Webinar - ‘Micro Frontends’

Micro App

Micro App

Micro App

By Om Singh

Open Mentoring Group

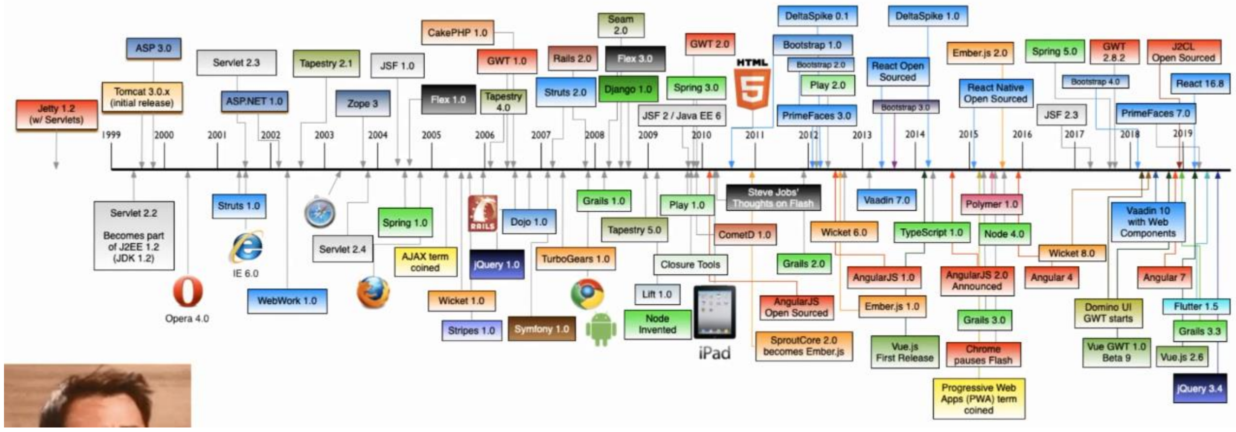
Happy2Help

What is Micro Frontends ?

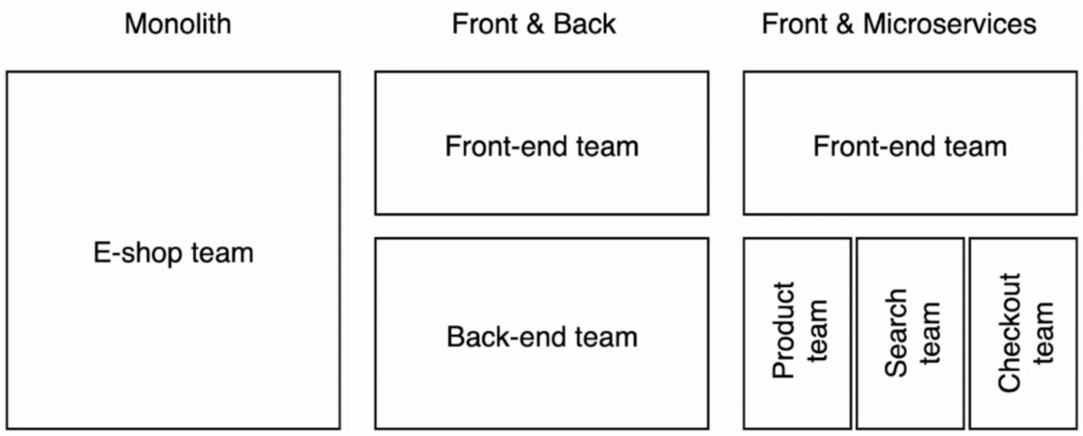
Domain driven design approach.

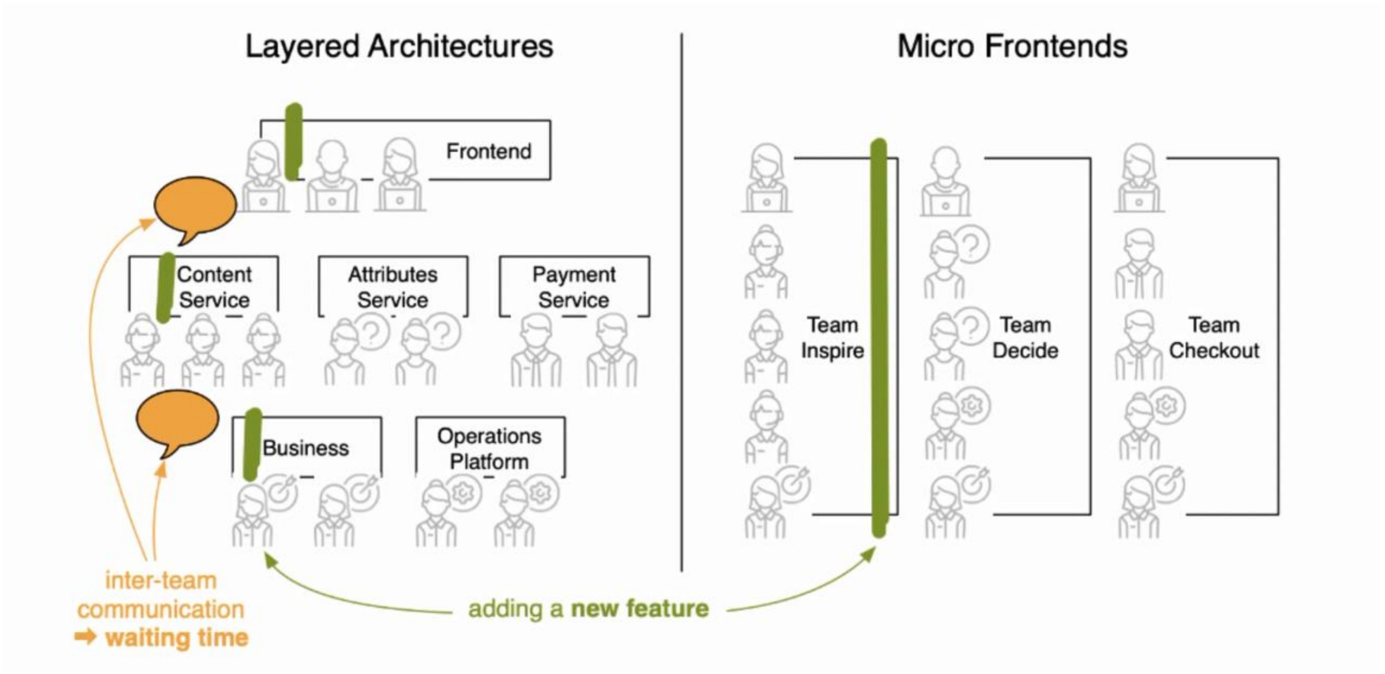
Extending the microservices architecture pattern to Frontend-development.

History of web frameworks timeline

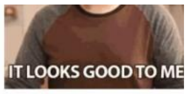


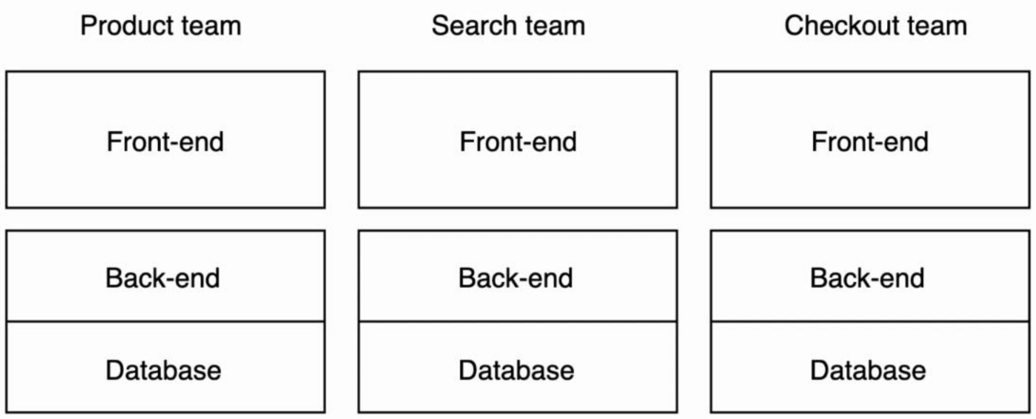
The Traditional Approach



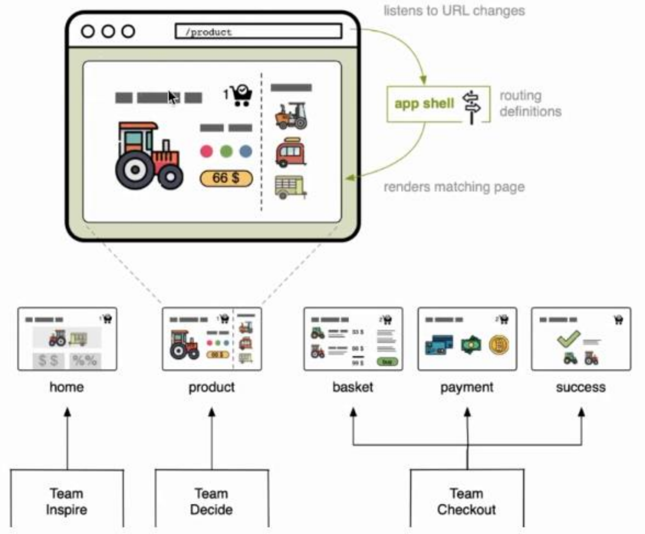


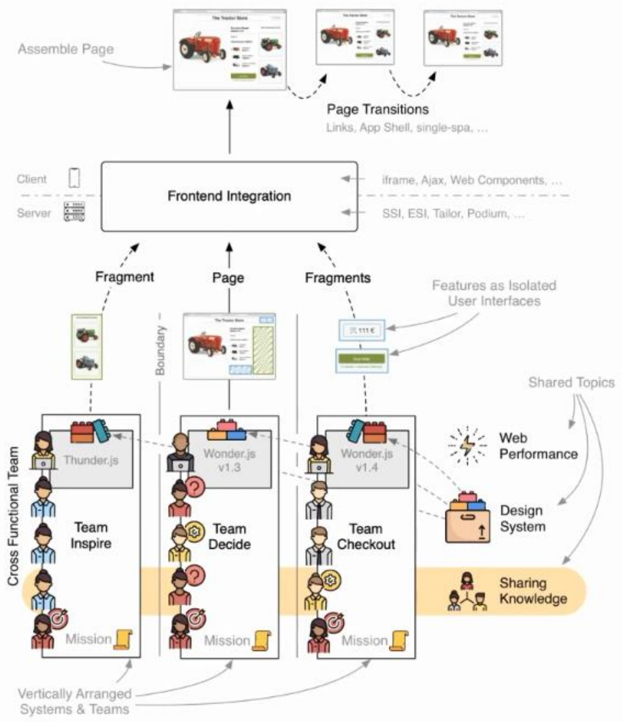
The Micro Frontends approach





Vanilla Architecture





Why micro frontends?

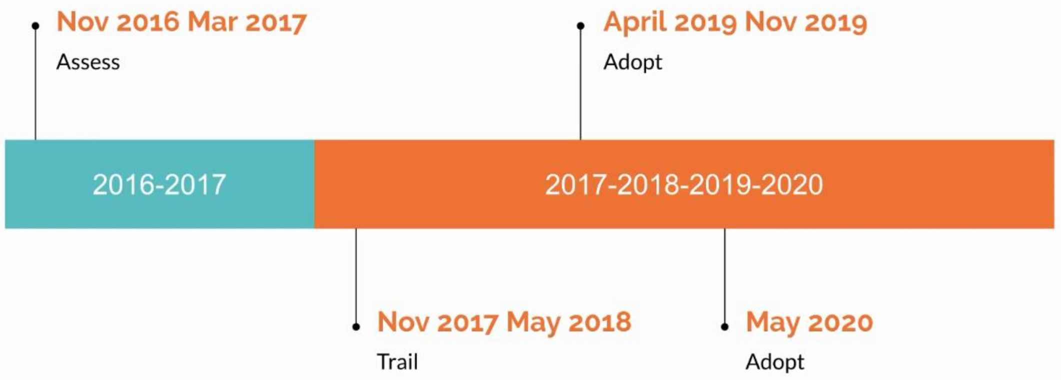
Because traditional approach doesn't work for larger web apps

Tip

Micro-frontend is an architecture pattern.

"An architectural style where independently deliverable frontend applications are composed into a greater whole"

Thoughtworks Technology Radar



https://www.thoughtworks.com/radar/techniques/micro-frontends

Benefits

★ Incremental upgrades

★ Simple, decoupled codebases

★ Reducing communication overhead

★ Independent deployment != Independent release

★ Autonomous Teams

★ Freedom to innovate

Incremental upgrades

For many organisations this is the beginning of their micro frontends journey.

The old, large, frontend monolith is being held back by yesteryear's tech stack, or by code written under delivery pressure, and it's getting to the point where a total rewrite is tempting.

In order to avoid the perils of a full rewrite, we'd much prefer to strangle the old application piece by piece, and in the meantime continue to deliver new features to our customers without being weighed down by the monolith.

Simple, decoupled codebases

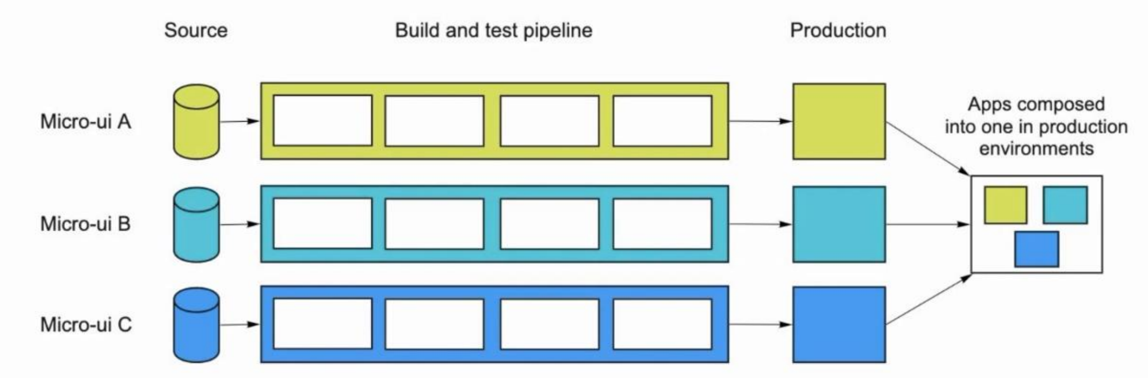
These smaller codebases tend to be simpler and easier for developers to work with.

In particular, we avoid the complexity arising from unintentional and inappropriate coupling between components that should not know about each other. By drawing thicker lines around the bounded contexts of the application, we make it harder for such accidental coupling to arise.

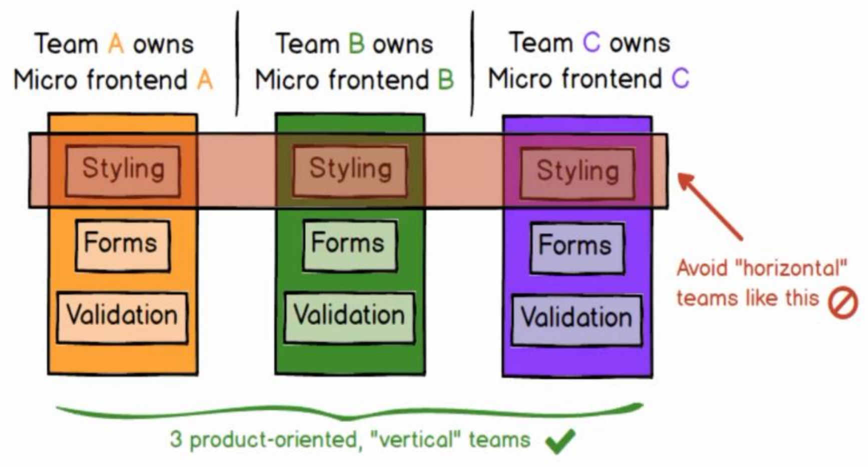
For example, sharing domain models across bounded contexts becomes more difficult, so developers are less likely to do so.

Similarly, micro frontends push you to be explicit and deliberate about how data and events flow between different parts of the application

Independent deployment



Autonomous Teams


Team A owns Micro frontend A
Team B owns Micro frontend B
Team C owns Micro frontend C
Styling
Forms
Validation
Styling
Forms
Validation
Styling
Forms
Validation
Avoid "horizontal" teams like this
3 product-oriented, "vertical" teams

Micro Frontends Core Principles

- ★ Should solve specific business problem
- ★ Reusable smaller app/fragment/component
- ★ Micro app should be independent and isolated
- ★ Minimum dependencies
- ★ Technology or framework agnostic
- ★ Easier to understand, develop, test and deploy
- ★ Faster and reliable CI/CD

Downsides

- ★ Payload Size due to library but Module federation
- ★ Redundant codebase
- ★ Environment differences
- ★ Operational and governance complexity
- ★ No standard framework but

  https://opencomponents.github.io/ is interesting

Payload size

Independently-built JavaScript bundles can cause duplication of common dependencies, increasing the number of bytes we have to send over the network to our end users.

For example, if every micro frontend includes its own copy of React, then we're forcing our customers to download React n times.

By building independently, any single page-load will only download the source and dependencies of that page. This may result in faster initial page-loads, but slower subsequent navigation as users are forced to re-download the same dependencies on each page.

It's important to consider the performance impacts of every architectural decision, be sure that you know where the real bottlenecks are.

Environment differences

We should be able to develop a single micro frontend without needing to think about all of the other micro frontends being developed by other teams.

If our development-time container behaves differently than the production one, then we may find that our micro frontend is broken, or behaves differently when we deploy to production.

The solution here is not that different to any other situation where we have to worry about environmental differences.

This will not completely solve the problem, but ultimately it's another tradeoff that we have to weigh up: is the productivity boost of a simplified development environment worth the risk of integration issues? The answer will depend on the project!

Operational and governance complexity

The final downside is one with a direct parallel to microservices.

As a more distributed architecture, micro frontends will inevitably lead to having more stuff to manage - more repositories, more tools, more build/deploy pipelines, more servers, more domains, etc.

The main point we wish to make is that when you choose micro frontends, by definition you are opting to create many small things rather than one large thing.

You should consider whether you have the technical and organisational maturity required to adopt such an approach without creating chaos.

Operational and governance complexity

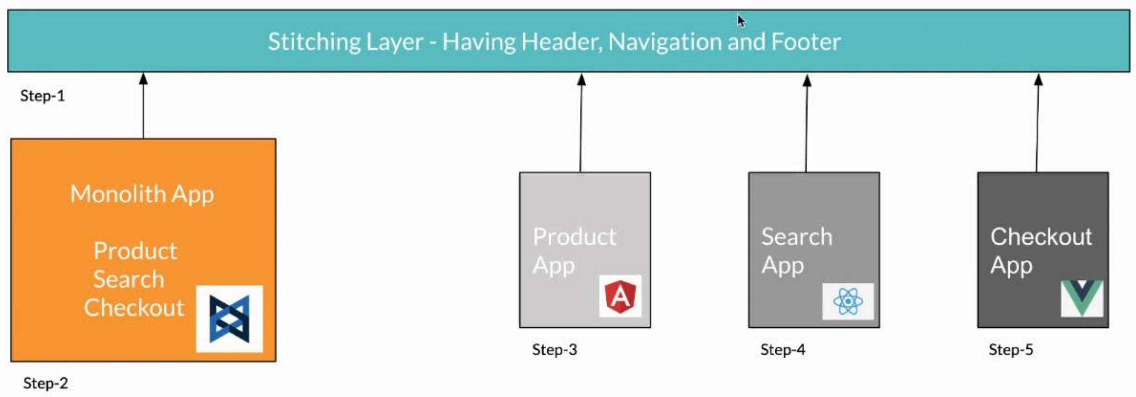
Do you have enough automation in place to feasibly provision and manage the additional required infrastructure?

Will your frontend development, testing, and release processes scale to many applications?

Are you comfortable with decisions around tooling and development practices becoming more decentralised and less controllable?

How will you ensure a minimum level of quality, consistency, or governance across your many independent frontend codebases?

How to migrate legacy App



Problems to solve

- ★ How to compose stitching layer
- ★ Cross-application communication
- ★ Backed communication
- ★ Testing
- ★ Common/shared content
- ★ Infrastructure
- ★ Consistent theme/style

How to compose stitching layer

Client side orchestration	Server side orchestration
client routing	server routing with proxying requests
state sharing	register all applications
register all applications	resolve shared dependencies if any
resolve shared dependencies if any	serving and composing fragments from different micro frontend apps
initialise the main app	
compose fragments from different micro frontend apps	

Cross-application communication

Whatever approach we choose, we want our micro frontends to communicate by sending messages or events to each other, and avoid having any shared state.

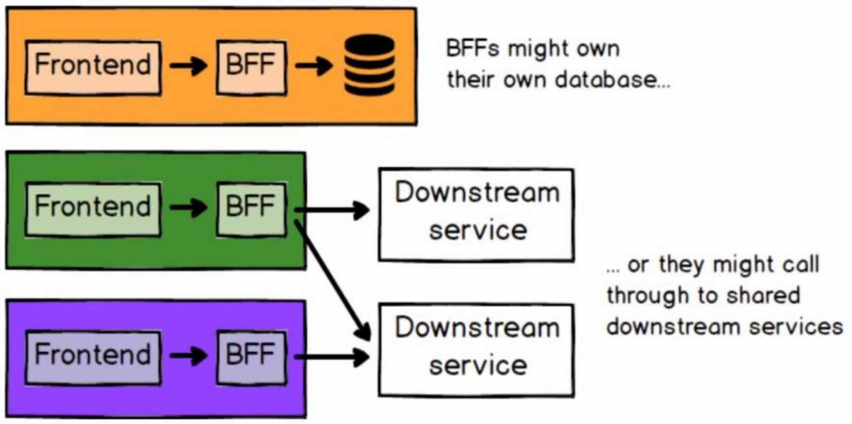
Just like sharing a database across microservices, as soon as we share our data structures and domain models, we create massive amounts of coupling, and it becomes extremely difficult to make changes.

An event bus implements the publisher/subscriber pattern.

It can be used to decouple the components of an application, so that a component can react to events fired from another component without them having direct dependencies with each other.

https://github.com/chrisdavies/eev

Backend communication


Frontend
BFF
BFFs might own their own database...
Frontend
BFF
Downstream service
Downstream service
... or they might call through to shared downstream services
Frontend
BFF

Testing

Use unit tests to cover your low-level business logic and rendering logic, and then use functional tests just to validate that the page is assembled correctly.

For example, you might load up the fully-integrated application at a particular URL, and assert that the hard-coded title of the relevant micro frontend is present on the page.

If there are user journeys that span across micro frontends, then you could use functional testing to cover those.

But keep the functional tests focussed on validating the integration of the frontends, and not the internal business logic of each micro frontend, which should have already been covered by unit tests.

Common content

While we want our teams and our micro frontends to be as independent as possible, there are some things that should be common.

We can create repository of common content, including images, JSON data, and CSS, library which are served over the network to all micro frontends.

The first step is to choose which dependencies to share.

Sharing code across teams is always a tricky thing to do well. We need to ensure that we only share things that we genuinely want to be common, and that we want to change in multiple places at once.

Infrastructure

The application is hosted on AWS, with core infrastructure (S3 buckets, CloudFront distributions, domains, certificates, etc), provisioned all at once using a centralised repository of Terraform code.

Each micro frontend then has its own source repository with its own continuous deployment pipeline on Travis CI, which builds, tests, and deploys its static assets into those S3 buckets.

Note that each micro frontend (and the container) gets its own bucket.

This means that it has free reign over what goes in there, and we don't need to worry about object name collisions, or conflicting access management rules, from another team or application.

Frameworks to implement

Server Side Frameworks

SSI (Server Side Includes)

Varnish ESI (Edge Side Includes)

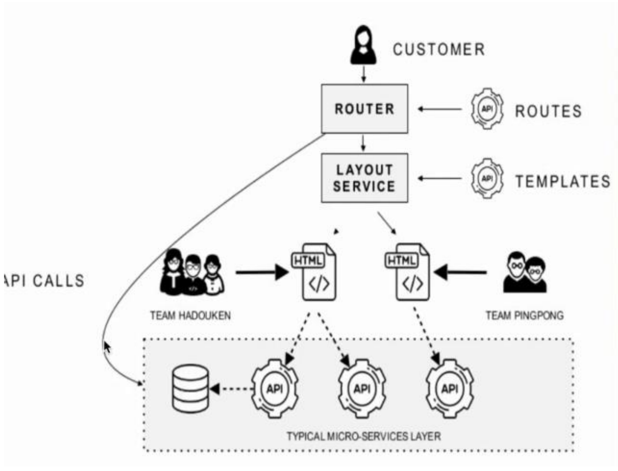
Tailor - https://github.com/zalando/tailor

Client Side Frameworks

Single-spa - https://single-spa.js.org/

Frint JS - https://frint.js.org/

<iframe>



Zalando's solution

It is called the Project Mosaic9 https://www.mosaic9.org/

They are loading every fragment in a separated iframe.

For communication and for coordinating the events across different iframes they are using event bus.

Every fragment has its own dependencies and data. The downside of this solution is that some dependencies are loaded more times.

Spotify's solution

Request parsing: web server parses and sanity checks the HTTP request.

Data fetching: web server fetches data from storage tier.

Markup generation: web server generates HTML markup for the response.

Network transport: the response is transferred from web server to browser.

CSS downloading: browser downloads CSS required by the page.

DOM tree construction and CSS styling: browser constructs DOM tree of the document, and then applies CSS rules on it.

JavaScript downloading: browser downloads JavaScript resources referenced by the page.

JavaScript execution: browser executes JavaScript code of the page.

Facebook's solution

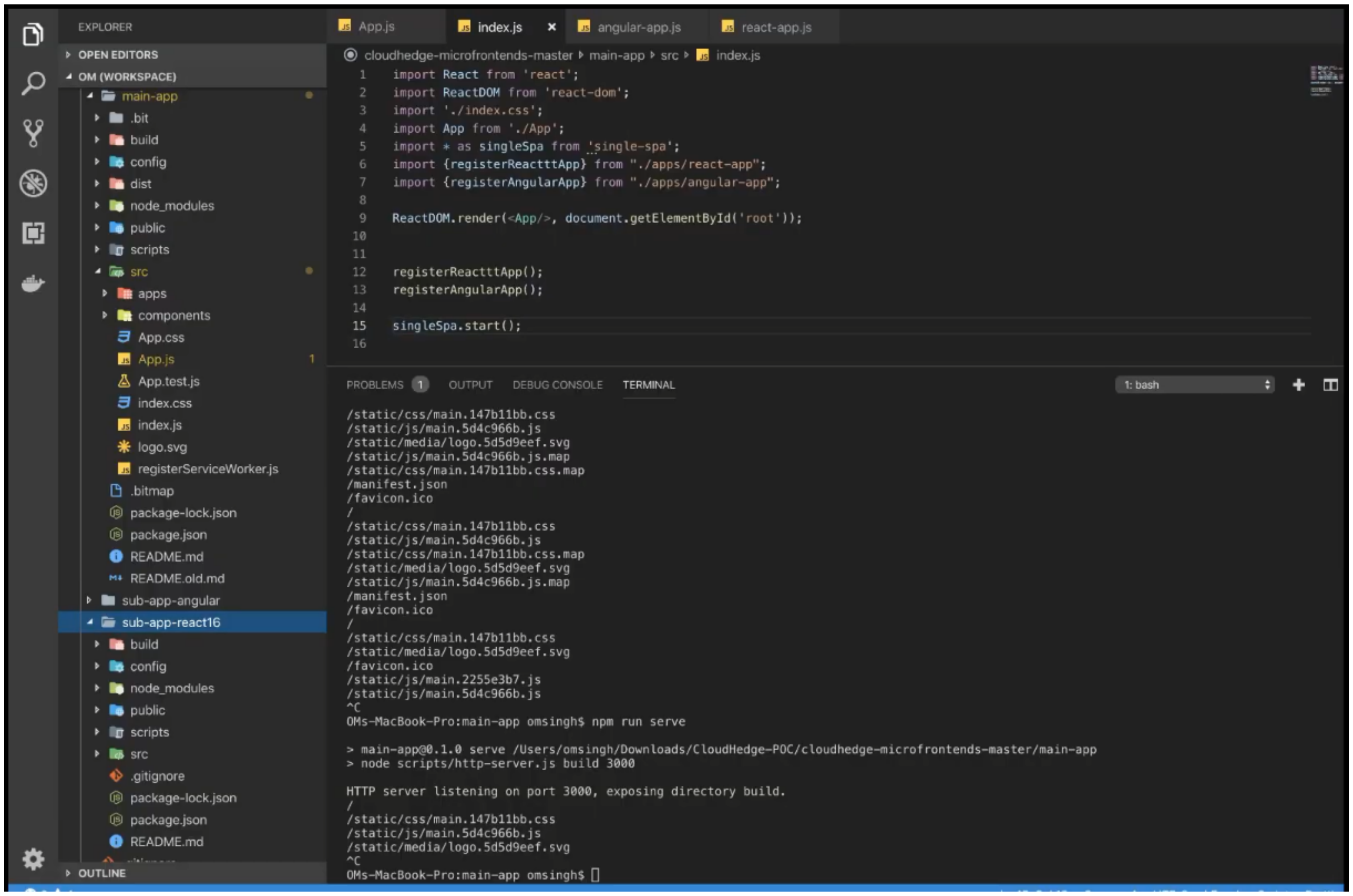
They call it BigPipe.

https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919/

Single-Spa

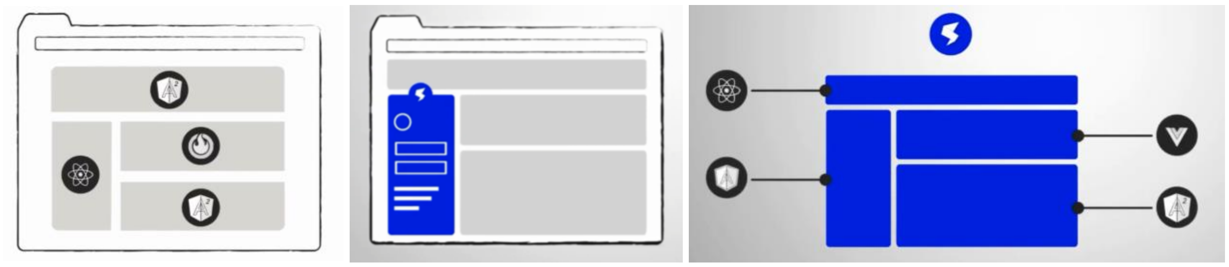
A javascript router for front-end microservices

See a short video on Single-Spa

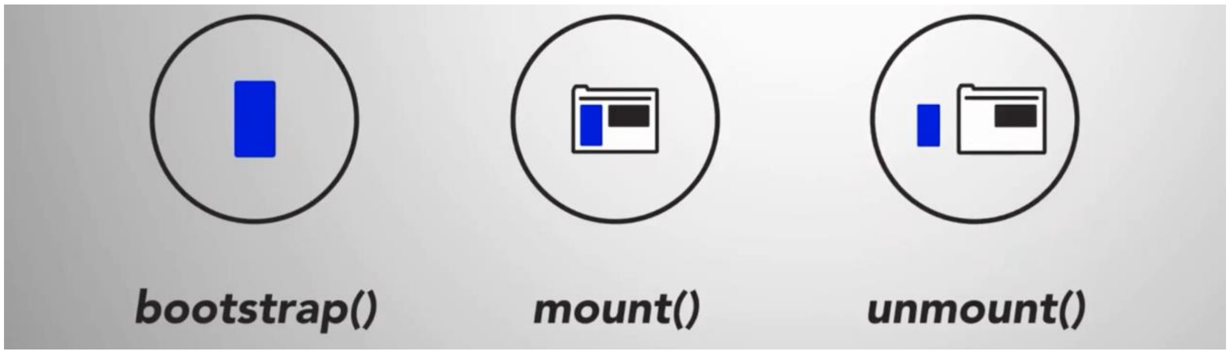


```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as singleSpa from 'single-spa';
import {registerReactttApp} from "./apps/react-app";
import {registerAngularApp} from "./apps/angular-app";

ReactDOM.render(<App/>, document.getElementById('root'));


registerReactttApp();
registerAngularApp();

singleSpa.start();
```

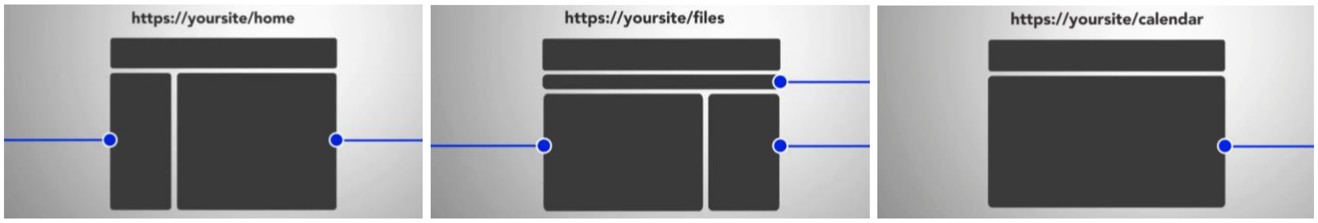
```
/static/css/main.147b11bb.css
/static/js/main.5d4c966b.js
/static/media/logo.5d5d9eef.svg
/static/js/main.5d4c966b.js.map
/static/css/main.147b11bb.css.map
/manifest.json
/favicon.ico
/
/static/css/main.147b11bb.css
/static/js/main.5d4c966b.js
/static/css/main.147b11bb.css.map
/static/media/logo.5d5d9eef.svg
/static/js/main.5d4c966b.js.map
/manifest.json
/favicon.ico
/
/static/css/main.147b11bb.css
/static/media/logo.5d5d9eef.svg
/favicon.ico
/static/js/main.2255e3b7.js
/static/js/main.5d4c966b.js
^C
OMs-MacBook-Pro:main-app omsingh$ npm run serve

> main-app@0.1.0 serve /Users/omsingh/Downloads/CloudHedge-POC/cloudhedge-microfrontends-master/main-app
> node scripts/http-server.js build 3000

HTTP server listening on port 3000, exposing directory build.
/
/static/css/main.147b11bb.css
/static/js/main.5d4c966b.js
/static/media/logo.5d5d9eef.svg
^C
OMs-MacBook-Pro:main-app omsingh$ 
```



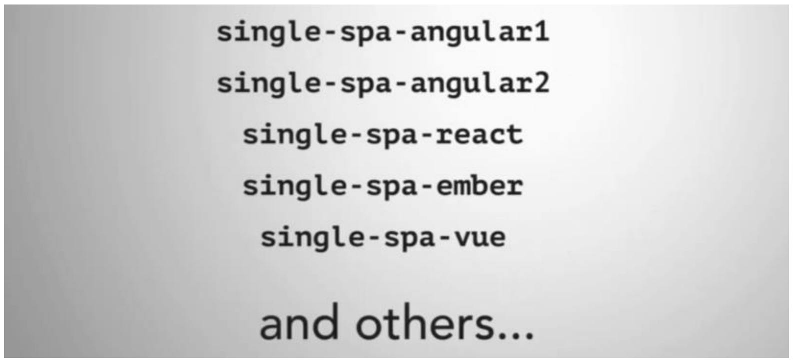
An application is a group of UI components registered to single-spa that controls part of the web page. These apps co-exist no matter what UI framework they are written in.



Once your code is registered as a single-spa app, you can use the common API made up of 3 main parts. The **bootstrap()** function for initialization, a **mount()** function that activates the application, and an **unmount()** function that deactivates the application. Your code does the 3 main things and single-spa does the rest.



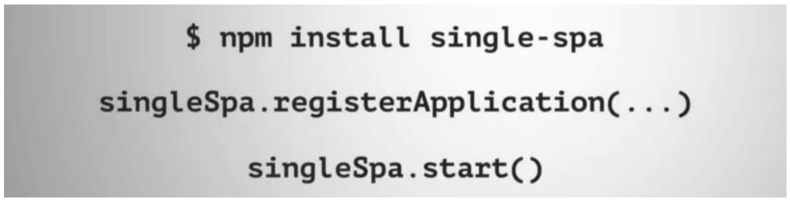
The Single-Spa JS library hooks into your client-sdie routing so that the correct applications are active at any time.



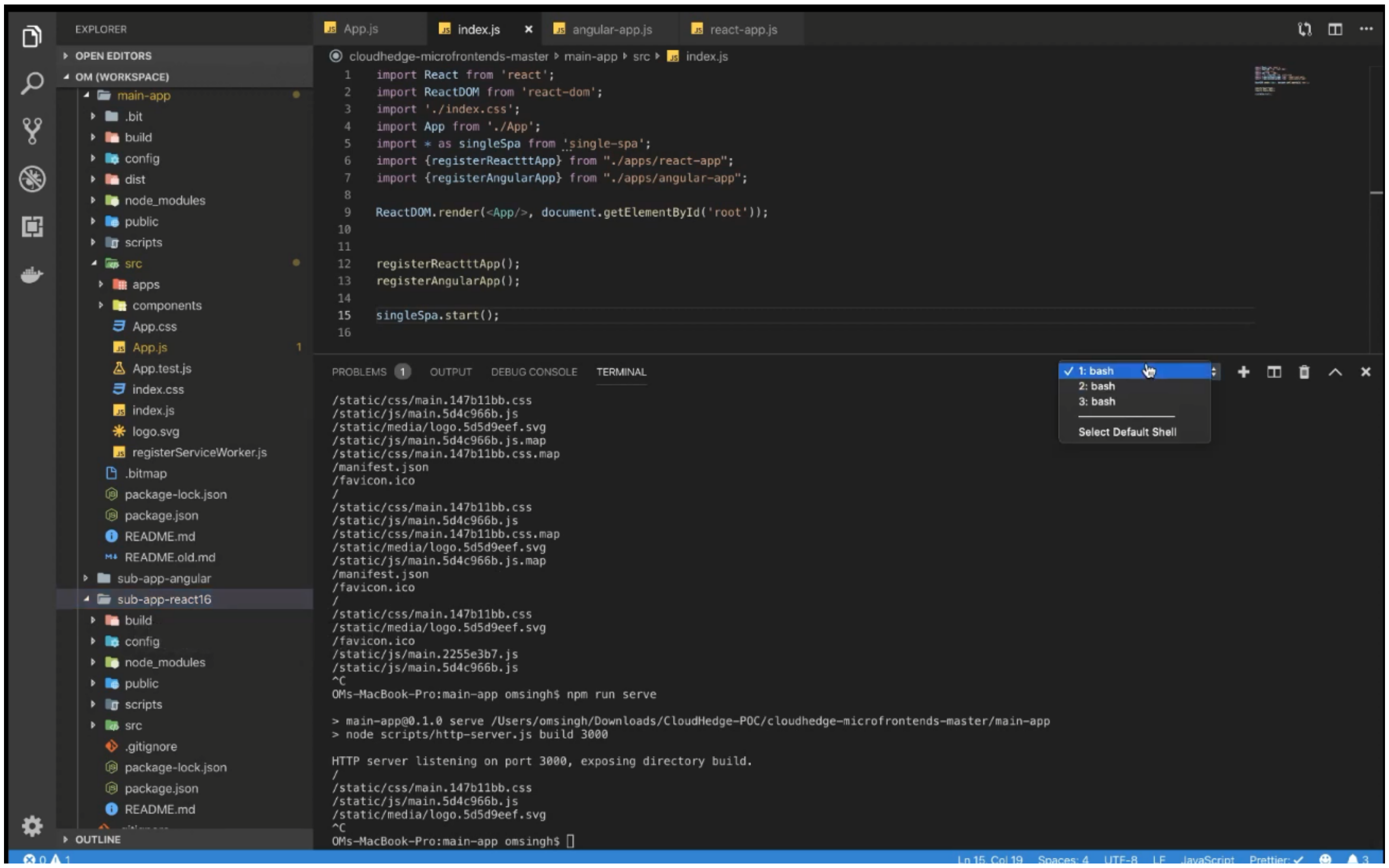
```
single-spa-angular1
single-spa-angular2
single-spa-react
single-spa-ember
single-spa-vue
```

and others...

Your code does not need to change, you simply wrap it with some helper libraries



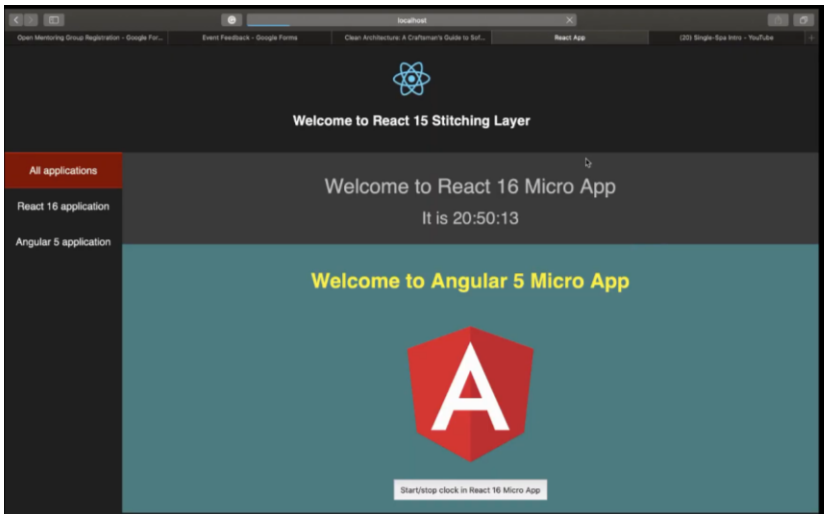
```
$ npm install single-spa
singleSpa.registerApplication(...)
singleSpa.start()
```

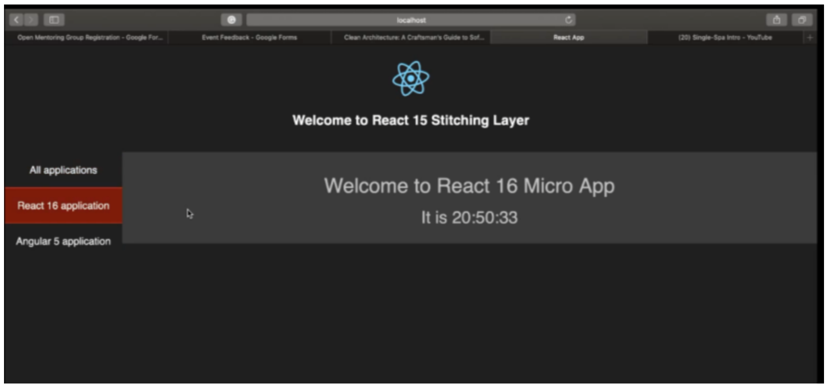


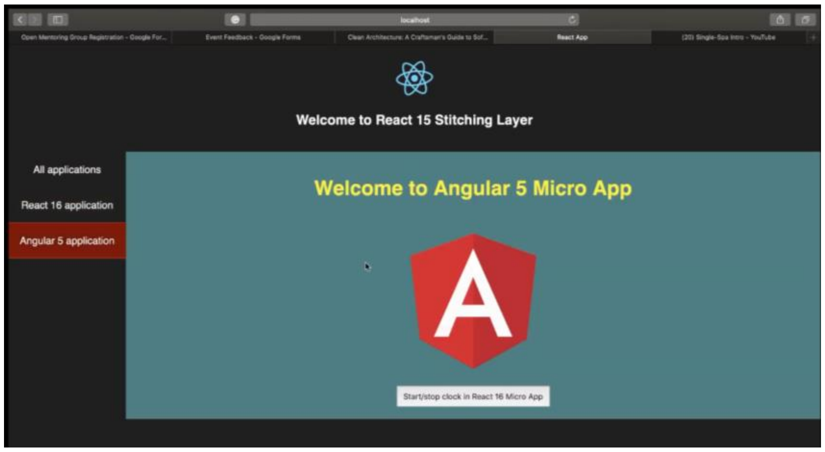
```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as singleSpa from 'single-spa';
import {registerReactttApp} from "./apps/react-app";
import {registerAngularApp} from "./apps/angular-app";

ReactDOM.render(<App/>, document.getElementById('root'));


registerReactttApp();
registerAngularApp();

singleSpa.start();

```

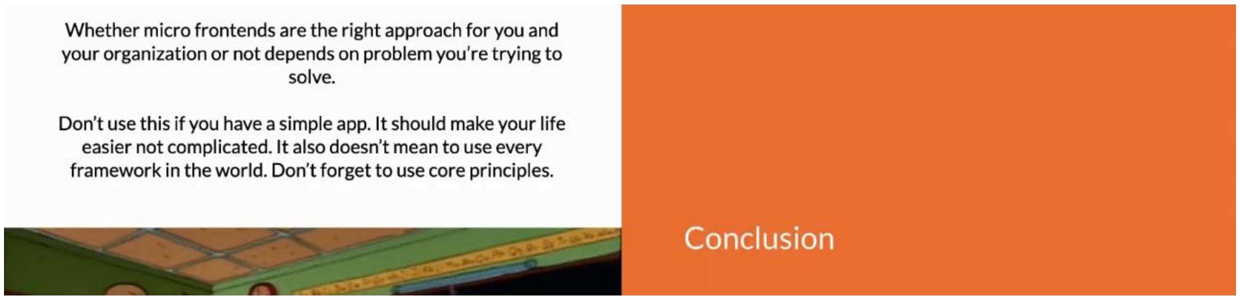
```
/static/css/main.147b11bb.css
/static/js/main.5d4c966b.js
/static/media/logo.5d5d9eef.svg
/static/js/main.5d4c966b.js.map
/static/css/main.147b11bb.css.map
/manifest.json
/favicon.ico
/
/static/css/main.147b11bb.css
/static/js/main.5d4c966b.js
/static/css/main.147b11bb.css.map
/static/media/logo.5d5d9eef.svg
/static/js/main.5d4c966b.js.map
/manifest.json
/favicon.ico
/
/static/css/main.147b11bb.css
/static/media/logo.5d5d9eef.svg
/favicon.ico
/static/js/main.2255e3b7.js
/static/js/main.5d4c966b.js
^C
OMs-MacBook-Pro:main-app omsingh$ npm run serve

> main-app@0.1.0 serve /Users/omsingh/Downloads/CloudHedge-POC/cloudhedge-microfrontends-master/main-app
> node scripts/http-server.js build 3000

HTTP server listening on port 3000, exposing directory build.
/
/static/css/main.147b11bb.css
/static/js/main.5d4c966b.js
/static/media/logo.5d5d9eef.svg
^C
OMs-MacBook-Pro:main-app omsingh$
```

There are 3 apps in this single-spa demo, the **main app** runs on **port 3000**, the **Angular sub-app** runs on **port 3001** and the **React sub-app** runs on **port 3002**. We have registered the 2 sub-apps and started the main app.


Welcome to React 15 Stitching Layer
All applications
React 16 application
Angular 5 application
Welcome to React 16 Micro App
It is 20:50:13
Welcome to Angular 5 Micro App
Start/stop clock in React 16 Micro App




localhost
Open Mentoring Group Registration - Google For...
Event Feedback - Google Forms
Clean Architecture: A Craftsman's Guide to Sof...
React App
(20) Single-Spa Intro - YouTube
Welcome to React 15 Stitching Layer
All applications
React 16 application
Angular 5 application
Welcome to Angular 5 Micro App
Start/stop clock in React 16 Micro App



Whether micro frontends are the right approach for you and your organization or not depends on problem you're trying to solve.

Don't use this if you have a simple app. It should make your life easier not complicated. It also doesn't mean to use every framework in the world. Don't forget to use core principles.

Conclusion