cognizant softvision

PROGRAMMERS' WEEK 2022

Micro-frontends

Sharing standard web components across multiple frontends

What we will be covering

What are micro-frontends?

Use cases

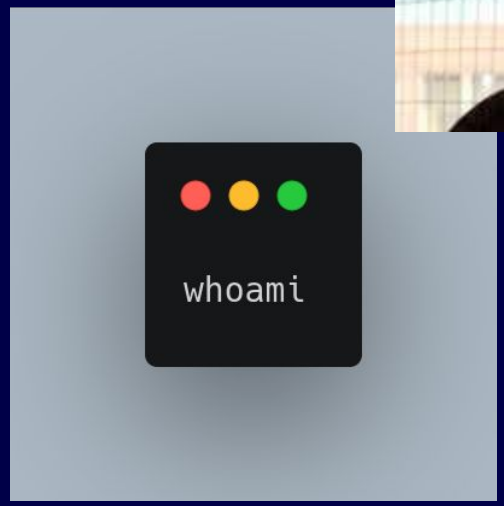
Why and how

Different known alternatives

Implementation

Live demo

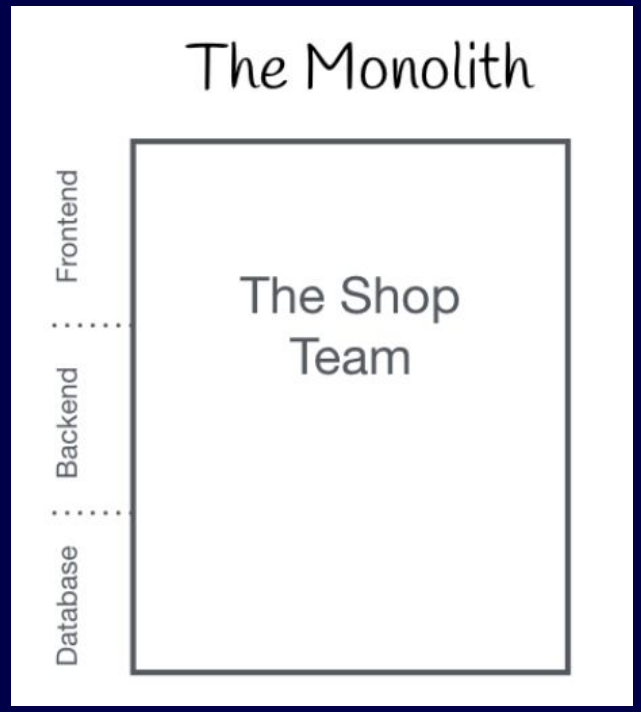
But first…

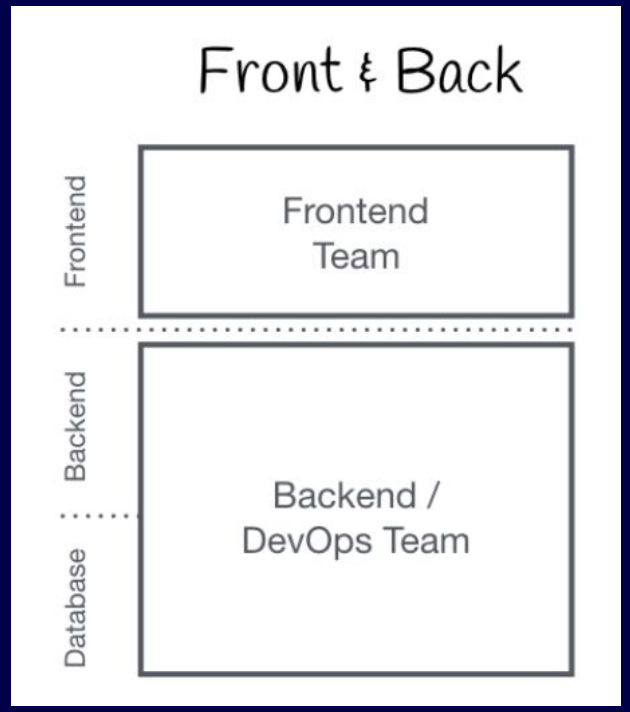


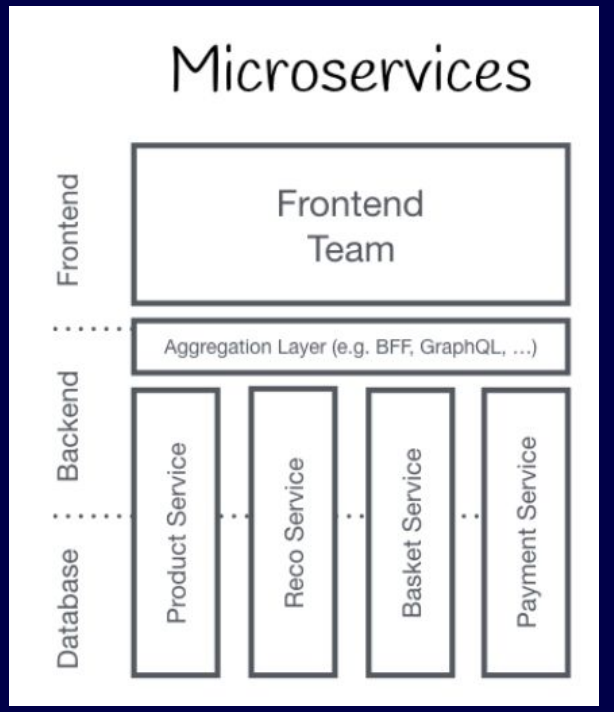
- Web developer, mid to Sr
- I choose linux and open source when I can
- Focused on the front end side of the stack
- Consider myself a generalist, but with experience working on vanilla, React and Angular
- (Yes, that includes JQuery)

Micro-frontends (the short explanation)

The old ways

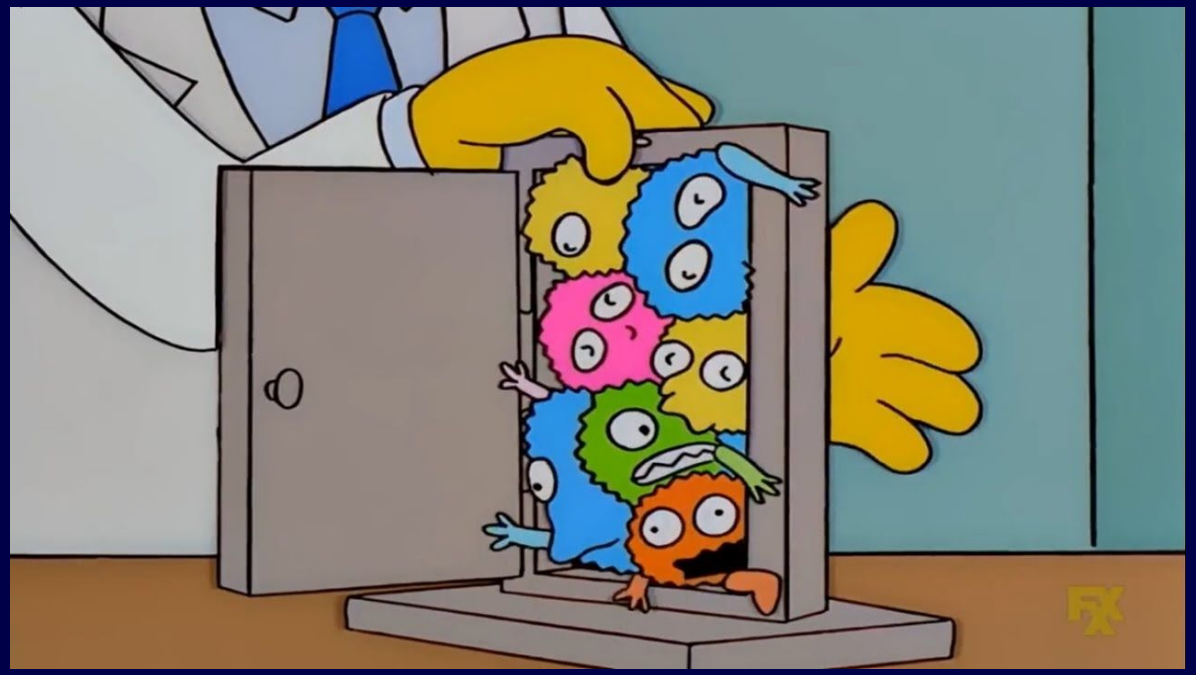






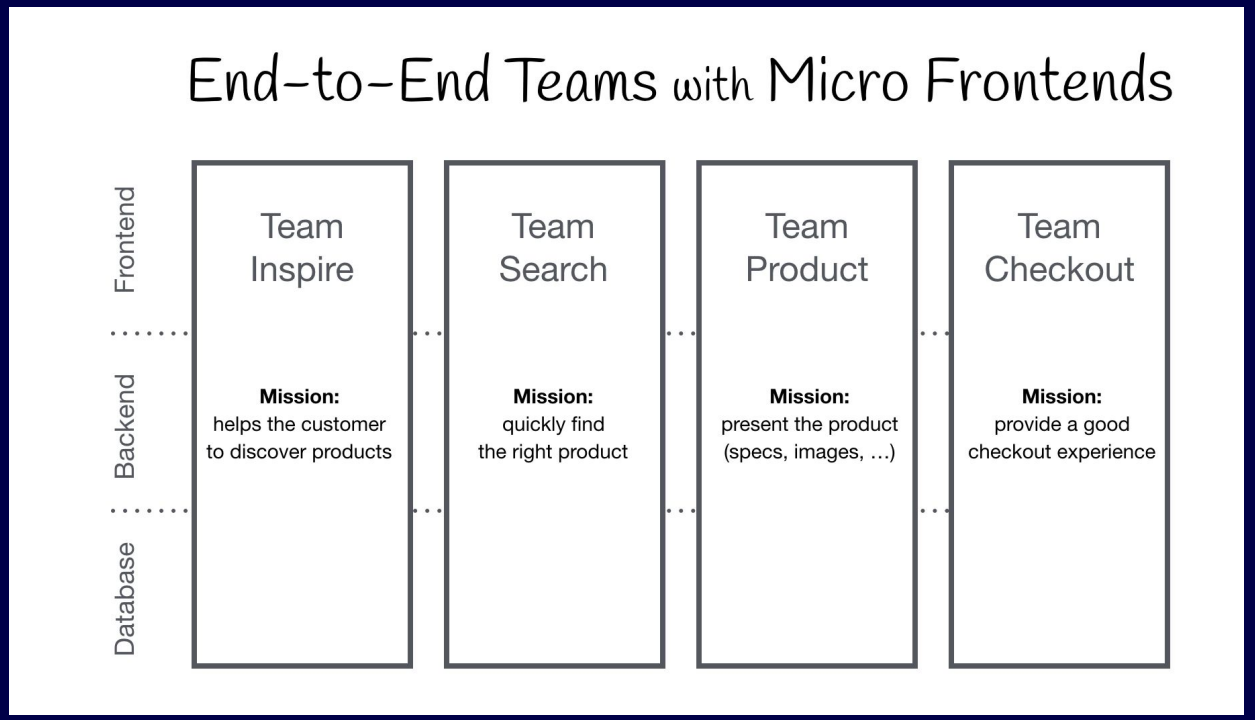
The problems with the Monolith

- Distributed teams
- Async work
- Big teams with different concerns
- Pipeline congestion
- Gargantuan build processes to fix a typo



BUT WAIT

A New Hope



cognizant softvision

Micro-frontends advantages

Divide and conquer

Smaller builds and build times

Independent deploys

Avoid problems with sharing a repo

Great, another technology to learn. Does somebody actually use is?

Well, actually…

…and many others!

Great, how can we do it?

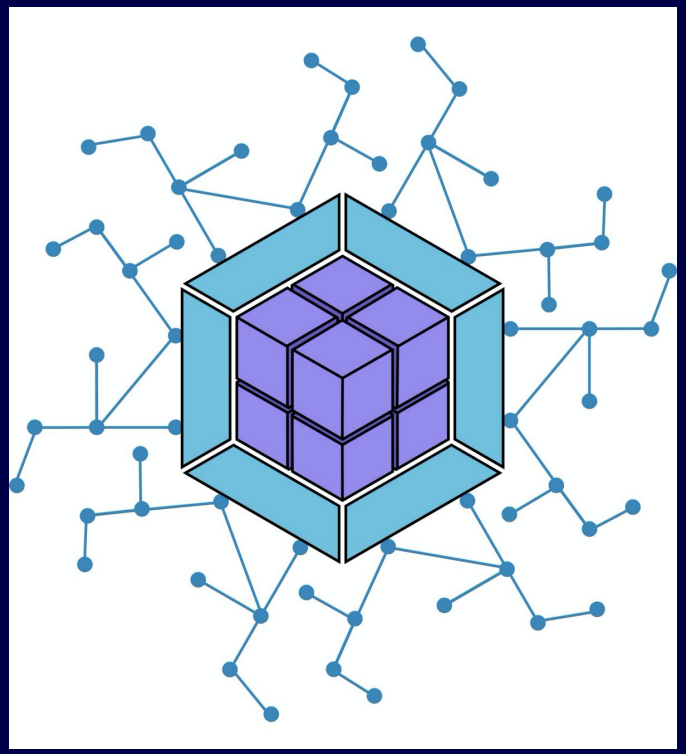
The runtime module approach

For example, with import maps:

```
{
  "imports": {
    "lit-html": "./node_modules/lit-html/lit-html.js",
    "lit-element": "./node_modules/lit-element/lit-element.js",
    "lit-html/lit-html.js": "./node_modules/lit-html/lit-html.js",
    "lit-html/lib/shady-render.js": "./node_modules/lit-html/lib/shady-render.js"
  }
}
```

And system.js: https://github.com/systemjs/systemjs

The build time module approach (Module Federation)



Which should I pick?

No clear winner

Adjust to your needs

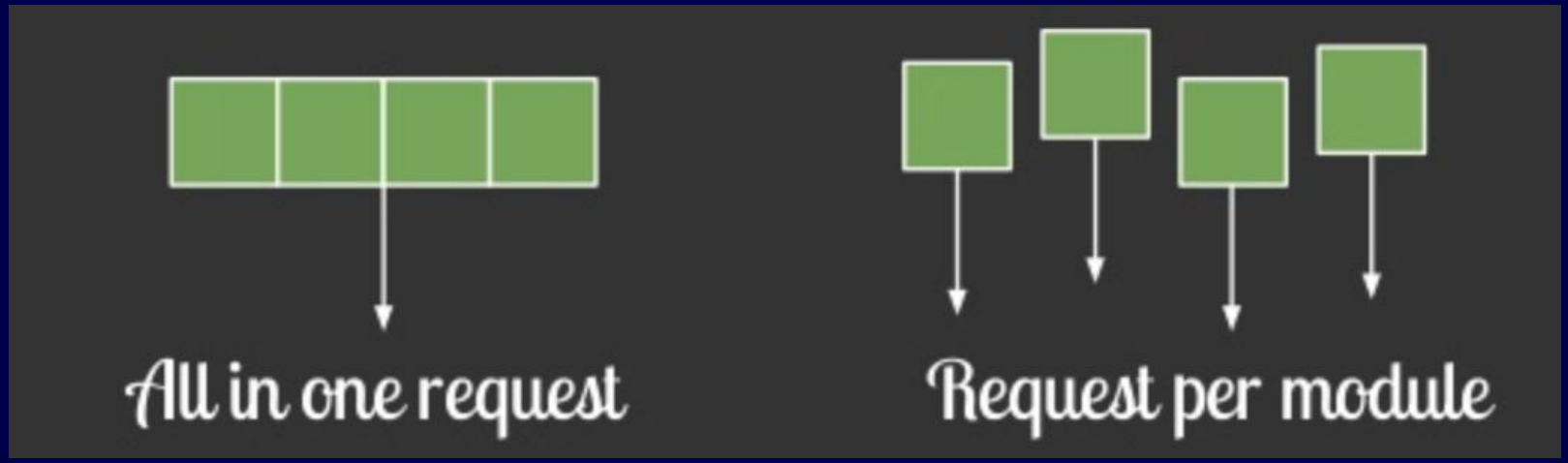
Beware of FOMO

First, code splitting… Why and how?

What it means?

Why?

Caching



But listen to uncle Ben on this one

Great, but what if I want to split code … and share it across the network?

Webpack Module Federation to the rescue!

Module Federation is code splitting across the network

What makes the network thing any different?

In code, that means…

We declare it like this:

```
remotes: {
  MyAwesomeRemote: `my_remote@http://location.com/fileName.js`,
},
```

And we import it like this:

```
import("MyAwesomeRemote/Exposed")
```

On the remote side…

We expose what we want to share like this:

```
{
  name: "my_remote",
  filename: "fileName.js",
  remotes: {},
  exposes: {
    "./Exposed": "path/to/file/Exposed.js",
},
```

The full config of the plugin should look something like this:

Host (consumer):

```
new ModuleFederationPlugin({
  name: "host_remote",
  filename: "filenameOfHost.js",
  remotes: {
    MyAwesomeRemote: `my_remote@$http://location.com/fileName.js`,
  },
  exposes: {},
  shared: {
    react: {
      singleton: true,
      eager: true,
      requiredVersion: deps["react"],
    },
  },
});
```

Remote:

```
new ModuleFederationPlugin({
  name: "my_remote",
  filename: "fileName.js",
  remotes: {},
  exposes: {
    "./Exposed": "path/to/file/Exposed.js",
  },
  shared: {
    react: {
      singleton: true,
      eager: true,
      requiredVersion: deps["react"],
    }
  },
});
```

Shared dependencies


```
shared: {
    react: {
        singleton: true,
        eager: true,
        requiredVersion: deps["react"],
    },
}
```

Bootstrapping the app

New entry file

```
import('./bootstrap' /* webpackChunkName: "bootstrap-app" */ );
```

Old entry file example, renamed to “bootstrap.js”

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { AppRoutes } from './App';
import { BrowserRouter } from 'react-router-dom';

import "./App.css";

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <AppRoutes/>
    </BrowserRouter>
  </React.StrictMode>
);
```

Does it really work?

Lazy loading


```jsx
import React, { Suspense, useContext} from "react";

import Footer from "components/footer/Footer";
import ErrorBoundary from "components/ErrorBoundary";
import ErrorComponent from  'components/errorComponent/ErrorComponent.jsx'
import { logout, UserContext } from "context/UserContext";

// REMOTE NAV BAR:
const TopBar = React.lazy(() => import("ReactComponents/Topbar"));

export const Layout = ({ children }) => {
  const [user, dispatch] = useContext(UserContext);

  return (
    <div className="" key="Layout">
      <ErrorBoundary errorComponent={<ErrorComponent />}>
          <Suspense fallback={<span>loading...</span>}>
            <TopBar name={user.name} logged={user.logged} logout={() => dispatch(logout())} />
          </Suspense>
        </ErrorBoundary>
      {children}
      <Footer />
    </div>
  );
};

export default Layout;
```

What can we share?

For starters, anything that is js, but there are common use cases:

-Sharing UI components

-Sharing apps that work as standalone inside a SPA

-Sharing common state across them

-Sharing assets

Module Federation seems pretty cool right?

So is there a way to prevent this?

So how to encapsulate?

We could use a library

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

…or we could leverage the power of the DOM

Standard Web Components & Shadow DOM as native encapsulation

One approach to this


```
const AppMFE = () => (
    <>
        <link rel="stylesheet" href={process.env.STYLES_URL} />
        <link rel="stylesheet" href={process.env.WC_DEMO_STYLE_URL} />
        <MemoryRouter>
            <AppRoutes />
        </MemoryRouter>
    </>

);


class StandaloneWebComponent extends HTMLElement {
    constructor() {
        super()
    }

    connectedCallback() {
        const mountPoint = document.createElement("section");

        this.attachShadow({ mode: "closed" }).appendChild(mountPoint);

        const root = createRoot(mountPoint);
        root.render(<AppMFE />);
    }
}

const init = () => {
    window.customElements.get("cwc-foundations") ||
window.customElements.define("cwc-foundations", StandaloneWebComponent);
    return Promise.resolve('cwc-foundations')
}


export default init();
```

On the host…


```
useEffect(() => {
  import("MyAwesomeRemote/Exposed")
    .then((module) => setExposedDefined(true))
    .catch((module) => setExposedDefined(false))
},[])
```

Adding styles to the Shadow Dom

Embedding strings to html tags

```
generateHTMLelementInnerHTML = (css, html) => {
  return `
    <style>
      ${css}
    </style>

    ${html}
  `
};


export function createTemplateForWebComponent({ html, css })
{
  const template = document.createElement("template");
  template.innerHTML = generateHTMLelementInnerHTML(css,
html);
  return template;
}
```

Linking to external stylesheets

```
const AppMFE = () => (
    <>
        <link rel="stylesheet" href="https://someorigin.com/styles.css" />
        <link rel="stylesheet" href="/app-styles.css" />
        <App />
    </>

);
```

So finally, a working example

Tips & tricks

.env file

```
PORT=3000
REMOTE=https://my.app/

REMOTE_TOPBAR=https://remote-topbar.app
REMOTE_STANDALONE_MFE=https://remote-wc-mfe.app
REMOTE_WEB_COMPONENTS=https://remote-wc-system.app

# for development, comment remote below to fetch from remote, uncomment to fetch from localhost
LOCAL_TOPBAR=http://localhost:3001
LOCAL_STANDALONE_MFE=http://localhost:3002
LOCAL_WEB_COMPONENTS=http://localhost:3003
```

On Webpack config

```
// Remote: `AwesomeRemote@http://localhost:3002/remoteEntry.js`,
ReactComponents: `topbar_remote@${LOCAL_TOPBAR || REMOTE_TOPBAR}/remoteEntry.js`,
WebComponents: `wc_system@${LOCAL_WEB_COMPONENTS || REMOTE_WEB_COMPONENTS}/remoteEntry.js`,
StandaloneMFE: `WCDemo@${LOCAL_STANDALONE_MFE || REMOTE_STANDALONE_MFE}/remoteEntry.js`
```

Final thoughts

You may not need it

This is not a swiss army knife

Try before you buy

You should give it a chance, if you can

“If the only tool you have is a hammer, it is tempting to treat everything as if it were a nail”

Abraham Maslow

Repository & resources:

https://github.com/cognizant-softvision/pw2022-mfe-wc

cognizant softvision

PROGRAMMERS´
WEEK 2022

Thanks