Micro Frontends

Good frontend development is hard. Scaling frontend development so that many teams can work simultaneously on a large and complex product is even harder. In this article we'll describe a recent trend of breaking up frontend monoliths into many smaller, more manageable pieces, and how this architecture can increase the effectiveness and efficiency of teams working on frontend code. As well as talking about the various benefits and costs, we'll cover some of the implementation options that are available, and we'll dive deep into a full example application that demonstrates the technique.

19 June 2019

Cam Jackson

Cam Jackson is a full-stack web developer and consultant at ThoughtWorks, with a particular interest in how large organisations scale their frontend development process and practices. He has worked with clients across multiple industries and countries, helping them to deliver web applications more efficiently and effectively.

APPLICATION ARCHITECTURE

FRONT-END

MICROSERVICES

In recent years, microservices have exploded in popularity, with many organisations using this architectural style to avoid the limitations of large, monolithic backends. While much has been written about this style of building server-side software, many companies continue to struggle with monolithic frontend codebases.

Perhaps you want to build a progressive or responsive web application, but can't find an easy place to start integrating these features into the existing code. Perhaps you want to start using new JavaScript language features (or one of the myriad languages that can compile to JavaScript), but you can't fit the necessary build tools into your existing build process. Or maybe you just want to scale your development so that multiple teams can work on a single product

simultaneously, but the coupling and complexity in the existing monolith means that everyone is stepping on each other's toes. These are all real problems that can all negatively affect your ability to efficiently deliver high quality experiences to your customers.

Lately we are seeing more and more attention being paid to the overall architecture and organisational structures that are necessary for complex, modern web development. In particular, we're seeing patterns emerge for decomposing frontend monoliths into smaller, simpler chunks that can be developed, tested and deployed independently, while still appearing to customers as a single cohesive product. We call this technique **micro frontends**, which we define as:

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

In the November 2016 issue of the ThoughtWorks technology radar, we listed micro frontends as a technique that organisations should Assess. We later promoted it into Trial, and finally into Adopt, which means that we see it as a proven approach that you should be using when it makes sense to do so.

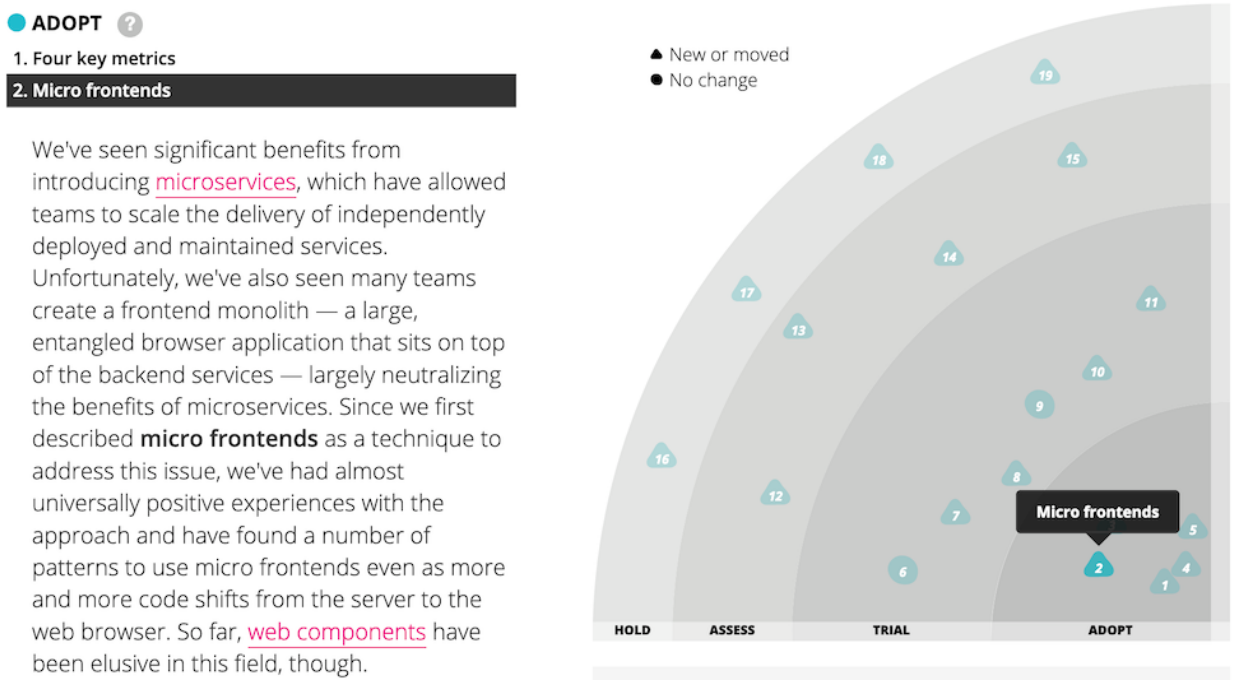

Figure 1: Micro frontends has appeared on the tech radar several times.

Some of the key benefits that we've seen from micro frontends are:

- smaller, more cohesive and maintainable codebases
- more scalable organisations with decoupled, autonomous teams
- the ability to upgrade, update, or even rewrite parts of the frontend in a more incremental fashion than was previously possible

It is no coincidence that these headlining advantages are some of the same ones that microservices can provide.

Of course, there are no free lunches when it comes to software architecture - everything comes with a cost. Some micro frontend implementations can lead to duplication of dependencies, increasing the number of bytes our users must download. In addition, the dramatic increase in team autonomy can cause fragmentation in the way your teams work. Nonetheless, we believe that these risks can be managed, and that the benefits of micro frontends often outweigh the costs.

Benefits

Rather than defining micro frontends in terms of specific technical approaches or implementation details, we instead place emphasis on the attributes that emerge and the benefits they give.

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.

This often leads towards a micro frontends architecture. Once one team has had the experience of getting a feature all the way to production with little modification to the old world, other teams will want to join the new world as well. The existing code still needs to be maintained, and in some cases it may make sense to continue to add new features to it, but now the choice is available.

The endgame here is that we're afforded more freedom to make case-by-case decisions on individual parts of our product, and to make incremental upgrades to our architecture, our dependencies, and our user experience. If there is a major breaking change in our main framework, each micro frontend can be upgraded whenever it makes

sense, rather than being forced to stop the world and upgrade everything at once. If we want to experiment with new technology, or new modes of interaction, we can do it in a more isolated fashion than we could before.

Simple, decoupled codebases

The source code for each individual micro frontend will by definition be much smaller than the source code of a single monolithic frontend. 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.

Of course, a single, high-level architectural decision (i.e. "let's do micro frontends"), is not a substitute for good old fashioned clean code. We're not trying to exempt ourselves from thinking about our code and putting effort into its quality. Instead, we're trying to set ourselves up to fall into the pit of success by making bad decisions hard, and good ones easy. 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, which is something that we should have been doing anyway!

Independent deployment

Just as with microservices, independent deployability of micro frontends is key. This reduces the scope of any given deployment, which in turn reduces the associated risk. Regardless of how or where your frontend code is hosted, each micro frontend should have its own continuous delivery pipeline, which builds, tests and deploys it all the way to production. We should be able to deploy each micro frontend with very little thought given to the current state of other codebases or pipelines. It shouldn't matter if the old monolith is on a fixed, manual, quarterly release cycle, or if the team next door has pushed a half-finished or broken feature into their master branch. If a given micro frontend is ready to go to production, it should be able to do so, and that decision should be up to the team who build and maintain it.

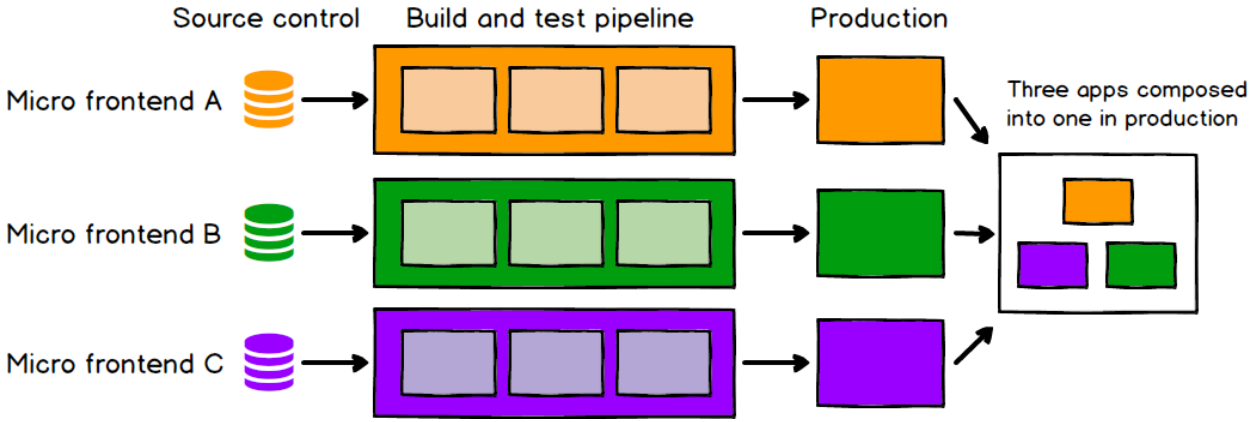

Figure 2: Each micro frontend is deployed to production independently

Autonomous teams

As a higher-order benefit of decoupling both our codebases and our release cycles, we get a long way towards having fully independent teams, who can own a section of a product from ideation through to production and beyond. Teams can have full ownership of everything they need to deliver value to customers, which enables them to move quickly and effectively. For this to work, our teams need to be formed around vertical slices of business functionality, rather than around technical capabilities. An easy way to do this is to carve up the product based on what end users will see, so each micro frontend encapsulates a single page of the application, and is owned end-to-

end by a single team. This brings higher cohesiveness of the teams' work than if teams were formed around technical or “horizontal” concerns like styling, forms, or validation.

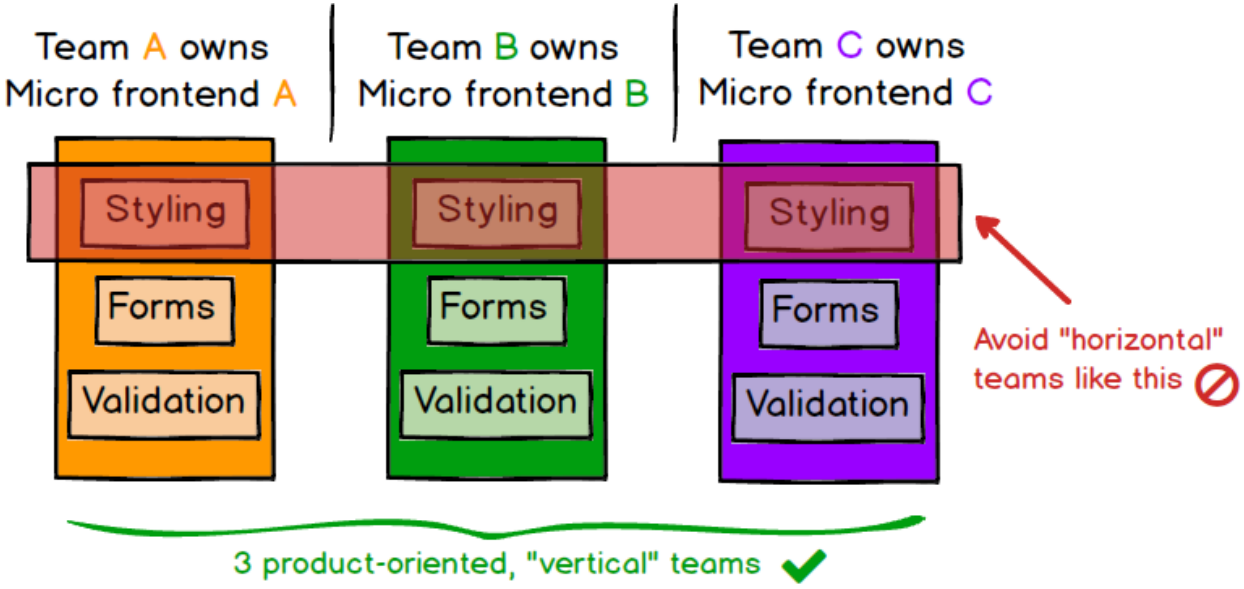

Figure 3: Each application should be owned by a single team

In a nutshell

In short, micro frontends are all about slicing up big and scary things into smaller, more manageable pieces, and then being explicit about the dependencies between them. Our technology choices, our codebases, our teams, and our release processes should all be able to

operate and evolve independently of each other, without excessive coordination.

The example

Imagine a website where customers can order food for delivery. On the surface it's a fairly simple concept, but there's a surprising amount of detail if you want to do it well:

- There should be a landing page where customers can browse and search for restaurants. The restaurants should be searchable and filterable by any number of attributes including price, cuisine, or what a customer has ordered previously
- Each restaurant needs its own page that shows its menu items, and allows a customer to choose what they want to eat, with discounts, meal deals, and special requests

- Customers should have a profile page where they can see their order history, track delivery, and customise their payment options

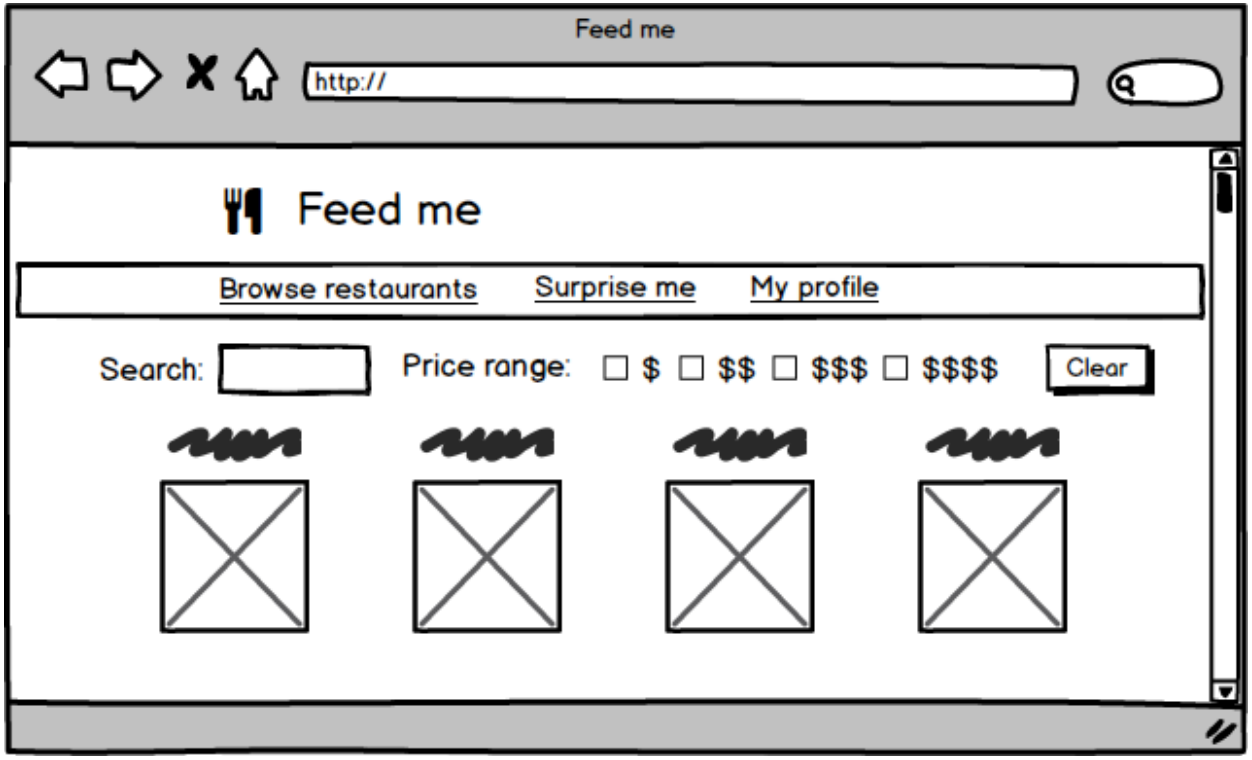

Figure 4: A food delivery website may have several reasonably complex pages

There is enough complexity in each page that we could easily justify a dedicated team for each one, and each of those teams should be able to work on their page independently of all the other teams. They should be able to develop, test, deploy, and maintain their code without worrying about conflicts or coordination with other teams. Our customers, however, should still see a single, seamless website.

Throughout the rest of this article, we'll be using this example application wherever we need example code or scenarios.

Integration approaches

Given the fairly loose definition above, there are many approaches that could reasonably be called micro frontends. In this section we'll show some examples and discuss their tradeoffs. There is a fairly natural architecture that emerges across all of the approaches - generally there is a micro frontend for each page in the application, and there is a single **container application**, which:

- renders common page elements such as headers and footers
- addresses cross-cutting concerns like authentication and navigation
- brings the various micro frontends together onto the page, and tells each micro frontend when and where to render itself

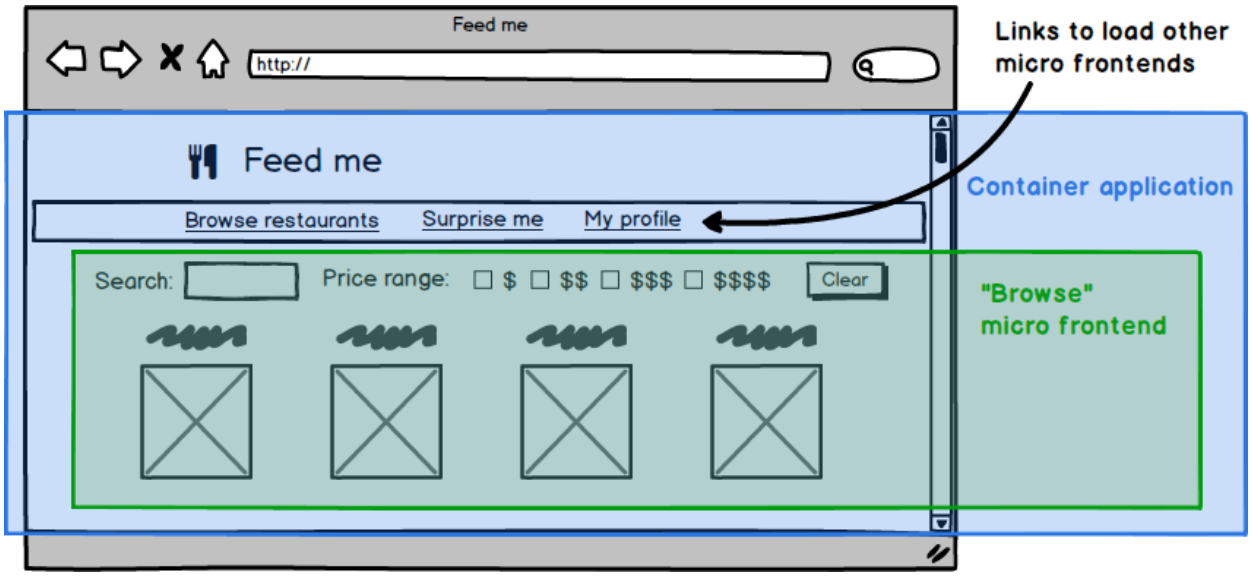

Figure 5: You can usually derive your architecture from the visual structure of the page

Server-side template composition

We start with a decidedly un-novel approach to frontend development - rendering HTML on the server out of multiple templates or fragments. We have an `index.html` which contains any common page elements, and then uses server-side includes to plug in page-specific content from fragment HTML files:

```
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
```

```
    <title>Feed me</title>
  </head>
  <body>
    <h1>🍽 Feed me</h1>
    <!--# include file="$PAGE.html" -->
  </body>
</html>
```

We serve this file using Nginx, configuring the `$PAGE` variable by matching against the URL that is being requested:

```
server {
    listen 8080;
    server_name localhost;

    root /usr/share/nginx/html;
    index index.html;
    ssi on;

    # Redirect / to /browse
    rewrite ^/$ http://localhost:8080/browse redirect;

    # Decide which HTML fragment to insert based on the URL
    location /browse {
      set $PAGE 'browse';
    }
    location /order {
```

```
      set $PAGE 'order';
    }
    location /profile {
      set $PAGE 'profile'
    }

    # All locations should render through index.html
    error_page 404 /index.html;
}
```

This is fairly standard server-side composition. The reason we could justifiably call this micro frontends is that we've split up our code in such a way that each piece represents a self-contained domain concept that can be delivered by an independent team. What's not shown here is how those various HTML files end up on the web server, but the assumption is that they each have their own deployment pipeline, which allows us to deploy changes to one page without affecting or thinking about any other page.

For even greater independence, there could be a separate server responsible for rendering and serving each micro frontend, with one server out the front that makes requests to the others. With careful caching of responses, this could be done without impacting latency.

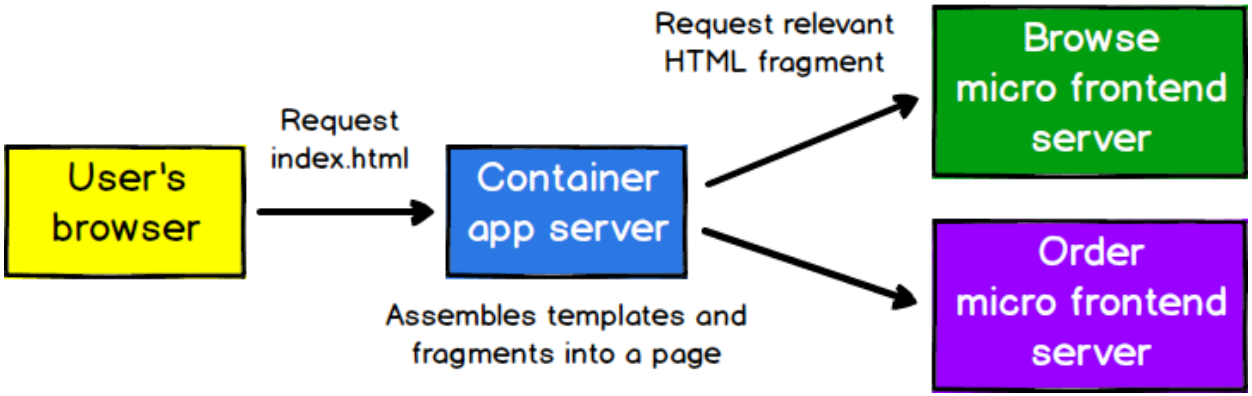

Figure 6: Each of these servers can be built and deployed to independently

This example shows how micro frontends is not necessarily a new technique, and does not have to be complicated. As long as we're careful about how our design decisions affect the autonomy of our codebases and our teams, we can achieve many of the same benefits regardless of our tech stack.

Build-time integration

One approach that we sometimes see is to publish each micro frontend as a package, and have the container application include them all as library dependencies. Here is how the container's `package.json` might look for our example app:

```
{
  "name": "@feed-me/container",
  "version": "1.0.0",
  "description": "A food delivery web app",
  "dependencies": {
    "@feed-me/browse-restaurants": "^1.2.3",
    "@feed-me/order-food": "^4.5.6",
    "@feed-me/user-profile": "^7.8.9"
  }
}
```

At first this seems to make sense. It produces a single deployable Javascript bundle, as is usual, allowing us to de-duplicate common dependencies from our various applications. However, this approach means that we have to re-compile and release every single micro frontend in order to release a change to any individual part of the product. Just as with microservices, we've seen enough pain caused by such a **lockstep release process** that we would recommend strongly against this kind of approach to micro frontends.

Having gone to all of the trouble of dividing our application into discrete codebases that can be developed and tested independently, let's not re-introduce all of that coupling at the release stage. We

should find a way to integrate our micro frontends at run-time, rather than at build-time.

Run-time integration via iframes

One of the simplest approaches to composing applications together in the browser is the humble iframe. By their nature, iframes make it easy to build a page out of independent sub-pages. They also offer a good degree of isolation in terms of styling and global variables not interfering with each other.

```
<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>
```

Just as with the server-side includes option, building a page out of iframes is not a new technique and perhaps does not seem that exciting. But if we revisit the chief benefits of micro frontends listed earlier, iframes mostly fit the bill, as long as we're careful about how we slice up the application and structure our teams.

We often see a lot of reluctance to choose iframes. While some of that reluctance does seem to be driven by a gut feel that iframes are a bit “yuck”, there are some good reasons that people avoid them. The easy isolation mentioned above does tend to make them less flexible than other options. It can be difficult to build integrations between different parts of the application, so they make routing, history, and deep-linking more complicated, and they present some extra challenges to making your page fully responsive.

Run-time integration via JavaScript

The next approach that we'll describe is probably the most flexible one, and the one that we see teams adopting most frequently. Each micro frontend is included onto the page using a `<script>` tag, and upon load exposes a global function as its entry-point. The container application then determines which micro frontend should be mounted, and calls the relevant function to tell a micro frontend when and where to render itself.

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

    <!-- These scripts don't render anything immediately -->
    <!-- Instead they attach entry-point functions to `window` -->
    <script src="https://browse.example.com/bundle.js"></script>
    <script src="https://order.example.com/bundle.js"></script>
    <script src="https://profile.example.com/bundle.js"></script>

    <div id="micro-frontend-root"></div>

    <script type="text/javascript">
```

```
      // These global functions are attached to window by the above scri
      const microFrontendsByRoute = {
        '/': window.renderBrowseRestaurants,
        '/order-food': window.renderOrderFood,
        '/user-profile': window.renderUserProfile,
      };
      const renderFunction = microFrontendsByRoute[window.location.pathr

      // Having determined the entry-point function, we now call it,
      // giving it the ID of the element where it should render itself
      renderFunction('micro-frontend-root');
    </script>
  </body>
</html>
```

The above is obviously a primitive example, but it demonstrates the basic technique. Unlike with build-time integration, we can deploy each of the `bundle.js` files independently. And unlike with iframes, we have full flexibility to build integrations between our micro frontends however we like. We could extend the above code in many ways, for example to only download each JavaScript bundle as needed, or to pass data in and out when rendering a micro frontend.

The flexibility of this approach, combined with the independent deployability, makes it our default choice, and the one that we've seen

in the wild most often. We'll explore it in more detail when we get into the full example.

Run-time integration via Web Components

One variation to the previous approach is for each micro frontend to define an HTML custom element for the container to instantiate, instead of defining a global function for the container to call.

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

    <!-- These scripts don't render anything immediately -->
    <!-- Instead they each define a custom element type -->

    <script src="https://browse.example.com/bundle.js"></script>
    <script src="https://order.example.com/bundle.js"></script>
    <script src="https://profile.example.com/bundle.js"></script>

    <div id="micro-frontend-root"></div>

    <script type="text/javascript">
```

```
      // These element types are defined by the above scripts
      const webComponentsByRoute = {
        '/': 'micro-frontend-browse-restaurants',
        '/order-food': 'micro-frontend-order-food',
        '/user-profile': 'micro-frontend-user-profile',
      };
      const webComponentType = webComponentsByRoute[window.location.path

      // Having determined the right web component custom element type,
      // we now create an instance of it and attach it to the document
      const root = document.getElementById('micro-frontend-root');
      const webComponent = document.createElement(webComponentType);
      root.appendChild(webComponent);
    </script>
  </body>
</html>
```

The end result here is quite similar to the previous example, the main difference being that you are opting in to doing things 'the web component way'. If you like the web component spec, and you like the idea of using capabilities that the browser provides, then this is a good option. If you prefer to define your own interface between the container application and micro frontends, then you might prefer the previous example instead.

Styling

CSS as a language is inherently global, inheriting, and cascading, traditionally with no module system, namespacing or encapsulation. Some of those features do exist now, but browser support is often lacking. In a micro frontends landscape, many of these problems are exacerbated. For example, if one team's micro frontend has a stylesheet that says `h2 { color: black; }`, and another one says `h2 { color: blue; }`, and both these selectors are attached to the same page, then someone is going to be disappointed! This is not a new problem, but it's made worse by the fact that these selectors were written by different teams at different times, and the code is probably split across separate repositories, making it more difficult to discover.

Over the years, many approaches have been invented to make CSS more manageable. Some choose to use a strict naming convention, such as BEM, to ensure selectors only apply where intended. Others, preferring not to rely on developer discipline alone, use a pre-processor such as SASS, whose selector nesting can be used as a form of namespacing. A newer approach is to apply all styles programatically with CSS modules or one of the various CSS-in-JS

libraries, which ensures that styles are directly applied only in the places the developer intends. Or for a more platform-based approach, shadow DOM also offers style isolation.

The approach that you pick does not matter all that much, as long as you find a way to ensure that developers can write their styles independently of each other, and have confidence that their code will behave predictably when composed together into a single application.

Shared component libraries

We mentioned above that visual consistency across micro frontends is important, and one approach to this is to develop a library of shared, re-usable UI components. In general we believe that this a good idea, although it is difficult to do well. The main benefits of creating such a library are reduced effort through re-use of code, and visual consistency. In addition, your component library can serve as a

living styleguide, and it can be a great point of collaboration between developers and designers.

One of the easiest things to get wrong is to create too many of these components, too early. It is tempting to create a Foundation Framework, with all of the common visuals that will be needed across all applications. However, experience tells us that it's difficult, if not impossible, to guess what the components' APIs should be before you have real-world usage of them, which results in a lot of churn in the early life of a component. For that reason, we prefer to let teams create their own components within their codebases as they need them, even if that causes some duplication initially. Allow the patterns to emerge naturally, and once the component's API has become obvious, you can harvest the duplicate code into a shared library and be confident that you have something proven.

The most obvious candidates for sharing are “dumb” visual primitives such as icons, labels, and buttons. We can also share more complex components which might contain a significant amount of UI logic, such as an auto-completing, drop-down search field. Or a sortable, filterable, paginated table. However, be careful to ensure that your shared components contain only UI logic, and no business or domain logic. When domain logic is put into a shared library it creates a high

degree of coupling across applications, and increases the difficulty of change. So, for example, you usually should not try to share a `ProductTable`, which would contain all sorts of assumptions about what exactly a "product" is and how one should behave. Such domain modelling and business logic belongs in the application code of the micro frontends, rather than in a shared library.

As with any shared internal library, there are some tricky questions around its ownership and governance. One model is to say that as a shared asset, "everyone" owns it, though in practice this usually means that *no one* owns it. It can quickly become a hodge-podge of inconsistent code with no clear conventions or technical vision. At the other extreme, if development of the shared library is completely centralised, there will be a big disconnect between the people who create the components and the people who consume them. The best models that we've seen are ones where anyone can contribute to the library, but there is a custodian (a person or a team) who is responsible for ensuring the quality, consistency, and validity of those contributions. The job of maintaining the shared library requires strong technical skills, but also the people skills necessary to cultivate collaboration across many teams.

Cross-application communication

One of the most common questions regarding micro frontends is how to let them talk to each other. In general, we recommend having them communicate as little as possible, as it often reintroduces the sort of inappropriate coupling that we're seeking to avoid in the first place.

That said, some level of cross-app communication is often needed. Custom events allow micro frontends to communicate indirectly, which is a good way to minimise direct coupling, though it does make it harder to determine and enforce the contract that exists between micro frontends. Alternatively, the React model of passing callbacks and data downwards (in this case downwards from the container application to the micro frontends) is also a good solution that makes the contract more explicit. A third alternative is to use the address bar as a communication mechanism, which we'll explore in more detail later.

> If you are using redux, the usual approach is to have a single, global, shared store for the entire application. However, if each micro frontend is supposed to be its own self-contained application, then it makes sense for each one to have its own

> redux store. The redux docs even mention "isolating a Redux app as a component in a bigger application" as a valid reason to have multiple stores.

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.

As with styling, there are several different approaches that can work well here. The most important thing is to think long and hard about what sort of coupling you're introducing, and how you'll maintain that contract over time. Just as with integration between microservices, you won't be able to make breaking changes to your integrations without having a coordinated upgrade process across different applications and teams.

You should also think about how you'll automatically verify that the integration does not break. Functional testing is one approach, but we prefer to limit the number of functional tests we write due to the cost of implementing and maintaining them. Alternatively you could implement some form of consumer-driven contracts, so that each

micro frontend can specify what it requires of other micro frontends, without needing to actually integrate and run them all in a browser together.

Backend communication

If we have separate teams working independently on frontend applications, what about backend development? We believe strongly in the value of full-stack teams, who own their application's development from visual code all the way through to API development, and database and infrastructure code. One pattern that helps here is the BFF pattern, where each frontend application has a corresponding backend whose purpose is solely to serve the needs of that frontend. While the BFF pattern might originally have meant dedicated backends for each frontend channel (web, mobile, etc), it can easily be extended to mean a backend for each micro frontend.

There are a lot of variables to account for here. The BFF might be self contained with its own business logic and database, or it might just be an aggregator of downstream services. If there are downstream services, it may or may not make sense for the team that owns the micro frontend and its BFF, to also own some of those services. If the micro frontend has only one API that it talks to, and that API is fairly stable, then there may not be much value in building a BFF at all. The guiding principle here is that the team building a particular micro frontend shouldn't have to wait for other teams to build things for them. So if every new feature added to a micro frontend also requires backend changes, that's a strong case for a BFF, owned by the same team.

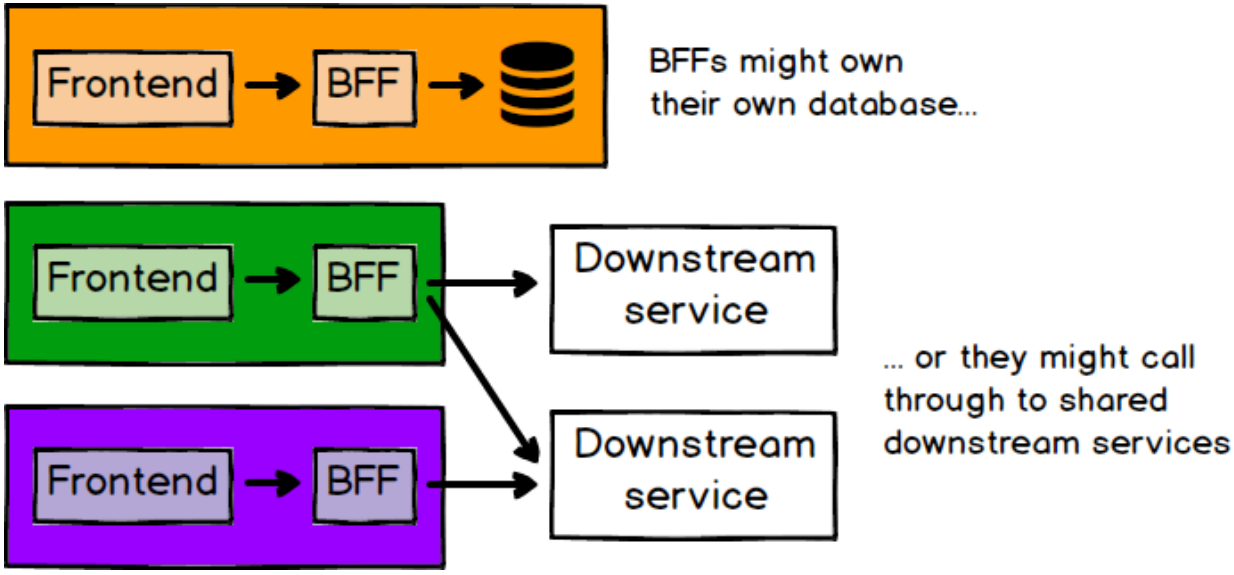

Figure 7: There are many different ways to structure your frontend/backend relationships

Another common question is, how should the user of a micro frontend application be authenticated and authorised with the server? Obviously our customers should only have to authenticate themselves once, so auth usually falls firmly in the category of cross-cutting concerns that should be owned by the container application. The container probably has some sort of login form, through which we obtain some sort of token. That token would be owned by the container, and can be injected into each micro frontend on initialisation. Finally, the micro frontend can send the token with any request that it makes to the server, and the server can do whatever validation is required.

Testing

We don't see much difference between monolithic frontends and micro frontends when it comes to testing. In general, whatever strategies you are using to test a monolithic frontend can be reproduced across each individual micro frontend. That is, each micro frontend should have its own comprehensive suite of automated tests that ensure the quality and correctness of the code.

The obvious gap would then be integration testing of the various micro frontends with the container application. This can be done using your preferred choice of functional/end-to-end testing tool (such as Selenium or Cypress), but don't take things too far; functional tests should only cover aspects that cannot be tested at a lower level of the Test Pyramid. By that we mean, 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. As mentioned above, consumer-driven contracts can help to directly specify the interactions that occur between micro frontends without the flakiness of integration environments and functional testing.

The example in detail

Most of the rest of this article will be a detailed explanation of just one way that our example application can be implemented. We'll focus mostly on how the container application and the micro frontends integrate together using JavaScript, as that's probably the most interesting and complex part. You can see the end result deployed live at https://demo.microfrontends.com, and the full source code can be seen on Github.

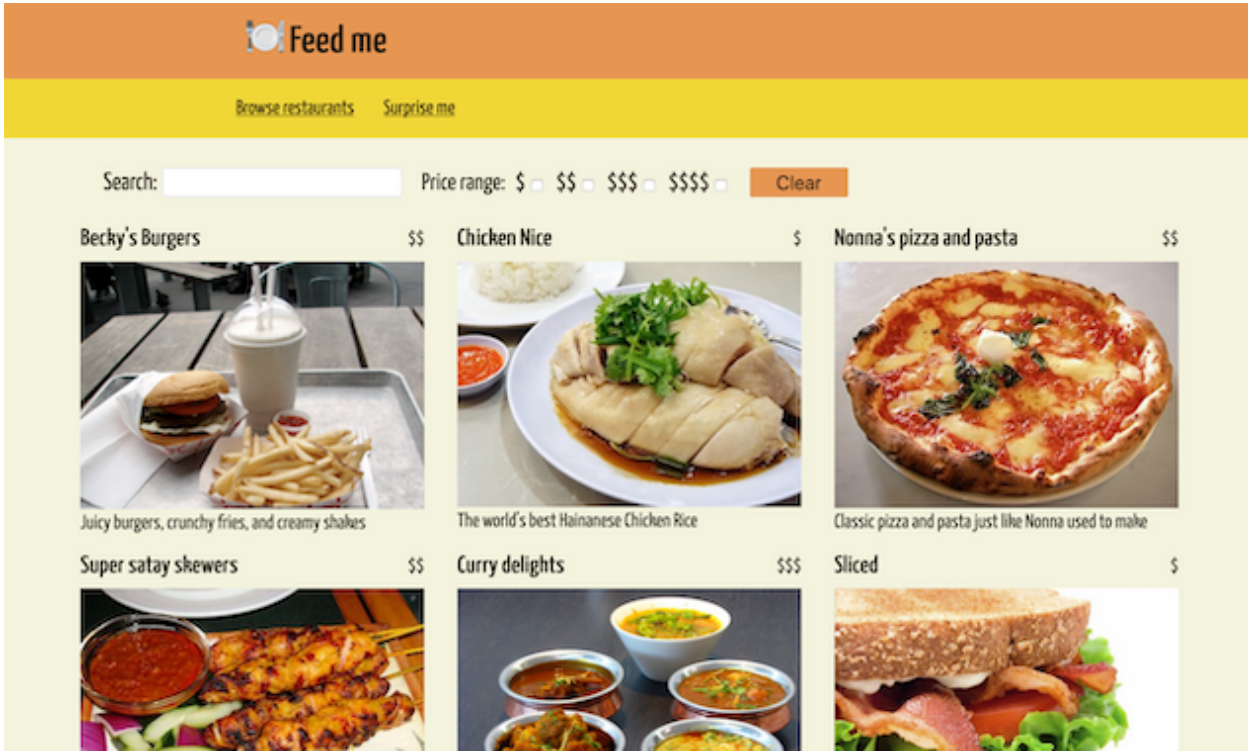

Figure 8: The 'browse' landing page of the full micro frontends demo application

The demo is all built using React.js, so it's worth calling out that React does *not* have a monopoly on this architecture. Micro frontends can be implemented with with many different tools or frameworks. We chose React here because of its popularity and because of our own familiarity with it.

The container

We'll start with the container, as it's the entry point for our customers. Let's see what we can learn about it from its `package.json`:

```
{
  "name": "@micro-frontends-demo/container",
  "description": "Entry point and container for a micro frontends demo",
  "scripts": {
    "start": "PORT=3000 react-app-rewired start",
    "build": "react-app-rewired build",
    "test": "react-app-rewired test"
  },
  "dependencies": {
    "react": "^16.4.0",
    "react-dom": "^16.4.0",
    "react-router-dom": "^4.2.2",
    "react-scripts": "^2.1.8"
  },
  "devDependencies": {
    "enzyme": "^3.3.0",
    "enzyme-adapter-react-16": "^1.1.1",
    "jest-enzyme": "^6.0.2",
    "react-app-rewire-micro-frontends": "^0.0.1",

    "react-app-rewired": "^2.1.1"
  },
  "config-overrides-path": "node_modules/react-app-rewire-micro-frontenc
}
```

> In version 1 of `react-scripts` it was possible to have multiple applications coexist on a single page without conflicts, but version 2 uses some webpack features that cause errors when two or more apps try to render themselves on the one page. For this reason we use `react-app-rewired` to override some of the internal webpack config of `react-scripts`. This fixes those errors, and lets us keep relying on `react-scripts` to manage our build tooling for us.

From the dependencies on `react` and `react-scripts`, we can conclude that it's a React.js application created with `create-react-app`. More interesting is what's *not* there: any mention of the micro frontends that we're going to compose together to form our final application. If we were to specify them here as library dependencies, we'd be heading down the path of build-time integration, which as mentioned previously tends to cause problematic coupling in our release cycles.

To see how we select and display a micro frontend, let's look at `App.js`. We use React Router to match the current URL against a predefined list of routes, and render a corresponding component:

```
<Switch>
  <Route exact path="/" component={Browse} />
  <Route exact path="/restaurant/:id" component={Restaurant} />
```

```
  <Route exact path="/random" render={Random} />
</Switch>
```

The `Random` component is not that interesting - it just redirects the page to a randomly selected restaurant URL. The `Browse` and `Restaurant` components look like this:

```
const Browse = ({ history }) => (
  <MicroFrontend history={history} name="Browse" host={browseHost} />
);
const Restaurant = ({ history }) => (
  <MicroFrontend history={history} name="Restaurant" host={restaurantHos
);
```

In both cases, we render a `MicroFrontend` component. Aside from the history object (which will become important later), we specify the unique name of the application, and the host from which its bundle can be downloaded. This config-driven URL will be something like `http://localhost:3001` when running locally, or `https://browse.demo.microfrontends.com` in production.

Having selected a micro frontend in `App.js`, now we'll render it in `MicroFrontend.js`, which is just another React component:

```
class MicroFrontend extends React.Component {
  render() {
    return <main id={`${this.props.name}-container`} />;
  }
}
```

> *This is not the entire class, we'll be seeing more of its methods soon.*

When rendering, all we do is put a container element on the page, with an ID that's unique to the micro frontend. This is where we'll tell our micro frontend to render itself. We use React's `componentDidMount` as the trigger for downloading and mounting the micro frontend:

> `componentDidMount` is a lifecycle method of React components, which is called by the framework just after an instance of our component has been 'mounted' into the DOM for the first time.

class MicroFrontend...

```
  componentDidMount() {
    const { name, host } = this.props;
    const scriptId = `micro-frontend-script-${name}`;

    if (document.getElementById(scriptId)) {
      this.renderMicroFrontend();
```

```
      return;
    }

    fetch(`${host}/asset-manifest.json`)
      .then(res => res.json())
      .then(manifest => {
        const script = document.createElement('script');
        script.id = scriptId;
        script.src = `${host}${manifest['main.js']}`;
        script.onload = this.renderMicroFrontend;
        document.head.appendChild(script);
      });
  }
```

We have to fetch the script's URL from an asset manifest file, because `react-scripts` outputs compiled JavaScript files that have hashes in their filename to facilitate caching.

First we check if the relevant script, which has a unique ID, has already been downloaded, in which case we can just render it immediately. If not, we fetch the `asset-manifest.json` file from the appropriate host, in order to look up the full URL of the main script asset. Once we've set the script's URL, all that's left is to attach it to

the document, with an `onload` handler that renders the micro frontend:

class MicroFrontend...

```
renderMicroFrontend = () => {
  const { name, history } = this.props;

  window[`render${name}`](`${name}-container`, history);
  // E.g.: window.renderBrowse('browse-container, history');
};
```

In the above code we're calling a global function called something like `window.renderBrowse`, which was put there by the script that we just downloaded. We pass it the ID of the `<main>` element where the micro frontend should render itself, and a `history` object, which we'll explain soon. **The signature of this global function is the key contract between the container application and the micro frontends**. This is where any communication or integration should happen, so keeping it fairly lightweight makes it easy to maintain, and to add new micro frontends in the future. Whenever we want to do something that would require a change to this code, we should think long and hard about what it means for the coupling of our codebases, and the maintenance of the contract.

There's one final piece, which is handling clean-up. When our `MicroFrontend` component un-mounts (is removed from the DOM), we want to un-mount the relevant micro frontend too. There is a corresponding global function defined by each micro frontend for this purpose, which we call from the appropriate React lifecycle method:

class MicroFrontend...

```
componentWillUnmount() {
  const { name } = this.props;

  window[`unmount${name}`](`${name}-container`);
}
```

In terms of its own content, all that the container renders directly is the top-level header and navigation bar of the site, as those are constant across all pages. The CSS for those elements has been written carefully to ensure that it will only style elements within the header, so it shouldn't conflict with any styling code within the micro frontends.

And that's the end of the container application! It's fairly rudimentary, but this gives us a shell that can dynamically download our micro frontends at runtime, and glue them together into something

cohesive on a single page. Those micro frontends can be independently deployed all the way to production, without ever making a change to any other micro frontend, or to the container itself.

The micro frontends

The logical place to continue this story is with the global render function we keep referring to. The home page of our application is a filterable list of restaurants, whose entry point looks like this:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

window.renderBrowse = (containerId, history) => {
  ReactDOM.render(<App history={history} />, document.getElementById(cor
  registerServiceWorker();
};

window.unmountBrowse = containerId => {
  ReactDOM.unmountComponentAtNode(document.getElementById(containerId));
};
```

Usually in React.js applications, the call to `ReactDOM.render` would be at the top-level scope, meaning that as soon as this script file is loaded, it immediately begins rendering into a hard-coded DOM element. For this application, we need to be able control both when and where the rendering happens, so we wrap it in a function that receives the DOM element's ID as a parameter, and we attach that function to the global `window` object. We can also see the corresponding un-mounting function that is used for clean-up.

While we've already seen how this function is called when the micro frontend is integrated into the whole container application, one of the biggest criteria for success here is that we can develop and run the micro frontends independently. So each micro frontend also has its own `index.html` with an inline script to render the application in a "standalone" mode, outside of the container:

```
<html lang="en">
  <head>
    <title>Restaurant order</title>
  </head>
  <body>
    <main id="container"></main>
    <script type="text/javascript">
```

```
      window.onload = () => {
        window.renderRestaurant('container');
      };
    </script>
  </body>
</html>
```

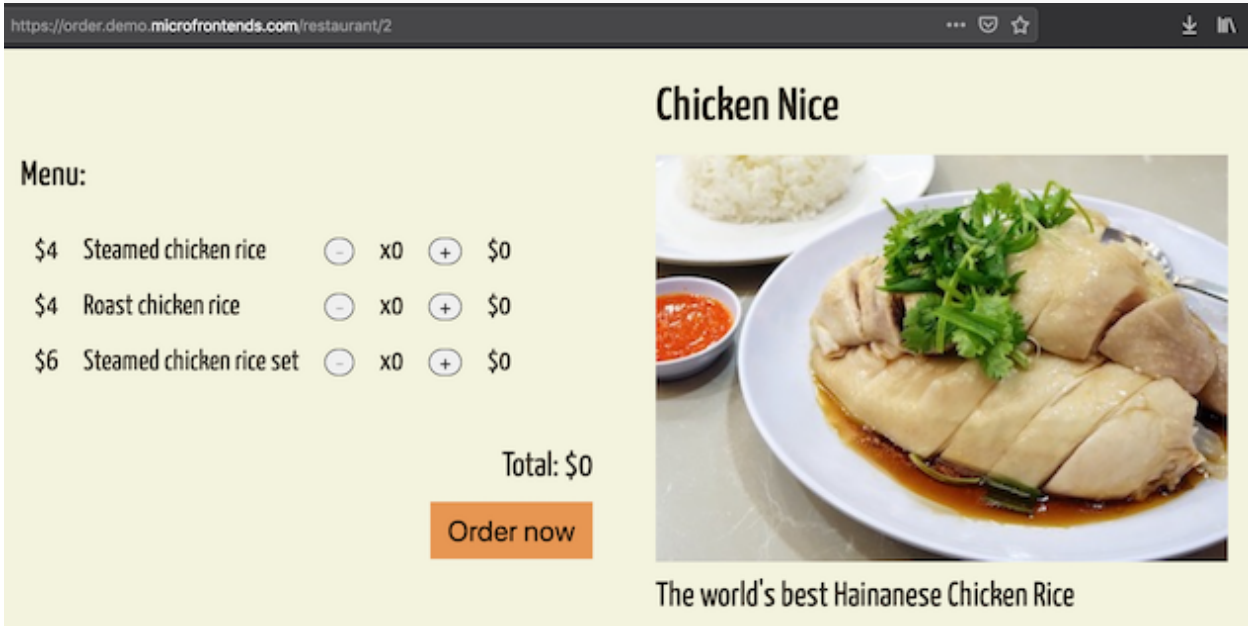

Figure 9: Each micro frontend can be run as a standalone application outside of the container.

From this point onwards, the micro frontends are mostly just plain old React apps. The 'browse' application fetches the list of restaurants from the backend, provides `<input>` elements for searching and filtering the restaurants, and renders React Router `<Link>` elements,

which navigate to a specific restaurant. At that point we would switch over to the second, 'order' micro frontend, which renders a single restaurant with its menu.

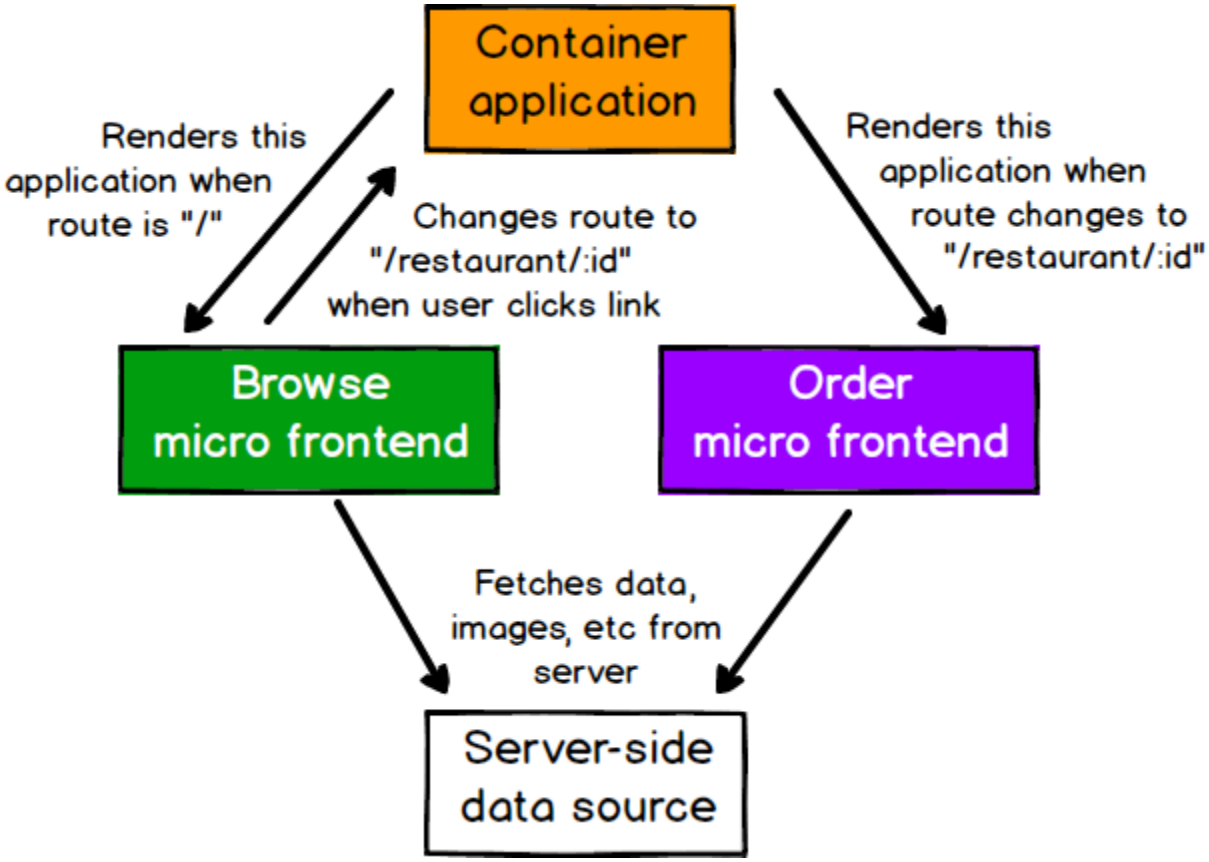

Figure 10: These micro frontends interact only via route changes, not directly

The final thing worth mentioning about our micro frontends is that they both use `styled-components` for all of their styling. This CSS-in-JS

library makes it easy to associate styles with specific components, so we are guaranteed that a micro frontend's styles will not leak out and effect the container, or another micro frontend.

Cross-application communication via routing

We mentioned earlier that cross-application communication should be kept to a minimum. In this example, the only requirement we have is that the browsing page needs to tell the restaurant page which restaurant to load. Here we will see how we can use client-side routing to solve this problem.

All three React applications involved here are using React Router for declarative routing, but initialised in two slightly different ways. For the container application, we create a `<BrowserRouter>`, which internally will instantiate a `history` object. This is the same `history` object that we've been glossing over previously. We use this object to manipulate the client-side history, and we can also use it to link multiple React Routers together. Inside our micro frontends, we initialise the Router like this:

```
<Router history={this.props.history}>
```

In this case, rather than letting React Router instantiate another history object, we provide it with the instance that was passed in by the container application. All of the `<Router>` instances are now connected, so route changes triggered in any of them will be reflected in all of them. This gives us an easy way to pass "parameters" from one micro frontend to another, via the URL. For example in the browse micro frontend, we have a link like this:

```
<Link to={`/restaurant/${restaurant.id}`}>
```

When this link is clicked, the route will be updated in the container, which will see the new URL and determine that the restaurant micro frontend should be mounted and rendered. That micro frontend's own routing logic will then extract the restaurant ID from the URL and render the right information.

Hopefully this example flow shows the flexibility and power of the humble URL. Aside from being useful for sharing and bookmarking, in this particular architecture it can be a useful way to communicate intent across micro frontends. Using the page URL for this purpose ticks many boxes:

- Its structure is a well-defined, open standard
- It's globally accessible to any code on the page
- Its limited size encourages sending only a small amount of data
- It's user-facing, which encourages a structure that models the domain faithfully
- It's declarative, not imperative. I.e. "this is where we are", rather than "please do this thing"
- It forces micro frontends to communicate indirectly, and not know about or depend on each other directly

When using routing as our mode of communication between micro frontends, the routes that we choose constitute a **contract**. In this case, we've set in stone the idea that a restaurant can be viewed at `/restaurant/:restaurantId`, and we can't change that route without updating all applications that refer to it. Given the importance of this contract, we should have automated tests that check that the contract is being adhered to.

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 wrote earlier about how shared component libraries can help with consistency across micro frontends, but for this small demo a component library would be overkill. So instead, we have a small repository of common content, including images, JSON data, and CSS, which are served over the network to all micro frontends.

There's another thing that we can choose to share across micro frontends: library dependencies. As we will describe shortly, duplication of dependencies is a common drawback of micro frontends. Even though sharing those dependencies across applications comes with its own set of difficulties, for this demo application it's worth talking about how it can be done.

The first step is to choose which dependencies to share. A quick analysis of our compiled code showed that about 50% of the bundles was contributed by `react` and `react-dom`. In addition to their size, these two libraries are our most 'core' dependencies, so we know that all micro frontends can benefit from having them extracted. Finally, these are stable, mature libraries, which usually introduce breaking changes across two major versions, so cross-application upgrade efforts should not be too difficult.

As for the actual extraction, all we need to do is mark the libraries as externals in our webpack config, which we can do with a rewiring similar to the one described earlier.

```
module.exports = (config, env) => {
  config.externals = {
    react: 'React',
    'react-dom': 'ReactDOM'
  }
  return config;
};
```

Then we add a couple of `script` tags to each `index.html` file, to fetch the two libraries from our shared content server.

```
<body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="root"></div>
  <script src="%REACT_APP_CONTENT_HOST%/react.prod-16.8.6.min.js"></scri
  <script src="%REACT_APP_CONTENT_HOST%/react-dom.prod-16.8.6.min.js"></
</body>
```

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. However, if we're careful about what we do share and what we don't, then there are real benefits to be gained.

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. This balances the convenience of centralised infrastructure management with the flexibility of independent deployability.

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.

Downsides

At the start of this article, we mentioned that there are tradeoffs with micro frontends, as there are with any architecture. The benefits that we've mentioned do come with a cost, which we'll cover here.

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. There is a direct relationship between page performance and user engagement/conversion, and much of the world runs on internet infrastructure much slower than those in highly-developed cities are used to, so we have many reasons to care about download sizes.

This issue is not easy to solve. There is an inherent tension between our desire to let teams compile their applications independently so that they can work autonomously, and our desire to build our

applications in such a way that they can share common dependencies. One approach is to externalise common dependencies from our compiled bundles, as we described for the demo application. As soon as we go down this path though, we've re-introduced some build-time coupling to our micro frontends. Now there is an implicit contract between them which says, “we all must use these exact versions of these dependencies”. If there is a breaking change in a dependency, we might end up needing a big coordinated upgrade effort and a one-off lockstep release event. This is everything we were trying to avoid with micro frontends in the first place!

This inherent tension is a difficult one, but it's not all bad news. Firstly, even if we choose to do nothing about duplicate dependencies, it's possible that each individual page will still load faster than if we had built a single monolithic frontend. The reason is that by compiling each page independently, we have effectively implemented our own form of code splitting. In classic monoliths, when any page in the application is loaded, we often download the source code and dependencies of every page all at once. 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. If we are disciplined in not bloating our

micro frontends with unnecessary dependencies, or if we know that users generally stick to just one or two pages within the application, we may well achieve a net *gain* in performance terms, even with duplicated dependencies.

There are lots of “may’s” and “possibly’s” in the previous paragraph, which highlights the fact that every application will always have its own unique performance characteristics. If you want to know for sure what the performance impacts will be of a particular change, there is no substitute for taking real-world measurements, ideally in production. We've seen teams agonise over a few extra kilobytes of JavaScript, only to go and download many megabytes of high-resolution images, or run expensive queries against a very slow database. So while 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. We may even be able to run our micro frontend in a

“standalone” mode, on a blank page, rather than inside the container application that will house it in production. This can make development a lot simpler, especially when the real container is a complex, legacy codebase, which is often the case when we're using micro frontends to do a gradual migration from old world to new. However, there are risks associated with developing in an environment that is quite different to production. 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. Of particular concern are global styles that may be brought along by the container, or by other micro frontends.

The solution here is not that different to any other situation where we have to worry about environmental differences. If we're developing locally in an environment that is not production-like, we need to ensure that we regularly integrate and deploy our micro frontend to environments that *are* like production, and we should do testing (manual and automated) in these environments to catch integration issues as early as possible. 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. So before adopting such an architecture there are a few questions you should consider:

- 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?

We could probably fill another entire article discussing these topics. 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.

Conclusion

As frontend codebases continue to get more complex over the years, we see a growing need for more scalable architectures. We need to be able to draw clear boundaries that establish the right levels of coupling and cohesion between technical and domain entities. We should be able to scale software delivery across independent, autonomous teams.

While far from the only approach, we have seen many real-world cases where micro frontends deliver these benefits, and we've been able to apply the technique gradually over time to legacy codebases as well as new ones. Whether micro frontends are the right approach for you and your organiation or not, we can only hope that this will be

part of a continuing trend where frontend engineering and architecture is treated with the seriousness that we know it deserves.

Acknowledgements

Huge thanks to Charles Korn, Andy Marks, and Willem Van Ketwich for their thorough reviews and detailed feedback.

Thanks also to Bill Codding, Michael Strasser, and Shirish Padalkar for their input given on the ThoughtWorks internal mailing list.

Thanks to Martin Fowler for his feedback as well, and for giving this this article a home here on his website.

And finally, thanks to Evan Bottcher and Liauw Fendy for their encouragement and support.

Significant Revisions

19 *June* 2019: Published final installment on downsides

17 *June* 2019: Published installment with the example

13 *June* 2019: Published installment with sections from styling to testing.

11 *June* 2019: Published installment on integration approaches.

10 *June* 2019: Published first installment: covering benefits