector.

That randomly generated class name will be applied

to the element that you are trying to style.

So it's going to make sure that,

although you've got some very plain looking CSS here,

because that is a unique, in theory,

and randomly generated, class name,

only say this particular h1 is going to have a color

of green applied to it.

This entire random generation of the class name,

and applying it to that h1, generally occurs

a little bit behind the scenes,

and you don't have to usually do a whole lot

of heavy lifting around it.

So this would be a pretty good way

of accomplishing this entire idea of scoping.

We've got some styling that's being applied to that h1,

but it's not affecting some h1 that we might display

over on our pricing page.

So that's the first option.

We're going to take a deeper look at css-in-js libraries

in just a moment so you'll get better idea

of what's going on there.

The other two options are to make use of some scoping

that might be built into your front-end framework for you.

So this is done automatically by view and by angular.

So if you're using either of those frameworks,

anytime you write out some custom CSS,

it is going to be automatically scoped for you,

and you don't have to worry about that CSS

somehow leaking out and affecting

other elements on your page.

Another option you can use is to manually

namespace all of your CSS.

Let me show you what I mean by that.

So if we want to manually namespace our CSS,

at the root element of all of our different applications,

we might add in a class of something like, say,

pricing for our pricing page,

or maybe something like auth

for our entire authentication application.

So, in other words, the most root element in each

of these different sub-projects will have

a very general class name.

Then, anytime we want to write out any custom CSS

for just our sign-in page,

we're going to make sure that

when we write out the selector,

we're going to scope it so it only tries

to affect h1 elements within that div.

So we would write out .auth, and then h1.

That would give us some selector that would apply

just to that h1 right there,

and not affect some h1 that might be displayed

over on, say, the pricing page.

This is a really easy way to accomplish some basic scoping.

The one downside to it is that it really

does require you to always remember

to have to prepend every single selector

with . whatever your, kind of, root class name is.

The reason that's kind of bad

is that you might eventually have some kind of,

say, new engineer come onto your project,

and they might just not know to do that.

So they might decide, hey, I'm going to do something

to all the h1's on this page.

Maybe put on a font size of 36 pixels,

and all of a sudden this selector right here

is of course going to affect that h1

over on the pricing page,

which is exactly what we do not want.

So, little bit risky to use this scoping technique,

but it is definitely the easiest one to set up.

If you do use this scoping technique,

you can make use of some CSS preprocessors like,

say, sass to kind of make this entire process

a little bit more automated and easy.

But it's definitely something to just be aware of.

Okay, so those are some different techniques

for scoping CSS that you and I

write ourselves into a project.

Now we need to think about CSS

that is coming from some outside source.

Say, for example, a component library,

or maybe a CSS library, something like say Bootstrap

or Semantec UI, or Bulma or anything like that.

Our first option is,

if we are making use of a component library,

we can simply choose one that makes use of css-in-js.

And as a matter of fact,

on the components I have given you,

so the marketing page components I gave you,

and the header I just gave you as well,

they are making use of a component library

called Material-UI.

This is a React component library that actually

makes use of css-in-js.

So in theory right now,

we should not have any CSS conflicts,

because we are using some scoped CSS.

In theory,

that's not what we're actually seeing in production,

but I just want to point out that in theory,

this library we're using should kind of

solve these issues for us.

We do, of course, have to figure out why

we are seeing these problems,

and we're going to investigate that in just a moment.

If you're trying to make use of something like

say Bootstrap directly,

you can still technically make this work,

but it is rather challenging.

So if you're trying to make use of Bootstrap,

there are a lot of selectors in libraries

like say Bootstrap or Bulma,

or whatever else that just say:

for all the h1's on the page, apply this style;

for all the h2's on the page;

for all the buttons on this page;

apply this style.

So as soon as you start to import or make use

of something like Bootstrap,

you're going to import a lot of global CSS

that is going to change every single element

across every different page of your application,

which is probably not super ideal.

To get around that, you can,

as one workaround here,

manually build the CSS library yourself,

and apply some of those same namespacing techniques to it.

This is a really advanced thing,

and it kind of requires you to have a good idea

of how the CSS library works

and obviously to build it yourself,

which can be definitely a little bit challenging.

Now in all of this,

you might immediately think to yourself:

Hey Steven, you're kind of ignoring

a really easy solution to all this stuff.

You might think: there is another solution here;

and you might think it's to just use the exact

same CSS library everywhere.

So maybe we could say that our entire project

is going to make use of, say, Bootstrap.

So we can have some version of Bootstrap

that is loaded up through, say, our container

to make sure that it's being made use of inside

of every single page and just say:

hey, we're using Bootstrap across the board,

that's what we're making use of.

Now this is definitely an option, without a doubt,

but I just want to remind you one

of the original design goals of our project.

We had said that we want to make sure

that all of our different sub-projects,

so authentication and marketing,

can be stand alone projects and that we should

be able to make changes to them,

upgrades, all that kind of stuff,

without affecting any other project.

So let's kind of think down the line here.

Let's say that at some future point in time,

Bootstrap V5 comes out, a new version of Bootstrap,

and we want to make use of it on our project.

Well, if we start to import Bootstrap V5

into our project, and we have two versions

of Bootstrap being loaded up at the same time,

we would fall into this exact same problem,

where maybe Bootstrap V4 has a h1 rule

of a font size of 26 pixels.

And so on the pricing page we might want to apply that rule.

And then on the sign-in page,

we might have a different rule coming in

from Bootstrap V5 that has an h1 with a font size of 36.

So in this case, as a user starts to navigate around,

they would be loading up different versions of Bootstrap,

and that would cause some different elements

on the screen to change their styling,

which again is definitely what we don't want.

Now the obvious answer here would be to just say:

hey, when we make use of Bootstrap V5,

let's just drop Bootstrap V4.

Well, that's definitely a possibility,

but remember, now we're saying that as soon as

we want to upgrade Bootstrap,

we have to make a change across

all of our different sub-projects.

Once again, that is exactly what we want to avoid

with a micro front-end architecture.

So to just make a change to the version of Bootstrap

we're using, we would have to go to every single

different project and make sure that it works correctly

with this newer version of Bootstrap,

and that could be a very serious effort.

Remember, in a lot of the cases when you make changes

between major versions of CSS libraries,

you are not just changing some very specific rules

that are being applied to some element.

In some cases, let's say in say Bootstrap V4,

there might be some selector

that you're trying to make use of,

and it might be something like card.

You might be trying to show some kind

of card element on the screen.

Well, in Bootstrap V5, rather than giving

this kind of thing a class of card,

they might have decided to rename it to like card-light,

or something like that.

And so that means that when you go to upgrade

to Bootstrap V5, you would have to go back in your project,

find every place where you use that class name of card,

and change it out to be card-light, or something similar.

So when you upgrade between versions,

it's not necessarily just rip out one version,

put the other version in.

It might require a very major re-write

of a lot of the HTML structure

and different class names you're using

inside of your project.

So, in my opinion, if we are sticking

to our original goal here of making sure

that we can make changes to one project

and not break another,

deliberately sharing CSS libraries

across different sub-projects is really not

a very good option.

So, if that's a goal for you,

you probably don't want to try to use

just the same version of Bootstrap everywhere,

'cause you're going to fall into this trap eventually.

And so that kind of takes us back to these original options.

Okay, so with that in mind now that we've gone through

some different options here,

well, this doesn't really answer what's going wrong

in our project between development and production,

so we have to figure out what's going on there.

So, let's take a pause and start to do

a little bit more investigation

into how these css-in-js libraries work,

and why it's not quite correctly

working out on our project.

Topic 81. Understanding CSS in JS Libraries

Instructor: We just had a really long discussion

around different CSS scoping techniques.

But in all that discussion, we didn't eventually come down

to some reason that we're seeing some different results

in our app between development and production.

So in this video, we're gonna get a better understanding

of how some of these different CSS in JS libraries work.

Once we understand that,

we can then use that knowledge to understand why

our app is breaking in production.

Okay, so to get started, we're gonna take a look

at how CSS and JS is used

inside this material UI library that is being used

on our homepage right now.

So back inside of my code editor,

I'm gonna find the packages marketing directory

SRC components and then we're gonna take a look

at landing JS.

Inside of here at the very top,

tremendous number of imports from the Material UI library.

This is a react component library that makes use

of CSS and JS to style all these different components.

It also exposes a CSS and JS feature that you

and I can use inside these components to write

out our own custom styling as well.

Right now, both the marketing and the container applications

are both making use of material UI.

So here is our marketing app.

Here's a container they're both making use

of material UI and therefore the same CSS and JS library.

Anytime that you have multiple different sub-projects

that are using the exact same CSS and JS library,

you might eventually run into a bug

that we can call something like class name collision.

And that is what is happening right now inside

of our project when we deploy it to production.

We are seeing different results

because we are using these same exact CSS JS library

between these two different projects.

Okay, so let's take a look, get a better idea of how

the library works first and then understand what

that collision thing is all about.

So back inside of landing JS,

I'm gonna scroll down a little bit to about line 27 or so.

You'll notice that there is a function call

to something called make styles right here

and we're eventually passing in or technically

returning an object that has a lot

of different property names.

And then for each property we've got what looks

like some CSS rules.

So I see stuff that looks like background color, padding,

margin top, definitely some CSS looking stuff here.

Let's first understand how this object is actually

being used and how it eventually applies some styling

to the different elements that appear on the screen.

For this example, we're gonna focus on around line 36 here.

We've got hero content and that's applying a style

of background, color, and padding.

Okay, so first off, quick diagram.

So we are calling make styles and providing a big object.

One of the different key value pairs inside there

is a key of hero content and it has a rule

of padding and the background color as well.

We're just going to ignore the background color

styling rule just for simplicity.

When we call make styles, two different things happen.

First off, we get back an object that has a couple

of different keys and values on it.

The keys of this object are identical

to the keys in the object that we are returning

from make styles.

So first we've got this object with a key of hero content

and the value is what we really care about here.

The value right here is a somewhat randomly

generated class name.

This class name needs to be applied

to some element that we are trying to display on the screen.

So let's first begin by trying to trace out this process

and see where this step really occurs.

Okay, so you'll notice that when we call make styles here,

we return that object that gets processed

by make styles and eventually assigned

or the result is assigned to something called use styles.

Use styles is made use of on line 67.

We call use styles that return something called classes.

Let's do a console log of classes really quickly.

So I'm gonna put in the console log, save it,

and then flip back over to local host 80 80.

I'm gonna open up my console, refresh the page,

and there's that object.

So we can see very easily

that it has that hero content key right there

and it has a string assigned to it

of make styles hero content two.

So you and I have to take that somewhat randomly

generated string and apply it to some element

that we are creating inside of our component.

We can see that being done very easily on line 73.

On line 73, there's a div, we're assigning a class

to it and we are providing the classes dot hero content.

So ultimately this div right here

you can imagine being assigned a class name

of make styles hero content dash two.

So you can kind of imagine that we end

up here with Make Styles Hero content dash two.

That's what's happening on the JavaScript side

of things.

Simultaneously as all this stuff occurs,

this mix styles function is also going

to eventually generate some CSS.

This generated CSS is going to have a class name that is

identical to the class name that was randomly generated

for this object over here.

So it's gonna be mixed styles, hero content dash two,

and it's gonna contain the rules that we had assigned back

inside this object.

So as soon as we apply that class name to some HTML

element that is being generated by our component,

that set of rules right there is going to apply

to that element.

Once again, we can see all this occurring very easily back

inside the browser.

If we go back to the browser and open up

our Element inspector panel,

I'm going to inspect this top element right here.

The thing that is wrapping homepage.

So I'll see inside of main we have a div with a class

of make styles hero content dash two,

and if I open up the style inspector,

we'll see the exact same two rules

that we had set up inside of the initial make styles object.

So there's the padding and there's a background

color being assigned.

Okay. So that is what is going on behind the scenes.

That's how these CSS and JS libraries work.

You provide some kind of configuration that says,

yeah we wanna have some hero content thing,

we wanna apply these styling to it.

We then get back some object.

We need to take that somewhat randomly generated class name

and assign it to some element.

And then simultaneously we also get some generated CSS.

Now that we understand this process

let's take a brief pause, come back to the next video,

and understand how this process changes ever so

slightly when we start to move into production.

And that's where we're gonna start to see where

this really big bug that we currently have comes from.

Topic 82. So Whats the Bug?

Speaker: Time to finally figure out

what is going on between development and production.

So remember, one of the big critical steps

inside this entire makeStyles process

is that at some point in time, our CSS

and JS library is going to randomly generate a class name.

Something like makeStyles heroContent dash two.

And the assumption here is that this is kind of

sufficiently random, so to speak.

Now one thing I wanna point out here

is that this class name that is generated is rather long.

There's a lot of characters in it.

So the CSS and JS library that we're using

and many others that you'll probably make use of as well

make a really big assumption.

They assume that in production when you extract all the CSS

or when the CSS is generated, you might want to extract all

of it into a separate CSS styles sheet.

When you start to extract all that CSS

you might not want to have a really, really long class name

for every single element that you're trying to make use of.

Because for every different class name,

every single time you make use of it,

well that's just additional characters

that represents additional space that has to be used

up inside of a CSS file.

And that means it takes ever so slightly longer

to send your CSS assets down to some user.

So as a space saving technique

these CSS and JS libraries decide

that when you are building everything for production

they're not going to use a randomly generated class name

that is really long like that right there.

Instead to save just a tiny amount of space

they decide to change that randomly generated selector

into something like jss1, jss2, jss3, and so on.

So rather than having a really long randomly

generated class name like that, it will instead be jss1.

The one is an identifier.

So it'll start off at one, and then the next time

we need to randomly generate class name

it'll go to jss2, then jss3, 4, 5, and so on.

We can very easily see this in action

if we go over to our production site.

Let's do that right now.

So at my CloudFront page,

I'm going to inspect that same kind of header element

right here or so, not the header,

but the main content thing.

And you'll notice that over here we have a class name

of jss2 for that element.

Whereas back in development we had a class name

of makeStyles heroContent two.

So again, this different class name is simply

because we are building for production.

It is simply to make that class name ever

so slightly shorter.

So now that we understand that, I now want you to think

about what is possibly going on behind the scenes here.

Lemme show you a diagram just to make sure it's super clear.

So we are building a production version

of our container and a production version of landing page.

These are two completely separate build processes.

They have no idea that the other project really exists

at build time outside of the fact

that we were loading up that module federation plugin.

But what's more important,

they've both got these separate copies

of the same exact CSS and JS libraries.

So when we start building

both these projects for production

they're gonna start to use those shortened class names.

And most importantly, they both start off

with a kinda random number on here of one.

So as soon as you have your first

randomly generated class name

due to you writing out some custom CSS

both projects when they are built, are going to use a random

and I use the term random here,

it's not truly random, that's the whole problem.

They both start off with jss1.

So your first randomly generated custom class name

when you build up your landing page might be say jss1

and that might refer to some CSS.

That is say border 3px solid black.

When you build your container

it's a totally different build process.

So the CSS and JS library generates its first somewhat

random class name and gives it the same exact name, jss1.

And this selector right here might say apply a padding

of 30 pixels.

So now when we load up our application, we load

up all the CSS for both our container and the landing page.

And between both these we have two different sets of CSS.

They both have identical class names of jss1

and jss1, and both these contain some very

different styling that are probably intended

for very different types of elements.

So on our landing page, we might have a div

with the class of CSS, or excuse me, jss1

and we might have a very similar div

inside of our container.

These divs are going to get the sum total

of both of these different sets of rules.

So both this div right here and this div right here

are going to have applied to them

a border of 3px solid black

and a padding of 30 pixels as well.

And that's the whole problem.

The whole problem is that we're using the exact same CSS

library between projects.

They generate these same somewhat random,

not really random class names.

And so when we go into production, we're seeing collisions

in CSS across our different projects.

And we can see that very easily.

If we go back over to our browser

and inspect this kind of header thing right here,

you'll notice that we're

getting this tremendous amount of padding.

That's really why we're seeing

this big difference in styling.

So the padding is the green on top and bottom.

So I'm gonna select that element, go to my element inspector

and I'll see that there are two jss2,

sets of rules being applied to the element.

The first set of rules right here has a padding

and a border color, and the second has a flex wrap

and a justify content.

So jss2 right here, the second one with the flex wrap

and the justify content, that is styling that is actually

coming from our container and that is styling that we want.

But the first jss2 is styling that is coming

out of our marketing application.

This styling right here is actually intended

for this element.

So we are applying some styling to our header that is

intended for a totally different element.

Now if we inspect this kind of hero section right here

so this thing, we have a totally different

element with the same exact class name, jss2.

And so this one is correctly getting the padding

and background color.

We want that.

But we don't want on this element

a flex wrap or a justified content.

Okay, so that is the problem.

Anytime we are making use

of the same exact CSS and JS library

chances are they're going to generate the same

not so random selectors when you go into production.

And as soon as you load out these same selectors

on the same page, you're going to get these

style collisions and everything in production

is gonna start to look totally wild,

not at all what you expect.

All right, so now that we understand the problem

it all comes down to the fact that we're kind

of reusing these same, not so random selectors.

How do we fix it?

That's a big question.

Well, let's take a look at how we're gonna fix this up

in just a moment.

Topic 83. Fixing Class Name Collisions

Instructor: At long last, we understand

what the issue is.

Hooray. So how are we gonna fix it?

Well, it's gonna be pretty simple and straightforward.

We're gonna make sure that when we build each

of our different sub-projects, rather than having our CSS

in JS library use a not so random class name

of JSS 1, 2, 3, 4.

We're gonna have it generate a slightly different style

of class name right here.

We're gonna make sure

that this string is actually sufficiently random

that it's not gonna collide against some randomly

generated class name for another project.

So to do so, we'll first get started

by going back into our code editor, finding packages,

marketing source app dot JS.

Inside of here, you might recall that we added

in this Styles Provider thing

and I had said previously,

oh yeah we're gonna eventually come back

and talk about what this thing is all about.

Styles Provider is a React component that is

used to customize all this CSS and JS generation stuff.

So we're gonna give this component a little extra option

and make sure that it's going to generate

all these class aims

for production in a slightly more randomized fashion.

To do so, we'll find the import statement

for Styles Provider.

I'm gonna add in a comma right after that.

And we're also going to import something

called create generate class name.

Then after all of our import statements,

I'm gonna create a new variable called generate class name

and that's gonna be the result

of create generate class name.

We're going to provide it a configuration object,

give it a production prefix property,

I'm gonna assign that a string of MA and then save the file.

So what does this really do for us here?

Well, this just means that whenever we start

to build our application for production,

rather than generating all these different class names

with a prefix of JSS, it's going to instead

generate the them all with a prefix of ma.

So in other words, we're going to eventually

see these class names rather than being JSS.

They'll be like ma one, ma two, ma three, and so on.

These are only going to affect the randomly generate

class names for our marketing application.

It's not gonna have any effect

on our container and that's actually good.

It means that our container is gonna have randomly

generate class names of JSS 1, 2, 3, 4

and our marketing application will have randomly

generate class names of MA 1, 2, 3, 4, 5 and so on.

And so we will not have the same collision.

Last thing we have to do is make sure we take this

and provide it to the Styles Provider.

On Styles provider, I'm going to throw in here,

generate class name and then assign it generate class name.

And that should be it.

Now we could just deploy this

as is and everything should resolve nicely in production.

We should now have separation in our randomly

generated class names between production

and between our marketing application and the container.

However, I would like to do the same thing

inside of our container as well.

We don't technically have to.

The only reason we're gonna do this is that

if we eventually add in another sub-project that makes use

of Material UI, if we forget to do the same option

then we'll once again have collisions

between the container and whatever that new project is.

So whenever you make use of a CSS and JS library

it's definitely worth checking out whether

or not you should probably set

up this kind of production prefix option.

So we're gonna go ahead

and do it to our container right now.

We don't technically have to

or we're just gonna do it to make sure

that we kind of future proof our application.

So inside of my container I'll find the SRC app dot js file.

You'll notice that inside

of here we had never set up that Styles Provider thing,

so we're gonna add that in right now.

At the very top we'll add in an import statement

for Styles Provider and create generate class name

from at material dash ui slash core slash styles.

I'm gonna save that right away just

so it wraps all that text.

We'll then create the same generate class name

being the result of create generate class name.

We'll set up the production prefix

and this case I'll give it characters of about co.

Now just so you know, in the last one we did prefix

of ma and this one we did co.

You can use any characters here you want.

I only used ma and co because they're short

for essentially marketing and container.

So just kind of matching the prefix

with whatever project we're setting this thing up in.

All right, so then we're going to set

up our Styles provider down inside of our component.

So right around the div I'll place a Styles Provider

and then close it off on the other side of the div.

And to the first element right here

we'll provide our option of generate class name,

generate class name, and that should be it.

Okay, so now each project is gonna have its own

uniquely kind of randomly generated, not really name space

that's not really good term.

I guess it's kind of technically name spaced, whatever.

They're gonna be sufficiently name spaced.

So the class names are not gonna collide.

So the last thing we have to do is redeploy

to production and just make sure this fixes everything up.

I first wanna just open

up local host 80 80, do a quick refresh,

make sure our application still least works in development.

Yeah, it does.

So I definitely don't have any typos or anything like that.

We'll then go back over to my terminal.

As usual, a quick get add, commit.

I'll say randomized production.

Last names, again, not really randomized per se

but it's a good enough message.

I'll do a get push to origin master.

Let's let that push occur, let the deployment happen.

We'll come back to the next video and just make sure

that this fixes up all of our production styling issues.

Topic 84. Verifying the Fix

Instructo

r: My deployment is complete, so I'll go back

to my cloud front production deployment,

refresh the page,

and now sure enough, styling is fixed.

Awesome.

If I inspect the header, I'll now notice

that I've got the randomly generate class name

of CO1, CO2, and so on.

And if I take a look at that hero section,

I've got ma2.

So no longer do we have any collision

between these class names.

Okay, well that's definitely a good fix.

Now the last thing I wanna mention here, just very quickly

just to make sure it's super clear, we are making use

of a CSS in JS library that is specifically embedded

into Material ui.

So when you go and build your own applications

if you are making use of a CSS in JS library

you might be making use of a very, very different one.

And these other libraries might have a better system

for class name generation.

So you might get some truly randomized class names

which means you might not have to worry

about the same issue in production.

In addition, these other libraries are probably

gonna have a different way

of customizing that class name generation as well.

So if you do realize

that this is gonna be a problem in your own projects

as you're using some other library, chances

are there's not gonna be some generate class name thing.

There's probably not gonna be a styles provider.

You're gonna have to take a look at the documentation

for your particular CSS and JS library to figure

out how you actually customize the class name generation

if it is going to be an issue.

So again, this is just overall

in this project we're working on.

I just wanted you to understand this is something

you really, really need to think

about around class name collision.

And if it is gonna be something that's gonna be a problem

you're gonna have to figure out how to fix it

up on your own projects if you're using different libraries.

Okay, so that is it.

Let's take a pause here and move on to our next really big

and challenging problem in the next video.

Section 10: Implementing Multi-Tier Navigation

Topic 85. Small Required Change to historyApiFallback

**Small Required Change to historyApiFallback**

Before starting on the next section, we will need to fix up a bug related to the historyApiFallback settings. Otherwise, you will be met with 404 errors in certain situations such as directly accessing http://localhost:8081/pricing.

Find this code in the webpack/dev.js file in both marketing and the container:

1. devServer: {
2. port: 8081,
3. historyApiFallback: {
4. index: "index.html",
5. },
6. },

You may resolve the issue by adding a / to the front of index.html:

1. devServer: {
2. port: 8081,
3. historyApiFallback: {
4. index: "/index.html",
5. },
6. },

Or, by setting to true:

1. devServer: {
2. port: 8081,
3. historyApiFallback: true,
4. },

After making this change, remember to restart both of your servers.

Topic 86. Inflexible Requirements Around Navigation

Instructor: I'm back at our application

in the development environment at localhost:8080.

Once here, I can very easily see

the home page on the screen.

Now, I want to show you a little bug

that we have inside of our app right now.

If you click on pricing, you'll see the pricing page,

but if you then click on the App link up here,

you'll notice that the url changes back to localhost:8080,

but we still see pricing on the screen.

When we click on App right here,

I would have expected to see

the main content on the screen update,

and we should probably go back

to seeing the homepage instead.

So it's pretty clear that navigation

inside of our app right now is kind of broken.

We're gonna spend this entire section

getting a better idea of how navigation works

inside of a micro-frontend app.

Before we discuss anything around navigation whatsoever,

I want you to understand

that there are a tremendous number

of different ways of implementing navigation.

So it really comes down, once again,

to you, as an engineer, sitting down and deciding

what are the requirements of navigation inside of your app.

And that's what you and I are going to do.

We're gonna sit down and say,

"How should we treat navigation?

"What are the must-have features and requirements

"of navigation inside of our app?"

Once we generate that set of requirements,

we can then start to think about a design

that actually satisfies those requirements.

So we're gonna go through a list of big requirements

in this video and the next,

so let's get to it right away.

Our first big requirement is that

we need to have some kind of routing or navigation logic

inside both our container

and inside of individual subapplications.

We're gonna add in some routing logic to the container

so that a user can navigate around

and see different subapps appear on the screen.

So for example, as a user starts to click around,

I want to have some routing logic inside the container

that decides whether to show the marketing application

or the authentication, or the dashboard apps.

Likewise, once a user sees

a subapplication appear on the screen,

we then want the user to change

between different pages of that subapplication.

So once a user loads up, say, the marketing subapplication,

I want them to be able to navigate between

the main home page and the pricing page.

Once a user loads up the authentication app,

I want them to be able to navigate between

the signup screen and the signin screen, and so on.

The last thing I want to mention here

is that not all subapps are going to require routing.

For example, right now, our dashboard application

doesn't really have any kind of multiple different pages

being displayed, and it never will.

We haven't really spoken about

what's going to go into the dashboard just yet,

but let me just tell you right now that

we don't have any requirements for navigation

inside of our dashboard app.

Okay, so that's inflexible requirement number one.

Everything needs to have the ability to have routing.

So here's our next big requirement.

Our subapplications might need to add in

new pages or routes all the time.

So for example, right now, we've really said

that our marketing application is just gonna have

the homepage right here at just / as a route,

and we've also said that it should have

a pricing page at /pricing.

But for our marketing application,

we might eventually have some kind of

"how it works" page or "about us" page,

or something like that,

and those might be additional pages

that need to be displayed at new URLs or new paths.

So we might decide to add in

some new pages to our marketing app,

and when we do so, we don't necessarily want to have to do

a complete re-deploy of the container.

So the goal of this requirement right here

is just to point out that we probably want to have

our subapplications control some amount of routing.

We might want to say that our container

should just decide which major application or which subapp

to show on the screen at any given time, but beyond that,

the container might not have any additional routing logic,

and it will be up to the subapp to decide

which individual page to actually display

once it is displayed on the screen

rather than having the container

tell our subapp which page to display.

Okay, onto number 3.

Now, this is a requirement that we're not going to

actually have to implement inside of our application,

but I just want to point it out

as something that you should keep in mind.

We might need to show two or more micro-frontends

on the screen at the same time,

and this is going to occur very frequently

if you ever have some kind of

consistent or persistent element on the screen.

So for example, let's imagine that

you are building some kind of application like this,

where you have some kind of link

or navigation stuff on the left-hand side.

So maybe this is a standalone micro-frontend app

on the left-hand side,

with some links to some other different subapps.

Whenever a user clicks on, say,

Messages or Orders or Products,

we're gonna load up that appropriate subapplication

on the main panel of the screen.

So in this case, we've really got

two distinct micro-frontends being displayed.

One is the kind of navigation elements

over here on the left-hand side,

and another is the actual content,

or the main app, being displayed on the right.

So this is a very simple example

where we just need to be cognizant of the fact

that we might need to have different routing solutions

that are kind of somehow running at the exact same time.

Okay, just one or two more.

The next big one: we want to be able to use

off-the-shelf routing solutions.

So in other words, I do not want to have to

build out an entire custom routing library

just to get this micro-frontend stuff to work.

We should be able to use existing routing solutions

such as React Router or Vue Router,

or Angular Router and CLI.

These are all open-source libraries

that are extremely popular

for each of these different micro-frontends,

or, excuse me, these different frontend frameworks.

And I want to be able to use those different frameworks

and not have to build our own custom framework

just to handle the navigation inside of our app.

So if it feels like we have to make

our own kind of custom framework around navigation,

we are probably doing something not quite so correct.

Now, having said that, there might be one or two scenarios

where we will have to write out

at least a little bit of custom code

just to add in support for one or two small things

in these different routing libraries.

And we will take a look at an example of that

inside this app.

Okay. One or two more requirements.

Let's see, is that...

Okay, two more.

So the fifth big requirement here

is that we need to have navigation features in subapps

that are running both in hosted mode

and when we are running these in isolation.

So in other words, whenever I am running our application,

the marketing app, through our container,

we need to have some very easy-to-use navigation stuff.

In other words, if I go to localhost:8080,

I should be able to see a very specific page

from our marketing application appear on the screen.

And likewise, if I go to

just our marketing app in isolation,

which right now is running at localhost:8081,

I should also be able to see the homepage

at just the root route, or go to /pricing,

and see the pricing page there as well.

So very simply, I want to be able to

kind of test out and develop all this navigation stuff

when we are running our application in isolation

and when we are running it through the container.

There should effectively be no very big difference.

All right, last one.

So if at any point in time,

we decide that we need to somehow communicate

routing information between our different apps,

that communication should be done in

as generic a fashion as possible.

We're gonna very quickly start to see

that we do have to start to share some information

between our child apps and the container.

And we're gonna realize that, well,

as we start to implement this communication,

we have a lot of different ways we can implement it.

So we just need to make sure

that we just are going to implement this communication

in as generic a fashion as we possibly can.

And the reason for this is very, very simple:

first off, our different subapplications in the container

might have different routing solutions.

Second off, even if they are using

the same routing libraries,

we might decide to use different versions between them.

Entirely possible.

And if we decide to upgrade the version

of the routing library inside, say, the container,

I would not want to have to go

and refactor all of our routing logic

inside of all of our different containers.

So in other words, we just need to make sure

that if we are doing any communication around routing,

we have to make sure it is extremely generic in nature.

Okay, so I apologize for all the lecturing here.

I know it's really boring stuff.

But again, the big reason we're going through all this

is that you're going to see

that all this routing stuff is extremely flexible.

There are a ton of different solutions we can use,

tons of different approaches, so you really have to say,

"What are the requirements of our navigation solution?"

That's the only way that you can

eventually sit down and say,

"Okay, here's the actual design we're going to use."

So now that we've established all these requirements,

let's start to refine our design in the next video.

Topic 87. A Few Solutions

Instructor: We've got our

six big inflexible requirements.

So now in this video we're gonna take a look

at how we're going to satisfy the first three.

This will be really fast 'cause I think

that these first three are pretty easy

goals to meet.

So first off, we need to make sure

that both the container

in each individual sub app can have access

to some routing features.

So for that we're gonna make sure

that our container and marketing,

and authentication and dashboard and so on,

can all have their own

independent navigation library.

In our case, for right now,

we are really just concerned

about our container and marketing.

They're both gonna have an independent,

and separate copy of React Router.

These two copies of React Router

can very easily be different versions

of the library

because we're not going to have

any direct communication

between the two that really relies

upon any built-in features

inside of React router.

Remember, if we need any kind of communication

between these two things,

it will be done in as generic

a fashion as possible.

Second big requirement.

Our sub-applications might need to add

in new routes or pages all the time.

So to make sure this is the case,

we're gonna put in

some rather basic routing logic

inside of our container.

The container is going to take

a look at some just kind of root part

of the URL and use that to decide

exactly which micro front-end it's going to show.

So if the current path is just slash by itself

or slash pricing, the container is going

to try to show the marketing application

on the screen and it's not going to try

to provide any further configuration

to the marketing sub app

or try to tell it exactly what page to display.

Instead, it'll be up to the marketing application

to implement some routing logic to say,

"Okay, if we're currently at forward slash

or slash pricing,

we should show the landing page

or the pricing page.

So really two levels of routing logic

inside a container.

Just decide which micro front-end to show

inside of marketing or the sub app,

just decide which page to show.

And then number three,

remember we're not gonna actually implement this

'cause we don't have any pages inside of our app

that need to be displayed simultaneously.

But if we ever had the need to show two

or more micro front ends at the very same time,

we can use the same style of approach

where we would have our container say,

if we are at a root route of just slash,

then let's try to show both marketing,

and pricing on the screen at the same time.

And it would then be up to

these two sub applications to decide

which individual page to display.

So this kind of two tiered routing approach

will work just fine,

even if we are trying to show more than one micro

front-end at the same time on the screen.

All right, so like I said,

very quick and easy for these first three.

For the next three,

are going to require a little bit

more explanation and we need to really start

to understand some internals around

how a lot of these different routing libraries

work and what's more,

we're gonna have to understand very quickly

a really big gotcha around how they work as well.

So let's take a look at

the next three in just a moment.

Topic 88. Which History Implementation?

-: We're not gonna cover items four, five, and six just yet.

First, we're going to instead have a quick side discussion.

I wanna give you a quick overview

on how a lot of these different routing libraries work.

So I specifically am talking about routing libraries such

as React-Router or Vue-Router or Angular Router and so on.

All these different routing libraries can really be thought

of as two separate parts.

The first part is something called a history object.

This history object is an object that is used to figure

out what path the user is currently visiting

inside our application and then eventually make changes

to that path as well.

The other big major component

of these routing libraries is something called a router.

The router is used to figure out what content is showed

to the user based upon the current path

that they are visiting.

In this video, I'm gonna focus

on exactly what this history object is all about.

So we're gonna take a look at a couple

of diagrams that are going to more closely

explain what's going on with history.

All these different routing libraries generally

implement three different kinds of history objects.

When I say three different kinds, I mean to say

that there are generally three different methods

of deciding what route a user is currently visiting inside

of our application.

The first major kind of history, and by far the most

popular is something called a browser history object.

This is an object that is gonna decide upon

or decide what path a user is visiting inside

of our application by taking a look

at the URL inside the address bar.

Browser history is gonna take a look

at everything after the domain and use that portion

of the URL right there to figure

out what path the user is currently visiting.

So in this case, browser router would say

that a user is visiting marketing slash pricing.

Browser history would then take that string right there,

pass it off to a router,

and the router would decide what content to show

on the screen based upon this string.

The next kind of very common history is called Hash history.

We're not gonna use this inside of our application

so we're just gonna skip over it for now.

And then finally, there's another kind

of history object that is called memory history

in React-Router, and it's referred

to as abstract history inside of Vue-Router.

This is a history object that stores the current path

that a user is visiting inside

of memory or essentially inside of your code.

So there's no kind of visual indication

of what path a user is visiting.

A abstract or memory history is not gonna use the address

in the address bar in any way to figure out

what the user is currently visiting inside of your app.

So what does all this stuff have to do with our application?

Well, remember we had said that we're gonna have kind

of a multi-tier navigation approach.

We're gonna have some kind of routing logic inside

of our container that's gonna decide what sub app we want to

show at any given time.

And then inside of all of our sub applications

we're gonna have some further routing logic

to decide what actual page to display on the screen.

So in this case, we're gonna be using React-Router

in three separate locations.

Each copy of React-Router is a hundred percent separate.

And remember the whole reason for that is that we

should be able to have different versions of React-Router

or even different routing libraries entirely inside

of each of these different applications.

Whenever we create a copy of React-Router,

we have to tell it exactly what kind

of history we want to use.

So we can tell React-Router to use browser history,

memory history, or hash history.

So for our application, we need to figure

out and decide what kind

of history we are going to use in every location.

The most common way of setting up navigation inside

of Micro Friend and apps when it comes

to history is to make use of browser history

inside of your container and then memory history

inside of all the different sub applications.

The reason for this is very simple.

We've been talking about browser history

and memory history and so on.

Just to make sure it's really clear,

all these different navigation libraries such

as React-Router and Angular Router

and Vue-Router provide their own custom implementations

of stuff like browser history and memory history and so on.

Even though they all have very similar names,

each library has its own custom implementation.

So that means that inside of our authentication app

if we eventually decide to use Vue-Router over here

and let's imagine for a second

that we were using browser history instead,

even though this thing is called browser history here

and browser history here,

these are two different versions of browser history.

One is browser history

as it is implemented inside of React-Router.

The other is Browser history

as it is implemented inside of Vue-Router.

These different browser history objects as we discussed

take a look at the URL inside

of the address bar, and most importantly

they eventually change that URL as well.

So we are talking

about possibly multiple different objects that are

trying to change that URL

inside the address bar at the exact same time.

And each of these different objects are implemented

in a slightly different way and you don't necessarily know

or have any guarantee about how that implementation is done.

So you could very easily get in a scenario where

there's some kind of race condition inside of your app where

you might try to navigate to some location and maybe

this browser history object would decide to update the URL

at some point in time.

And then this one would also decide to update the URL,

and the order in which those updates occur might be slightly

different and they might decide to update the URL

in slightly different ways.

So if you are using multiple different copies

of browser history inside of your application, you might run

into some big trouble at some point in time.

So to get around that, we make use of memory history inside

of all the children instead.

So the browser history inside of Container is the only copy

of history that will access the address bar and take a look

at the URL inside there and eventually update it as well.

All these other history objects inside of all

of our different children are going to have their own

kind of separate copy of what the current URL is.

And they can change that, update it as they please,

and we will eventually add in some code to make sure we sync

that path across all the different copies

of history inside of our application.

But what's most important is that as these different copies

of history decide to start to navigate around,

they are not also going to go

into the address bar and try to change it in some way.

Again, that could lead to some big trouble down the road.

Okay, so now that we understand

our general history approach here,

we'll start to implement this inside

of our project in just a moment.

Topic 89. Surveying Our Current History Setup

Instructor: We're gonna make use of Browser History

inside the container and then Memory History

inside of all the children.

To implement all this stuff,

we're gonna first take a look

at what we've done around React Router

inside of our application already

and then plan out the different

updates we need to make.

I say that because some

of the code I gave you

actually has some React Router

related stuff inside of it.

So we just need to figure out

what we have right now

and exactly how we need to change it.

So to get started

I'm gonna open up my editor,

find the container directory,

and then open up the app dot js file inside there.

Inside of here,

if we scroll down to our React component

you'll notice that there's a tag

inside here called Browser Router.

This is a React component that is provided

to us by React Router dom.

It creates a router object

and behind the scenes

it also creates a browser history object

for us as well.

So whenever you see a Browser Router,

that essentially means

that we have this situation right here.

We are making use of React Router and internally

it is creating a copy of Browser History for us.

So that means our container

is already set up as we want.

So we don't need to make any changes

right now to our container.

Let's now repeat that same process

inside of marketing.

Inside of marketing,

I'm gonna find my app dot js file

and inside there unfortunately,

you'll see that we are also making use

of a copy of Browser Router here as well.

I say unfortunately because that means

that we are currently making use

of browser history inside

of our marketing project,

but we want to use Memory History instead.

So we're going to have to do

a little refactor to marketing.

As we start to replace that copy

of Browser History with Memory History,

we're gonna start to break a lot

of navigation inside of our project.

So in the short term

as we start to implement Memory History

inside of here in the next video or two,

it's gonna seem like

we are breaking everything for no good reason,

but don't sweat it.

Immediately after putting Memory History inside of here

we're then going to have

a deeper discussion around

how we're going to keep

all these different history objects in sync

and make sure that navigation

is a very seamless experience inside of our app.

So let's take a pause right here.

When we come back to the next video,

we're gonna start to customize

our marketing project and make sure

that we make use of a Memory History object

inside of here rather than Browser History.

Topic 90. Using Memory History

-: I'm inside of our marketing app.js file.

So as we just discussed, we do not want to make use

of Browser Router inside of here anymore.

We want to use a different kind of router

that's going to use a memory history object instead.

So to do so, we're gonna go up to our import statement

at the very top from React Router dom, and rather

than importing something called Browser Router right here

which again internally creates a copy of browser history

I'm gonna replace it with just simply Router instead.

A simple Router like this allows us to

provide the history object that we wanted to use.

Rather than having React Router dom,

create the history object for us.

I'm then going to go down to Browser Router,

and replace those two tags with Router instead.

So now we are creating a copy of Router,

that doesn't make its own history.

That means we have to provide it some history to use.

A very obvious thing to do here, would be to

create a history object inside this file and,

then pass it into this Router.

Where we're gonna take a slightly different approach.

Rather than creating our memory history inside the

app.js file, we're gonna create it inside of our

bootstrap.js file instead.

This is done, because we're gonna eventually come back

and customize this history object quite a bit,

and we're gonna want to write out all the code

for that inside of our bootstrap.js file.

So we are gonna create a copy of memory history

inside of here, then pass it down to our app component,

receive that memory history and provide it to this

router thing right here.

So let's write out the code for that right away.

Inside of bootstrap js I'm gonna add in an import statement

at the top for, create memory history,

from history.

You'll notice that I imported this

from a library called history, not React router dom.

It turns out that React router Dom internally makes use

of this history library

for getting access to all this history related stuff.

So we're then going to use this create memory

history function to create a copy

of memory history inside of our mount function.

So inside if you're gonna say const history

is create memory history,

and then going to pass that

as a prop down into our app component.

So I'll put in history is history like so.

I'm then going to save this file, go back over to app.js.

I'm going to receive that history object as a prop.

So curly braces history, and then we are going to

provide that to this router thing right here.

So history, history, like so.

So all we've done at this point in time is take out

the browser history that was being created automatically.

We are now creating a memory history instead,

and we are creating that inside of our bootstrap.js file.

Again, we are creating it inside of here

cause we're going to eventually add in a lot of code

to this file to sync the current state of our history inside

of marketing with the history object inside of container.

You'll see all that in action very quickly.

Right now, let's save all this, go back over

to the browser and just test everything out.

Okay, so back over here, I'm gonna make sure I

refresh the page. I'm at local Host 80 80,

and then I'm gonna click on pricing.

Now when I do so, I do continue to see pricing appear

on the screen, but you'll notice that the URL did not

update at all. It still says local host 80 80,

when it should say, local host 80 80 slash pricing.

So it looks like this change we just made,

started to break something.

If I click on app, it looks like nothing happens at all.

And finally, if I try to navigate

to local host 80, 80 slash pricing

rather than seeing the pricing page appear

I see the homepage instead.

So it looks like immediately we just broke a

lot of navigation stuff.

So this is what I refer to just a moment ago.

Soon as we make this memory history change

a lot of stuff starts to get really weird really quickly.

So let's figure out exactly what went wrong

wrong at this point in time.

And then we're gonna start to kind of fix

up some of the different routing logic inside of our app.

Topic 91. Why the Strange Results?

Instructor: All of our routing logic is now broken,

so let's make sure it's super clear why that is.

Okay, so a couple of quick diagrams.

So in these diagrams,

we're going to figure out exactly what is occurring

behind the scenes whenever we go to some address

and then eventually click a link on the screen.

So I first wanna think about exactly what happens

when we go to localhost:8080 by itself,

and then eventually click on the pricing link

inside of the Marketing app.

So first off, when we go to localhost:8080,

we create two separate copies of history.

One is browser history, the other is memory history.

Both these have an initial location value

stored inside them.

Something that says, here is the current path

that we are at right now.

Your browser history object

is always gonna use an initial path right here

equal to whatever the path is inside the address bar.

So right now, the path inside the address bar is just slash

and that means that the initial value

of the browser history object is slash.

The initial value of a memory history object

on the other hand is always slash no matter what.

So either way, both of our history objects

start off at just slash.

We then eventually click on the pricing link

in the Marketing app.

That is going to update the current location

that memory history thinks we are at to /pricing.

We then see some appropriate content up here on the screen.

So if I go to localhost:8080 and click on PRICING,

I see the pricing component.

That is because we updated

our Marketing application's memory history,

and our router that is being created

inside the Marketing app has now responded

to this new URL or this new path

and is showing the appropriate content on the screen.

However, clicking on that link makes no change whatsoever

to the current URL or whatever the browser history object

inside of container thinks that we are at.

So as far as our browser history is concerned,

we are still at a route of forward slash,

and that's why we do not see the URL update at all.

That's why the address bar stays at localhost:8080.

So now let's consider the other scenario that we looked at,

where we navigate initially to localhost:8080/pricing.

So I'm gonna repeat that experiment really quickly.

There we go.

And once again, we see the homepage by default.

So once again, our browser history

is going to take its initial path from the URL.

So it starts off at /pricing.

By end, once again, memory history, no matter what,

always starts off at just slash by itself.

So then eventually, we click on pricing.

We're gonna update the location

that memory history thinks we are at,

and now they both think we are at /pricing

and we see the appropriate content up here on the screen.

But once again, if I try to navigate away,

even if the URL up here changes,

our memory history object still thinks we are at /pricing.

Okay, so clearly, stuff is a little bit awkward

and very broken right now, and you might be thinking,

"Why are we using memory history at all?"

Don't worry, we're gonna add in a little bit of code

to sync all these different history objects together

in a pretty smooth way in just a little bit.

The last thing I wanna mention very quickly here

is to just make it really clear

how this navigation stuff is kind of scoped.

So right now, our container router

is eventually showing some different content on the screen,

including a link that is placed into our header.

So that is the header up here, this little App thing.

If we click on App, that is an element,

and its immediate parent router is going to be the router

that was created inside of container.

So if you click on the link inside the header,

that is gonna make a change

to the history object tied to our container router.

At the same time,

we have some links that are being displayed

inside of our Marketing application.

Their closest parent router is the marketing router.

So if we ever click

on a link inside of our Marketing application,

such as the PRICING button right here,

the closest parent router is gonna be the marketing router

which is tied to our memory history object.

So in other words, clicking on a link

inside of our Marketing application

is going to update the memory history only,

as opposed to trying to change anything

inside the container history or the browser history.

So I just wanna point out that all these links

that are now being displayed are kind of scoped in nature.

Okay, so now that we've kind of laid out what's going on,

we've got a better idea of some

of the issues with having multiple distinct history objects

on the screen at the same time, like I said,

we need to somehow sync these different objects together.

We need to make sure that as we start to navigate around,

we detect in other history objects

that navigation just occurred

and update the current path inside of those as well.

So let's take a look

at how we're gonna approach this in just a moment.

Topic 92. Communication Between Apps

Instructor: One last very quick lecture,

and then we're gonna write out the code

to sync these two history objects.

The reason that we're doing

another just talking kind of quick presentation lecture

rather than writing out the code

to sync these history objects

is that we're gonna very quickly start

to tackle two very big topics inside of microfinance.

The first is, well, all this navigation stuff.

The second is, we need to do some communication

between Marketing and Container,

communication between services.

That is definitely something that we need to consider.

So let's first kinda make sure it's super clear

what we are trying to achieve.

So two different things need to happen.

First off, if a user ever clicks a link

inside the Container that is governed

by our browser history, we need to communicate that change

down to our Marketing sub-application.

And our Marketing's Memory History object

should update its current path.

When it updates its current path,

that should automatically update the content

inside of our Marketing application.

So as long as we can communicate this change

and the current path down to Marketing,

everything should work as expected.

But then we also need to handle the reverse situation.

So whenever a user clicks a link that is governed

by Marketing or really the Memory History,

then we need to make sure

that we communicate this change up to our Container.

Once that change gets communicated up,

then we need to make sure

that our Container's Browser History

updates its current path

and reflects whatever the current path is

inside of our Marketing application.

So again, two different problems at the same time.

We're not only talking about syncing these history objects,

but now we are kind of introducing

and saying we need to handle communication

from the Container down to Marketing

and from Marketing up to the Container.

So just a quick reminder

before we start to write this code out.

I want you to remember,

whenever we are doing this communication, we want it to be

in as generic a method of communication as possible.

So one way that we could very easily handle this stuff

is by taking these history objects

and just directly exchanging them

between these two sub-applications.

That is a hundred percent feasible,

but we definitely do not want to take that approach.

We don't want to exchange these very complicated objects.

The reason for that is very simple.

At some point in time, we might decide

that we don't want to use React Router DOM

inside of our Marketing application anymore.

We might decide that we want to use, say, Vue Router

or we might decide that we want to use a different version

of React Router altogether.

If the communication between Marketing

and History occurs through some complex object,

well, as soon as we start to replace the library

that that object depends upon,

we're gonna have to make a change

to the other application, in this case, Container.

So we need to make sure that all this communication is done

with very simple objects, with very simple mechanisms.

So we're thinking about stuff like simple events,

simple callbacks, and so on.

Okay, so now we understand the big challenges here.

Let's start to focus on how this communication

is going to take place in the next video.

Topic 93. Communicating Through Callbacks

Instructor: Let's take care

of our first communication between container and marketing.

So we're gonna first think about handling the case

in which a user clicks a link inside

of our marketing application.

To handle this, we're gonna make sure

that whenever our container shows the marketing application,

whenever container calls the mount function, in other words,

we're gonna have the container pass down

a call back function called onNavigate.

Then at some future point in time, whenever a user clicks

on, say, the pricing link inside the marketing app,

as usual, we will allow our memory history's current path

to be updated to /pricing.

That is going on behind the scenes.

We don't actually have to do that,

but as soon as that occurs, we're gonna make sure

that we also call this on navigate callback,

and that is gonna communicate back to the container

and tell the container that the current path has changed

in some way.

So again, the entire key of communication here

is that our container is gonna pass down a callback

to the marketing application

when it calls the mount function.

So let's try this out.

Back inside my code editor, I'm gonna go

to my container and then the marketingapp.js file.

So here's where our container calls the mount function

for our marketing application.

So I'm gonna pass

into there a second argument that's gonna be an object

because we eventually might pass

down a lot of different options.

Inside this object,

I'm gonna pass in a function that we will call onNavigate.

So whenever we call onNavigate, right now,

let's just do a very simple console log and say something

like, the container noticed navigation in marketing.

So we are providing our callback right here.

Now we need to go into our marketing application.

We need to make sure that we receive this callback function.

And again, as soon as some navigation actually occurs,

we need to call

onNavigate inside of our marketing application.

So back inside of my marketingSRCbootstrap.JS file,

here it is right here.

Here's where we define our mount function.

So as we just saw,

mount is now going to be receiving a second argument.

It's gonna be an object that has a key inside

of it of onNavigate.

So that is our callback function right there.

So now all we have to do is make sure

that whenever some navigation occurs,

we call the On Navigate function.

To do so, we're gonna use some functionality that is built

into the history object right here itself.

The history object that comes back

from calling Create Memory History,

has an event listener tied to it called Listen.

We're gonna call History.listen.

Whenever some navigation occurs,

this history object is going to call any function

that we have provided to this listen thing.

So we're gonna put in onNavigate like so.

Okay, so now let's trace out the entire series

of steps are going to go on; at some point in time,

our container application is gonna decide

to mount our marketing application.

It will create a callback and then call the mount function

and provide that callback as onNavigate right here.

We then create a memory history object,

and we tell the memory history object

that whenever the URL changes, whenever the path changes,

we want it to automatically call the onNavigate function.

So then whenever a user clicks on the link,

we're going to update our memory history and memory history

will automatically call the callback,

and we should then see this console log

at our, inside of our console.

And that's the entire flow.

So let's save all this.

Go back to browser and see what happens.

So back over here, I'm gonna click on the Pricing button.

I'm gonna refresh first.

There we go.

And then click on the Pricing button.

And as soon as I do so, I see my console log.

So the container noticed a navigation attempt inside

of marketing.

So now inside of our callback,

inside the marketing app file,

rather than doing that console log right there,

we could attempt to figure out

what URL the marketing application navigated

to and then use that to update our browser history

inside the container.

So let's take care of that last step in the next video.

Formularbeginn

Formularende

Topic 94. Syncing History Objects

-: I'm inside of our MarketingApp.js file.

So here's our onNavigate callback

that gets called automatically

whenever some navigation occurs inside of our marketing app.

So now inside of here

we need to figure out where the marketing app navigated to.

We need to then take that new path and use it

to update the current path inside of our container.

So this is the kind of synchronization step.

So the first thing we really need to figure out here

is how do we tell where the marketing app navigated to?

Well for that turns out that the listen function,

that is eventually calling onNavigate,

is going to give us an argument

which we refer to as the location.

This is an object that has some information

about where we are about to navigate to

inside the marketing application.

So rather than doing a console log of that plain string

let's do a console log of location instead.

I'm gonna save this file, flip back over,

make sure I refresh the page,

and then again click on pricing.

So here's the location object.

And again, this is what is being communicated

from our memory history inside of marketing

up to our container.

So you'll notice inside of here

there's one property that we really care about for right now

it's called pathname.

And that's the path that our marketing application

is attempting to navigate to.

So you and I want to take that pathname property

and use it to update the current path

that we are visiting inside the container application.

So back inside my onNavigate function.

Rather than receiving the entire location object

let's instead only receive that pathname property.

Now notice that pathname has a lowercase 'n' inside of it,

not uppercase.

I'm gonna do one other thing inside of here

that right now it's gonna look a little bit mysterious

but in just a moment it's gonna make a lot of sense.

Rather than just destructuring this pathname property,

I'm gonna destructure it

and simultaneously rename the property as well.

You can rename a property that you are destructuring

by putting in a colon

and then the name that you want to rename this thing to.

So I'm gonna rename it to nextPathname, like so.

So now inside of our onNavigate function,

we have access to a variable inside of here

called nextPathname, and this represents the path

that our marketing app is attempting to navigate to.

I'm gonna try to console log that out just by itself

and just make sure we got all this syntax here

a hundred percent correct.

So again, refresh, click on pricing

and now I see just slash pricing by itself.

Very good.

Okay, so again, we just did a little bit of renaming there.

Again, the reason for that will be clear in just one moment.

So now that we know exactly where

the marketing application is attempting to navigate to

we need to make sure that we update the current location

of our container as well.

And that's gonna be the actual synchronization step.

So to update the current path that we are looking at

inside of our container, we need to get access

to the history object that is being created inside

of our container.

So the actual browser history object.

That's how we navigate around.

To do so, we're gonna add in an import

at the very top of the file.

We are going to import, useHistory from react-router-dom.

Then right after this ref right here,

we'll say history is useHistory.

So this history object right here is the history object

that is currently being used inside of our container.

So in this case it is a copy of a browser history.

Now we're gonna take that history object

and update the current path that we are looking at

with nextPathname.

So I'm going to remove the console log right there.

We're going to call history.push

and then provide nextPathname, like so.

This push function essentially means,

"Hey, history object, we want to navigate to this new path

and the path we are attempting to navigate to

is nextPathname."

Okay, so we're gonna save this and do a very quick test

but then immediately come back

and make a very small change to this function.

So I'm gonna go back over refresh once again.

And now when we click on pricing

our goal is to see the URL update.

That would be an indication that we are correctly

telling browser history inside of our container

about this new URL that we're attempting to navigate to.

So I'll click on pricing and sure enough there's the update.

Awesome.

So we've now satisfied one direction of communication.

Now whenever a user clicks on a link

that is governed by marketing

we are communicating that change up to container

and we are updating the browser history inside of container.

Now like I said

there's one very quick change we want to make.

Turns out that, well,

if you really consider this flow right here

we're essentially saying whenever one history

object changes, automatically change the other.

It turns out that you can very easily get

into an infinite loop here where whenever

we tell one history object to update, it'll tell the other.

That one's gonna update

and then tell the other immediately back that it updated.

And we're gonna get into this infinite flow

where each history object just keeps on telling itself

that it just made a change.

So to prevent that infinite flow,

we're going to add in a little check inside of here

to decide whether or not we actually want

to have our history object navigate.

So to add in the check,

we're gonna take a look at the current pathname

that we are at inside of our container application

and we're going to verify and make sure

that our current pathname

and the nextPathname are different.

If they're different, that means that we probably

need to do some navigation

but if they're the same, that means we are already

at the URL that we're supposed to go to

and therefore we should not attempt any navigation.

So to figure out what path we are currently at

we're going to do const, pathname, history.location.

So this right here is gonna be the path

that we are currently at inside of our container app.

So then we're going to check and see

if pathname is not equal to nextPathname.

If they are not the same

that means that we need to do some navigation.

We should cut and paste that line inside the if statement.

And there we go.

Now maybe it's a little bit more clear

why we rename that pathname property,

because now we have a variable named nextPathname,

that's where we are being told to navigate to.

And we have pathname, which is where we are currently at.

So again, only if these things are different

should we attempt any navigation.

So let's save this and test one more time.

I'll do another refresh.

And actually I'm gonna first go back manually

to localhost.8080.

I'll click on pricing.

There we go.

We see the new content and the URL is updated as well.

Perfect.

Well we just satisfied one direction of communication.

Now we need to take care of the other direction.

So whenever a user clicks on a link

that is governed by the container

we're going to automatically update our browser history

and we have to communicate that change down to marketing

and tell marketing that it needs to update

its memory history's current path.

This line of communication going downwards

is gonna be just a little bit more difficult

but at the end of the day, it's gonna be very similar

in nature in that we're going to eventually call some kind

of function that says, "Hey, go update something."

So let's get started on this other flow in just a moment.

Topic 95. Running Memory History in Isolation

Instructor: Before we handle this other line

of communication,

there's one very quick fix I want to add in.

And I'm only putting this right here inside this video

just in case you see this error

and get really concerned about it.

So I'm gonna go back over to my browser

and I'm gonna take a look

at my marketing application in isolation.

So remember that's at localhost 80 81.

Once you do so, you might see an error like this,

"Cannot read property 'onNavigate' of undefined."

So this error is coming up for a very simple reason.

If we go back over to our editor

and find our bootstrap.js file inside of marketing,

remember we are now saying that our mount function is

going to receive an options object as the second argument.

And then inside there we are assuming

that we are going to get some kind of "onNavigate" function.

Well remember, mount is not only going to be called

by our container app.

Mount also gets called if we are running our app

in isolation.

So down here a little bit lower, we are calling mount

but we are not providing an options object

as the second argument.

And that's why we are seeing that error of,

"Cannot read property 'onNavigate' of undefined."

So to get that error to go away for right now,

we can add in an empty options object

as the second argument.

And then in addition, back inside of our mount function

before calling history.listen,

we're gonna make sure that there actually is

a "onNavigate" function that was provided.

So I'm gonna wrap this with, if onNavigate.

In other words,

if we actually did provide a navigate function,

then let's go ahead and call that listen function like so.

Okay, let's save that.

Go back over, take a look at localhost 80 81.

And once again, everything is working as expected.

One thing I would mention, in addition,

is that if you click on pricing

again, you're not going to notice the URL right here

updating if you're at localhost 80 81.

That is because when you are running our application

in isolation, we only have a memory history object.

We're going to eventually come back

and make sure that we use a browser history instead

if we are running this application in isolation

just to make it a little bit easier to develop

because that will allow developers to see

what current path they are visiting at any given time.

But like I said, we'll take care of that in just a moment.

Okay, So quick pause and then back over

to our main path right here with our container

and we're gonna make sure that we are setting up

that additional line of communication.

Topic 96. Container to Child Communication

Instructor: All right, let's take care of that

other line of communication.

So the case in which user clicks a link

that is governed by our container.

So in this case, we need to somehow detect that navigation

and communicate the change

down to our MarketingApplication.

This is actually gonna be a little bit more challenging

for one single reason.

If we go back over to our MarketingApp.js file.

Remember this is rendered by our container application.

Here's where we call mount.

That is where our actual MarketingApp is created.

When we look at all this code right now

we do not currently have any kind

of handle on our MarketingApp.

In other words, we have no way to reference it.

We have no way of communicating any values to it.

We call mount and that's pretty much it.

The first thing we really have to do here

is figure out some way to communicate information

down to our MarketingApplication.

The one is that is created whenever we call mount.

To do so I'm gonna go back over to my bootstrap.js file.

So this is the one inside of our marketing project.

And to have some line of communication

from the container down to our MarketingApp

I'm gonna have this mount function return in object.

So inside of here we're gonna place some number of functions

that our container project can call

to somehow update or change our MarketingApp.

So the first function we're going to add in here

and for right now, really the only one

I'm gonna add in a new function side of here

that we'll call onParentNavigate.

So the thought here is that anytime a parent navigates

we want to call this function,

but in other words anytime the container

does some kind of navigation

we want to have our container call this onNavigate

or excuse me, onParentNavigate.

But right now, I'm gonna add in a simple console log

and say something like about container, just navigated.

So then we can go back over to our MarketingApp.js file.

And now whenever we call our mount function right there

remember, we now get back some kind of object

that is gonna have that onParentNavigate function

inside of it.

So I'm going to receive that object

and destructure out onParentNavigate.

So onParentNavigate.

So now inside this useEffect

we need to somehow detect navigation occurring.

And whenever navigation occurs

we have to call onParentNavigate

and then eventually pass in the new route

that we are taking a look at or the new path.

So after that entire mount function call down here

at the bottom of useEffect,

we're gonna set up an identical kind of listener

on the history object that we have

inside of our container applications.

Remember history right here?

That is our browser history object.

A browser history object has the same exact .listen function

that the memory history does that we made use of

back inside of our MarketingApplications bootstrap.js file.

So inside of our MarketingApp.js after the useEffect.

Down here, we'll do history.listen.

And anytime there is any change to our browser history

we want to call onParentNavigate.

So I'll put in onParentNavigate.

And the last thing I wanna do inside of here

just in case I forget

notice how we are using the useEffect function right here.

If you're a little bit savvy with React,

you might recall that the useEffect function

is going to run itself anytime this component right here

is updated or changed in any way.

And the way that we limit

how often the useEffect function is executed

is by putting in a dependency array as the second argument.

Right now we probably only want to have

our useEffect function right here run exactly one time

when our MarketingApp is first rendered.

In other words, the MarketingApp component, I mean to say.

So, to limit how often this useEffect function

actually gets invoked, I'm gonna put in

a second argument right there of an empty array.

That means only try to run this function

when our MarketingApp component

is first rendered to the screen.

Okay, so that looks good.

Let's save this.

We'll then go back over to our browser and do a quick test.

Second refresh, I'm gonna click on pricing.

Okay, it looks like everything still works.

And now if we do some navigation inside of our parent

by clicking on that app right there inside of our header

we see "Container just navigated".

Our URL updated which is entirely expected,

but now we are getting a signal

down to our MarketingApplication.

Something to tell the MarketingApp,

"Hey, we just did some navigation up here

inside the container, just like you'd want to know."

Now we need to go back down to our MarketingApplication

and whenever some navigation occurs inside of container

we need to make sure

that we also navigate inside of our MarketingApplication.

So again, just the exact opposite

of what we did a little bit ago.

Okay, so back inside of our bootstrap.js file.

Right here, rather than doing that console log,

whenever we call onParentNavigate,

because this is being called by the history.listen function.

We're going to receive that same exact

kind of location object.

I'm gonna do a quick console log of that location object

just to remind you what it looks like.

Okay, I'm gonna refresh, I'll click on app up here.

And there's the history object,

so exactly identical to what we saw previously.

The property inside of here that we care about is pathname.

So just as we did before,

I'm gonna destructure out just pathname

and again, I'm going to rename it to be nextPathname.

Just to be clear that this is some incoming path name

that we might want to navigate to.

And then once again, to make sure

that we don't get into some kind of infinite loop,

we need to make sure that the nextPathname

is different than the path name

that we are currently looking at.

So inside of here, we're going to use our memory history.

We're gonna take a look at what our current location is

and we're just gonna make sure that our current location

is different than the one that we are trying to navigate to.

So, we'll Destructure off pathname, from history.location.

We are gonna delete the console log

and we'll say if pathname is not equal to nextPathname

then we probably want to do some navigation.

So we'll do a history.push to nextPathname like so.

And that should be it.

Let's flip back over and do a quick test.

I'm gonna make sure I refresh the page.

I'll then click on pricing.

We go over to pricing.

I'll click on app inside the header

and we go back to our homepage.

Awesome, so we can now toggle back and forth

every single time our URL updates

which is exactly what we want.

And if I start to click on my back button

that works almost as expected as well.

You'll notice that there might be just like,

well, I guess I got just a touch of lag inside there.

To really test this out

I should probably open up a totally new browser tab

just to clear out my navigation history.

I'm gonna do that really quick.

And I'll just very quickly make sure that navigating,

I go pricing, home, pricing, home, pricing, home

and I'll click the back button a couple times

I go back, back, back, back, perfect.

Okay, so I think that going back

and forward works exactly as expected.

Okay, so that's it.

We've now set up a line of communication in both directions

and we are keeping these history objects in sync.

So this is really good progress.

Let's take a pause right here and continue in just a moment.

Topic 97. Using Browser History in Isolation

Instructor: The last thing

I think we really need to handle

around all this navigation stuff

is making sure that we can run marketing in isolation

and see the actual URL that we are currently visiting.

So remember right now at Local Host 8081,

we can navigate around,

but we are making use of our memory history,

which means we see nothing inside the address bar,

and that would make developing

the marketing application a little bit of a pain,

because developers would be navigating around

and not really knowing where they are visiting

at any given point in time.

So I think that we need to

do a little bit of configuration here.

Maybe we should say that if we are running

our marketing application in isolation,

in development, we should make use

of a browser history object instead of memory history.

So that would allow us to have

much easier development

when we're at local host 8081.

So to do so,

I'm gonna go back to my bootstrap.js file.

So this is the bootstrap file inside

of our marketing project.

At the very top,

where we have our import from history,

I'm gonna add in

a create browser history, from history as well.

This function as you'd guess

allows us to create a browser history object.

I'm then going to go down

to our if statement right here,

where we decide whether or not we are in development

and in isolation.

Then on my mount function,

I'm gonna say that my options object right here

might take in an additional option,

let's call it default history,

and then provide it, create browser history.

So in short, whenever we call the mount function

when we are in isolation,

we're going to create a browser history instance

and provide it on the options object

as a property called default history.

So then inside of our mount function,

back up here,

I'm gonna destructure off default history.

We are only providing a default history

if we are in development.

So I'm going to say,

if we were given a default history, let's use it.

I'm going to assign that to history.

Otherwise, if we did not get one,

we will default to creating our own memory history.

So that's what the "or here"s gonna do.

If we provided a default history, go ahead and use it.

Otherwise, make use of a memory history.

A default history is only going to be provided

if we are calling the mount function

from right here for right now.

So obviously, our container could decide to give us

a different default history if it wanted to,

but right now the container is not doing that.

So if our application is being rendered

through the container,

we're still going to be creating our own memory history.

Okay, so let's save this, pull it back over, and do a test.

So now if I refresh the page at local host 8081

and click on pricing,

now the URL updates,

because we are now using a browser history in isolation.

So again, this would allow us

to do a lot more easy development,

cause we can actually see that URL

and make changes to it.

But if we go back over to local host 8080,

back over here,

we are still making use of a memory history.

So I can refresh the page,

and then click on pricing,

back and forth, back and forth,

and everything still works as expected.

All right, well I'd say that is a definite improvement

for our development experience.

So I think that's just

about it in the short term for navigation.

So let's continue in just a moment.

Section 11: Performance Considerations

Topic 98. Starting the Auth Project

Speaker 1: I think that we've done a pretty good job

of setting up our entire marketing application.

So we've now got all the routing, all the CSS,

everything is pretty much in place.

That means we're gonna start to move on to our next part

of our app, which is the authentication piece.

So this is going to mainly consist of

a sign in page and a sign up page.

These two screens right here, I really wanna clarify

just make sure it's really clear,

these are going to allow the user to

authenticate themselves inside of our app.

And we're not gonna actually wire up any kind of

authentication or anything like that,

but we're still going to have kind of a brief discussion

around some techniques around handling auth inside

of a micro frontend application. Because there are one

or two things I want you to be aware of.

So to get started, let's go back over to our code editor.

I'm gonna find my auth project directory.

And the vast majority of code that we're gonna write out

inside the auth directory is gonna look extremely similar

to what we've already put inside of marketing.

So rather than rewriting a ton of code from scratch

that you probably are already somewhat familiar with,

we're gonna do a little bit

of copy pasting inside this video.

So I'm going to take my config directory

inside of marketing.

I'm going to copy it and then paste it into auth.

I'll then do the same thing with the public directory.

I'll copy it and paste it into auth.

I'm then going to manually create an src directory

inside of auth.

And then I'm going to copy over the App.js, bootstrap.js,

and index.js files.

I'm gonna copy all three of those and paste them into src.

Okay. Now, before we start to run webpack or anything

like that at our terminal, we do have to go through

all these changed files and make a couple of updates.

Essentially, anytime we have a reference

to something particular around marketing,

we need to update the reference to be

about the auth project instead.

So we'll first begin by starting off

inside the config directory.

We're gonna go through every single file

that we just copy pasted over

and make changes to them one by one.

The first auth is the webpack.common.js file.

Inside of here I don't think we have any references

to anything around marketing or anything like that,

so we can close this one right away.

Next up is webpack.dev.

Inside of here, we have designated a port

that our marketing project is going to run on.

Now, because we are inside of auth

we need to use a different port.

So I'm gonna have our authentication piece run on port 8082.

Next up, we'll take a look at our Module Federation Plugin.

Rather than using a name of marketing,

we'll change it to auth.

Rather than exposing something called MarketingApp,

I'll expose AuthApp,

and then we can leave bootstrap the same.

Okay, I'm gonna save that file and close it.

We'll move on to the production webpack file.

I'm gonna change the publicPath to auth.

We'll change the name on the

Module Federation Plugin to auth.

And then update the name of our exposed module as well,

to AuthApp.

Okay, that looks good.

We'll take a very quick look at our index.html file.

We don't really have to change this ID right here

but let's do it anyways.

So we'll call it auth-dev-root instead.

We'll then move into the App.js file.

So we no longer have a component for landing or pricing,

so I'm gonna comment those out.

I'm gonna update our productionPrefix to be something

that will be unique for the authentication project.

I'll change it to just au, short for auth.

And then I'm going to delete the two routes

that we currently have inside of that switch statement.

We're gonna come back and replace those routes

in just a little bit.

Inside of bootstrap.js.

I think that we might not actually need to

make any changes inside of here whatsoever.

I think that all this code is actually gonna be

just a total reuse from what we had inside of marketing.

So I'm taking a look inside of here.

I think the only thing we really have to change

is the selector right here.

Rather than trying to render our application

into marketing-dev-root, in isolation we want to put it

into \_auth-dev-root. I think that's pretty much it.

So then finally inside of our index.js file...

Yep. Still want that to be the same.

Okay, so that looks good.

Well I'm gonna go back over to my terminal.

I'm gonna open up a new terminal window,

change into the auth project directory

and then just try to start my project up.

Go back in my terminal.

Here we go.

I'm gonna change into my packages/auth directory

and then do an npm start.

Oh, okay. That was the one thing I did forget.

We forgot to update the package.json file.

Let's take care of that really quickly.

So back inside my editor,

I'm gonna find the package.json file.

We just need to add in our two startup scripts,

one for starting and one for building.

I'm going to just copy those once again directly

from the marketing package.json file.

So there's the start and build scripts.

Just gonna copy those and paste them

into the marketing, or I mean the auth.json file.

Okay, let's save that again and give this another shot.

So I'll do another npm start,

and as usual, I'm just gonna make sure that I don't have

any distinct error messages.

So I might see a deprecation or a warning or two

like these right here. Totally fine.

What we really care about is making sure that we eventually

get something that says Compiled successfully.

So it looks good to me.

Let's go back over to our browser and just

make sure we can open up our app at local host 8082.

So inside the browser, when I navigate there

I don't see any content on the screen.

That is because we deleted those two routing statements.

So we're not actually showing any pages

or anything like that right now.

But if I open up my console

I definitely don't see any errors, which is good.

Okay, well that's the start of our auth project.

We'll take a quick pause right here.

We're gonna wire up the two pages that we need here,

the sign in and the sign up screens,

and then we'll start to think about authentication

at a higher level in the next video after that.

Topic 99. Component Files

**Component Files**

Hi!

**Attached to this lecture is a file called 'components.zip'. Please download it - we will use it in the next video.**

Resources for this lecture

Topic 100. Adding Signin and Signup Forms

Instructor: In the lecture right before this one,

you should have found a zip file called components.zip.

I'm gonna extract that really quickly.

And inside there is a components directory.

Inside of that is a signin and signup .JS file.

These are the two react components that we're going to

use to display a sign in form and a sign up form.

Once again, I'm giving you these two files just

to save a little bit of time.

There's nothing inside them that is directly related

to Micro FrontEnds.

So rather than spending 10 or 15 minutes writing

out all the React logic, all the custom styling and whatnot

I'm just gonna give you the two components directly.

So let's take these two files

and wire them up to our authentication project.

Back inside my code editor, I'll find the SRC directory.

Inside there are gonna make a new folder called Components.

I'm then going to drag

and drop those two files into the components directory.

Okay, there we go.

Now we need to open up our app.js file

and add in some custom routing rules that will

tell the React Router that is running inside

of our off project exactly when we want to show these

two components.

So inside of my app.js file

I'm gonna delete the two import statements right here

that we had commented out just a moment ago

and replace them with an import for

Signin from components/Signin

and Signup from components/Signup.

We'll then go down and find our switch statement.

We're gonna add in two routes inside

of here to decide when we want to show those two components.

My first route, I'll put in

a path of /auth/signin.

So whenever we go to that path inside of our browser,

I want to show my sign in component.

Simple enough.

We'll then do the same thing.

So if we ever go to at -- auth/signup, excuse me

component

sign up.

And by the way, I put in an extra little forward

slash right there at the end of the path.

I'm gonna make sure I clean up that one.

Okay, so that looks good.

Now we're gonna go back over to our browser,

go to localhost:8082,

where we are running our authentication app in isolation

and to see if we can get these two components to show

up on the screen if we navigate directly

to auth/signin or auth/signup.

So back inside my browser, I'm at localhost:8082

and I'm going to change the URL manually

to go to auth/signin.

When I do so, if I have my console open,

I'm gonna very quickly

see an error message appear right here.

Well, that's definitely unfortunate.

We don't see any content up here.

Let's try the same thing for sign up as well.

So if I go to auth/signup, yep, I get an error over

here again as well.

So what's going on?

It feels like everything around configuration is just

about identical to our existing marketing project.

And everything around navigation

and whatnot worked absolutely fine inside of marketing.

So I think it's pretty clear that there's some kind

of difference between our marketing project

and the auth one, even though we really just

copy pasted just all the exact same code

and replaced one or two things here or there.

Well, it turns out that this comes back

to a topic that we discussed earlier on inside the course.

Something around our Webpack configuration

in the development environment.

So we're gonna have to go back and revisit that topic

and figure out what is going on here and how to fix it

up a little bit better than what we did previously.

So let's take a quick pause right here.

Come back in the next video

and do a deeper dive into Webpack in just a moment.

Topic 101. A Deeper Dive on PublicPath

Presenter: In this video,

we're gonna have a somewhat long discussion

and figure out why we are seeing

this error message, right here,

around a failure to load up our, main.js file.

Let's just dive right into this discussion

and figure out what is going on.

All right, so the first thing I wanna do

is go back over my code editor,

and I'm gonna take a look at my marketing project,

specifically, the Webpack prod.js file.

So inside of here, we had set up

that public path property right here.

Remember why we had done that?

That was specifically for production,

and it was to make sure that

whenever we loaded up our remoteEntry.js file

in production through the container

that remoteEntry file would know

where to find all the different files that were created

by Webpack and stashed inside of the

marketing/latest folder, inside of our S3 bucket.

Essentially the goal of public path,

was to make sure that some different files that had to

kinda reference the output from Webpack,

such as our remoteEntry file,

knew where to find those things.

So it turns out that public path property is also

rather important when we are running our project

in development as well.

So right now, let's imagine that

we are running just Webpack by itself

and let's not really worry about the off app

or anything like that.

Let's just say that we've got two inputs going into Webpack.

One input is a file called maybe, app.js,

and the other is the, index.html file.

We're going to assume that we are

making use the HTML Webpack plugin.

And remember, the goal of that plugin is

to have that plugin kind of take a look

at this HTML file and automatically add in a script tag

to it that's gonna reference all the different outputs

from the Webpack process.

So if Webpack spits out a main.js file,

we want or would expect our index.html file

to have a script tag that references main.js.

So, that is expected behavior,

that's what we want to have happen behind the scenes.

Remember, this process occurs not only for the

index.html file, but it also occurs for

our remoteEntry files as well.

Those remoteEntry files have to be told

exactly where to find all the different

JavaScript files that were thrown out by Webpack.

So even though, the diagrams I'm going to show,

are referencing this index.html file,

same kind of thing applies

for the remoteEntry file as well.

Okay, so now here is why we are seeing

an error right now inside the browser.

Whenever you have a script tag

with a SRC of just a file name,

your browser is gonna try to load up that file

by taking a look at the current domain and path

inside of your browser or in the address bar.

And it's gonna try to go to the current domain

plus the current path and try to find the file

at that location.

So right now, we are trying to navigate

to local host 8082/off/signup,

because we do not have a slash

at the very end of the url,

this is interpreted as actually technically

being kind like local host

8082/off.

And then to actually get the main.js file

your browsers gonna put on main.js like so.

And it's gonna try to find that main.js file at that path.

We could see that very easily

by just looking at the error message.

It lines up current totally perfectly.

So our main.js file is not hosted at this path.

The main.js file is hosted

at local host 8082/main.js.

If we tried to manually go to that url

we will see our main.js file appear on the screen.

So in essence, the error that we're seeing right now

is very simple and very straightforward.

We're just trying to find our main.js file

at the incorrect path, that's all.

So how would we fix this?

Well, one thing we could do

is to make use that public path property

to customize how that little SRC right there is generated.

So for example, if we update our Webpack config file

and add in a public path of just forward slash,

that forward slash right there

will be used when this SRC property is generated.

So the SRC, and I said property,

is technically an attribute.

So, the SRC attribute will be turned

into /main.js because we have the slash on there.

That means that we want to load up

our script relative to just the current domain,

and not the current domain plus the current path.

So by putting in a public path of forward slash

we would end up with a SRC attribute of /main.js,

which would cause our browser to try to load up

our script at local hosts 8082/main.js.

And that of course is exactly what we want.

We would successfully load up our JavaScript file.

So that's good.

Kind of.

It's kind of good.

It turns out that down the line this would actually

cause us a little bit of pain.

So, lemme tell you exactly why that is

and why just putting in a public path

of forward slash is not a fix

specifically when we are working on

micro front end applications.

So, if we were just doing a normal monolithic app

using a public path

of forward slash would work out totally fine.

Everything would be no problem whatsoever.

But, because we are writing this out

in a mono, or a micro front end app architecture

well using a public path

like this would not work quite as we expected.

Let me show you why.

Okay, so I wanna think down the line

when we eventually set up our container

at local host 8080 to load our authentication app.

So remember our container is gonna load up

inside the browser first, at local host 8080

and the container is eventually gonna say,

Oh I need some code from the off app.

So I'm gonna go to the remoteEntry file

load it up and try to figure out

what different JavaScript files

are available inside the off app.

So remember inside that remoteEntry file

there are some directions on how to load up

the other JavaScript files,

that are part of our authentication app.

The remoteEntry file needs to have some paths

or URLs or directions, inside of it to say,

Hey here's how to get this main.js file.

So here's a little bit unfortunate reality.

If we just put in a public path of forward slash by itself

that public path property is also

going to be used, remember,

inside this remoteEntry.js file.

So if we use a public path of forward slash

the remoteEntry file's gonna say,

Oh you want to load up the main.js file, no problem.

You can do that by making a request to /main.js.

So that string right there is generated

in the exact same way

that this SRC attribute gets generated.

Essentially means just take the name

of the file you're trying to load

and put the public path right in front of it.

So if we put a public path forward slash

into our Webpack config file,

our remoteEntry file is gonna say the exact same thing.

So now here's the problem.

This path right here

when we try to load it up a /main.js file

just like with a script tag SRC attribute,

it is assumed that we want to load up

a main.js file relative to the current URL

in the address bar, the current domain.

So we would try to load up,

local host 8080/main.js.

Think about that for a second.

What is at local host 8080?

At local host 8080, we are running our container.

The container itself has nothing to do with the off app.

So if we go to local list 8080 and try to load up

main.js, we're gonna get the wrong MainJS file.

We would get the MainJS file coming

from the container application

and not the one that we want from the off app.

So, if we just put in a public path of forward slash

everything is gonna work out as expected

in the development environment

when we are running in our app in isolation.

But as soon as we start to load our app up

through the container,

in the address bar we have local host 8080.

We're gonna load up that remoteEntry file.

We're gonna try to load up the main.js file

and we're going to get the wrong MainJS file.

Only when we are running our project through the container.

So it's a really nasty bug, because you can set

that public path right now,

for development when you're running your app

in isolation and it's only down the line

when you try to integrate everything together

you're gonna start to get some really weird errors.

'Cause you're loading up the wrong JavaScript files,

which would be completely unexpected.

Okay, so at this point in time

you might have two big questions.

First big question is, how do we fix this?

How do we deal with it?

How do we make sure that we get

our main.js file loaded up correctly

in our development environment

and when we are running it through the container as well?

That's an obvious question.

The second question you might have

and this I think is the real important one,

why wasn't this an issue on the marketing app?

Why didn't we run into this problem at all?

On the marketing app when we were running

everything in development,

we didn't have to worry about the public path property

and we had no issues loading everything up in isolation

or when we were running marketing through the container.

Let's first address the easy question.

So how do we fix this?

Fixing it is gonna be really easy.

Inside of our Webpack DEV config file.

We're gonna set a public path

of just the entire domain, the entire URL

of where we should get this main.js file from.

So put in there http local host 8082.

Then the outputs from our Webpack process for the off app.

The index html file is gonna have the fully

qualified URL right there.

So the full absolute URL

of exactly where to go to get the main.js file.

This same exact kind of path right here the full URL

will be used inside the remoteEntry file as well.

So by putting the full URL for the public path

in our Webpack DEV file, this will solve the entire problem

for both development in isolation and when we run it

through our container and development as well.

That's a pretty easy fix.

So now the other big question here

why wasn't this an issue on the marketing app?

Well this is the real gotcha and this is why

I made this video such a big deal and so long.

Here's the gotcha and this is why this is so darn tricky.

It turns out, that if you

do not set your public path property

whenever you load up your remoteEntry file

the remoteEntry file is gonna try to point or load up

all of your other JavaScript files inside that project

by using the domain that you fetched

your remoteEntry file from.

So if we load it up our remoteEntry file

for the marketing app from local host 8081

then when we try to load up the main.js file,

remoteEntry would say,

Oh well I loaded myself from local host 8081,

that's where I was served up from.

So you know what, I'm gonna try to load up

this main.js file, relative to that same domain,

local host 8081.

And that is in fact where we have

the correct main.js file for our marketing app.

So this behavior occurs,

if we never set up the public path property.

If you don't set public path,

everything gets loaded up relative

to where we got remoteEntry from.

And that's why everything worked correctly

with the marketing app.

Even though we didn't ever worry about

any kind of this stuff, everything worked fine

and we didn't have to touch anything at all.

So it's only now, because we are trying

to navigate to these kind of nested paths

like let's say, off/signup,

that we are required to set up

the public path property at all.

And as soon as we set up that property

that changes how some other parts of our project work.

And that's why this entire public path thing

is so darn confusing.

Okay, so at this point in time

I think that we've got a reasonable idea

of at least how to fix the problem.

If it's not clear why this

is working out for the marketing app,

even though we didn't have to worry about it at all.

Don't sweat it quite so much.

All you really need to understand is that in general

whenever we set or setting up our Webpack DEV file

for a sub-project, we usually want

to set up a public path property

of wherever our application is gonna be hosted

at in the DEV environment.

So let's take a pause right here.

Come back to the next video, set up the public path.

We're gonna make sure that everything works in development.

And then start to integrate our project into the container

and we should see everything working just fine

over there as well.

Topic 102. Small Required Change to historyApiFallback

**Small Required Change to historyApiFallback**

Similar to the previous section, we will need to fix up a bug related to the historyApiFallback settings. Otherwise, you will be met with 404 errors in certain situations such as directly accessing http://localhost:8082/auth/signup.

Find this code in the webpack/dev.js file of auth:

1. devServer: {
2. port: 8082,
3. historyApiFallback: {
4. index: "index.html",
5. },
6. },

You may resolve the issue by adding a / to the front of index.html:

1. devServer: {
2. port: 8082,
3. historyApiFallback: {
4. index: "/index.html",
5. },
6. },

Or, by setting to true:

1. devServer: {
2. port: 8082,
3. historyApiFallback: true,
4. },

After making this change, remember to restart your server.

Topic 103. Last Time for PublicPath

Narrator: I apologize for the very long video,

but hopefully now this entire public path property

is a little bit more clearer.

And you understand how important it is.

So in this video, we're gonna go back

through all three of our different projects

and set up the public path property specifically

for the dev environment.

We don't technically have to do this right now

for our container or for marketing

but if we ever start to add

in any kind of nested routes to those

we're gonna very quickly run into

the exact same issue.

So we're just gonna solve this problem

once and for all.

But to get started,

I'm gonna go back over to my editor.

I'll find the auth config webpac.dev.js file.

Inside of here, I'm gonna add in an output section.

And then public path of

http://localhost8082.

Don't forget the slash at the very end,

that's extremely important.

Notice that we are using a port right here

identical to the port that the Dev server is running on.

We always need to make sure

that these two ports are absolutely identical.

Okay, so I'm gonna save this file and like I said

we're just gonna go through

the other two Dev Webpac files as well,

and add in this output path property right now

just so we don't ever have to worry

about this again.

So back inside of container

I'm gonna find the webpac.dev.js file.

I will add in a output

a public path of http://localhost8080.

And save it.

And then I'll do the same thing

for marketing as well.

So here's the marketing Webpac.dev.js file

put in my output, the public path of

http://localhost8081/

And you know I'm not sure if I put in the slash

at the end of the container webpac file.

You know, right after saying how important

it was to get that slash

at the very end, I left it off the container one.

So gonna make sure I get the fourth slash

at the very end of the container.

Webpac.dev.js file. My mistake.

So here's the Marketing Webpac dev file.

Yep, looks good.

80, 81 with a slash at the very end.

Okay, so now very last thing.

I'm gonna go back over to my terminal

and restart the web pra

Webpac process for all of these

different projects.

The back of my terminal,

I'm gonna stop off, stop container,

and stop marketing and then do an NPM start.

NPM start

and NPM start.

So hopefully, we'll, we will not have to worry

about any kind of public path property stuff

again in the development environment, so long

as we always put in that full absolute URL.

All right, now that we've got this all fixed up

I'm gonna go back over to my browser.

I'm at localhost8082/auth/signup.

I'm gonna refresh the page and sure enough

we see the form appear. Awesome.

Exactly what we want.

Well, this looks pretty good.

You'll notice that there are

some links down here that you can click on

to go to the sign in screen

and then back over to sign up.

Back and forth very easily.

If you click on the sign up buttons right now,

nothing is gonna happen.

We're eventually going to make sure that clicking

on these buttons sends a signal up

to our container app and tells it,

Hey, we are currently now authenticated.

We'll get around to that in just a moment.

We should also be able to go back to localhost8081,

where we have our marketing app in isolation.

Refresh the page and still see everything work.

And we should be able to go to localhost80, as well.

Refresh the page and still see everything work, as well.

Okay. So hopefully, like I said last time

to worry about public path.

So let's take a pause right here.

In the next video we're gonna start to integrate

the Auth app into our container.

So let's take care of that in just a moment.

Topic 104. Integrating Auth into the Container 5min

Okay, so like I said,

adding in the AuthApp.js file alone

doesn't really wire anything up.

We do need to make sure, now,

that there's some routing logic inside of our container

that decides when to show the AuthApp.

So inside of our container SRC App.js file,

we're gonna add in some routing rules to this

to decide when to show some application

besides MarketingApp.

Right now, MarketingApp is being displayed

a hundred percent of the time.

So we need to add in a routing tier, so to speak.

Remember, all the routing logic inside of our container

is going to decide which child application to show.

So to add in some routing logic,

we're gonna go up to our import statement

from react-router-dom, and get Route and Switch.

I'm also going to import AuthApp from components/AuthApp.

And then, finally, I will delete MarketingApp right there,

replace it with a Switch, and then add in two Routes.

So for the first Route,

we want to say if a user ever goes to a Route

that starts with a path of /auth,

we want to show the AuthApp.

And then for the second one,

if a user ever goes to just forward slash,

we want to show the MarketingApp.

Now, quick note here,

just as a very fast reminder around react-router

and how these paths are interpreted.

The path right here is meant to match up

to the first part of a path.

So, as a very quick example,

you don't have to write this out, just a quick example,

if we ever go to auth, that's going to show this Route.

If we go to auth/signin,

well, we are still matching the first part of the path

to the path prop that we are passing in right there.

So when we go to auth/signin,

this Route will still be activated.

As a matter of fact, we can go to auth/anything,

and it's going to always match up to that first Route.

But if we go to anything besides /auth,

such as just forward slash,

or if we go to pricing or, say, /about-us, or whatever else,

it's going to go to the second Route.

So these two Routes right here are going to handle

deciding which of our different sub-applications

we want to show on the screen.

All right, so that looks pretty good.

Now, I do want you to know,

our application is not going to work correctly right away.

It turns out that there's a real small gotcha

around the history that we actually made use of

inside of auth and marketing.

Remember we made use of the, what was it,

memory history inside of both those.

The name escaped me for just a moment.

So there's a very small gotcha around those

that we kind of discussed very, very briefly

when we went over the history stuff,

but now it's gonna kind of

rear its ugly head, so to speak,

and show us a really nasty bug.

So, just want you to be aware that everything's

not gonna work a hundred percent correctly right away,

but we're gonna fix it up very soon.

Okay? So let's do a test back at localhost 8080.

I can refresh the page, everything looks good.

I can go to pricing, everything looks good.

But now, if I click on login,

unfortunately, I see an empty screen.

If I click on login a second time,

you will notice that we correctly see

the sign in screen appear, and if I click on the link,

I see sign up,

and I can navigate away just fine.

It's the fact that I have to click on login twice

to get the AuthApp to show up.

That's the bug I was just mentioning a moment ago.

Like I said, just a nasty little thing.

Just a little gotcha around making use of memory history.

So let's figure out why we're seeing this very small bug

and fix it up in just a moment.

Topic 105. Adding Initial State to Memory History

-: So what's going on with this bug?

Well, like I said,

it's just a very small little kind of "gotcha" thing.

So as soon as I tell you what's going on,

I think it's gonna be pretty obvious what is happening

and perhaps even a way to fix it.

So let's just figure out exactly why this bug is occurring.

Why do we have to click on the login button twice

in order to see the signin form?

All right, so quick diagram

that we had actually looked at a while ago.

Remember this is a diagram talking about how

we had a browser history inside the container

and a memory history inside of marketing,

was what we were talking about at the time.

We had said that if we ever go to localhost:8080/pricing,

Browser History would immediately

take a look at the address bar

and see that we were at /pricing.

However, memory history does not have

any kind of default initial starting path.

It always starts off, no matter what,

at an initial path of just /,

even if you have a different address

or a different path inside of your address bar.

The bug that we're seeing right now

is pretty much directly related to this issue.

When we, back inside of our editor really quickly,

I'm gonna find my auth/src/bootstrap.js file.

So when we create a memory history right here,

the memory history initially thinks

that we are at a path of just /.

We have wired up all these different history listener things

but remember, these listeners don't actually fire or run

until the current path changes in some way.

So when we currently show all

of our different subapplications,

in this case just auth and marketing right now,

they all think that they start off

at an initial path of just /.

Right now, our auth application is configured

to show nothing at all when we go to /,

and that's why we see nothing appear on the screen.

But then as soon as we click on that link in the header,

we are navigating inside of our browser history

once again to the auth/signin or /signup routes.

That triggers an event from the browser history

that we are listening to inside this bootstrap.js file.

And so we go ahead and update our history,

or the memory history in this case, to that new path.

This entire description is just a little bit complicated.

So I'll tell you what, inside of our auth bootstrap.js file,

I'm gonna add in a console log here

of nextPathName, like so.

So let's see when onParentNavigate is actually executed.

And we're gonna see when we actually are told

that we are trying to, like, navigate around somewhere.

So I'm gonna go back over.

I'm gonna go back to localhost:8080 and refresh the page.

I'll then click on the login button.

When I click on login,

notice how we don't get any console log.

So we are not being told inside of the auth app

that any navigation has occurred

when we first navigate to the auth app.

We have no kind of initial state,

nothing to say that we are auth/signin.

And so our auth application thinks

that we are at just / by itself,

and at / by itself, we don't have any routing rules.

It's only when I click on login a second time

that we finally get that console log.

So only when we click on the button the second time

do we get some kind of event telling us,

"Hey, we just navigated over to auth/signin,"

and that's why only then do we see

some content up here on the screen.

So to fix this, all we have to do, really easy fix:

when we create this memory history right here,

we're going to give it an initial starting path.

We're gonna say, "Hey, just so you know,

"don't think that you're at just / when you first start up.

"Instead memory history, we want you to understand

"we are actually at auth/signin."

That's how we're gonna fix it.

We're just gonna give our memory history some initial state.

So for that, I'm gonna find createMemoryHistory right here.

We're gonna add in an object

and provide an option called initialEntries.

This is going to be an array,

and inside of here we're gonna put in one single string.

That's gonna be what our current,

or I should say initial, path is for MemoryHistory.

So to figure out what our initial history is going to be,

we're going to assume that our mount function

is going to be called with one additional option

in addition to the onNavigate and defaultHistory

that they currently have.

We're going to assume

that it gets one additional option called initialPath.

So we're gonna take that initial path

and provide it as the only element inside this array.

Okay, so then back inside of our container, AuthApp.js file.

Now whenever we call our mount function,

in addition to the onNavigate option right here,

we're also gonna tell the mount function

what its initial path should be.

And its initial path should be whatever our

history's location.pathname property currently is.

Because remember, history.location.pathname,

that is the current path

that we are visiting inside of our application.

So for the initial path we're putting in

history.location.pathname,

and don't forget, pathname is all lowercase.

Okay, so let's save this.

We're gonna flip back over.

I'm gonna go back to localhost:8080 and refresh the page.

I'll then click on login, and sure enough,

we see our authentication app appear right away.

Perfect. So that's exactly what we want.

Now we are going to very quickly go

and do this same fix for the marketing application.

Just because of the different routes

we have inside the marketing app,

it turns out that the marketing app

appears to not suffer from this issue,

but it technically does.

Whenever our marketing app first starts up,

the memory history inside there

is always gonna think that it's at the root route

of just, well, root route, just / by itself.

So right now, our marketing application always says,

"You know what, whenever we come to just /,

"I'm gonna show the marketing page."

So even though we have the same issue,

it just turns out that everything works as expected.

But if we ever had any other routes associated with it

or if we didn't have a root route of just /,

we would have the same kind of issue or bug appear.

So we're just gonna apply the same fix

to the marketing app as well.

So let's first begin inside of

our container components MarketingApp.js file.

So I'm going to add in that

additional option here of initialPath,

and that will be history.location.pathname.

And then back inside of our marketing project,

if we go to the bootstrap.js file inside of marketing,

totally identical.

We're going to receive that option of initialPath,

and then provide some options to create memory history.

We'll put in initialEntries.

It'll be an array with just initialPath inside of it.

And that should be it.

Okay, so one quick test here.

Gonna refresh the page, go to pricing.

Everything looks good.

Go to login.

Looks good. Go back to app.

Yep, I think we got the entire flow.

Okay, well, I'd say this looks pretty good.

I do apologize, I just wanna mention very quickly,

for only showing you that kind of

final fix to the public path a couple videos ago

and only showing you the fix

to this kind of initial entries thing

way later on inside the course.

The only reason that I'm showing you this stuff now

is that you only really start to see these problems

once we add in that second application.

And that's why I couldn't just give you the full answer

and just say, "Hey, let's just write out

"this kinda mysterious code right from the get-go."

'Cause if we wrote out all this mysterious code right away,

you might have said, "Hey,

"what's the deal with the public path?

"Why are we adding this in?"

All that kind of stuff.

So again, I apologize for kind of delaying

the explanation a little bit,

but I wanted you to see what would happen

if you did not put those options in.

Okay, well, overall I would say that

we've got the auth app appearing, which is very good.

The last thing we really need to think about here

is how we actually handle authentication.

In other words, what really happens

when we click on Sign in right here.

We gotta figure that out.

So we'll take care of that in just a moment.

Topic 106. Lazily Loading SubApps

Instructor: I'm back inside my browser

at localhost:8080.

I've got my network request tab open,

and I'm only taking a look at requests for JavaScript files.

Now remember, we have had a discussion or two

about the number of JavaScript files

that we're loading up here.

Right now, just to load up the homepage,

well, we're sending down an awful lot of JavaScript code.

And every time that we refresh the page,

well, definitely nothing changes.

Now, there's one thing I really wanna point out here.

At this point in time, whenever a user comes

to our root route, in other words, localhost:8080,

the user doesn't really need to see anything

around our authentication application.

It is only whenever a user clicks on the login button

right here that we need to really get access

to some code related to authentication.

So with that in mind, I've got a little question.

I'm kind of wondering if there's something we could do

to maybe optimize performance of our application.

Maybe there's some way that we could figure out

to only load up code related to the current application

that we're showing on the screen.

In other words, if a user opens up their browser

and types in localhost:8080,

is there some way that we can somehow only load up

code related to our container and the marketing app

during that initial page load?

And then maybe only when a user clicks on the login button

right here and tries to go over to the auth app,

maybe it only at that point in time could we load up

some code related to the auth app.

This kind of approach would make sure

that our initial page load would go rather quickly

because we would load up a rather minimum amount

of JavaScript just to show the content

that the user is looking for.

And then only whenever a user starts to navigate around

our application would we start to load up

some additional code.

So I think that we should give this a shot.

It turns out that implementing a system like this

with Module Federation and React together

is extraordinarily easy and straightforward.

Let show you exactly how to do it.

Back inside my code editor.

I'm gonna find the container app.js file.

So inside the app.js file,

remember we've got some import statements inside

of here for marketing app and auth app.

I wanna take a look at just one

of those app files really quickly.

Let's take a look at auth app since we just worked on it.

Here's auth app right here.

When we load up or import the authapp.js file,

that is the point in time in which we start to reach

into the auth app itself.

So this import statement right here,

remember, that means we're gonna load up

the remote entry file for the auth project

and then start to load up some associated

JavaScript code with it.

So if we wanna make sure that we don't just eagerly load up

all the code for our application, what we really wanna do

is not load up the marketing app right here,

or the auth app, until we are really sure that we need to

show some content related to it.

So to make that happen, we're gonna use, like I said,

a little feature inside of React.

At the very top of the screen or this, excuse me,

my editor, this file.

I'm gonna find my import statement for React

and we're going to import something else

from this thing called lazy and suspense.

Suspense is a React component and lazy is a function.

These two things work together to allow us to lazily load

different components inside of our application.

So we can use these two tools right here to make sure

that we only attempt to load the code related

to our marketing app when a user starts to go to a route

related to the marketing application.

To make use of lazy and suspense,

I'm going to delete our two import statements

for marketing app and auth app.

After the header import,

we're then going to create a new variable

called MarketingLazy.

And we're going to assign that,

the result of calling the lazy function.

We're gonna pass it in arrow function.

Inside this arrow function, we're gonna write out import

in a sort of function form.

So we're gonna put inside of your ./components/MarketingApp.

So this whole lazy thing right here,

make sure that we only try to load or import code

related to our marketing app component

when we try to show this thing on the screen,

this marketing lazy thing right here.

MarketingLazy itself is a React component.

So we can choose to only show this component in some cases,

and whenever we show it,

well, this lazy thing is gonna go and try to import

the code related to that file.

I'm gonna create a second one of these.

Called AuthLazy.

That will be a lazy

import

components auth app.

Like so.

So now we've got these two components that are

going to lazily load code related to either the auth app

or marketing app.

Now to make use of them, we're going to go down

to our component down here

and do a little bit of a refactor.

First, I'm going to wrap this entire switch thing right here

with the suspense component that we had imported

just a moment ago.

I'll put the other closing tag on the other side of Switch.

On suspense, I'm gonna add in a prop called fallback.

And for right now, I'm gonna put in a div

that just says, "Loading,"

like so.

This fallback thing is going to be shown to the user

only when we are in the process of trying to load up

some code related to either marketing app

and/or the auth app.

So when we actually load up these modules,

it's gonna take, perhaps, a very small amount of time,

maybe even a couple of seconds

if the user's on a low quality connection.

While we are loading up that code,

by default, React is gonna show this div right here

with text of loading.

So then finally, rather than showing

a component of auth app, we will show

either AuthLazy

or

MarketingLazy,

like so.

And that's it.

Believe it or not, that's all we have to do

to introduce this idea of lazily loading up some section

of our code base.

Let's now save this.

Go back over to localhost:8080.

I'm gonna go back to my homepage.

I'm gonna clear out out my network request log

and then refresh the page.

Now when I do so, it's gonna appear

that we're still loading up a lot of JavaScript code,

but I want you to really take notice

that now we are only loading up two bootstrap files.

The first bootstrap file is right here.

And you'll notice we are fetching that from localhost:8080.

That is the bootstrap file related to our container.

The only other bootstrap file that we're loading up

is coming from localhost:8081.

So that's related to our marketing app.

So we are currently not really loading up

any code besides the remote entry file

related to our authentication app.

Now, if I clear my network request log

and go over to the login pages,

only at that point in time are we gonna load up

this additional JavaScript code that is needed

to actually show the authentication app on the screen.

So in this case, we have saved loading up

a tremendous amount of JavaScript

by delaying loading that stuff only when we need it.

So clearly, a pretty good solution.

And one thing I wanna show you very quickly,

if I refresh the page again,

I can then slow down my internet connection arbitrarily

by finding this little dropdown right here that says online.

And I'm going to change it to slow 3G really quickly.

So now whenever I make a request,

just simply gonna take some amount of time

to complete the request.

I'll now click on the login button again

and now I can see that loading text right there

as I start to load up these additional JavaScript files.

When they're all finally loaded,

and I can kind of see them being loaded right now,

but very slowly.

When they're finally all loaded,

we're then going to see our content appear on the screen.

That's where the fallback gets displayed.

I just wanna point out here really quickly,

this kind of fallback text is just a little bit ugly,

just showing the word loading,

maybe not the best experience.

So maybe instead of that, we could show

some kind of, like, spinner or progress bar,

just to let the user know that we actually are

still loading up something related to their app.

So let's try putting

in a better fallback element inside there in just a moment.

Topic 107. Adding a Loading Bar

Instructor: Let's very quickly create a React component

that will give a better loading experience

for our users than this very simple text that says Loading.

So inside of my components directory

of our container project,

I'm gonna make a new file called Progress.js.

So this is going to show our users a progress bar of sorts.

We're gonna have to write out just a little bit

of React code.

So in this course, in general, I've given you just about all

of the different components that we need.

This time around,

we're just gonna write this thing out together

because it's just a very small amount

of code that we need to write out.

So inside this Progress.js file,

I'm gonna first begin by importing React from 'react.'

I'm going to import makeStyles and createStyles

from @material-ui/core/styles.

After that,

I'm then going to import something called LinearProgress

from @material-ui/core/LinearProgress.

So this LinearProgress thing is what's going to

show an actual progress bar to our users.

We're then going to set up a little bit of custom styling.

So we will create a new variable called useStyles,

and assign that the result of makeStyles.

We're going to receive a property

or argument to this function of theme.

And then inside this function,

we're going to return the result of createStyles.

We're gonna pass that in object.

I'm gonna put on a bar property to this thing.

We'll give it a width of 100% inside of a string.

And then we're gonna put a little bit

of a complicated selector here.

So inside of a string, we're going to write

in '& > \* + \*', like so.

And then, we're going to assign that

marginTop: theme.spacing(2).

Now I know all this code right here is a little bit nasty.

It requires you to understand some of this CSS and JS stuff.

Essentially all we're really doing here

is creating a custom style that's going to make sure

that our little progress bar can have some margin top

and give it a width of 100% as well.

Okay. So then after that,

we're going to create and export our component,

which will be very easy.

So export default down here.

We'll then say const classes,

and that's gonna be the result of calling it usedStyles.

And then finally return a div with a className

of classes.bar.

I'm gonna close off that div.

And then finally, inside the div,

we will place LinearProgress like so.

Okay, that's it.

So let's save this file.

I'm going to go back

to the App.js file.

Right by our current header,

I'm going to import Progress from ./components/Progress.

And then finally,

add our suspense element right here.

Rather than doing the div with Loading,

I'm gonna replace that with Progress, like so.

And that's it.

Okay, so let's save this,

flip back over to the browser and do a quick test.

Now remember, at this point in time,

or I should say at the end of the last video,

I had turned on throttling just for very quick testing.

So if you did that as well,

you might still have Slow 3G selected right here.

Make sure you change it back to Online really quickly.

We're then going to go back to our root route.

So back to our main local host 8080 page.

I'm then going to refresh the page.

I'm then going to re-enable Slow 3G.

And now, let's click on that Login button.

And so, now we get this really nice progress bar right

at the top.

It kind of meshes really nicely with the header as well.

And only after we load up all of our code

related to the authentication app

are we gonna see the progress bar go away

and see our form appear on the screen.

So I'd say this looks pretty darn good, without a doubt.

All right, so this is definitely a very good improvement,

and we've definitely added

in some good performance enhancements as well.

So now that we can transition between our different apps

in a rather performant matter,

we're gonna start to work on our authentication system

in just a moment.

Section 12: Authentication in Microfrontends

Speaker: On your network request tab,

do make sure that you change throttling

back over to Online,

so that you do not take a really long time

to refresh the page or transition between different pages.

Okay, so in this section,

we're gonna start to focus on our authentication system

inside of our app.

Now, I had said previously that we're not really

gonna build authentication in.

When I said that,

I meant that we're not gonna have a backend API

or anything like that, that's going to actually

store or verify user credentials,

like an email or password or anything like that.

However, we are still gonna have some kind of basic

authentication system just built into the front end.

So whenever a user comes to our application by default,

in other words, the first time they come,

a user will be considered to be not authenticated,

or logged out.

So we will want to make sure,

that a user sees the LOGIN button up here,

and if a user tries to go to our Dashboard page,

we should redirect them over to

the Sign in page automatically.

Once a user clicks on SIGN IN right here,

or on the SIGN UP button, on a Sign up page,

I want to make sure that the LOGIN button goes away,

or turns into Log out or something like that.

And we should redirect the user over to our Dashboard page.

So we're gonna have some kind of limited,

kind of front end only authentication system.

So we need to first understand some things

around handling authentication inside

of a micro-frontend app.

Let me give you a couple of quick notes.

So these first two notes on here are pretty much

the entire point of this entire section.

Something that I wanna make sure is super clear right away.

We currently have an Authentication app.

And so you might think that we're going

to put a lot of logic inside there,

around maybe permissions and governing user access

to different pages or stuff like that.

That is not the case.

Our Authentication app is only for,

signing in or signing up users.

That is it and nothing else.

It is a 100% not at all to be used for doing anything like

permissions, allowing access

to some certain routes inside of our application,

or to have some kind of utility functions

that decide whether or not your user

is signed into your application.

These two first points right here,

are both for one extremely important and very simple reason.

Remember, just a moment ago,

we put in some lazy loading to our application.

So if a user comes to localhost:8080,

and refreshes the page, at that point in time,

we have loaded up extremely little code,

besides the remoteEntry file,

related to our Authentication app.

So with that in mind, picture the following scenario.

So imagine that a user opens up their browser

and navigates directly to app.com/dashboard.

Now remember,

we probably want to somehow restrict access to this page.

So if a user comes directly here,

we need to know whether or not they are logged in.

But here's the issue,

if a user navigates directly to this page,

they have only loaded code related to our Container

and our Dashboard.

They have not loaded any code related to auth.

So we cannot rely upon code written inside the Auth project

to decide whether or not a user is logged in

or whether or not they should be given access

to the dashboard.

We, in theory, of course we could.

We could place code inside there,

and it'd just always load up the Authentication app

no matter what.

But well, that would be possibly bad for performance.

That means we would be loading up our Authentication project

way more often than we need to,

just to figure out whether or not the user is logged in.

So this right here is the sole reason

that we're not going to stuff in a ton of business logic

around authentication into our Auth app.

The only thing we're gonna do inside there,

is allow user to sign in or sign up.

That's it.

Okay, so with that in mind,

we're going to talk about two general approaches

for handling authentication inside of micro-frontend apps.

So the two general approaches here are to,

Number 1, add in some code to each sub application

that gives it some knowledge of how authentication works,

and allows each sub app to decide whether or not

the user is logged in.

The other possibility is to just centralize

a lot of this authentication logic inside the Container

and have the Container tell each sub app whether or not

the user is logged in and whenever they're logged in

or authentication status changes.

We're gonna focus on Number 2 inside this course.

The reason for this,

is that if you go with option Number 1 right here,

you're essentially saying,

that you're gonna start duplicating a lot of code

between your different projects.

Because now, each individual app needs

to have some authentication related code

to decide whether or not the user is logged in,

whether they're signed out, whether they have permission

to go to some page, and so on.

And so rather than doing that, we're going to centralize

all of our authentication logic inside the Container.

So that means having some logic inside there

to decide whether or not the user is logged in.

And we'll have the Container communicate down

to each sub app the user's current authentication status.

So overall, it's gonna be kinda straightforward

to be totally honest with you.

So let's take a pause here and start writing

out some auth related code in the next video.

Topic 109. Implementation Strategies 2min

-: Let's take a look at a very quick diagram to understand

the general strategy that we're going to use

for handling authentication.

Okay, here we go.

So we're going to have our container

that has a property inside of it

that decides whether or not the user is currently signed in.

We'll call it something like isSignedIn.

If you are building out a real application,

you might decide to call it something like

currentUser, instead, and have its value point to

some object that describes

some stuff about the current user.

Maybe their id, their email and stuff like that.

But because our application doesn't have

quite so complicated a record around the current user,

we'll have a very simple property called isSignedIn,

and have that be either true or false.

Our container is then going to pass down

a very special callback to our authenticational app.

We'll call that something like onAuthChange.

Whenever a user successfully signs in or signs up,

we're going to have the authentication app

invoke that callback.

Whenever that thing is called,

it's going to be a sign that

the user has now signed into the application

through the authentication app.

So we will update inSignedIn over to true.

Then whenever we show some other application,

such as the dashboard or something like that,

we can decided to pass down inSignedIn

down to that application,

to tell it whether or not it should consider

the user to be authenticated.

We might also want to use isSignedIn

inside of our container to decide whether or not

the user should even get access to the dashboard app,

before we load any code related to it.

So, that's the general idea.

So let's start this implementation in the next video.

Topic 110. Communicating Auth Changes

Instructor: Okay, let's get started

on our implementation.

So first, we're going to create a callback

and pass it down into our authentication app.

Back inside my code editor,

inside my container, SRC components directory,

I'll find the AuthApp.js file.

Inside of here, we call mount, right there.

And remember, we pass in that configuration object.

So we are gonna add in an additional option to this object.

We're gonna put in onAuthChange.

And you know what, in this case,

we kind of might as well call it something

like maybe onSignIn, or something like that,

because we're only gonna call this callback

whenever a user actually signs into the application.

Let's just call it onSignIn.

Just keep it very simple.

Right now, I'll have this callback

just do a console log of user signed in.

Now we need to open up our authentication app.

We're going to receive this callback,

and we're going to make sure that whenever a user submits

that sign in or sign up form, we call the callback.

That's it, that's all we have to do.

So back inside my auth directory,

I'm gonna find the bootstrap dot js file.

Here's our mount function.

So we're going to receive that callback

that is now coming in that second argument right there.

And we called it onSignIn,

so I'm gonna receive onSignIn,

and then we'll pass that down into our app component.

So onSignIn is onSignIn.

Then, inside of our authentication app component,

so here's the app dot js,

we can receive the prop onSignIn,

and now all we really have to do is pass this callback down

to wherever it needs to be invoked from.

In this case, we need to pass it down

into our sign in and sign up components.

And that's pretty much it.

And right now, we are writing out these two routes

in with a particular syntax that prevents us

from providing any props to sign in or sign up.

So we're gonna do a little refactor

that will just allow us to communicate this callback

down to those two components.

To do the refactor, I'm gonna delete component sign in.

I'm gonna remove the closing tag right there,

and then put in a new closing route tag.

Inside of that, I'm gonna write out sign in,

so we wanna show our sign in component,

and we're going to pass down into it onSignIn.

We'll then repeat the process for the other route as well.

So I'm gonna delete component,

remove the closing tag,

create a new closing tag,

put in sign up,

and then provide onSignIn, onSignIn.

All right, that looks good.

Now, inside of our components directory,

we can find sign in and sign up js.

We'll start off first with sign in js.

I'm gonna scroll down a little bit,

and you're gonna notice

that the main component inside of here,

I actually kind of already took care

of this for us in the code that I gave you.

So the SignIn component already expects

to receive a callback right here of onSignIn.

We can scroll down a little bit more

and find underneath the form, or in the form,

there's a button inside of here.

And whenever you click on that button,

it's going to try to call that onSignIn function.

So the signin js file, or the signin component,

already makes use of onSignIn.

We'll also take a look at signup js.

Here's signup js, so sign up component.

And you'll notice I already wired up

that callback inside of here for you as well,

just to save a little bit of time.

We can go down to the button inside of here as well.

There's the button, there's on click.

Okay, so we are now passing that callback all the way down

from the auth app component inside of our container.

Here's the auth app component inside of our container.

We're passing that down into mount,

and that callback is going all the way down

to a final button that will be clicked

on by the user at some point in time,

and in theory, that should run this callback right here,

and we should see a console log.

Let's go test this out in our browser really quickly.

Okay, so back at local host 8080.

I'm gonna refresh the page.

I'll open my console, I'll go to login,

and then click on the login button.

Sure enough, there's the user signed in.

We can test out the same thing

over on the sign up page as well.

Okay, well that looks pretty good.

So, now all we really have to do here

is create some new piece of state inside of our container

that's going to track whether or not the user is signed in.

And whenever a user does sign in,

we're going to communicate that fact down to our dashboard,

or home, or whatever else needs to know about it.

So we'll take care of that piece of state in the next video.

Topic 111. Communicating Authentication State

Instructor: Let's add in that new piece of state

to our container.

So back inside my editor, inside my container project,

I'll find the app.js file.

Here it is right here from react.

I'm gonna add in an additional import of use state.

So we're gonna use the use state hook

to create a new piece of state inside of our app component.

Down at our component,

I'll create a new piece of state

we'll call it isSignedIn

and we'll get the setter as well.

So you'll notice I'm kind of assuming

you're somewhat familiar with react hooks here.

If you're not, don't sweat it.

We only need to write out

a very small amount of code for this.

We'll have this be the result of use state

and we're gonna start off the user as not being signed in.

So use state of false.

We then need to make sure that at some point in time

whenever we show our auth lazy component right here,

we have to make sure that we somehow

communicate this call back down into it

or make sure we give it the ability

to change this piece of state.

So for that we're going to once again

have to refactor this routes syntax.

I'm going to delete the component auth lazy.

Remove the closing slash, put down slash route,

put an auth lazy right there,

and then we'll we will pass down a callback.

We'll call on sign in this.

This prop will provide a function to it.

Whenever someone calls it

we're going to automatically invoke set is signed in

and provide it a value of true.

We'll do set is signed in and provide it true like so.

Now we can go into our auth app

inside of our container and receive this prop,

whenever we see our callback invoked

we're gonna call this prop right here.

You could kinda see how all these callbacks are getting

like just a little bit nasty

'cause we have to keep track of all them.

You could start to pull in

some kind of state management library

like Redux for something like this,

it's totally up to you.

But obviously for this application

I just wanna keep some of this stuff

a little bit more straightforward and simple.

Let's take a look at our auth app now.

So inside of our container components auth app.js file,

we will receive that new prop that we're passing down

which we called on sign in.

And then inside the call back

that we're passing down to our authentication app.

Instead of doing the console log,

we'll just call on sign in.

And you'll notice that we've kind of got

some like duplicated prop names here.

We've got a prop coming in of on sign in

and then we are passing down a callback

of on sign in as well.

So one thing we could actually do here is just

collapse this down to just be on sign in like so.

And now we are essentially passing down this

set state update right here directly down

into our child application.

So just a little bit more straightforward.

Okay, let's go back over and do a quick test.

So I'll go back to my landing page

I'll do a quick refresh, go to login.

I'm gonna click on sign in

and I don't see any errors right now

which is definitely good.

Maybe the last thing we should really do to

really test this thing out

is provide that is signed in flag into our header

and use it to update the text

and the target that we're linked to with a login button.

So for that,

I'm gonna take a look at our header component very quick.

Here's the header.js file.

You'll notice that it actually turns out

I kind of already put in that callback for you right there.

So signed in.

I called it is signed in as opposed to is signed in.

So let's just update the name of it really quickly.

I'm gonna change that to is signed in.

It looks like it's used right there as well,

so I'll change it to is signed in,

and there's one other location

where this thing was actually used.

We scroll down a little bit inside the header right there

instead of signed in and signed in,

we will change it to is signed in and is signed in.

So now back inside of app.js.

When we show our header, we're just gonna communicate

whether or not the user is currently authenticated.

We'll pass down a prop of is signed in

is signed in and that's it.

All right, I know that all these changes

have been rather confusing

because some of the code I provided for you,

some of it we had to write out ourselves.

The words is signed in

and just the words sign in has been appearing everywhere.

So I know this stuff is a little bit nasty right now,

probably a little bit confusing.

At the end of the day

all we are really doing is declaring a piece of state

inside of our app component.

We are consuming it inside the header

and we are passing down some callbacks

to this auth lazy thing,

and then eventually into the authentication app itself

to eventually update the value

of the is signed in piece of state.

So at the end of the day,

this is all some kind of basic react stuff,

it's just the fact that the terminology

is a little bit confusing,

and I already gave you some of this code

that I think makes it a little bit more nasty.

But if you had to reproduce this on your own project,

I'm absolutely positive you'd be able to do so.

This stuff is not the worst thing in the world.

All right, so time for a quick test.

I'm gonna go back to off slash sign in.

Refresh the page, click on sign in, and sure enough,

take a look at that.

Changed over to logout.

I can now click on logout

and when I click on logout

I get redirected back to the homepage.

When I click on logout,

we didn't actually have this button toggle right here,

so there's still just a little bit of work

that needs to be done,

but at the end of the day

I would say that this is definitely good success

because we are able to change the text inside that header.

So clearly our authentication state

is being updated in some way.

So I think the last big thing we need to take care of,

is make sure whenever we click on logout right there

we just reset the is signed in value.

We could do that in this video or the next one.

Let's just take care of it in a moment.

Keep this video a little bit more on the shorter side.

Topic 112. Allowing Signout

Instructor: Like we were just discussing,

let's make sure that clicking on that "Logout" button

actually resets our authentication state.

So once again, if we take a look at our "Header.js" file

and find our component inside there,

here it is right here, there is a prop

that we are receiving called "onSignOut."

So this function will be called automatically,

whenever a user clicks on the sign out button.

So all we have to do is make sure that our "App.js" file

provides some kind of callback down to the header

of "onSignOut," whenever that thing is called,

we're gonna update our "IsSignedIn" status to be false.

So on our header right here,

we'll provide "onSignOut," put in a call back,

and inside there, we will run "setIsSignedIn,"

and we'll update that value to be false.

Let's now flip back over, refresh once again,

we can click on "LOGIN," "SIGN IN."

Looks like we got the update. We'll click on "LOGOUT."

That automatically navigates us back to our root route

and we see the text inside the button change as well.

Well, I'd say that's pretty good.

Okay, well that's pretty much it

for wiring up some basic authentication.

The last thing we really have to do here is make sure

that we govern access to our dashboard app,

depending upon whether or not the user

is currently signed in.

So of course for that, we're going to actually

have to implement our dashboard application.

Let's take a pause right here.

We're going to start to deploy everything we've put together

in the last couple of sections.

Once we've done the deploy, we'll then start to work

on that final part, our dashboard app.

Topic 113. Adding an Auth Deploy Config

Instructor: I would really like

to do a deployment of our application

because it's been quite a while

since we deployed anything.

We have added in our authentication stuff.

We've added in lazy loading,

we've fixed up some webpack config,

I think it's definitely time to do a quick deployment,

and just make sure nothing is totally broken.

Before we do so however,

remember we do need a workflow file

for our auth project.

We don't have a GitHub workflow just yet

to actually deploy anything related to auth.

So to create a new workflow,

we'll again go into the .github directory.

Find workflows,

and I'll make a new file inside there called auth.yml.

I'm going to open my marketing.yml,

I'm going to copy everything inside of here,

and paste it into auth.yml.

And inside of auth.yml,

we'll just go through wherever we see the term marketing,

we're going to update it to be auth instead.

So I'll go to a deploy-auth,

packages/auth/\*\*,

packages/auth.

Then down here, when we sync our distribution directory

with our S3 bucket,

we're going to sync it into the auth folder

instead of marketing.

And then remember, we do have to do

that invalidation for our auth project as well.

Specifically, for the remote entry file.

Because it's contents will be updated,

but it's name never changes.

So at the very end of this line,

we'll make sure that we update

or create the invalidation

for paths of auth/latest/remoteEntry.

Okay, I'm going to save this file.

I do want to open up our auth,

config,

webpack reduction,

or prod.js really quickly.

So I just want to point out

that we've already set up the publicPath inside of here.

So all that discussion that we had around publicPath,

it should work out just fine

because we already put in the appropriate configuration

to the production file.

So this looks good right here,

we shouldn't have any issues around any publicPath

or anything like that.

Now the last thing we have to do,

is commit all of our changes and push them out to GitHub.

So as usual, I'm going to go backward to my terminal.

I'll open up a new terminal window.

I'm going to change back to our root project directory.

So for me, I called it mfp.

We can do a git status,

and yeah, a lot of changes inside of here,

we should have done some more granular git commits,

but oh well, it's totally fine.

So I'll do a git add.,

a git commit,

and I'll give it a message of "added auth subapp".

And then we'll push that off to GitHub.

So git push origin master.

Okay, so as usual,

we should be able to go to GitHub,

find our Actions tab,

we're going to see possibly a couple

of different things being deployed here.

So I see my container,

I see marketing, and auth,

as we made changes to all three.

So I'm going to let all these deployments run,

we'll come back together in the next video,

make sure that all the deployments occurred successfully,

and also take a look at our cloudfront site,

just make sure that we see

the updated content over here as well.

Topic 114. Verifying Deployment

Instructor: It appears that all three of my deployments

were done successfully,

so I'm gonna open up my CloudFront deployment URL

and just make sure that we can see the authentication piece

and all that kind of good stuff.

So here's my CloudFront URL,

remember if you lost this,

you can open up your CloudFront dashboard.

So here's my CloudFront dashboard,

and then find your CloudFront distribution,

there it is right there,

and remember, you're looking

for that domain name right there.

Okay, so back at CloudFront,

I'm gonna refresh the page,

I should see my application up here.

Ooh, I saw the progress bar at the top,

that was neat.

So now I've loaded everything up,

I'm gonna go to pricing, yep, looks good.

I'll go to login,

okay, I'm loading up some code

related to our authentication app,

I can toggle between sign in and sign up,

I can eventually click on sign in,

and I see logout get updated,

or I guess the login button get updated to logout.

Then if I click on logout,

there is some code inside of header

that's going to automatically do some navigation for us,

that's why we get navigated back to the route route.

You do have to write that out on your own,

I guess I mean proverbially on your own,

I wrote out for you inside the header.

Long story short, the header has code

that will handle the navigation for you when you log out.

So I'll click on log out and I go back over to the homepage,

and I see that button turn back into login.

Okay, well I'd say this is

just about as good as it can possibly be.

This looks fantastic.

The last major thing we need to do inside of our app

is add in our dashboard,

which is what we're gonna try to govern access to,

so let's get started on that dashboard in the next video.

Section 13: Using Other Frontend Frameworks

Topic 115. Initial Dashboard Setup

Instructor: In this section we're gonna start

to put together our dashboard page.

This page is gonna have a lot of different pretty charts

and tables on it.

The dashboard is gonna be built using Vue

as opposed to React.

The only reason we are using Vue here is to just say,

"Hey look, we can make use of multiple different front

and frameworks on the same micro frontend application."

There's not really gonna be anything very special

done with Vue.

We're just gonna use it to show some content.

There's not even gonna be any kind of routing logic

or anything like that.

It's gonna be just a very simple application, very basic,

very straightforward, just to make use of Vue.

Pretty much no other good reason besides that.

So, in this video we're gonna do a little bit

of initial setup for this dashboard page.

We're gonna copy over the contents of some other files

from our other projects,

and we're gonna update their contents just a little bit,

just as a head start.

We absolutely could write out all the code

for dashboard from scratch, but if we did,

honestly, it would take a pretty good amount of time

to write out all the code.

It would look very similar to everything we already put

inside of marketing and auth,

and at the end of the day

it would be just a little bit boring.

So, that's why we're gonna do

a decent amount of copy, paste here.

So to get started, inside of my dashboard directory,

I'm gonna create three new folders.

I'll create config,

I'm gonna make an src,

and then finally a public.

And then going to open up auth,

and we're gonna copy a couple of different files

from the auth project over to dashboard.

So first off, I'm gonna copy all three webpack config files

into the config directory of dashboard.

Inside of public,

I'm gonna take the auth, public, index dot html file,

and then paste it into the public directory of dashboard.

And then finally, out of the src directory,

from auth, we are going to copy

both the index and bootstrap dot js files.

I'll copy both those and paste them into src.

Okay. Once we've added all those files,

we have to go through each of them

and make a pretty good number of changes.

So even though we have saved time,

because we don't have to write out

all this stuff from scratch,

we still have to spend a pretty good amount of time

to do a fair number of updates.

We're gonna first begin with our webpack common js file.

This file is where we need to make the vast majority

of changes.

So, we're gonna spend just a little bit of time

inside of here, but after this file,

everything else will be pretty smooth sailing.

So inside of here at the very top,

we're going to first specify an entry file

of dot

slash

src

index dot js.

We're gonna put in an output

with a file name

square brackets,

name

dot

content

hash

dot

js.

We're gonna make sure I get a comma after that.

We're gonna add in a resolved section.

Make sure you get a comma after that as well.

Inside of here, we'll add in extensions,

put in an array with a dot js string and a dot vue string.

If you've never worked with Vue before,

rather than writing out code inside of JavaScript files,

we can instead write out what are called

single file components,

which are files that end with an extension of dot vue.

So, this right here is just going to make sure that

webpack understands that it should attempt to load up

dot vue files.

That's pretty much it.

Next up, we're going to make a real big set of changes

to this rules section.

We're gonna add in a couple of different things here.

First off, we're gonna add in a new object.

I'll give it a test property.

And, we're gonna write out a rather long regular expression.

So, forward slash, back slash, dot,

then inside of a set of parenthesis,

we're gonna put in png,

a pipe,

that is the character right above the enter key

or the return key on your keyboard,

not an I, not an L.

Make sure you got a pipe right there.

You'll put in jpe, question mark, g, pipe,

gif pipe,

woff pipe,

svg,

eot,

ttf.

I'll then close off the parenthesis,

dollar sign,

forward slash, I.

After that, we'll then put in a use array.

Put an object in there

with a loader property of file dash loader.

So, what is this thing right here all about?

Well, the Vue project we're going to make use of

has a couple of different custom fonts,

images and whatnot inside of it.

We are making use of this file loader thing

so that webpack can understand

whenever we try to import a font or an image and so on.

This is something that is only required by our dashboard

in this case.

We could have just as easily had to put in a loader

like this for our React projects.

We just happen to not need to add in any imports

around using any fonts or anything like that.

Okay.

Next object, we're gonna put inside the rules array,

we'll add in test of forward slash, back slash, dot vue,

dollar sign, forward slash

and then a use of vue dash loader.

And then, just one more big one here.

Right after that, just one more object.

Put in a test, forward slash, back slash,

dot

scss

pipe,

back slash dot css,

dollar sign,

forward slash.

We'll then put in a use array.

This one's gonna have a couple of strings.

We're gonna put in vue style loader,

style loader,

css dash loader,

and finally,

sass dash loader.

Let me collapse that sidebar so you can see that whole line.

Okay, very last change inside of here.

I'm gonna find preset react right there

inside the presets array for babel.

And we're just gonna remove that entire preset,

cuz we're not making use of React at all.

Okay, I did kinda lie.

One very last small change at the very top.

This file, we're gonna add in a required statement

or vue loader plugin,

require vue dash loader.

Don't forget the curly braces on there.

We'll then go back down to the very bottom of the file,

and right above that last curly brace right there,

I'm gonna add in a plugins array,

and put in new vue loader plugin,

like so.

Okay, so that is it for this webpack common file.

So, I know this has been very tedious.

I apologize for all the typing,

going a little bit slow there.

I just wanna make sure you get everything put in correctly.

But that is pretty much it for the really nasty config.

Everything else we're gonna put inside of here

is gonna be pretty simple and straightforward to nature.

Just a small tweak or two, here and there.

So, be going a lot faster than inside this video.

So quick pause, come back and do a little bit more setup

on our dashboard project in just a moment.

Formularbeginn

Formularende

Topic 116. A Touch More Config

Narrator: Like I just said a moment ago,

the webpack.common.js file

was gonna be the most tedious change.

So now we're just gonna go through

all of our remaining files rather quickly

and make a quick tweak here or there.

So we're gonna move on to our webpack.dev.js file.

Inside of here you're gonna find publicPath,

we'll update the path, or excuse me, the port

at the very end to 8083.

On devServer change our port to 8083.

So now our dashboard is running on,

you guessed it, port 8083.

In the devServer section,

we're gonna add in another section to this thing of headers.

I'm gonna provide a string of Access-Control-Allow-Origin

and then star, like so.

We only need this on our dashboard project

because we're going to attempt to load up

a couple of different font files and whatnot.

And loading up these different files

will be subject to a security policy called Cores.

If you're not familiar with Cores at all,

don't sweat it, not a big deal.

Basically just a little configuration thing

to allow us to load up some fonts

when we are loading our application

or viewing it from inside the container.

After that, we'll go down to our ModuleFederationPlugin.

So I bet you could guess

what we're gonna change inside of here.

We'll change the name to 'dashboard'.

We're going to expose a file called 'DashboardApp'

and then we can leave bootstrap just the same.

That is it for our webpack.dev.js file.

Let's move on to prod.

So this one, all we have to really change here

is our publicPath

to dashboard/latest

and then the same exact changes

to the ModuleFederationPlugin.

We'll go to 'dashboard' and 'DashboardApp'.

Okay, let's save this.

Next up, our index.html file.

Remember, we've got that id inside there.

Not really important that we update it,

but what the heck, let's do it anyways.

So I'll change that to 'dashboard'.

Remember, that is gonna be the target

that we rendered to

when we are running our app in isolation.

So now, the last really big change we need to make

beyond our package.json file

is inside the bootstrap file.

Inside of here, we've got a ton of code

that is assuming that we're making use of React right now.

So obviously we have to take out a lot of stuff

around React inside of here

and replace it with some code around view and whatnot.

We'll take care of that in just a moment.

Right now, let's add in some scripts to our

package.json file,

just so we can make sure that we can run this thing

in development and in production.

And, excuse me,

build it for production is what I mean to say.

Once again,

rather than writing out these scripts from scratch,

let's just copy them over from how about,

our marketing project?

So inside of marketing, I'll find our package.json file,

we're gonna take our two scripts inside there

for starting and building.

I'm gonna copy those, go back over to my dashboard

package json and paste them into the script section,

there we go.

All right, well that is it for the initial setup,

outside of our bootstrap file.

So we're gonna take a pause right here.

In the next lecture

I'm gonna attach a zip file with some component

one or two that we're gonna show inside this View app.

So make sure you download that zip file.

We'll make use of it in the next video,

and very quickly get some really interesting content

visible on the screen.

Formularbeginn

Formularende

Topic 117. Component Files

omponent Files

Hi!

Attached to this lecture is a file called 'components.zip'. Please download it - we will use it in the next video.

Resources for this lecture

Topic 118. Showing the Dashboard

Instructor: Once again, attached to the previous lecture

you should see a zip file called Components.

I'm going to double-click that and extract it.

Inside there, we're gonna find one file

called Dashboard.vue.

Back inside of our Dashboard directory,

I'm gonna find the src folder.

Inside there, I'm gonna make a new directory

called Components.

We'll then drag and drop Dashboard.vue

right inside there, like so.

So inside this file,

you're gonna see a tremendous amount of markup,

and a little bit of JavaScript,

and then eventually some CSS at the very bottom.

So this is everything required

to get some kind of fake data Dashboard

to appear inside of our application.

So we now need to write out just a little bit

of vue-related code to create a new vue app

and eventually display it on the screen.

So I know that you might not be at all interested in vue,

even in the slightest.

Don't sweat it.

We're gonna get through this rather quickly.

We'll first get started inside of our bootstrap.js file.

So once again, inside of here we've got a lot of

react-related code that we need to clean up right away.

Also, we've got some stuff around creating

a memory history or a browser history.

In our application for the vue portion,

we're not gonna have any routing,

so we don't need to worry about creating any history

or anything like that.

In fact, we don't really need any of these imports.

The only thing we really need to import is vue itself

and we also need to create that or import that

Dashboard component that we just added in as well.

So I'm gonna delete all the imports at the top.

I'm gonna replace it with an import for createApp from vue,

and then I'll also import the Dashboard component

that we just added in.

We're gonna get that from ./components/Dashboard.vue,

like so.

Okay, onto the mount function.

We do not really expect any options

or anything like that to come down

into the mount function at all.

We don't have anything related to SignIn

that we're going to immediately make use of.

We don't have any navigation we need to worry about, either.

So I'm gonna remove that entire options object

that's coming in as a second argument.

We can then do a fair amount of cleanup

inside the mount function itself.

So we don't need to worry about history,

we don't need to worry about ReactDOM.render

or anything like that.

And we also don't really need to return anything.

All we have to do inside of here

is create a new vue application

and eventually show it inside of this el right here.

Remember, that's our HTML element.

So I'm gonna create a new app with createApp

and pass in the Dashboard,

and we'll do an app.mount to el.

Just so you know, this mount function right here

is totally unrelated to our mount function.

This mount function tied to app is something related to vue.

It's how we tell vue to actually show a component

inside the DOM.

Okay, that's it for our mount function.

So next up, we're gonna go down and make a very small update

to all of our dev-related logic down here.

Remember, we now no longer

have anything related to options as a second argument,

so I'm gonna remove that.

And then we also wanna make sure

we update that query selector

and try to render our app into dashboard-

dev-route.

And that should be it.

All right, let's go back over to our terminal.

We're gonna try to start up our project,

run it in isolation,

just make sure we can see the Dashboard on the screen.

After that, we'll start to integrate this app

into our container.

So back over at our terminal,

I'm gonna open up yet another terminal window.

We've now got several terminal windows going.

Inside of this latest one, I can reorder them.

There we go.

I'm going to change into our Dashboard.

I'll do an npm start

and we should- oh, okay.

Unfortunately, I must have a typo inside of here somewhere.

So it looks- oh, I got a little typo.

I put in O-U-P-U-T somewhere.

So that's probably inside my dev config file,

or maybe the common one.

That's where we made the vast majority of changes.

So I'm just gonna go and find that typo really quickly.

Chances are you don't have this typo,

unless you duplicate it on your own.

I'm just gonna very quickly flip back over

on my Dashboard comment.

Yep, there's the typo right there.

So let's make sure I change it to output. Much better.

I'll go and restart webpack.

Now hopefully, we can get this running without any errors.

All right, looks good.

So, no errors and I see a Compiled successfully.

Once again, you might have some

in a deprecations or warnings, stuff like that.

Totally fine.

We just don't wanna see anything

that says there's a distinct error.

Now we can test out our Dashboard

by going to localhost 8083.

Inside of our browser,

you guessed it, localhost:8083.

And there is our Dashboard.

Looks pretty cool, doesn't it?

Yeah, I just packed a ton of markup, ton of stuff,

directly into that vue component.

So you're seeing the result of a lot of vue code right here,

a lot of markup and whatnot.

Now of course, there's a fair number

of buttons and whatnot on the screen.

A lot of them might work,

but a lot of them are just not really gonna do anything,

which is totally fine.

Remember, this Dashboard is more or less fake.

If you wanted to,

you could absolutely continue to develop the Dashboard

and add in some interesting features and whatnot,

but not really relevant for learning about Micro Frontends.

Well, I'd say this looks pretty good.

If I open up my console, you might see a warning or two.

These are totally fine.

Now, the very last thing

we need to really do for our entire project

is integrate this vue project back into our container,

do a final deploy,

and we can wrap everything up.

Topic 119. Integration with the Container

Instructor: Let's take care of our integration

between the container and the dashboard.

We've gone through this process several times now

so I bet you can guess what is gonna happen.

Inside of our container SRC Components directory,

we are going to create a new file called

dashboard app dot js

and then going to open up our off app.

I'm gonna copy everything inside of here

and paste it into Dashboard App.

We'll then go back through and we're going to clean

up a pretty good amount of code 'cause

of course our dashboard app is far more simplistic

than the off app is.

So I'm inside of Dashboard app,

I see all the stuff on the screen.

First thing we need to do is make sure we

get the correct import, the correct mount function.

We wanna get the mount function

from dashboard slash dashboard app.

We still need to make use

of React inside of here 'cause remember a lot

of this React related stuff is really tied

to displaying our this sub application inside the container.

That's why we have a lot

of these different import statements.

However, because we are not making use

of any navigation or routing inside of our dashboard

we don't need anything related to use history.

So we can remove that.

We'll now go through our component itself

and do a little bit of cleanup inside of here.

First off, we don't need any prop around on sign in.

We don't need to get access to history.

When we call mount right here,

we only need to call mount and provide a reference

to the element that we want to render our application into.

So as a matter of fact, just about everything inside

of use effect can be deleted and we can replace it all

with just something around the mount function.

So I'm gonna take just about everything inside of here,

delete it and replace it with mount ref dot current.

Okay, that looks good.

We'll now go back over to our app dot js file.

We're gonna add in a little configuration to make sure

that we eventually show this dashboard app thing.

So inside of our container app dot js we need to get access

to the dashboard app component we just created.

We're going to make use of a lazy import once again.

So we do not import anything related

to the dashboard until we're very sure

that we want to show content related to it.

So I'm going to add in dashboard lazy

and that's gonna be the result

of lazy import components dashboard app.

After that we'll go on down

to our switch statement and I'm going to add

in a route to this thing with a path of slash dashboard.

And remember it's really important

that you put it above this particular route right here.

If you put this route above

then whenever you go to slash dashboard

this route will be matched first before slash dashboard.

We'll then put it in our component of dashboard lazy

and close off that route.

Then the very last thing we need to take care

of is adding in a little bit of configuration

around our module federation plugin for the container.

And remember, we have to tell the container exactly

where to go to load up stuff around our dashboard.

So inside of our config directory inside of container

we need to make a very small change

to the webpac dev and the webpac prod dot JS files.

First off our dev file, here it is right here.

We can go down to our remote section.

I bet you can guess what we're gonna put inside of here.

So dashboard, dashboard at http colon slash

slash local host 80 83

remote entry dot js.

We'll then repeat that over

inside of our webpac prod dot js file.

So we will put in dashboard and in this case

don't forget the back ticks dashboard at dollar sign

curly braces domain slash dashboard

latest remote entry dot js.

Okay, we can save this.

We now need to go and make sure

that we restart our container application

at the terminal so it uses the latest webpac

config that we just wrote inside there.

So back over at our terminal

I'm gonna find the terminal window running webpac

for a container.

You might see an air inside of it right now.

That's totally fine because you're currently running a copy

of webpac that doesn't see that latest configuration.

So it doesn't know how to load

up stuff related to the dashboard.

We should be able to stop the webpac process

for the container, restart it,

and then see everything start running without any errors.

Yep, looks good.

All right, here's the moment of truth.

Can we load up a view app inside of a React application?

Let's try it out.

So back inside my browser

I'll go to local host 80 80, I'm gonna refresh,

I'll click on login, I'll click on sign in.

Now unfortunately when we click on sign in,

we don't have anything to automatically redirect us

over to the dashboard just yet.

So unfortunately in reality to take a look at the dashboard,

we just have to navigate directly

to local host 80 80 slash dashboard.

Well, if we do that, we see our app load up

see the header at the top.

So there's the container related elements

and then there's our dashboard.

I can still click on app right there

and go back to our main homepage.

If I go back to dashboard, I can click on login.

That takes me to the authentication page.

I would say this looks pretty good.

So we've now got React together working with View

and there's no conflicts between them whatsoever.

Let's say this looks pretty good

definitely a very good success.

There's still one or two things we need to take care of.

So we do need to make sure that when we actually sign in,

we redirect over to the dashboard.

We'll take care of that really quickly.

And we also

we'll might want to eventually wire up something to say,

hey the user can only go to the dashboard

if they are signed in.

So let's take care of all the stuff in just a moment.

Topic 120. Protecting Access to the Dashboard

Instructor: The last big thing we need to take care of

is making sure that whenever user signs in,

they get redirected over to our dashboard

and whenever they sign out

we should send them back to maybe the sign in screen

or maybe the landing page or something like that.

So let's get to it.

We're gonna make all the changes required

for this inside of our container source app dot js file.

So here it is right here.

We're gonna first begin by adding

in a couple of different imports at the top.

So I'm gonna find our import from React

and I'm gonna add in an import of use effect.

I'm gonna find the React router dom import right after it.

And rather than importing browser router,

we're gonna get something else called the router.

We're also gonna add in an import for redirect.

One last import, after the first three

I'm gonna add in another one.

We'll put in import create browser.

Come on.

Browser history

from history.

I'll then go down right above our React component.

I'm gonna first create a new history instance using

that Create browser history function

and I'll tell you why we are doing this here

in just a moment.

First, let's just add in a little bit more code.

I'm gonna find browser router

and I'm gonna change that to simply router

and pass in history like so.

I'll then go down to the very bottom of the component,

I'll find the closing browser router tag

and change that to router as well.

All right, so why do we make that change?

Why are we not creating a browser router anymore?

Well, this is nothing related to micro front ends at all.

This is just how react router dom works.

Remember, whenever we create a browser router

that internally creates an instance

of browser history for us, we need to get access

to that history instance so that we can somehow

programmatically redirect the user around our application.

Namely whenever the value of say 'is signed in' changes

and we want to write out all the logic for this directly

into this component.

To get access to the history that is created

by browser router, is rather challenging to do so

if you're trying to get access to history

in the same component where you create the browser router.

In other words, getting access to the history when

we had browser router right here is challenging

from specifically this component.

It would be a lot easier if you're trying to write

out the logic for this in say our header or something else

but just because we are trying to write out this logic

it's just hard.

That's it.

And the way that we get around this is

by creating the history manually as we've done right here,

reading an instance of a generic router,

and then telling it which copy of history we wanted to use.

So again, this is something related to react router dom

nothing related to micro front ends or anything like that.

Now that we've made this change,

we're gonna add in a little bit of code to decide

when we want to redirect the user over to our dashboard.

So I'm gonna put in a use effect

right here.

As a second argument, I'll put in an array

with is signed in.

So this statement is going to run this function

right here whenever the value of is signed in changes.

So then inside this function, we can take a look

at the value of is signed in and say, if it is true

let's redirect the user over to the dashboard

because they must have just clicked

on that sign in or sign up button.

So inside of here we'll do an if is signed in

history dot push slash dashboard.

So if is signed in changes and it is true,

push the user over to the dashboard.

You might think that to handle a user signing out

and navigating them away from the dashboard

you might think we could put on an else case here

and then do maybe a history dot push to root route

or something like that.

Unfortunately, we do not want to do this.

The reason for that is that whenever user first comes

to our application,

they're always gonna start off as signed out.

This use effect function is going to run by default.

And if we put in that else case,

we're essentially saying whenever user first comes

to our application no matter what,

redirect them over to that given route.

Definitely not what we wanna do.

So we're gonna have to implement the redirect away

from dashboard whenever a user signs out

in a slightly different fashion.

For that, I'm gonna go down to our routing configuration.

We're gonna find our route for the dashboard right here.

We're gonna do another refactor of this just

like we've done several times now around all these routes.

So I'm gonna delete the component designation.

I'm gonna delete that little closing slash.

I'll close off the route.

I'm gonna still show dashboard lazy

but then right above it, I'm also gonna put

in a little check right here inside of some curly braces.

We're gonna say if the user is not signed in.

So if exclamation is signed in,

then I want to show an instance of a redirect component.

A redirect component does exactly what its name says.

If we ever show a redirect component on the screen.

react router dom is gonna automatically redirect our user

off to some other path.

In this case, we're gonna try to redirect the user

over to our landing page.

So if we ever try to go to slash dashboard

and the user is not signed in,

just redirect them to the landing page.

That's pretty much it.

Okay, so let's save this

and we'll do a very quick test.

Back inside the browser,

I'll make sure that I refresh the page.

I'll then attempt to sign in.

I get redirected over to the dashboard

and now if I sign out, I get kicked back to the homepage.

If I try to navigate directly to dashboard

well unfortunately I start off as not signed in

so I get automatically redirected over to the homepage.

So let's say that this works perfectly.

Well that is pretty much it for getting our dashboard

to show and controlling access to it.

Now the very last thing we really need to do

inside this project is add in some deployment configuration

around our dashboard and do an actual production deployment.

We can then do a final test

on cloud front and make sure everything works as expected.

Topic 121. Deploying the Dashboard

-: All right, my friends,

let's take care of this final deployment.

So to deploy our dashboard, it'll be the exact same process,

exactly what we have done several times before.

Inside of our workflows directory

we're gonna create a new file called dashboard.yml,

I'm then going to open up our auth.yml file,

I'm gonna copy everything inside of here

and paste it into dashboard.

Then inside of the dashboard.yml file,

we're gonna go around a little bit,

and wherever we find the word auth, we're gonna change it

to say dashboard instead.

So first on the first line right there, deploy dashboard,

we'll do packages dashboard, packages dashboard again.

We'll then go down to our two S3 commands

or I guess our one S3 command

and our one cloudfront command.

I'm gonna scroll over to the far right-hand side.

I'm gonna find auth right there and change it to dashboard.

And the same thing on our create invalidation line as well.

There's paths slash auth right there,

change that to dashboard.

And that should be it.

So as usual, we're gonna save this file.

We're gonna commit it,

push it off to GitHub, and then just verify,

make sure that the deployment occurred successfully.

So back at my terminal, inside my route project directory,

I'll do a git add, I'll do a git commit,

I'll give it a message of something like added dashboard.

I'll then push this off to GitHub.

Once we do the push, as usual,

we can take a look at our actions tab over on GitHub.

Here's actions right here.

And I should see a pending deployment

for a dashboard of deploy auth right there

and deploy container right there.

Okay, now you will probably also see a deployment

for deploy dashboard.

To be totally honest with you,

you know, actually, it was my mistake.

You should not see a deploy auth right there, I think.

Maybe you do, maybe you don't.

Long story short, the video right before this one,

I actually had to record a couple of different times,

as you can see.

And unfortunately, in one or two other times I recorded

the video, I made a little small mistake.

And I only realized that after I deployed the application.

So here's the original deploy the dashboard.

You probably see that.

I'm onto like deploy number two or three or something of it,

just because like I said, I've unfortunately had to

re-record the video right before this one,

and I accidentally deployed it a couple times as well.

Long story short,

all we really have to do is deploy our application

wait for it to complete, and then we can test it out

make sure everything works as expected.

So I'll see you in just a minute.

Topic 122. Verifying Deployment

Instructor: All right, deployment complete,

let's now open up our site

on CloudFront and make sure everything works as expected.

So I'll go back over to my CloudFront URL,

I'm gonna refresh the page.

I should then be able to take a look

at all the different flows,

so we should be able to go to pricing,

then log, sign out,

go to sign up, log in,

sign out, all that kind of stuff,

everything should work as expected.

So I'm gonna go to pricing, looks good.

I'll go to log in, looks good.

I'm gonna sign in, yep.

I'm gonna log out,

I get kicked back to our homepage, exactly what we want.

I'll go to log in,

this time I'm going to go the signup screen and sign up,

looks good.

I'll log out,

and then finally, last big test,

I should not be able to go to /dashboard.

If I go to /dashboard,

I should get redirected back over to our homepage.

It looks like that's exactly what happens.

All right, looks like everything works,

this is fantastic.

So I think our application is just about complete,

last thing we're gonna do is a quick wrap up

and kinda a review

on some of the very important topics

around micro-frontends,

so let's take care of that final review in the next video.

Topic 123. Wrapup

Instructor: All right, my friends.

I think that wraps up our application.

So I wanna finish this course out

with one last slide here,

and just do a quick review on some

of the most important things

we discussed during this course.

I think, bar none, the most important thing

for you to understand about microfrontends is

that you have to analyze your requirements.

You have to realize or figure out

what is important for your application.

There are a tremendous number of ways

to build out microfrontends,

and I showed you one very particular implementation

inside this course.

Our implementation was the result

of deciding what was really important

for us and our application.

And it was only once we sat down

and decided on what was important,

that we were able to come up

with a set of requirements for our application.

If your requirements are different

than the ones I showed inside this course,

you might end up with a rather different architecture.

So just keep in mind,

most important thing you can do,

when it comes to designing microfrontends,

is deciding upon what is important to you.

Next big thing I want to remind you of is that,

whenever you are trying to build up

or somehow couple together different microfrontends

or different subapplications,

you always want to be asking yourself,

"If I ever try to change X on something,

am I gonna have to change something

on another subapplication

at some point in time, in the future?"

We ran into this situation many different times,

and we ended up writing a pretty good amount of code.

Best example of this was probably

around the routing logic

to make sure that, if we decided to say,

"Change out the routing library inside

of our container,"

we would not have to go

and rewrite portions of each subapp.

And we accomplished that by wiring everything together

with a very generic and simple interface.

The third most important thing, without a doubt,

remember even though React is really popular today

and it's the total in framework,

eventually, in some number of years,

everyone's gonna forget React.

And if they don't forget it,

it might just be a very unpopular

and undesirable framework to work with.

So you want to be planning on,

perhaps, React,

maybe it's not gonna be so great

at some point in time,

and you might not have anyone at your company

who knows how to make changes to it.

So whenever you're trying to integrate together

your different apps,

maybe you don't want to just rely upon React

to shepherd information between each app for you.

Again, kinda comes down to your requirements.

All right, fourth item.

We saw a really nasty bug around handling CSS.

Remember that was a bug that only started to appear

once we went into production.

So just keep in mind that,

when you start to write out styling,

you probably want to apply some technique

of scoping your CSS between your different projects.

And very closely related to that,

we saw a great example of where everything

around our application was working

just fine in development,

and it was only when we built our project

and pushed it off to dev,

that stuff was not quite working so well.

And that, again, came down to a CSS issue.

But the point there was that

a lot of these different frameworks and whatnot

are gonna work slightly differently in production

than they do in development.

So it's always very important that,

if you develop your application,

everything looks good locally and you deploy it,

you still might want to do a little bit of a test

and just make sure that,

hey, everything in production is

working out as expected as well.

Okay, so that is it.

I hope you enjoyed this course.

I personally had a fantastic time recording it.

It really was a lot of fun for me.

I really enjoyed putting the application together.

I hope that I answered a lot of your big questions,

and I hope to talk to you

at some point in time, in the future.

Topic 124. Updating Project for React 19, React Router 6, MUI 5 1min

**Updating Project for React 19, React Router 6, MUI 5**

Attached to this lecture is a zip file containing a completed, working version of the MFP application upgraded to use React 19, React Router v6, and MUI v5 (formerly Material-UI v4). The major difference is how navigation works between our separate applications.

In the original project, React Router v5 gave us direct access to the history object. We could easily listen for navigation changes and sync them between our container and child applications.

With React Router v6, a history object is no longer exposed directly. Instead, navigation state is built into the Router component itself. To deal with this, we have added explicit state management to enable communication between the Microfrontends while keeping our Module Federation architecture intact.

Resources for this lecture