Report: Micro Frontends

Summary:

This article delves into the various integration strategies within the micro frontend architecture paradigm. It outlines the prevalent approaches and emphasises runtime JavaScript integration. Through a detailed example project supplemented with code snippets and GitHub repository links, readers gain practical insights. Additionally, it evaluates the significant advantages and drawbacks of this architecture, while offering solutions to prevalent challenges encountered during its implementation.

Keywords:

MFA implementation | MFA benefits | MFA Integration | MFA downsides

Research Questions:

1. **What are the benefits of Micro Frontends?**
2. **What are the Integration approaches?**
3. **What are the downsides of Micro Frontends?**

Key Points:

➔ **Micro frontends can be defined as**

- "*An architectural style where independently deliverable frontend applications are composed into a greater whole*" *[1]*

➔ **Benefits**

- Enables incremental upgrades on each sub-app
- Simple, decoupled - easily understandable codebases
- Independent deployment - for each micro frontend with its own continuous delivery pipeline, which builds, tests and deploys it all the way to production
- Autonomous teams - micro frontend encapsulates a single page of the application, and is owned end-to-end *[3-6]*

➔ **Integration 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 *[7-8]*

- **Server-side template composition**
    - Index.html contains any common page elements
    - uses server-side includes to plug in page-specific content from fragment HTML files *[8]*

- **Build-time integration**
    - publish each micro frontend as a package and the container application includes them all as library dependencies
    - produces a single deployable Javascript bundle
    - Change to single micro frontend causes re-compilation of the whole app *[10]*

- **Run-time integration via iframes**
    - Composing applications together in the browser using iframe
    - By their nature, makes it easy to build a page out of independent sub-pages
    - Difficult to build integrations between different parts of the application
    - Extra challenges with responsiveness *[10-11]*

- **Run-time integration via JavaScript**
    - 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 decides which micro frontend should be mounted, and calls the relevant function to tell a micro frontend when and where to render itself.
    - Each of the bundle.js files can be deployed independently.
    - Full flexibility to build integrations between micro frontends
    - Most often used approach *[11-12]*

- **Run-time integration via Web Components**
    - Each micro frontend defines an HTML custom element
    - Container then instantiates this element instead of calling global function *[13]*

➔ **Styling**

- CSS is inherently global, inheriting, and cascading, traditionally with no module system, namespacing or encapsulation which can cause problems in micro frontend architecture
- Pre-processor such as SASS should be used or apply all styles programmatically with CSS modules or one of the various CSS-in-JS libraries
- For a more platform-based approach, shadow DOM offers style isolation *[14]*

➔ **Shared component libraries**

- Consistent UI across micro frontends is important, to achieve this library of shared, reusable UI components can be developed
- Shared components should only contain UI logic, no business or domain logic to avoid high degree of coupling across applications
- 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. *[14-15]*

➔ **Cross-application communication**

- Applications should communicate as little as possible
- Custom events allow micro frontends to communicate indirectly, which is a good way to minimise direct coupling
- Alternatively, passing callbacks and data downwards (from the container to the micro frontends) is also a good solution
- Third alternative is to use the address bar as a communication mechanism
- Avoid having any shared state *[16]*

➔ **Backend communication**

- Preferable is BFF pattern, where each frontend application has a corresponding backend whose purpose is solely to serve the needs of that one frontend
- For authentication container obtain some sort of token and inject it into each micro frontend on initialisation, the micro frontend can send the it with any request *[17-18]*

➔ **Testing**

- Each micro frontend should have its own comprehensive suite of automated tests

- Integration testing should be done using your preferred choice of functional/end-to-end testing tool (such as Selenium or Cypress)
- Functional tests should only cover aspects that cannot be tested at a lower level of the Test Pyramid
- Consumer-driven contracts can help to directly specify the interactions that occur between micro frontends *[18-19]*

➔ **Downsides**

- **Payload size**
  - Independently-built JavaScript bundles can cause duplication of common dependencies
  - Increasing the number of bytes we have to send over the network

- **Environment differences**
  - Micro frontend can be run in a "standalone" mode, on a blank page, rather than inside the container application that will house it in production
  - There are risks associated with developing in an environment that is quite different to production
  - It can behave differently when deployed to production, particular concern are global styles

- **Operational and governance complexity**
  - Micro frontend architecture leads to having more stuff to manage - more repositories, more tools, more build/deploy pipelines, more servers, more domains, etc *[28-30]*

Infographics:

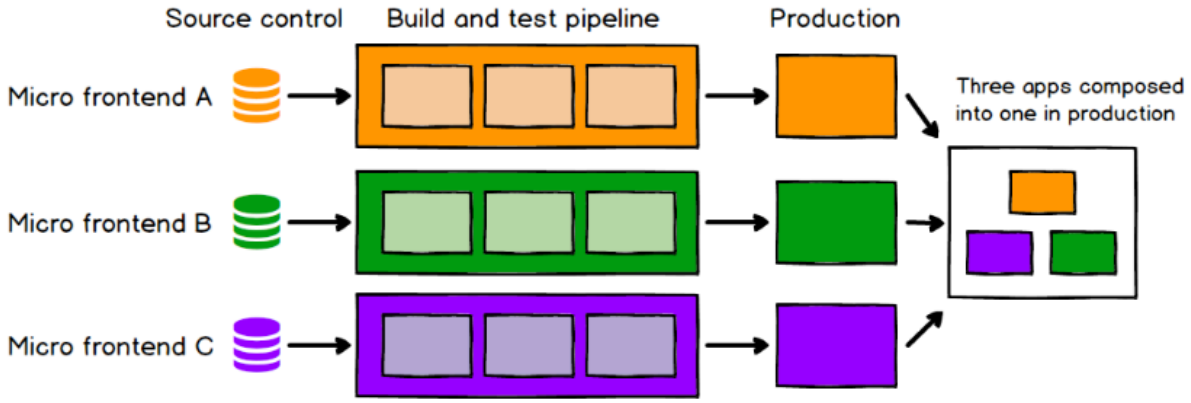

Figure 1: Microfrontend separate deployment [5]

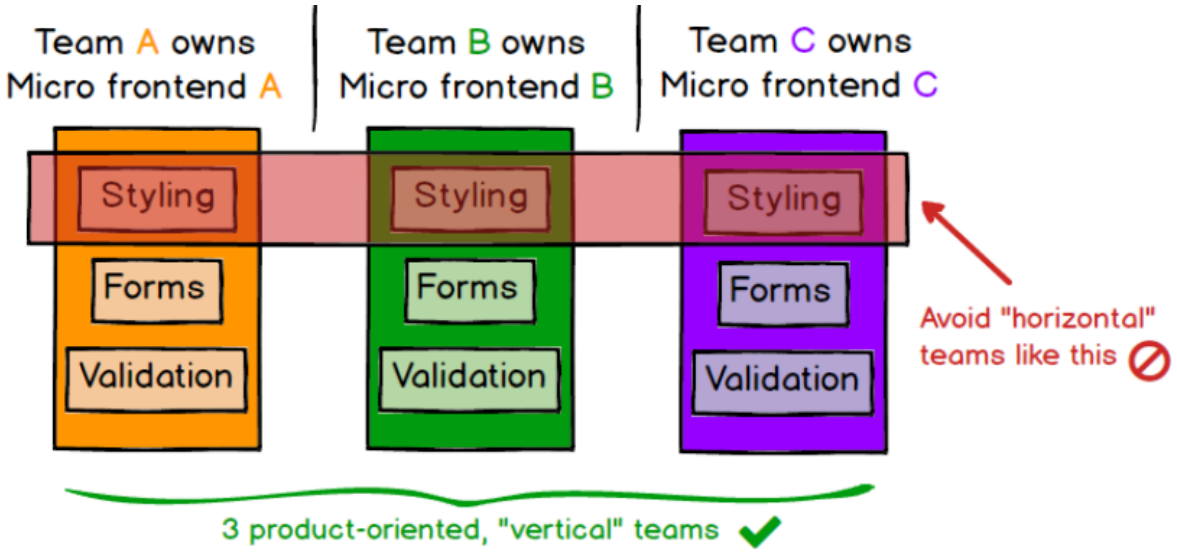

Figure 2: Each application should be owned by a single team [6]