Architecture

Technical Guide

Prepared by

Kadharmoideen Fadurudeen

In this guide

1 article · ~15 min total reading

Architecture January 31, 2025 15 min read

Buffet: A Delightful Microfront-end Architecture

Introducing Buffet, a revolutionary microfront-end architecture designed to dramatically increase app delivery velocity. Learn how this modular approach enables teams to work independently while maintaining a cohesive user experience.

Open Source Project
Check out the Buffet repository on GitHub

View on GitHub

Introduction

In modern web development, monolithic front-end applications often become bottlenecks as teams grow and features multiply. Buffet is a microfront-end architecture that solves this problem by allowing multiple teams to work on independent applications that seamlessly integrate into a cohesive user experience.

Think of it like a buffet restaurant—each station (microfront-end) operates independently, but together they create a complete dining experience. Teams can develop, test, and deploy their modules without blocking others, dramatically increasing development velocity.

Key Benefits

Independent Deployment

Deploy features independently without coordinating with other teams or waiting for full app builds.

Team Autonomy

Each team owns their microfront-end completely—from tech stack choices to deployment schedules.

Technology Flexibility

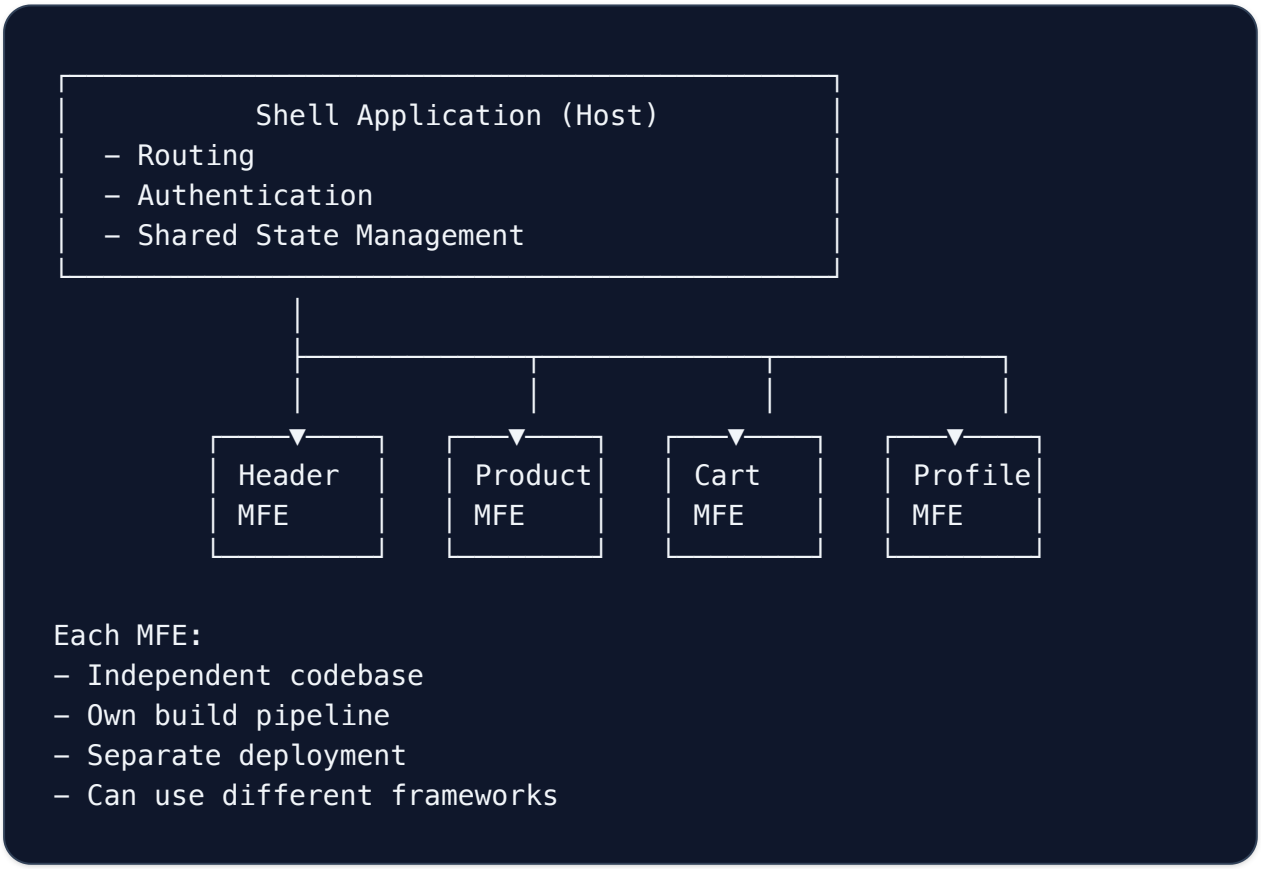
Use React, Vue, Angular, or any framework. Each microfront-end can use different technologies.

Faster Time to Market

Parallel development and independent deployments mean features reach users faster.

Architecture Overview

Buffet uses a shell application that orchestrates multiple microfront-ends. Each microfront-end is a self-contained application that can be developed and deployed independently.



Implementation with Module Federation

Buffet leverages Webpack Module Federation to enable runtime integration of microfront-ends. Here's how to set it up:

Shell Application Configuration

```
// webpack.config.js (Shell)
const ModuleFederationPlugin = require('webpack/lib/container/ModuleFe

module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'shell',
      remotes: {
        header: 'header@http://localhost:3001/remoteEntry.js',
        products: 'products@http://localhost:3002/remoteEntry.js',
        cart: 'cart@http://localhost:3003/remoteEntry.js',
      },
      shared: {
        react: { singleton: true, requiredVersion: '^18.0.0' },
        'react-dom': { singleton: true, requiredVersion: '^18.0.0' },
      },
    }),
  ],
}
```

Microfront-end Configuration

```
// webpack.config.js (Products MFE)
const ModuleFederationPlugin = require('webpack/lib/container/ModuleFe

module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'products',
      filename: 'remoteEntry.js',
      exposes: {
        './ProductList': './src/components/ProductList',
        './ProductDetail': './src/components/ProductDetail',
      },
      shared: {
        react: { singleton: true, requiredVersion: '^18.0.0' },
        'react-dom': { singleton: true, requiredVersion: '^18.0.0' },
      },
    }),
  ],
}
```

Loading Microfront-ends Dynamically

```
// Shell application
import React, { lazy, Suspense } from 'react'

const ProductList = lazy(() => import('products/ProductList'))
const Cart = lazy(() => import('cart/Cart'))

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <ProductList />
      </Suspense>

      <Suspense fallback={<div>Loading cart...</div>}>
        <Cart />
      </Suspense>
    </div>
  )
}
```

Communication Between Microfront-ends

Microfront-ends need to communicate without creating tight coupling. Buffet provides several patterns:

Event Bus Pattern

Use a custom event system for loose coupling between microfront-ends.

```
// Event Bus
class EventBus {
  events = {}

  emit(event, data) {
    if (this.events[event]) {
      this.events[event].forEach(callback => callback(data))
    }
  }

  on(event, callback) {
    if (!this.events[event]) this.events[event] = []
    this.events[event].push(callback)
  }
}

// Usage in Cart MFE
eventBus.emit('cart:item-added', { productId: 123 })

// Usage in Header MFE
eventBus.on('cart:item-added', (data) => {
  updateCartCount()
})
```

Shared State Management

Use a shared state management solution like Redux or Zustand for global state.

```
// Shared store
import { create } from 'zustand'

export const useCartStore = create((set) => ({
  items: [],
  addItem: (item) => set((state) => ({
    items: [...state.items, item]
  })),
  removeItem: (id) => set((state) => ({
    items: state.items.filter(i => i.id !== id)
  })),
}))

// Usage in any MFE
const { items, addItem } = useCartStore()
```

Best Practices

Define Clear Boundaries

Each microfront-end should own a specific domain or feature. Avoid overlapping responsibilities.

Share Dependencies Wisely

Share common libraries like React, but allow teams to use different versions of other dependencies.

Implement Error Boundaries

Wrap each microfront-end in error boundaries to prevent one failure from crashing the entire app.

Monitor Performance

Track loading times and bundle sizes for each microfront-end to maintain optimal performance.

Real-World Example: E-Commerce Platform

Consider an e-commerce platform with multiple teams working on different features:

1. **Header Team**
   Owns navigation, search bar, and cart icon. Deploys independently when adding new menu items.

2. **Product Team**
   Manages product listings, filters, and details. Can experiment with new layouts without affecting checkout.

3. **Checkout Team**
   Handles cart, payment, and order confirmation. Can update payment providers without coordinating with other teams.

4. **Account Team**
   Manages user profiles, order history, and preferences. Deploys new features on their own schedule.

Conclusion

Buffet's microfront-end architecture enables organizations to scale their front-end development by allowing teams to work independently while maintaining a cohesive user experience. By breaking down monolithic applications into smaller, manageable pieces, teams can move faster, experiment more freely, and deliver value to users more frequently.

The key to success with Buffet is establishing clear boundaries, implementing robust communication patterns, and maintaining shared standards for user experience and performance. When done right, microfront-ends can dramatically increase development velocity while improving code quality and team satisfaction.

Ready to Try Buffet?

Check out the GitHub repository for complete examples, documentation, and starter templates to get your microfront-end architecture up and running.

View on GitHub