Applause from you, Joel Denning, and 32 others

Joel Denning
@Joelbdenning
Dec 16, 2016 · 7 min read

A step-by-step guide to single-spa

Running Angular 1, React, Angular 2, and Vue.js side by side sounds pretty cool. And it seems appealing to have multiple applications coexisting on the same page, each lazily loaded.

But using single-spa for the first time can be tricky because you'll come across terms like "application lifecycles", "root application", "loading function", "child application", and "activity function."

This blog post will take you through setting things up and what choices you have when using single-spa. It's based on what I've seen at Canopy Tax where we went from an Angular 1 monolith to an Angular 1, React, and Svelte polyglot.

If you'd like to jump straight to a fully working, self contained code example, check out this webpack single-spa starter project.

. . .

Step One: choose a module loader.

Your module loader / bundler is the library you'll use to lazy load code. I recommend either Webpack or JSPM, if you're starting from scratch.

If you go with Webpack, try to use Webpack 2 if you can, since it has support for promise-based lazy loading. This will make things easier for you later on, since single-spa requires that your loading functions return promises. If you can't use Webpack 2, getting single-spa to lazy load your code with Webpack 1 will require some boilerplate code.

JSPM/SystemJS has worse documentation than Webpack, but is a great solution for module loading if you can get past that. I recommend using jspm@0.17—it's still in beta but has been worked on for over a year and at Canopy we find it stable enough to use in production.

If you're struggling to decide between the two, then ask yourself the following: Do I want multiple completely separate bundles? If you don't, I recommend Webpack because it has better docs, a larger community, and fewer gotchas. Otherwise, I'd go with JSPM, since Webpack has no plans to support dynamic runtime loading (See tweet below from Mr. Larkin, himself).

Sean Thomas Larkin
@TheLarkInn

Replying to @Joelbdenning @dan_abramov
indeed!! It's a static bundler! Module loading is something we won't do :)

Step Two: create a brand new html file

The next step is to create what single-spa calls your "root application." Really your root application is just the stuff that initializes single-spa, and it starts with an html file.

Even if you've got an existing project that already has it's own html file, I recommend starting fresh with a new html file. That way, there is a clear distinction between what is in your root application (shared between all apps) and what is in a child application (not shared with everything).

You'll want to keep your root application as small as possible, since it's sort of the master controller of everything and could become a bottleneck. You don't want to be constantly changing *both* the root application *and* the child applications.

So for now, just have a <script> to a single javascript file (root-application.js), which will be explained in Step Three.

Since Webpack is probably the more common use case, my code examples from here on will assume that you're using Webpack 2. The equivalent Webpack 1 or JSPM code has all the same concepts and only some minor code differences.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width"
    <title>A single-spa application</title>
  </head>
  <body>
    <div id="cool-app"></div>
```

Step Three: register a "child application"

Now it's time to finish up your root application by writing your "root-application.js" file. The primary purpose of root-application.js is to call `singleSpa.declareChildApplication(..)` for each of the applications that will be managed by single-spa.

If you're into analogies, you can think of single-spa as the operating system for your single page application, managing which "processes" (or "child applications") are running at any given time. At any moment, some of the child applications will be active on the DOM and others will not. As the user navigates throughout the app, some applications will be unmounting from the DOM and others will be mounting to the DOM.

Another way to look at it is that single-spa is a master router on top of your other routers.

To do this, first `npm install single-spa` and then call the declareChildApplication function:

```
import {declareChildApplication, start} from 'single-s

// Register your first application with single-spa. Mo
declareChildApplication("cool-app", loadCoolApp, isCoo

// Tell single-spa that you're ready for it to mount y
start();

// This is a "loading function"
function loadCoolApp() {
  return import("./cool-app/cool.app.js");
}
```

Because single-spa is *so very* cool, we've created an app called "cool-app" that will be lazy loaded and mounted to the DOM whenever the url hash starts with `#/cool`.

The `loadCoolApp` function is what single-spa calls a loading function. Inside of it, the `import` introduces a code splitting point—Webpack will create separate code chunks that will be lazy loaded by single-spa.

For your specific project, you probably won't have a hash prefix of "cool", but I recommend establishing some kind of convention that makes it easy to determine which apps are active. This will simplify the maintenance of your activity functions, as you add more and more child applications.

If you're going to start out with just one child application, then it might make sense to implement the activity function as `() => true`. You can worry about getting fancier once you have more than one application.

The last thing is to call `start()`. This is something you **must do for things to work.** The purpose is to give control over timing and performance. But until that is a concern, `start` is just one of those things you do, and then maybe read about it later if you ever need to.

Step Four: create ".app.js" file

When you open up your index.html file in the browser, you'll now see….. a blank screen! We're really close, but there's one crucial step left: building your app.js file.

After that, you'll have everything working for your first single-spa application.

An app.js file is a configuration file that you create for each child application. It is the code that is lazy loaded when your activity function returns true.

There are three things that you need to implement in the app.js file:

1. A bootstrap lifecycle
2. A mount lifecycle
3. An unmount lifecycle

A "lifecycle" is a function or array of functions that will be called by single-spa; you export these from the app.js file. Each function must

return a Promise so that single-spa knows when it is completed.

Here is a simple example:

```
// single-spa will import this file and call the expor

let user;

export function bootstrap() {
  return fetch('/api/users/0')
    .then(response => response.json())
    .then(json => user = json);
}

export function mount() {
  /* This is normally where you would have your framew
   * ReactDOM.render or angular.bootstrap(). The fact
   * into this function is what makes single-spa so po
   * can implement a "mount" and "unmount" to become a
   */
  return Promise.resolve()
    .then(() => {
      document.getElementById("user-app").innerHTML =
        <div>
          Hello ${user.name}!
        <div>
      `
```

At this point, you might be seeing the `document.getElementById` and `innerHTML =` and worry that you've been duped—maybe single-spa is really just a poor excuse for a ui component framework.

And really, don't we already have a lot of different ways to write UI components?

> *But getting all of those frameworks to work together.*

Using multiple frameworks is where single-spa really shines. It is not a ui framework itself, but a framework for using other frameworks.

Each child application can be written in any framework, so long as it implements application lifecycle functions. Then the mini-apps cooperate to form the entire single page application.

So going back to our previous example, we could choose to write our "cool.app.js" as an Angular 1 app, and choose something else for future apps:

```
import singleSpaAngular1 from 'single-spa-angular1';
import angular from 'angular';
import './app.module.js';
import './routes.js';

const domElementGetter = () => document.getElementById

const angularLifecycles = singleSpaAngular1({
  angular,
  domElementGetter,
  mainAngularModule: 'single-spa-app',
  uiRouter: true,
  preserveGlobal: true,
});

export const bootstrap = [aboutToBootstrap, angularLif

export const mount = [angularLifecycles.mount];

export const unmount = [angularLifecycles.unmount];
```

In this example, we use a helper library called `single-spa-angular1` which abstracts away the specifics of initializing Angular 1 apps. This blogpost doesn't show you the `app.module.js` or `routes.js` files, but you can see an example implementation here.

The pattern is to call `singleSpaAngular1` at the very beginning, which returns `bootstrap`, `mount`, and `unmount` lifecycle functions for you.

You might notice that this time the lifecycles are exported as arrays of functions instead of just functions—you can choose whichever works best for you.

The advantage of exporting an array of functions is that you can add in your own custom behavior (like `aboutToBootstrap` and `doneBootstrap`) that will run before or after the Angular 1 lifecycles. When you export an array, each item in the array must be a function that returns a promise. Single-spa will wait for each promise to resolve, in order, before calling the next function in the array.

To learn more about single-spa helper libraries, check out these github projects:

- single-spa-angular1
- single-spa-react
- single-spa-angular2
- single-spa-vue
- single-spa-svelte
- single-spa-preact

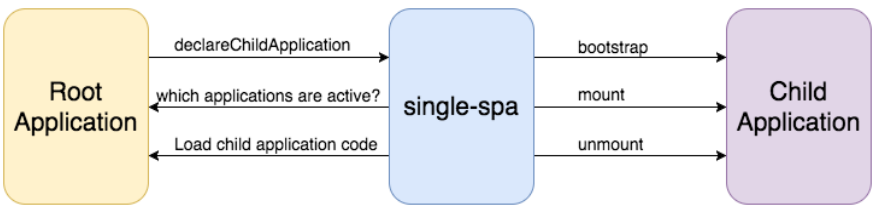
You can also see a fully working example of an angular app coexisting with other apps at the single-spa-examples repo or the live demo.

Step Five: test it out!

Refresh your page and you should now have a functioning single-spa application!

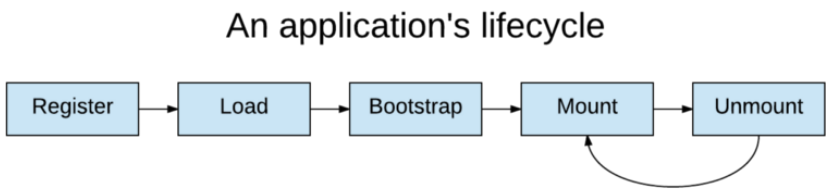
Try navigating to a url that your child app is active for (`#/cool`) and then navigating away from it. When you do so, the page will not refresh but you should see your application mount itself to the DOM and then unmount.

If you run into problems, try to narrow down whether the problem is in the root application or in the child application. Is your root application being executed? Are the declareChildApplication calls being made? Have you called `start()`? Is there a network request to download the code for your child application? Is your child application's `bootstrap` lifecycle being called? What about `mount`?



It may be helpful to add a navigation menu, so you can verify everything mounts and unmounts to the DOM correctly. If you want to level up your single-spa skills even more, make the navigation menu an entire child application whose activity function is `() => true`. An example that does just that is found here and here.

While you are verifying that everything is working, keep in mind that each application goes through five phases:



Conclusion

As you get your feet wet, you'll probably run into some (hopefully small) hiccups setting things up. When this tutorial is not enough, there are other resources on Github.

Single-spa is still a relatively new thing, and we'd love to hear your feedback and questions. We welcome contributions from everyone.

If you're excited about the possibilities, feel free to contact me (twitter @joelbdenning). And if you are not excited, then still feel free to contact me, but only after you leave some nasty comments :)