Platform freedom with Micro-frontends

Saravana Balaji Srinivasan

Senior Software Engineer @Red Hat

Full-stack Developer

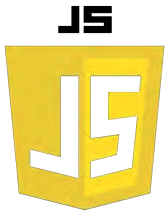
Red Hat Story

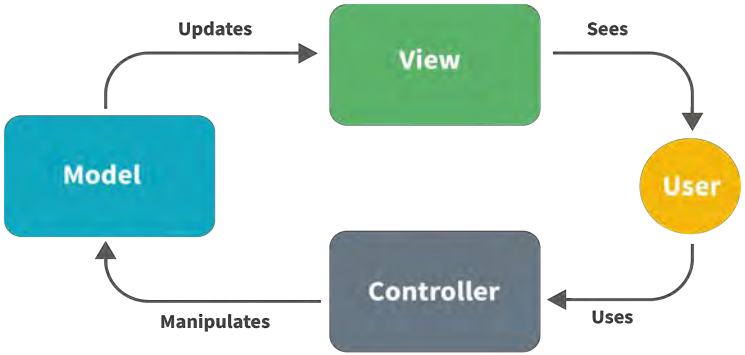
Landscape

We live in a world where **web technologies** have dominated software development.

Default choice for most applications.

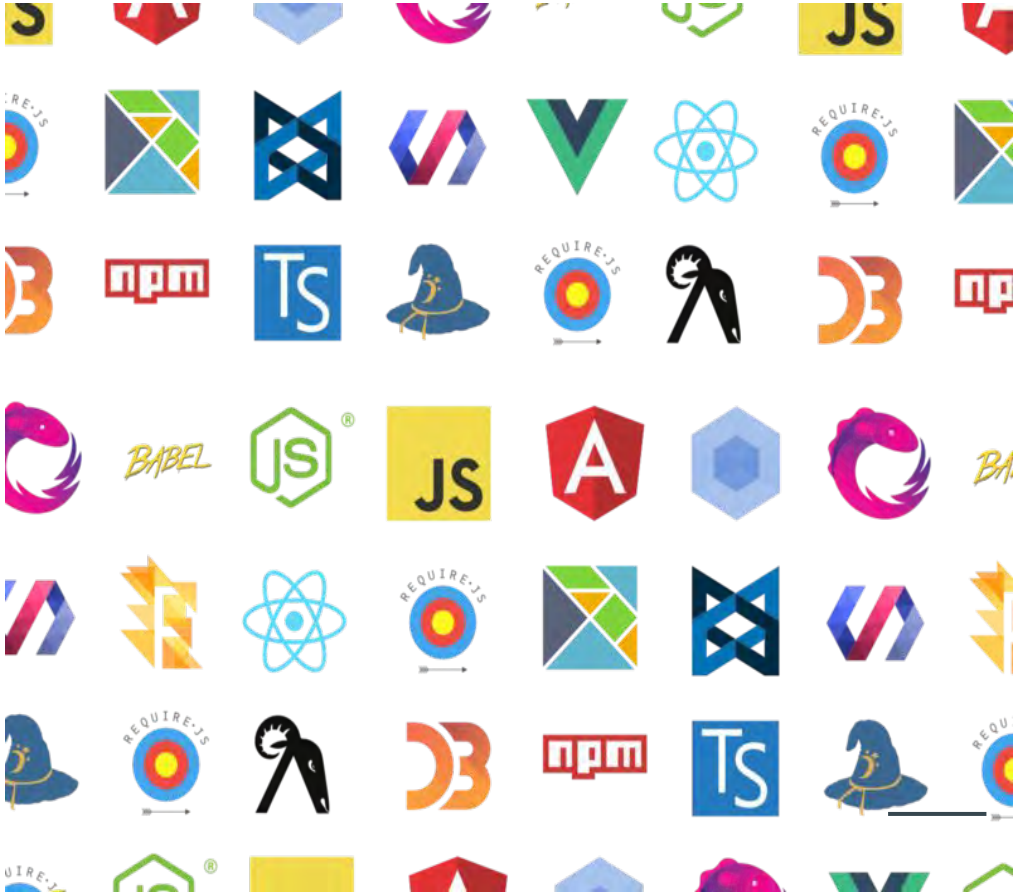
Standards, Patterns and Techniques





Foundation

Well though and understood set of Standards, Patterns and Techniques as a strong foundation.



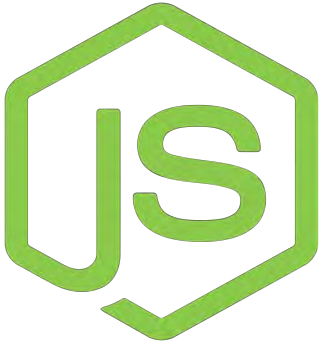
Ecosystem

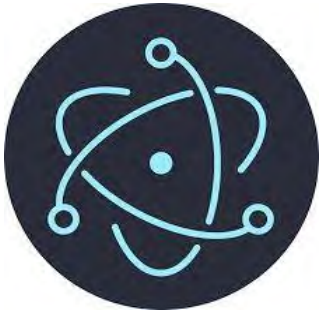
Rich ecosystem that has been maturing over the years

TypeScript

Static Typed Language

TypeScript created the perfect compromise for Static Type Languages believers.



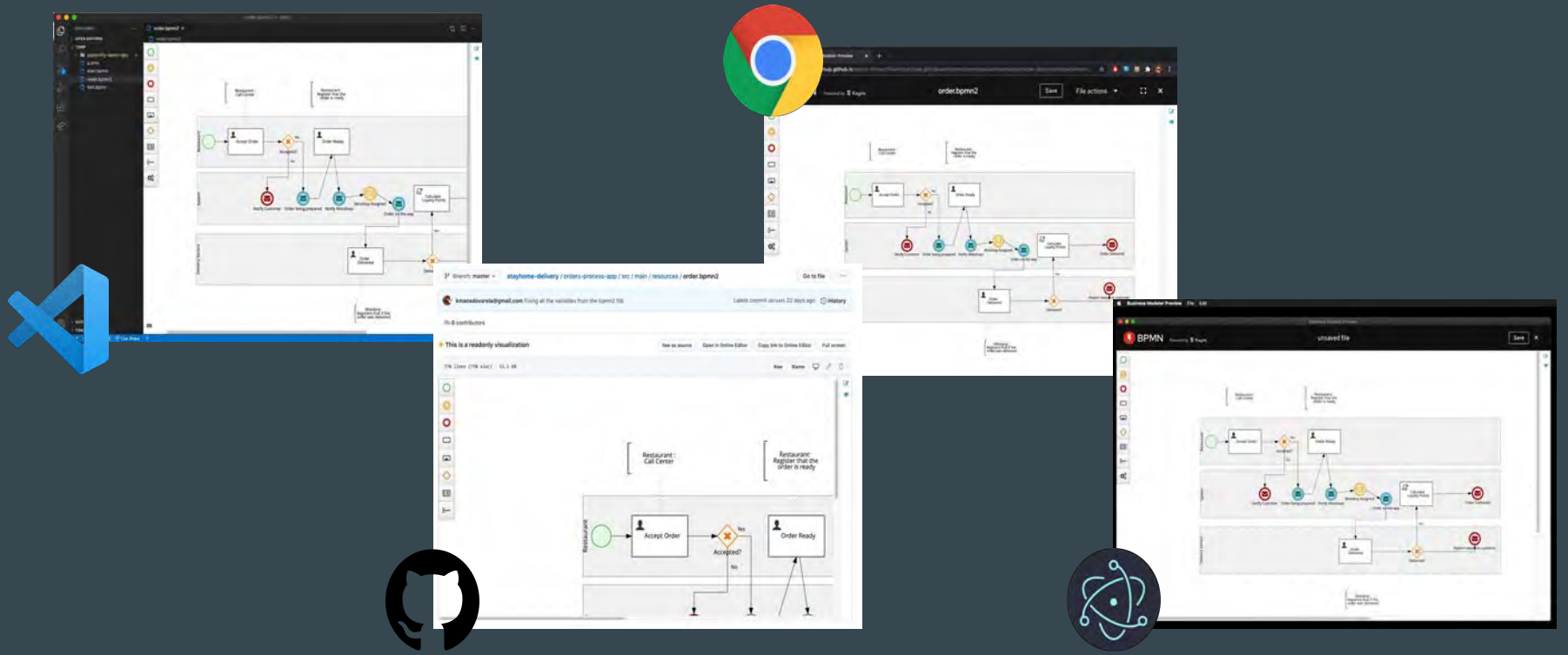


Browser Everywhere

Browser is now more than just the window for the internet.

Browsers became part of an important trend as the mechanism to distribute any Graphical User Interface based applications.

Multiple Distributions



Architectures

Evolutionary

An evolutionary architecture supports incremental, guided change as a first principle across multiple dimensions.

Microservices

Architectural style that structures an application as a collection of independent services.

Serverless

Incorporate third-party “Backend as a Service”, and/or that include custom code run as Functions.

Micro Frontends

Design approach in which a front-end app is decomposed into individual, semi-independent “microapps” working loosely together.

Micro-frontends

Micro-frontends

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

Cam Jackson

https://martinfowler.com/articles/micro-frontends.html

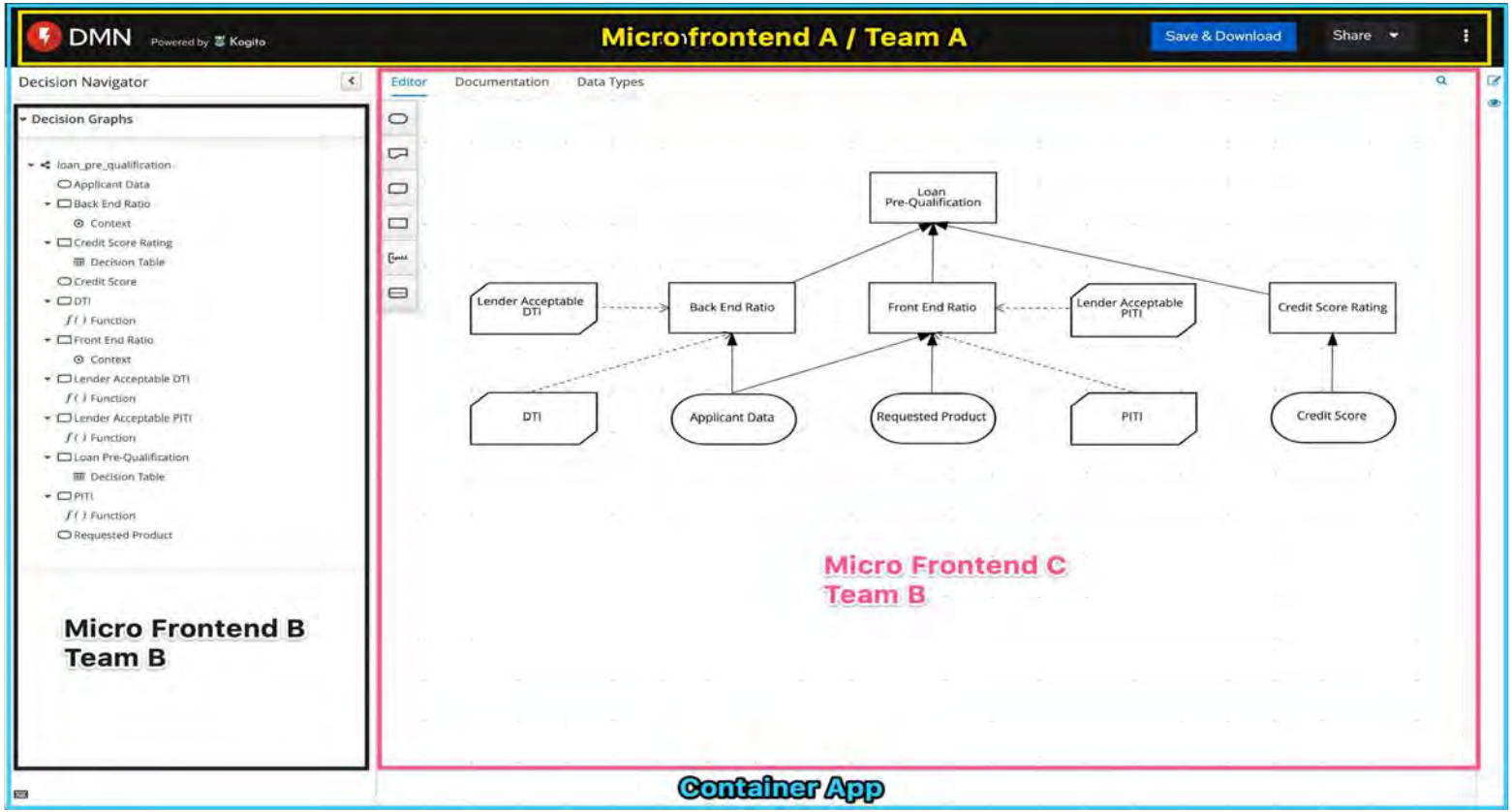
GOAL OF MICRO-FRONTENDS ?


Don't
PUT ALL
YOUR EGGS
IN ONE BASKET

Diversification

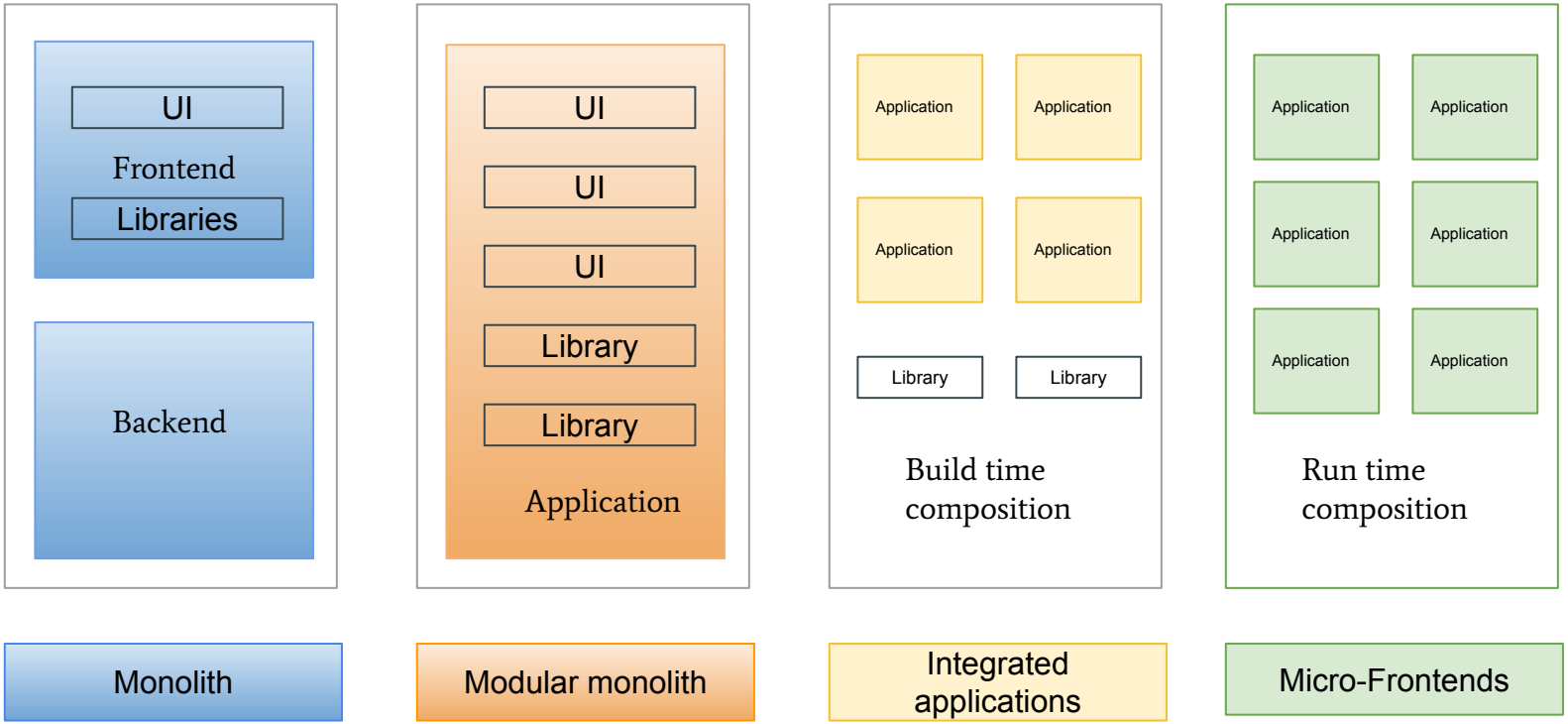
It is not only good for money but also for technology

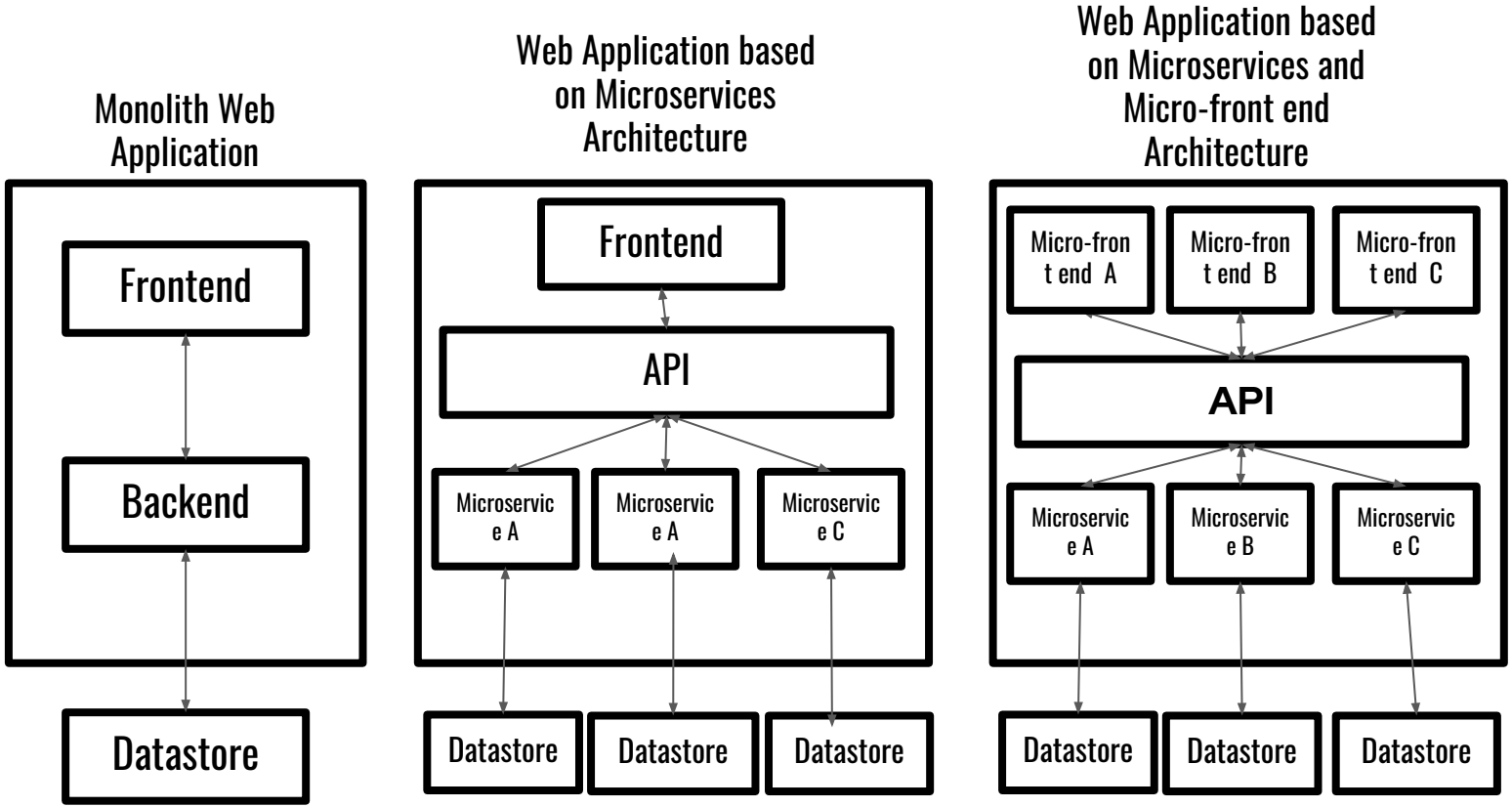
Example

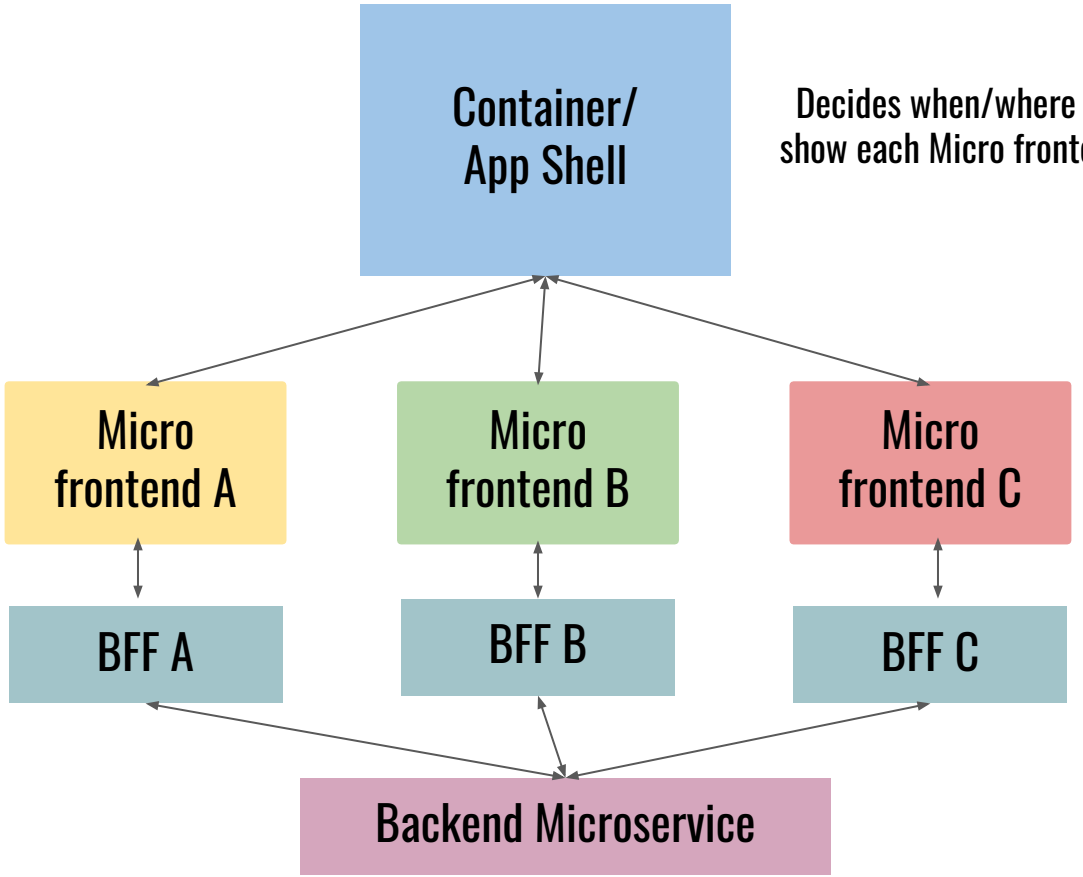


* DMN is a modeling language and notation for the precise specification of business decisions and business rules.

Micro-frontends




Monolith Web Application
Frontend
Backend
Datastore
Web Application based on Microservices Architecture
Frontend
API
Microservic e A
Microservic e A
Microservic e C
Datastore
Datastore
Datastore
Web Application based on Microservices and Micro-front end Architecture
Micro-fron t end A
Micro-fron t end B
Micro-fron t end C
API
Microservic e A
Microservic e B
Microservic e C
Datastore
Datastore
Datastore



Decides when/where to show each Micro frontend

No Micro frontend communicate directly to each other

BFF ensures seamless user interaction regardless of the platform the frontend application is running on

Types of Integration

Run-Time integration

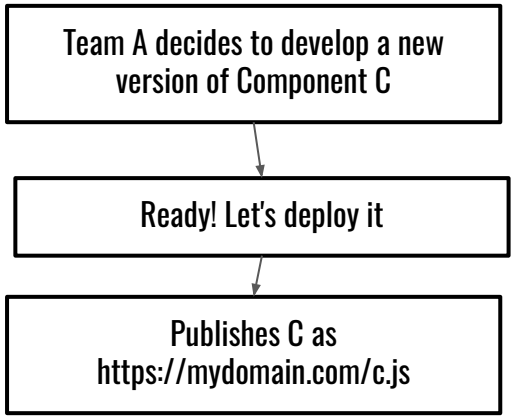
aka client-side integration:

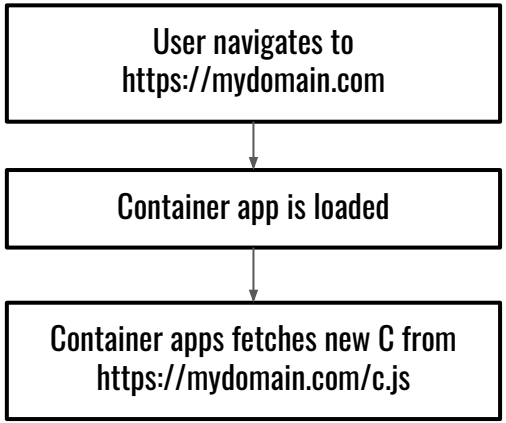
After the container gets loaded in the browser, it gets access to micro front end source code

✅ A can be deployed independently at any time and can deploy different versions of it, and Container can decide which one to use

❌ tooling + setup is far more complicated

Independent deployment makes it challenging to test/verify (build a good test suite for it)





Types of Integration

Build-time integration

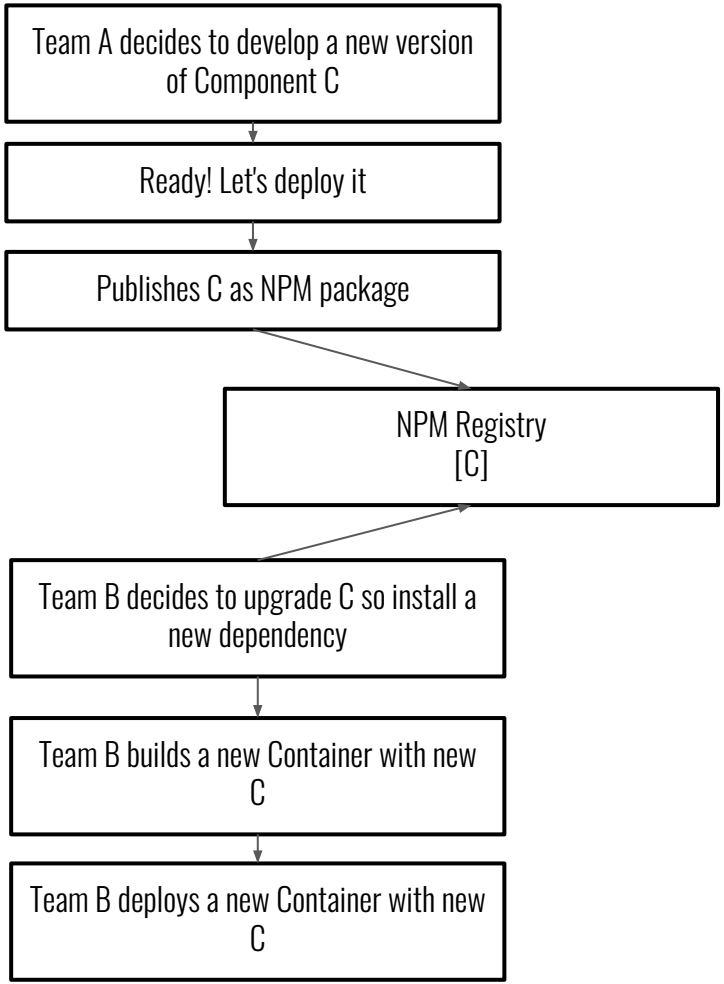
aka compile-time integration:

Before the container gets loaded in the browser, it gets access to micro frontend source code;

Foreign modules are accessible during build

✅ Easy to setup and understand

❌ Container has to be re-deployed every time child has updated and tempting to tightly coupled Container + child together



Another concerns - Styling

What you should do:

- *Custom CSS from your project:*
  - Use CSS-in-JS library
  - Use frameworks built-in component style scoping
    - Vue's and Angular has good ones
  - "Namespace" all your CSS
- *CSS coming from other libraries*
  - Use a component library that does css-in-js
  - Manually build the css library and apply namespacing techniques to it
  - Scope-it
    - Shadow DOM or iframes!

Micro-frontends Concerns

index.html

Raw

```html
<html>
  <head>
    <title>Feed me!</title>
  </head>
  <body>
    <h1>Welcome to Feed me!</h1>

    <iframe id="micro-frontend-container"></iframe>

    <script type="text/javascript">
      const microFrontendsByRoute = {
        '/': 'https://browse.example.com/index.html',
        '/order-food': 'https://order.example.com/index.html',
        '/user-profile': 'https://profile.example.com/index.html',
      };

      const iframe = document.getElementById('micro-frontend-container');
      iframe.src = microFrontendsByRoute[window.location.pathname];
    </script>
  </body>
</html>
```

Context Isolation via iframes

- Nothing new, exciting, even a bit of 'yuck"

- **Pros**
  - ✅ Great degree of isolation;
    - Styling
    - Global variables
    - Shadow DOM was not a option in 2019
  - ✅ Some libraries play directly with body of the page
  - ✅ We only use it when necessary

- **Cons**:
  - ❌ Makes your app feel 'old'
  - ❌ Less flexible than other options
  - ❌ Hard to integrate routing, history;
  - ❌ Challenging to make the app responsive
  - ❌ Not Content-Security-Policy friendly
  - ❌ Harder to make apps communicate

Why do we need a **new** architecture?

Cloud Native Tooling

requirements

Multiple Distributions

The origin of multiplying architecture is rooted in the need to distribute the same set of components in a myriad of platforms.

Minimize code changes

The components to be distributed should be preserved untouched and with avoiding feature flags.

Bridge

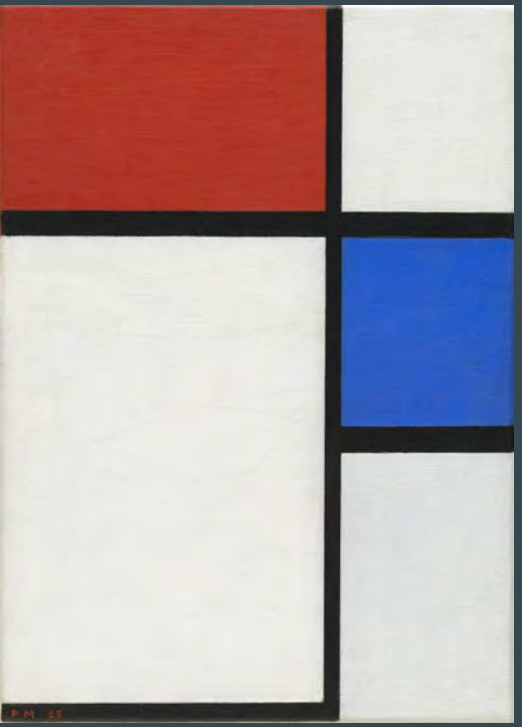
It has to embrace different generations of technology stack.

Introducing
Multiplying Architecture

What is Software Architecture?

“Architecture is about the important stuff. Whatever that is.”

Ralph Johnson



What is important for the Multiplying Architecture is the ***abstraction***.

The Abstractions

core

Channel

Top level abstraction that represents the hosting environment, like a website or a desktop application.

Envelope

Enable transparent communication between Components (View/Editor) and Channel

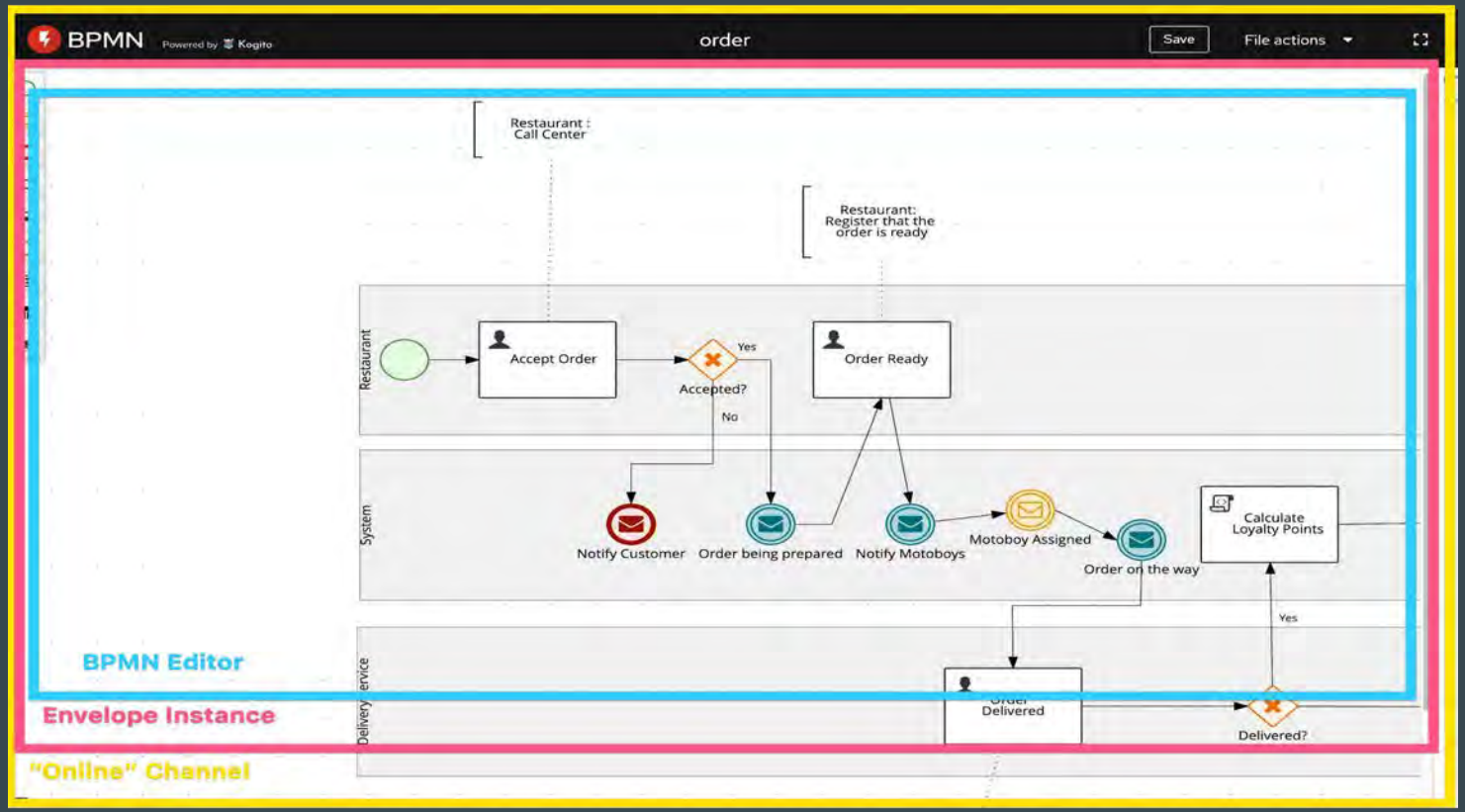
View

View is a portable set of widgets that are exposed as an unit to the Channel through the Envelope.

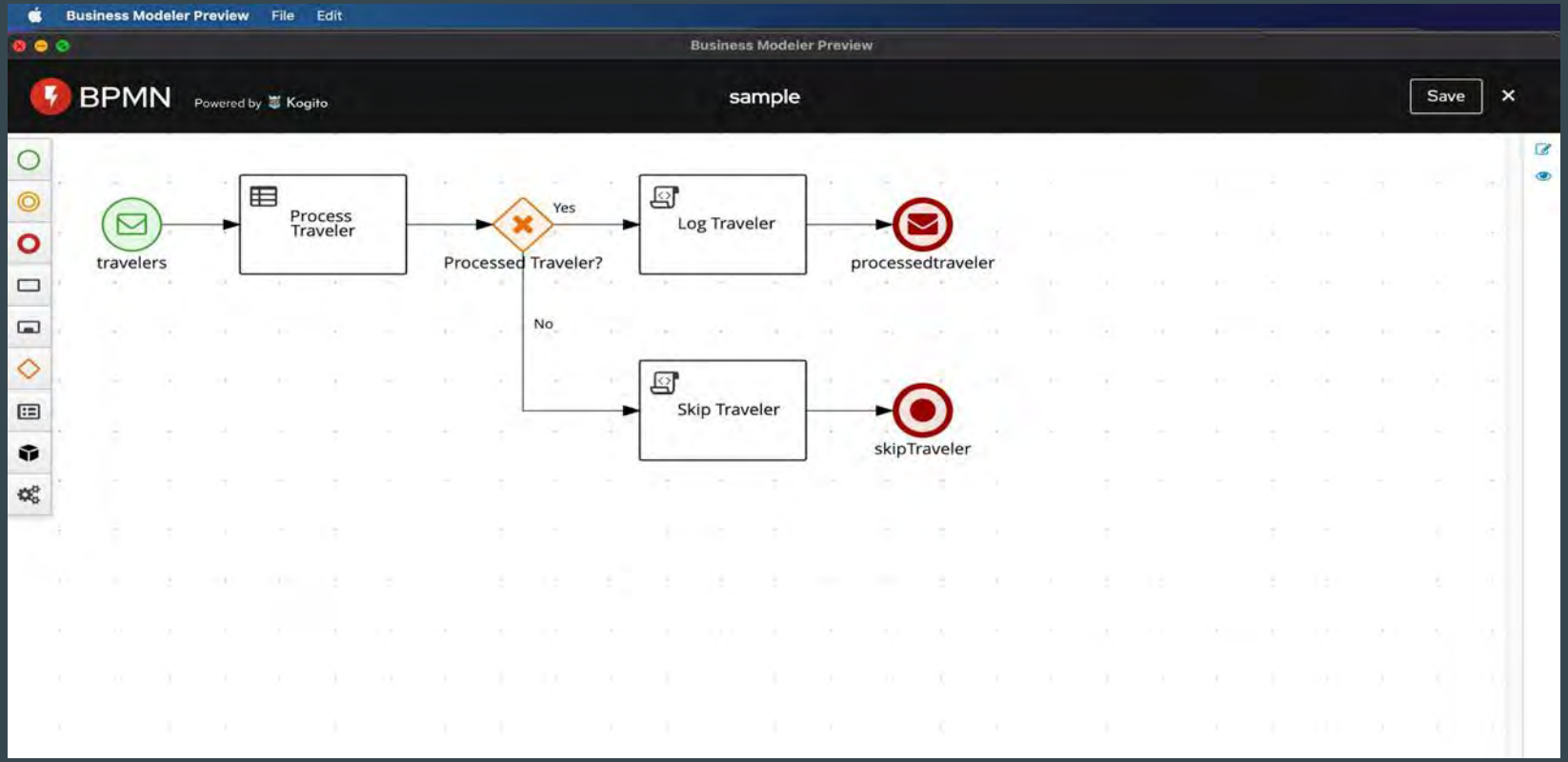
Editor

Editor is a specialized type of View, that gets a file content as input and is able to serve the content state back to the Channel through the Envelope.

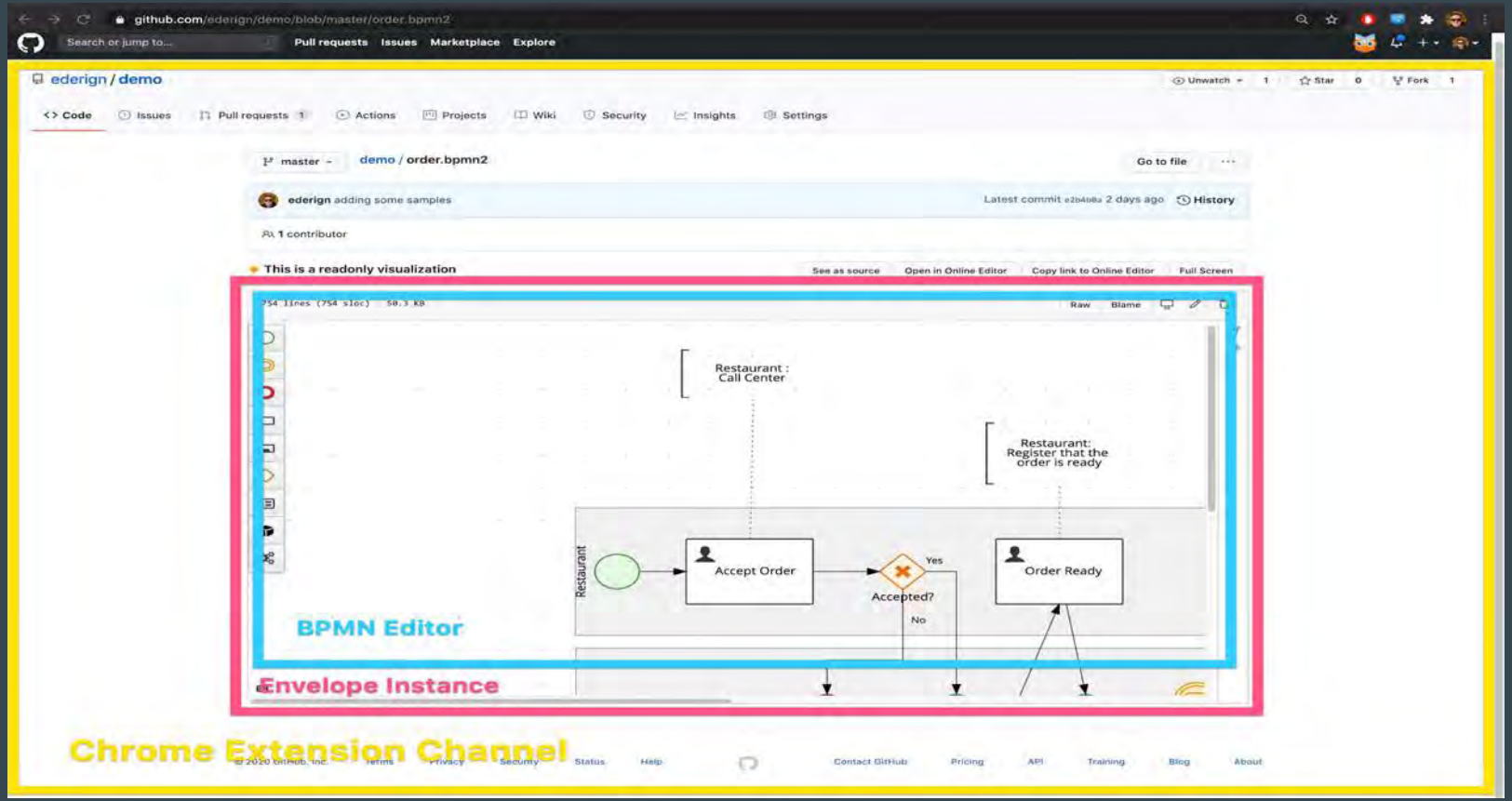
Online Channel



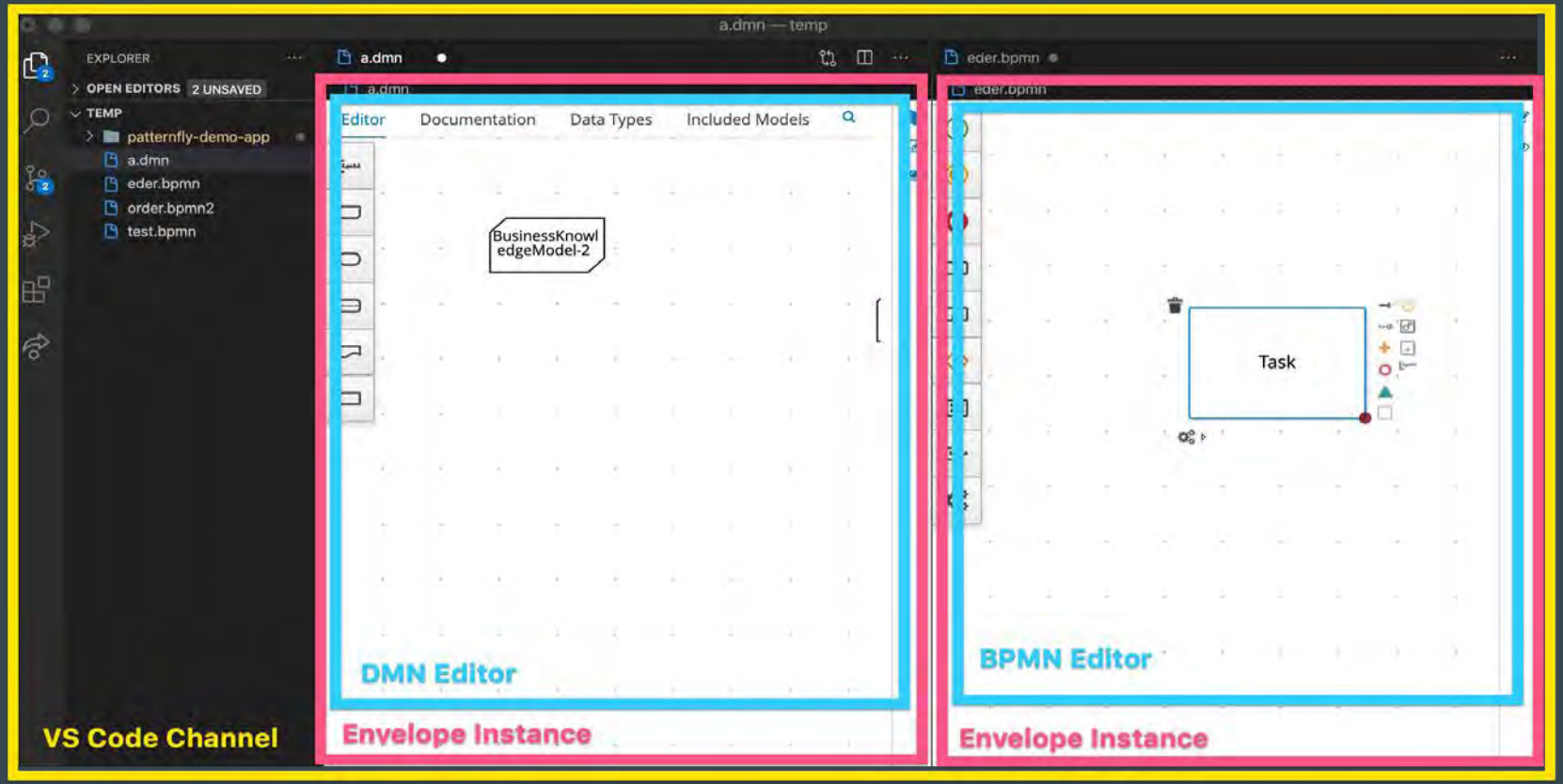
Browser extension Channel

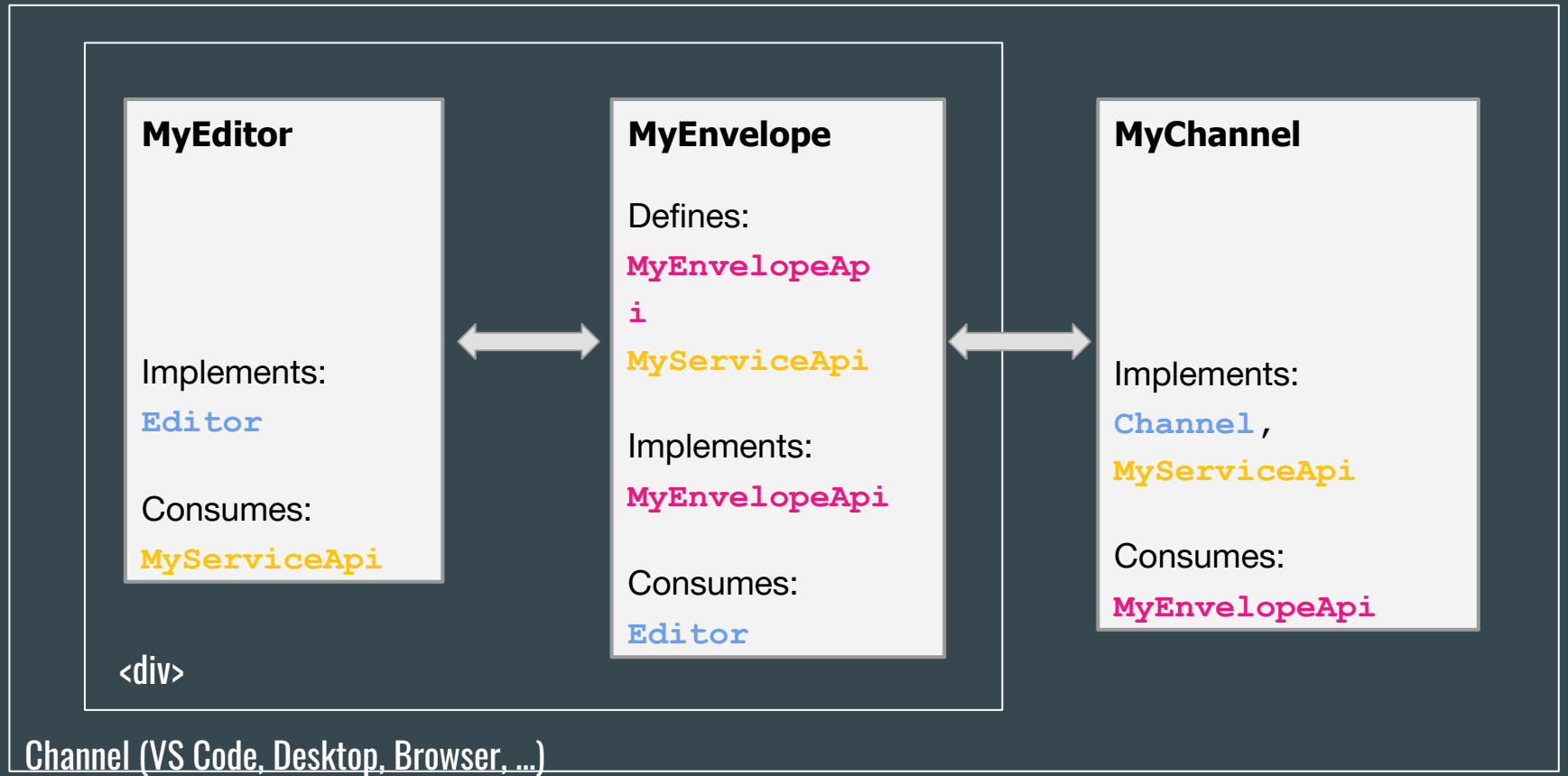


Github extension Channel



VSCode extension Channel




MyEditor
Implements:
Editor
Consumes:
MyServiceApi
MyEnvelope
Defines:
MyEnvelopeApi
MyServiceApi
Implements:
MyEnvelopeApi
Consumes:
Editor
MyChannel
Implements:
Channel,
MyServiceApi
Consumes:
MyEnvelopeApi

Channel (VS Code, Desktop, Browser, ...)

Envelope Advantages

Micro-frontend

Context Isolation (CSS and JS)

Autonomous Teams

Independent Release Cycles

Type Safe Communication

Multiplying Architecture in Practice

Built time x Runtime Integration

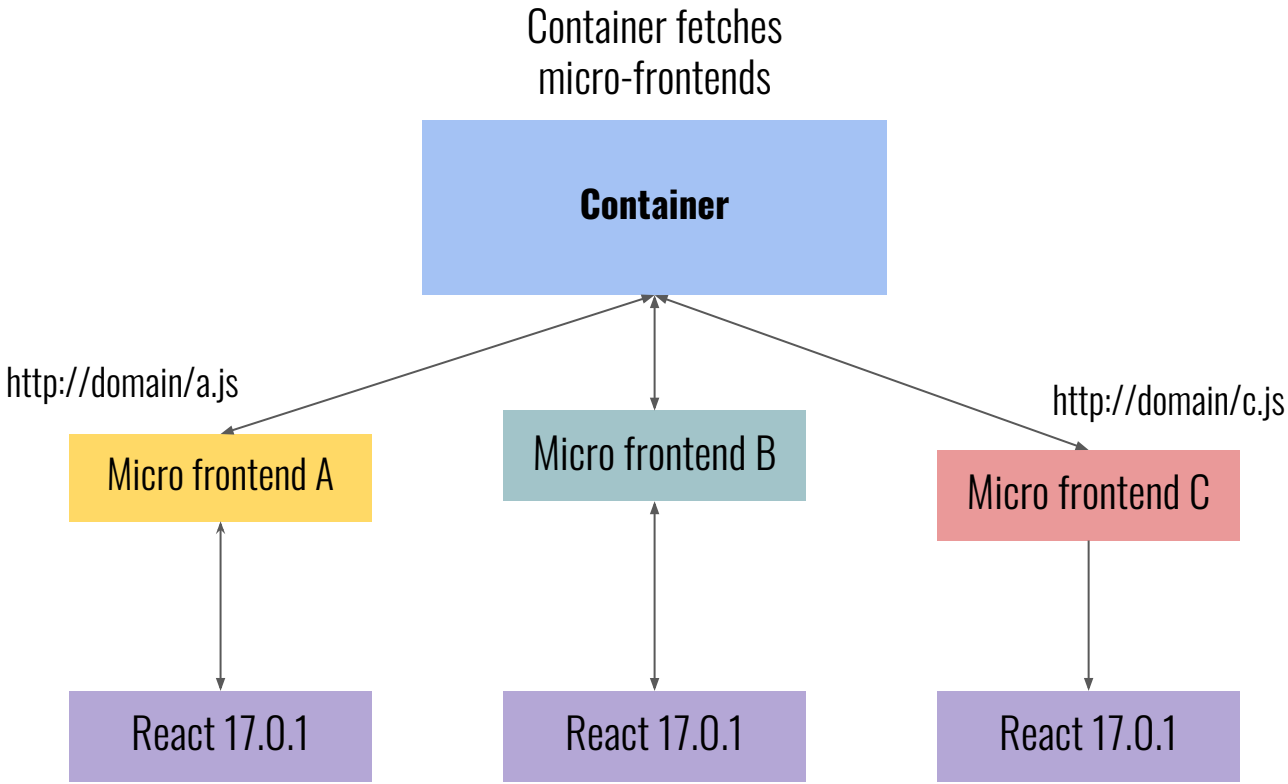
Build-time issues

- Foreign modules are accessible during build
- Container has to be re-deployed every time child has updated and tempting to tightly coupled Container + child together;
- One single change to prod. requires full a long rebuild
- Dependency versions alignment
- No clear app/team isolation
- **Duplication of library loading**

Runtime-time issues with iframes

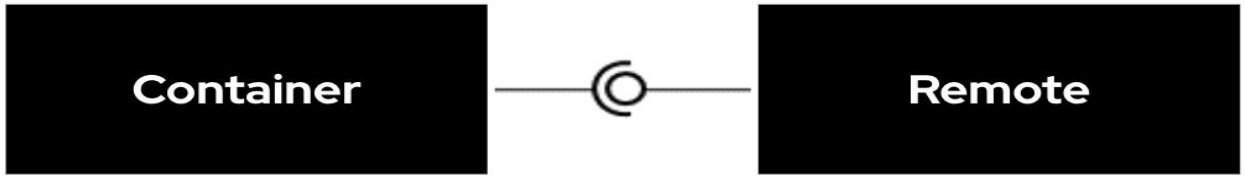
- Iframe doesn't help me share libraries/dependencies
- **Duplication of library loading**

Integration Issues



Federated Modules to the Rescue!

- Part of Webpack 5
- Allows loading separately compiled programs parts
- Solution for runtime integration of Micro frontends
- Allow referencing program parts that are not yet known at compile time.
- Each micro frontend can run in isolation



Federated Modules

```
const { merge } = require("webpack-merge");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");
const commonConfig = require("./webpack.common");
const packageJson = require("../package.json");

const domain = process.env.PRODUCTION_DOMAIN;

const prodConfig = {
  mode: "production",
  output: {
    filename: "[name].[contenthash].js",
    publicPath: "/container/latest/",
  },
  plugins: [
    new ModuleFederationPlugin({
      name: "container",
      remotes: {
        marketing: `marketing@${domain}/marketing/latest/remoteEntry.js`,
        auth: `auth@${domain}/auth/latest/remoteEntry.js`,
        dashboard: `dashboard@${domain}/dashboard/latest/remoteEntry.js`,
      },
      shared: packageJson.dependencies,
    }),
  ],
};

module.exports = merge(commonConfig, prodConfig);

```

-> ModuleFederation Plugin On webpack config

-> Remote Routes

-> shared dependencies

Federated Modules

App.js

```
import React, { lazy, Suspense, useState, useEffect } from "react";
import { Router, Route, Switch, Redirect } from "react-router-dom";
...
const MarketingLazy = lazy(() => import("./components/MarketingApp"));
const AuthLazy = lazy(() => import("./components/AuthApp"));
const DashboardLazy = lazy(() => import("./components/DashboardApp"));
...

  return (
    <Router history={history}>
      <StylesProvider generateClassName={generateClassName}>
        <div>
          <Header
            isSignedin={isSignedIn}
            onSignOut={() => setIsSignedIn(false)}
          />
          <Suspense fallback={<Progress />}>
            <Switch>
              <Route path="/auth">
                <AuthLazy onSignIn={() => setIsSignedIn(true)} />
              </Route>
              <Route path="/dashboard">
                {!isSignedIn && <Redirect to="/" />}
                <DashboardLazy />
              </Route>
              <Route path="/" component={MarketingLazy} />
            </Switch>
          </Suspense>
        </div>
      </StylesProvider>
    </Router>
  );
};
```

-> Import of Federated Modules

-> Lazy loading via Route

\+

Suspense

Goals of Multiplying Architecture

Solve a problem

Multiple Distributions

The origin of multiplying architecture is rooted in the need to distribute the same set of components in a myriad of platforms.

Minimize code changes

The components to be distributed should be preserved untouched and with avoiding feature flags.

Bridge

It has to embrace different generations of technology stack.

We Achieved Platform with Multiplying Architecture

- Microservices architecture
- Able to finally take advantage of runtime integration
- Each team can build/deploy their own micro frontend
- No duplication of library loading
- Ability to deploy multiple pieces of your application to different servers without iframes
- Have a portion of an application getting too big and wants a dedicated team? Split it out.
- That split you just made was a bad idea? Merge it back together.
- Finally we are able to real decoupling.
- Be able to evolve tech stack independently

Code sample -
https://github.com/kiegroup/kie-tools/tree/main/examples

Questions

Thank you

Saravana Balaji Srinivasan