Orchestrating a browser based Micro FE architecture

Here at Globant we encounter several challenges across different projects and client's requirements. Regardless, there's always one thing in common: *how to handle several client apps to provide a stremlined UX and release process?*

Welcome to Micro Frontends!

Table of contents

What are Micro Frontends?

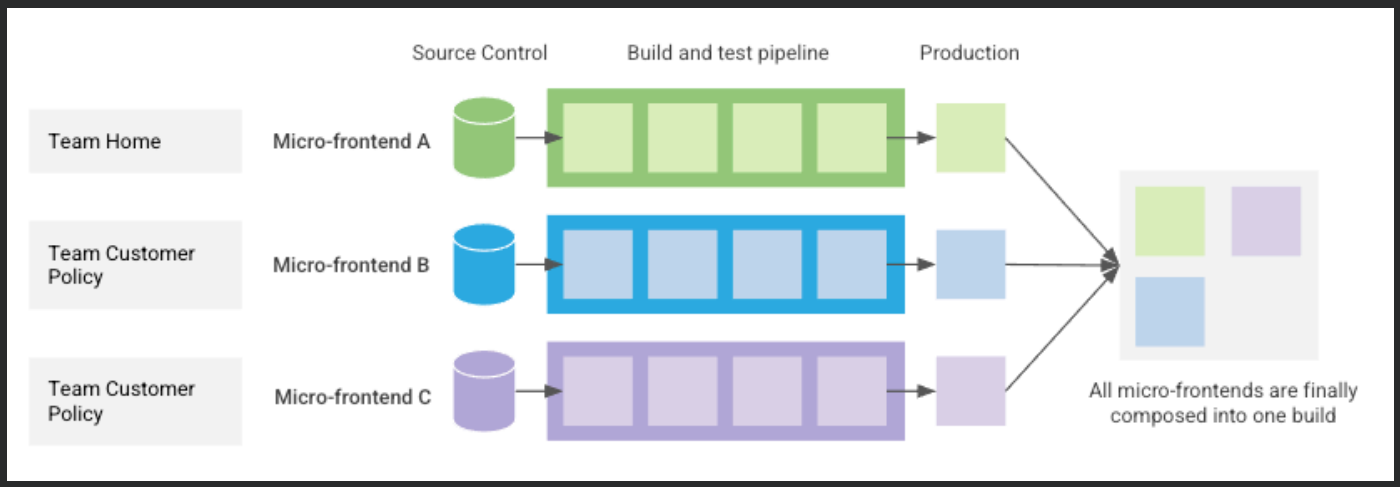
The term Microfrontends is a derivative of the microservices approach. It represents the architectural approach for the composition of multiple self contained and loosely coupled UI components (services), where each component is responsible for a specific UI element and / or functionality.

> A microfrontend is a microservice that exists within a browser.

Each microfrontend can be managed by a different team and may be implemented using its own framework.

Each microfrontend has its own git repository, its own `package.json` file, and its own build tool configuration. As a result, each microfrontend has **an independent build process** and **an independent deploy / CI**. This generally means that each repo has fast build times.

Micro frontends can be used to **empower product-oriented teams**Allows to structure teams around product-verticals (*bounded contexts*).Each team owns one or more micro frontend and are accountable for their quality.The frontend is then composed of micro-frontends developed by independent teams.

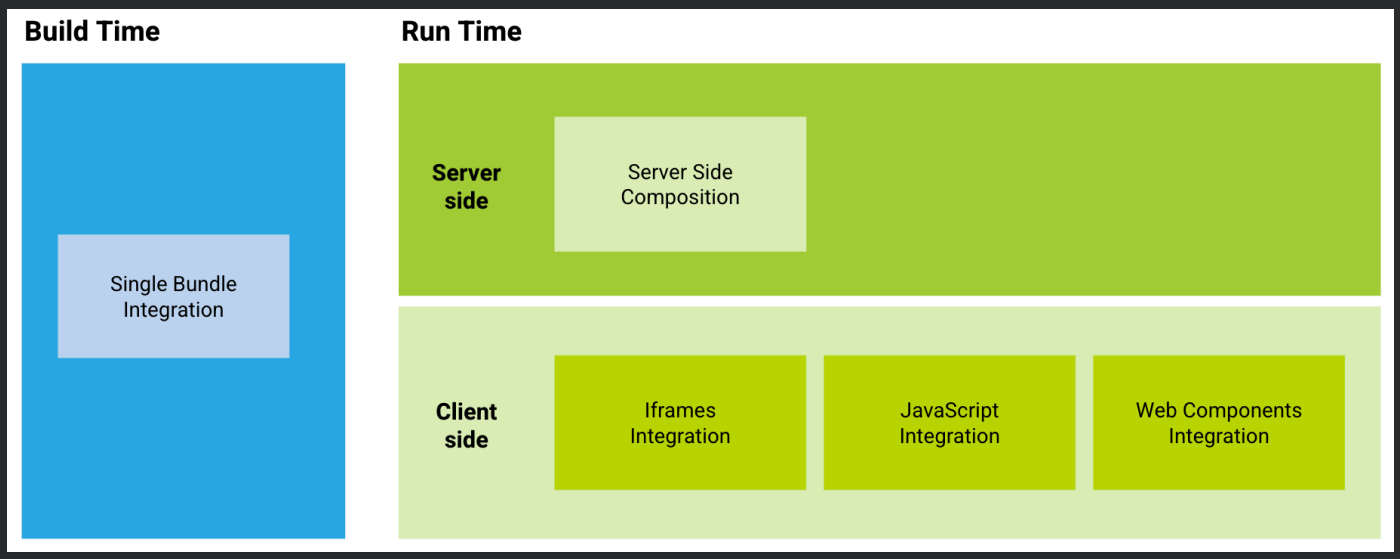


How do I chose the best strategy?

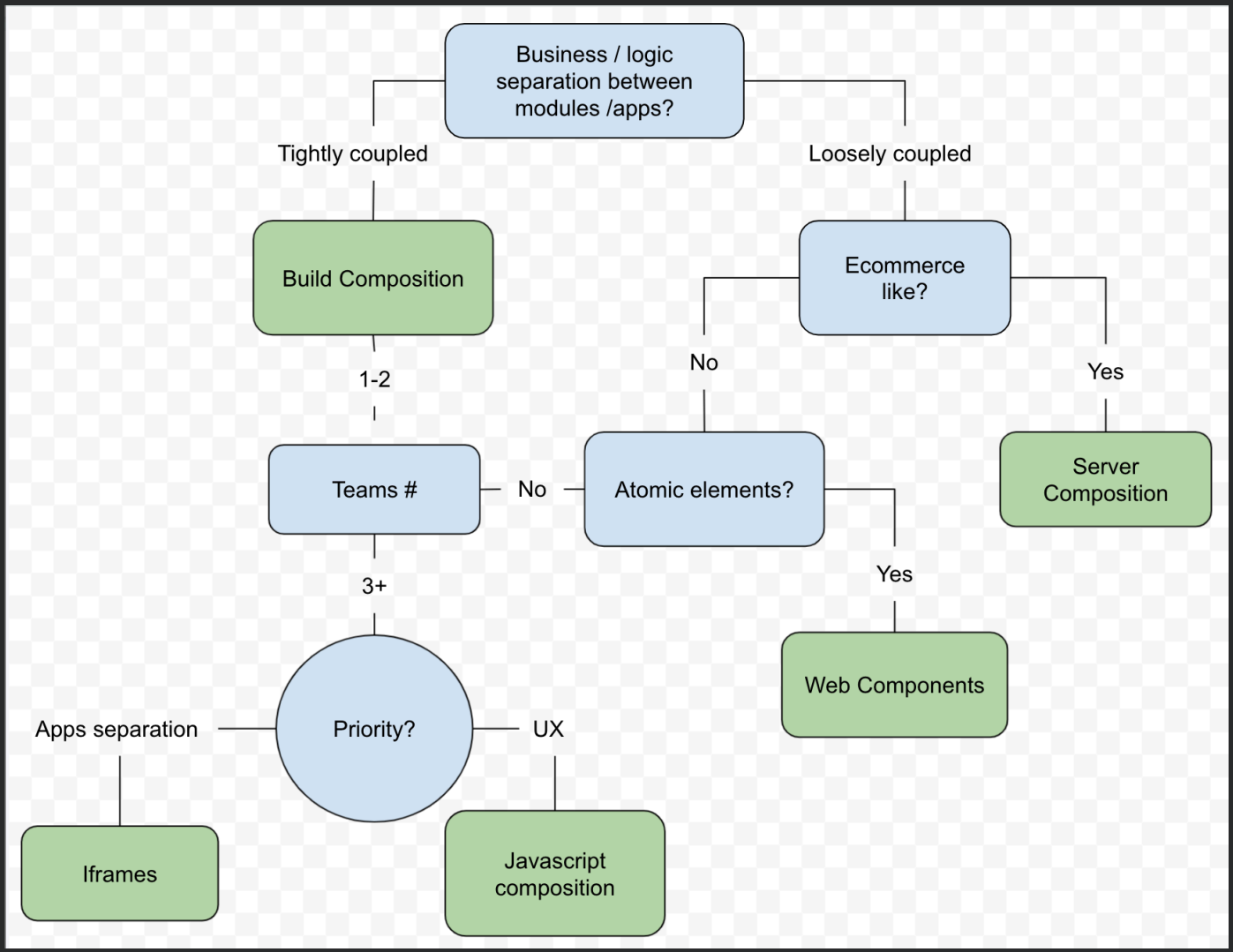
There's no wrong strategy here. The best solution is the one that fits your product and requirements.

You might want to have one big application that consumes the others as dependencies. That's **build time integration**.

You might want to do everything at **runtime**, either via a single rendering node (like a server) or in the browser itself.



The best decition comes from the proper answers. Here's a helpful chart that might make things easier:



Key considerations

Regardless of your choice, some key considerations must be taken into account:

Framework Compatibility

Most (if not all) of the modern frameworks have compatibility with web components and other micro frontend strategies. Reagardless you might want to check doccumentations just to be safe. All of them require some minor configuration though.

Avoid Framework Anarchy

There's no framework restrictions for making a micro frontend architecture. You can use a different one for each application. Regardless, **CAN** doesn't mean **should**; every decition has it's up and downs.

Using several frameworks can lead up to Framework Anarchy, wich can lead to different code standards, design patterns, and overall confusion between teams. That doesn't mean that sometimes a little anarchy is bad (like migrations). Use with caution.

Browser Support

Some *under the hood* APIs might not be supported by older browsers.

> Yes, we now it's 2021 and Edge took over IE and is basically Chrome under the hood. It'll surprise you how many people still use older browsers.

Remember to always check Can I use? for compatibility on the API of choice (Custom Elements, Shadow DOM, even Flexbox).

Just in case, there's always a Polyfill

Browser based JS orchestration: the default solution

As a rule of thumb, we suggest to use Browser based solutions. Most of the cases you can scale up from there, either integrating new applications or including Server Side Rendering nodes.

Having Javascript control what to require and load on-demmand gives us control of loading times, bundle sizes, and options. You can load different apps based on the client (mobile vs desktop), personalized apps for each user, do A/B testing. The combinations are limitless.

Of course, that means that you need to help the browser identify and require different modules.

> Browsers don't support ES imports at the moment. You need to look at other modularization options like SystemJS.

That means custom solutions, which takes **time**, **experience**, and **testing**. Sometimes we don't have the luxury to spend resources on that.

Luckly, there's ~~an app~~ a solution for that!

Introduction to Single SPA

Single SPA is an open source solution developed to solve this recurrent problem around handling multiple applications accross a single page environment.

According to their site:

> single-spa is a framework for bringing together multiple JavaScript microfrontends in a frontend application. Architecting your frontend using single-spa enables many benefits, such as:
>
> - Use multiple frameworks on the same page without page refreshing (React, AngularJS, Angular, Ember, or whatever you're using)
> - Deploy your microfrontends independently
> - Write code using a new framework, without rewriting your existing app

> - Lazy load code for improved initial load time

So they basically solve the base issue: *how do we route/control which app is active?*

Creating a streamlined scallable architecture with Single SPA

Let's explore how to configure our solution in order to support different kind of frameworks.

We'll go step by step, making changes and building on top of Single SPA.

First time generation

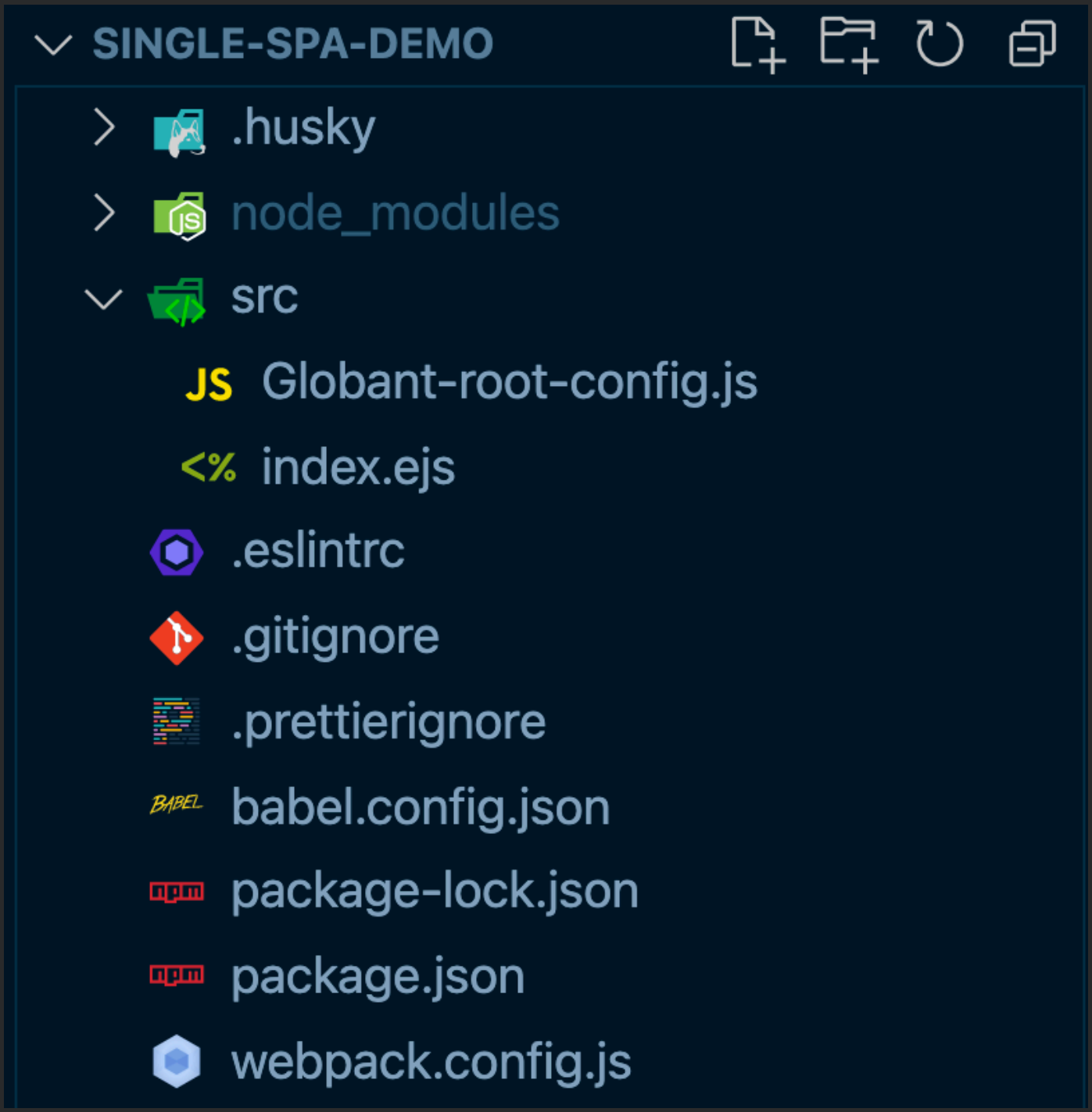
First things first, let's generate the base code scaffolding using Create Single SPA:

```
npx create-single-spa --moduleType root-config
```

A yeoman based CLI will takeover and you'll need to specify a few options. The crucial decitions are:

- Enable/Disable single-SPA Layout Engine -- used for Server Side Rendring (SSR)
- Organization Name -- used as namespace for module resolution

After the wizard finishes, navigate to the newly created folder. You should see a folder structure similar to this one:



Let's focus for a moment on the `index.ejs` file. That's your centralized module configuration file. There's actually a lot of configurations in the file, ranging from polyfills to CSP policies and ZoneJs fixes (for Angular); I recommend that once you generated an app you take the time to fully read them. We are going to focus to one section in particular: *the import map*.

```
<!-- Shared dependencies go into this import map. Your shared dependencies must be of one of the following formats:
  1. System.register (preferred when possible) - https://github.com/systemjs/systemjs/blob/master/docs/system-register.md
  2. UMD - https://github.com/umdjs/umd
  3. Global variable
  More information about shared dependencies can be found at https://single-spa.js.org/docs/recommended-setup#sharing-with
-import-maps.
-->
<script type="systemjs-importmap">
  {
    "imports": {
      "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-spa.min.js"
    }
  }
</script>
<link rel="preload" href="https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-spa.min.js" as="script">

<!-- Add your organization's prod import map URL to this script's src  -->
<!-- <script type="systemjs-importmap" src="/importmap.json"></script> -->

<% if (isLocal) { %>
<script type="systemjs-importmap">
  {
    "imports": {
      "@Globant/root-config": "//localhost:9000/Globant-root-config.js"
    }
  }
</script>
<% } %>
```

As you can see, by default we have the main bundle for Single SPA already imported as a dependency (by default that's imported from a public CDN, but you can always have that hosted somewhere else). and our app's root config imported as a local dependency.

Later, we'll need to update with our *production import map* this file (as you can see in one comment).

Now let's take a look at our `root-config.js` file.

```
import { registerApplication, start } from "single-spa";

registerApplication({
  name: "@single-spa/welcome",
  app: () =>
    System.import(
      "https://unpkg.com/single-spa-welcome/dist/single-spa-welcome.js"
    ),
  activeWhen: ["/"],
});

// registerApplication({
//   name: "@Globant/navbar",
//   app: () => System.import("@Globant/navbar"),
//   activeWhen: ["/"]
// });

start({
  urlRerouteOnly: true,
});
```

This is our main bootstraping file. In here we need to *register* our apps. As we can see, by default the *welcome app* is already there and a nice comment with an example is provided so we can see how to import an aliased module.

Let's run the app and see the result:

```
npm start
```

Logs will rain, a URL will be shared and navigated, and... *voila!*

Welcome

to your single-spa root config! 🥳

This page is being rendered by an example single-spa application that is being imported by your root config.

Next steps

1. Add shared dependencies

- Locate the import map in `src/index.ejs`
- Add an entry for modules that will be shared across your dependencies. For example, a React application generated with create-single-spa will need to add React and ReactDOM to the import map.

```
"react": "https://cdn.jsdelivr.net/npm/react@16.13.1/umd/react
"react-dom": "https://cdn.jsdelivr.net/npm/react-dom@16.13.1/u
```

Refer to the corresponding **single-spa framework helpers** for more specific information.

2. Create your next single-spa application

- Generate a single-spa application with create-single-spa and follow the prompts until it is running locally
- Return to the root-config and update the import map in `src/index.ejs` with your project's name

  > It's recommended to use the application's package.json name field

- Open `src/root-config.js` and remove the code for registering this application
- Uncomment the `registerApplication` code and update it with your new application's name

After this, you should no longer see this welcome page but should instead see your new application!

Learn more

- **Shared dependencies documentation on single-spa.js.org**
- **SystemJS** and **Import Maps**
- **Single-spa ecosystem**

Contribute

- **Support single-spa by donating on OpenCollective!**
- Contribute to **single-spa on GitHub**!
- Join the Slack group to engage in discussions and ask questions.
- Tweet **@Single_spa** and show off the awesome work you've done!

Adding our first app

Allright, let's get busy and start building upon our base.

Let's create a new app using the lovely `create-single-spa` CLI:

```
npx create-single-spa --moduleType app-parcel
```

The wizard will take over, for this example we'll generate a React SPA.

> Remember to use the same organizarion name as the root-config. Got's to have consistency.

Once finished, navigate to the app's folder (yes, that can be anywhere) and start the app. Take note of the served URL, we'll need that in the next step.

Let's go back to the `root-config.js` file. Remember the *import map*? We need to update that with our new app. Since this is for local development we'll update the local map.

```
<% if (isLocal) { %>
<script type="systemjs-importmap">
  {
    "imports": {
      "@Globant/root-config": "//localhost:9000/Globant-root-config.js",
      "@Globant/App1": "//localhost:8080/Globant-App1.js"
    }
  }
</script>
<% } %>
```

Let's talk dependencies for a moment.

For now, we have a React app that needs React (and React DOM) in order to run. What's gonna happen when we add a second React App? And a third? We don't want 3 apps bringing the same dependency! Fortunately, our friends at SSPA already tought about that and took advantage of Webpack and SystemJS.

All shared libraries can be imported once and used by all our apps, we just need to add them into the *import map*:

```
<script type="systemjs-importmap">
  {
    "imports": {
      "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-spa.min.js",
      "react": "https://cdn.jsdelivr.net/npm/react@16.13.1/umd/react.production.min.js",
      "react-dom": "https://cdn.jsdelivr.net/npm/react-dom@16.13.1/umd/react-dom.production.min.js"
    }
  }
</script>
```

What if one app needs a different version? Let's say... react 15? We can scope our dependencies to be served on specific modules!

```
<script type="systemjs-importmap">
  {
    "imports": {
      "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-spa.min.js",
      "react": "https://cdn.jsdelivr.net/npm/react@16.13.1/umd/react.production.min.js",
      "react-dom": "https://cdn.jsdelivr.net/npm/react-dom@16.13.1/umd/react-dom.production.min.js"
    },
    "scope": {
      "/app2": {
        "react": "https://cdn.jsdelivr.net/npm/react@15.2.1/umd/react.production.min.js",
      }
    }
  }
</script>
```

Anyways... Now we can go ahead and register our app in the `root-config.js` file:

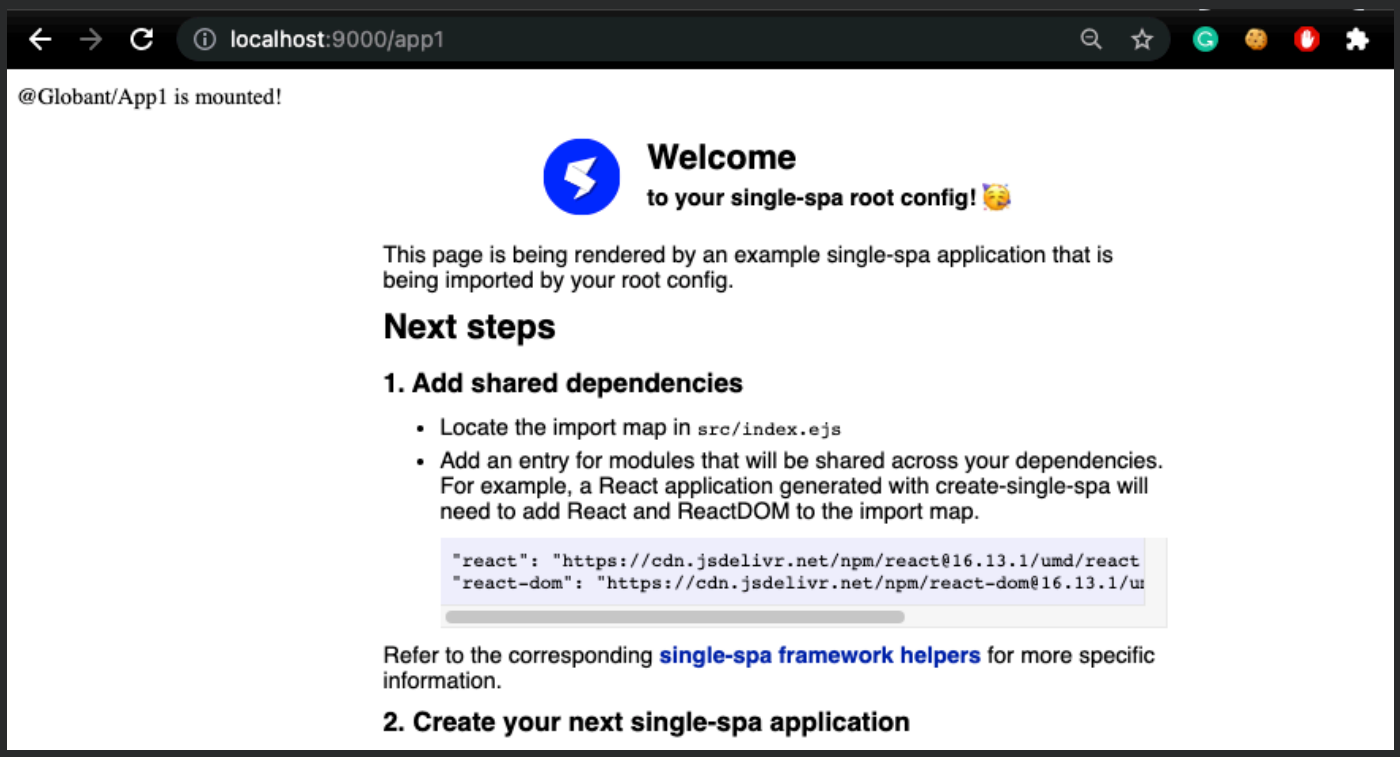
```
import { registerApplication, start } from "single-spa";

registerApplication({
  name: "@single-spa/welcome",
  app: () =>
    System.import(
      "https://unpkg.com/single-spa-welcome/dist/single-spa-welcome.js"
    ),
  activeWhen: ["/"],
});

registerApplication({
  name: "@Globant/App1",
  app: () => System.import("@Globant/App1"),
  activeWhen: ["/app1"]
});

start({
  urlRerouteOnly: true,
});
```

What did we do here? we added our app `@Globant/App1` and activated it only when the browser visits `/app1`. Here's the result:



So, why do we have the welcome app in the page?

All applications are sharing the DOM at the same time, so all of them are querying (and rendering) the same main DOM elements (in this case, the body). When we register, we are specifing the route where the app will be present. In this case we have:

- Welcome app
  - active when `/` is visited
- App1
  - active when `/app1` is visited

The `activeWhen` property accepts a *prefix* string. The validation is done on the start of the URL, **everything after the prefix will be valid**. The welcome app will always show after the `/` route.

How do we fix this?

There are several ways for us to tackle this. For now we are just going to remove the welcome app as we don't need it.

It's worth to note that Single SPA provides us with different modules types that enable different control mechanisms:

- single-spa applications: Microfrontends that render components for a set of specific routes.
- single-spa parcels: Microfrontends that render components without controlling routes.
- utility modules: Microfrontends that export shared JavaScript logic without rendering components.

Takeaway

Single SPA is a poweful solution that empowers the Micro Frontend architecture. We covered the very basics, but there's a lot more to continue. We do recommend to explore the docs and the examples to help you tailor your solution.

There's no *one solution for them all*, but some tools help us reach that baseline that we expect for all our projects. Single SPA is one of them.