

Micro-frontend Architecture avec Django Backend

E-commerce modulaire utilisant Webpack Module Federation



Travail élaboré par :
• Feriel Chaibi

Planning

01 Justification

02 Le backend

03 Le Frontend (MFE)

04 L'assemblage

05 Transversal : déploiement

06 Conclusion

Problématique & Objectifs du projet

?? Problématique:

- Complexité des applications web
- Monolithe Frontend (Couplage Fort)
- Déploiement Global (Risque Élevé)
- Conflits d'Équipe (Ralentissement du développement)



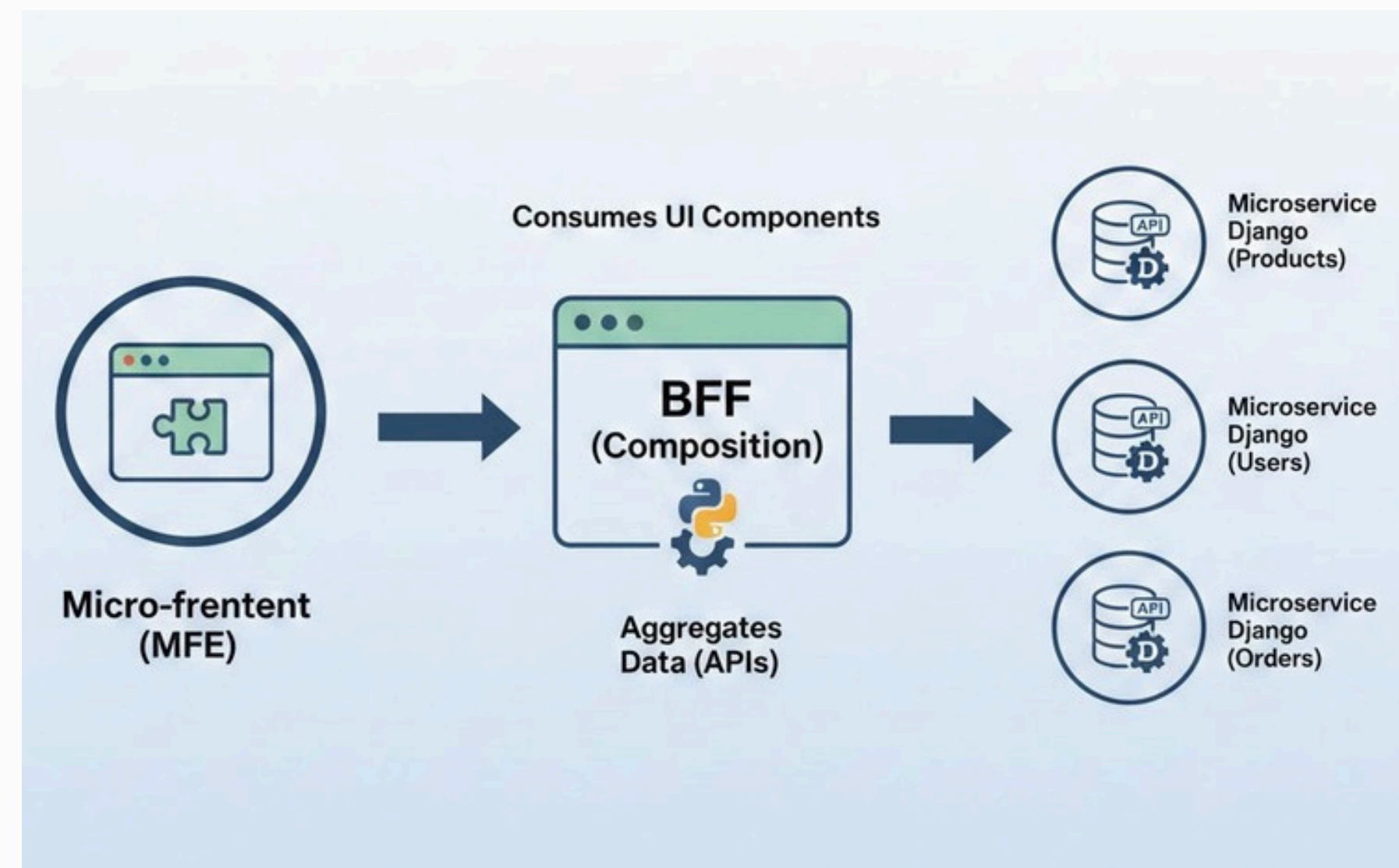
Objectifs :

- Décomposition en MFEs autonomes
- Déploiement Indépendant (CI/CD)
- Composition Dynamique (Container)
- Expérience Utilisateur Unifiée (SPA unique)
- Stack : Django + Webpack Module Federation

Le Réseau de Concepts (Micro-frontend, Django, Microservices)

4

- **Micro-frontend (MFE) :**
Décomposition du Frontend, Petites SPAs autonomes.
- **Django Microservices :** Le Backend décomposé, fournissant des APIs REST/GraphQL ciblées.
- **CI/CD & Déploiement Indépendant :** Mise à jour d'un MFE sans toucher aux autres.
- **Module Federation :** L'outil technique pour la composition des MFEs.



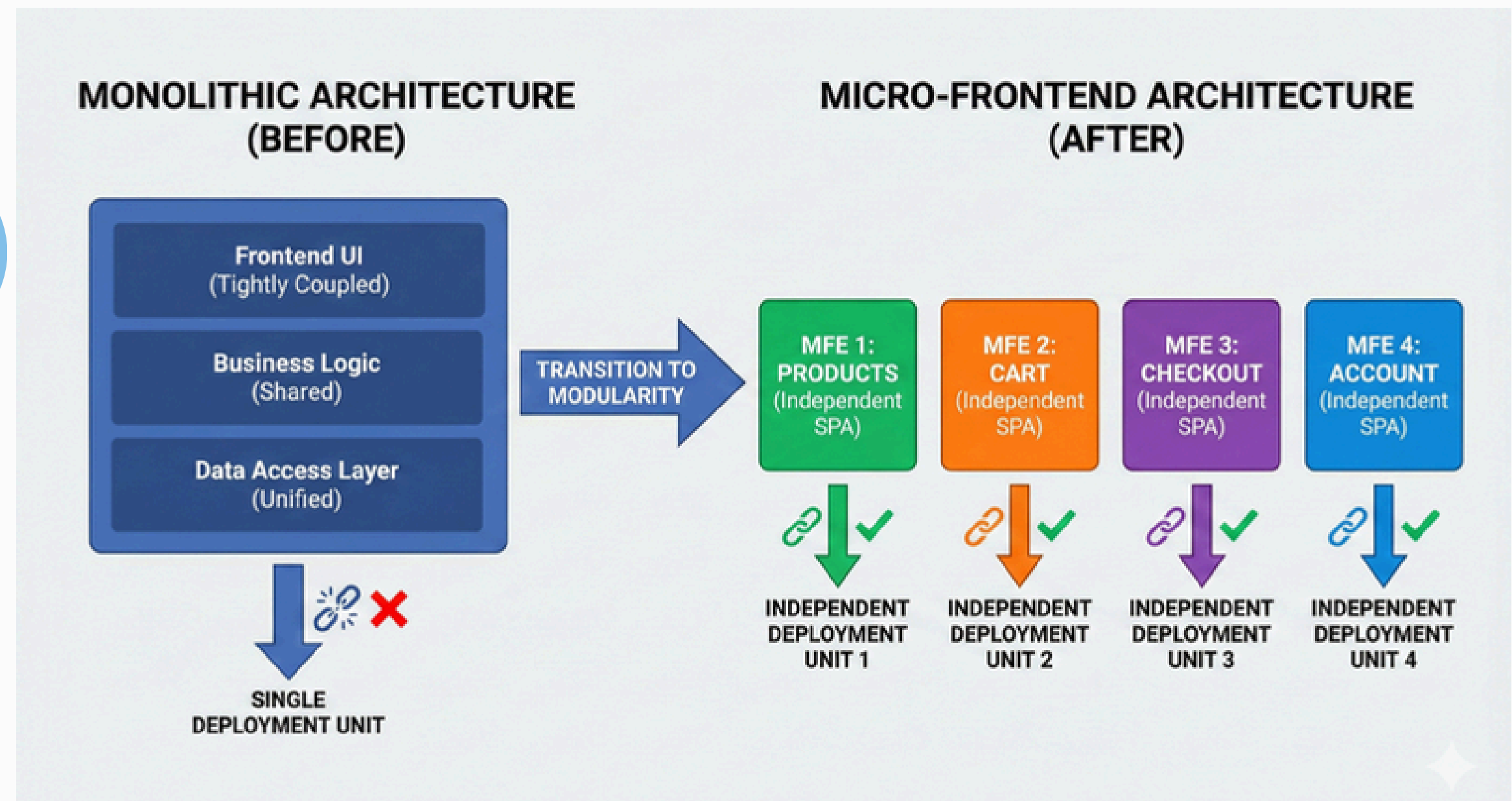
Monolithe vs Micro-frontends : Avantages Opérationnels

Monolithe :

1 Codebase, 1 Stack, 1
Déploiement. Risque élevé.

Micro-frontend :

N Codebases, Stacks
Hétérogènes possibles, N
Déploiements. Risque isolé.

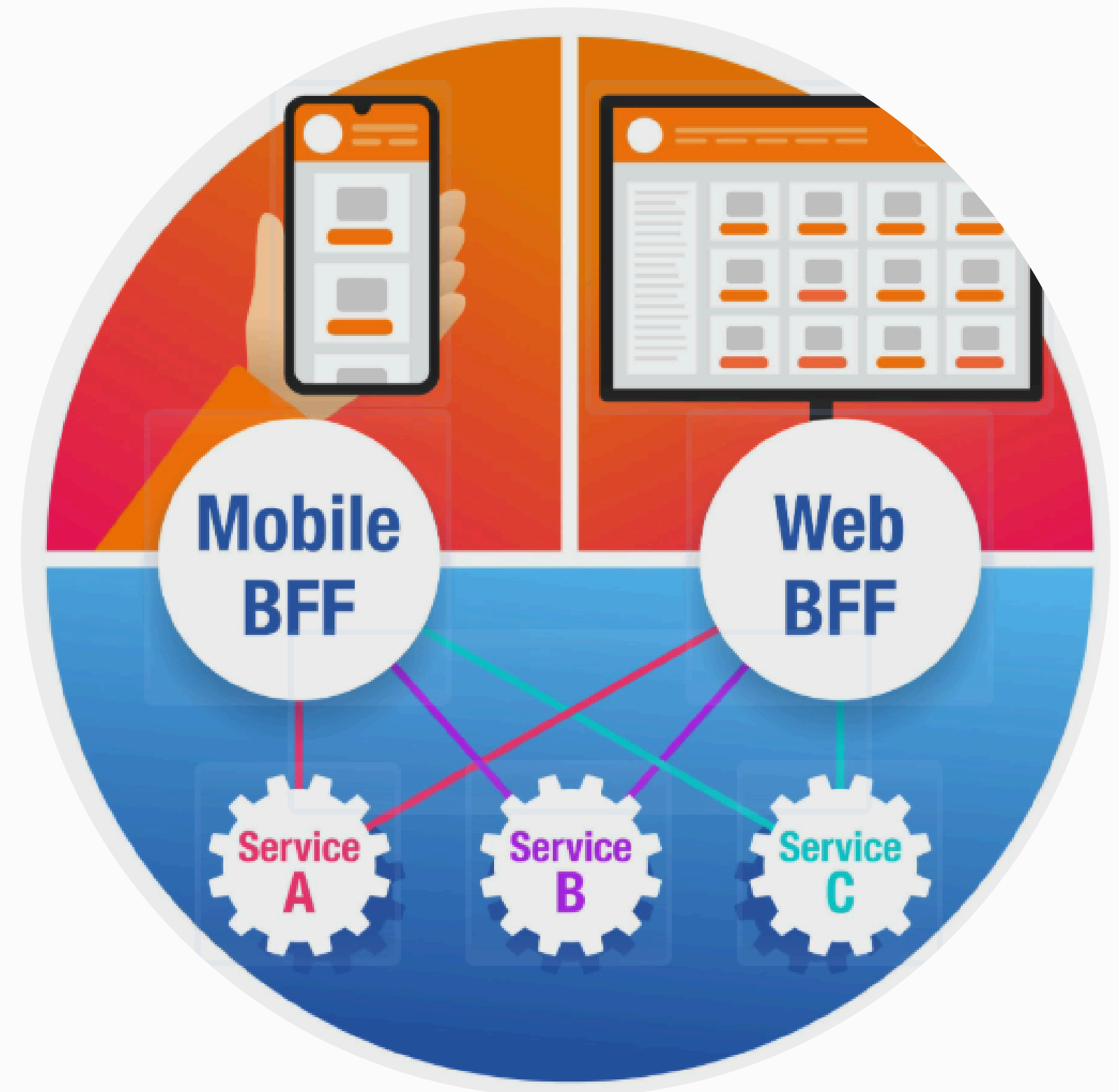


→ Scalabilité des équipes et choix technologique.

Composition Frontend et Backend-for-Frontend (BFF)

Le BFF (Backend-for-Frontend) est une couche Django qui s'intercale entre le MFE et les Microservices pour agréger et transformer les données.

Par exemple, le BFF du MFE Checkout peut combiner en une seule requête les informations des Microservices Utilisateur et Panier.



Composition et Backend-for-Frontend (BFF) : Simplifier l'Accès aux Données

Composition

01 Montage Frontal : Le composant MFE importé est monté dans le DOM du Shell, comme un composant local (grâce à Module Federation).

La Solution

03 BFF (Django) : Un service Backend dédié, agissant comme agrégateur de données spécifique au besoin du MFE.

Le Problème

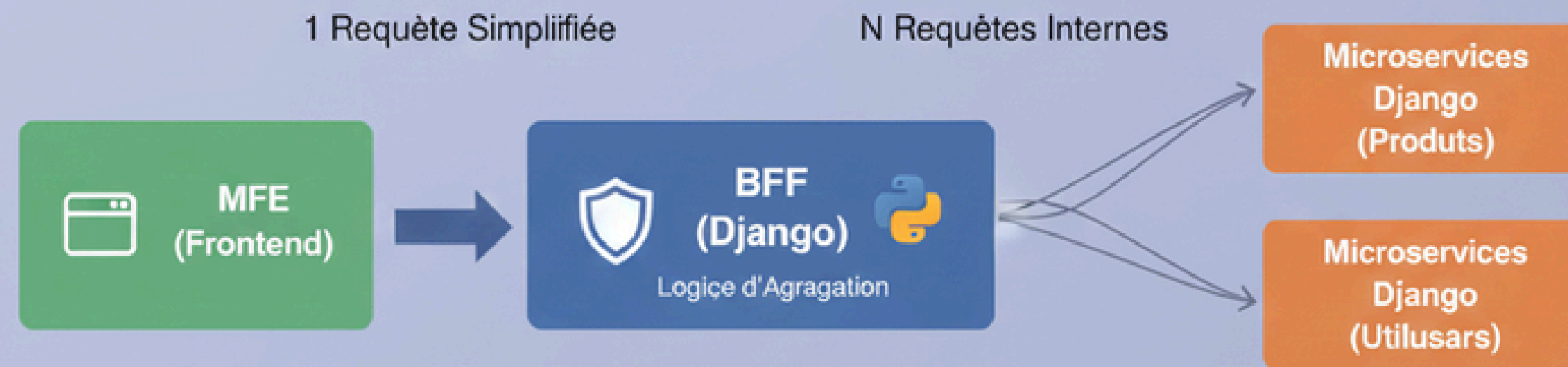
02 Sans BFF, un MFE doit faire plusieurs appels réseau vers différents Microservices Django (ex: chercher le produit, le stock, l'historique utilisateur)

Flux

04 Optimisation du Réseau :
MFE ----> BFF (Django) ----> Microservices Django. (Un seul appel côté Frontend).

Visuel

Composition et Backend-for-Frontend (BFF) : Simplifitir l'Accès aux Dontes

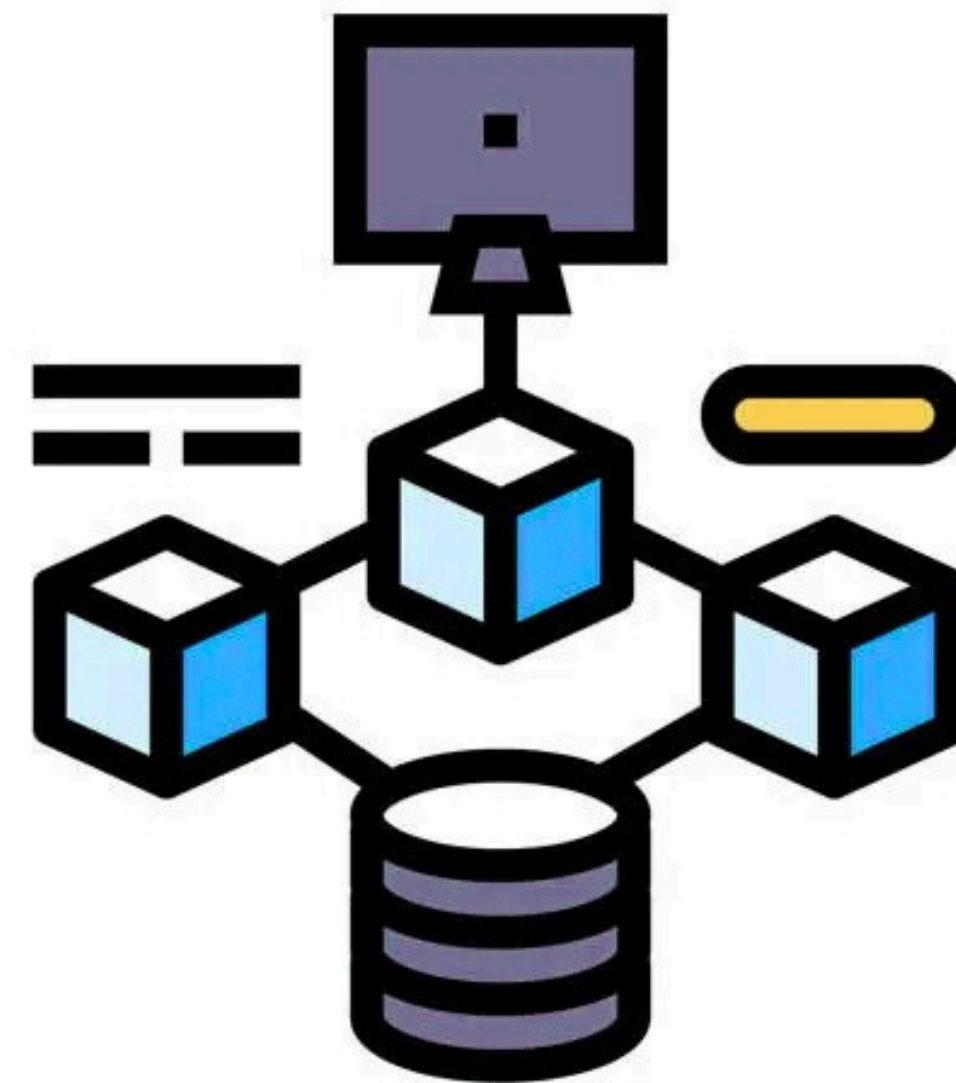


1 Réponse Enrichié

MFE → BFF (Dango) → Microservices Dango

Les Microservices Django (Le Backend)

- **Indépendance Fonctionnelle :** Chaque service est une application Django autonome, isolée.
- **Domaine Isolé :** Un service = Un métier (Catalogue, Panier, Commande).
- **Contrat d'API :** Exposition uniquement via des APIs REST bien définies (DRF).



```
from rest_framework import serializers
from .models import CartItem
import requests
```

```
CATALOG_SERVICE_URL = "http://127.0.0.1:8001/api/products"
```

Service panier doit connaître l'adresse de catalogue pour pouvoir lui parler

```
class CartItemSerializer(serializers.ModelSerializer):
    product = serializers.SerializerMethodField()
```

```
class Meta:
    model = CartItem
    fields = ['id', 'product', 'quantity', 'unit_price']
```

```
def get_product(self, obj):
    try:
        response = requests.get(f"{CATALOG_SERVICE_URL}/{obj.product_id}/")
        response.raise_for_status()
        product_data = response.json()
        return {
            "id": product_data.get("id"),
            "name": product_data.get("name", "Produit indisponible"),
            "price": float(product_data.get("price", 0)),
            "image_url": product_data.get("image_url", ""),
        }
    except:
        return {
            "id": obj.product_id,
            "name": "Produit indisponible",
            "price": 0,
            "image_url": "",
        }
```

DÉCOMPOSITION : Le Panier appelle l'API du Catalogue pour obtenir les données.

RÉSILIENCE : Si le service Catalogue tombe, le Panier ne crashe pas.

Le Frontend — Les MFEs Indépendants

(Autonomie des Équipes)

Philosophie

Décomposition métier :

Le Frontend est divisé en applications autonomes basées sur le domaine

→ Catalogue

→ Panier

→ Checkout

Structure

Indépendance Totale :

Chaque MFE est une SPA (React/Vue/etc.) avec son propre dépôt Git, ses propres dépendances et son cycle de vie.

Bénéfice

Agilité des équipes :

Les équipes travaillent sur leurs MFEs sans interfaces ni dépendances avec les autres équipes .

L'E-commerce Décomposé

Shell/Host

Fournit le cadre unifié : Header, Footer et Routage principal.

MFE Catalogue

Affichage de la liste de produits. Consomme l'API Django /products/.

MFE Panier

Gestion des articles, quantités et totaux. Consomme l'API Django /cart/.

MFE Checkout

Processus d'entrée des informations de paiement. Consomme l'API Django /payment/.

Structure des dossiers

```

  ✓ micro-frontends
    ✓ cart
      > node_modules
      > public
      > src
      B .babelrc
      ⚙ .env
      {} package-lock.json
      {} package.json
      ⚙ webpack.config.js
      > catalogue
      > checkout
      {} micro-frontends.code-workspace
```

micro-frontends/cart/src/app.jsx

```
// micro-frontends/cart/src/app.jsx
import React, { useEffect, useState } from "react";
import axios from "axios";
import { SESSION_ID, CART_API_BASE } from "../session";
```

```
const API_CART = `${CART_API_BASE}/cart/`;
```

```
export default function App() {
  const [cartItems, setCartItems] = useState([]);
  const [loading, setLoading] = useState(true);
  const [total, setTotal] = useState(0);
  const [error, setError] = useState(null);
```

→ **AUTONOMIE** : Le MFE gère son propre État (Panier) et son propre Backend (8002).

micro-frontends/cart/src/app.jsx

```
const loadCart = async () => {
  setError(null);
  setLoading(true);
  try {
    const res = await axios.get(` ${API_CART}?session=${SESSION_ID}` );
    setCartItems(res.data);
    const sum = res.data.reduce(
      (acc, item) => acc + ((item.product?.price || 0) * item.quantity),
      0
    );
    setTotal(sum);
    setLoading(false);
  } catch (err) {
    console.error("Erreur CartService :", err);
    setLoading(false);
    if (err.response) setError(`Erreur API ${err.response.status} : ${err.response.statusText}` );
    else if (err.request) setError("Le CartService n'a pas répondu. Vérifiez CORS et service démarré.");
    else setError("Erreur inconnue lors du chargement du panier.");
  }
};
```

→ **LOGIQUE ISOLÉE** : Le MFE contient toute sa logique métier et d'appel d'API.

Module Federation :

16

Mécanisme permettant à une application JavaScript de charger dynamiquement du code d'une autre application.

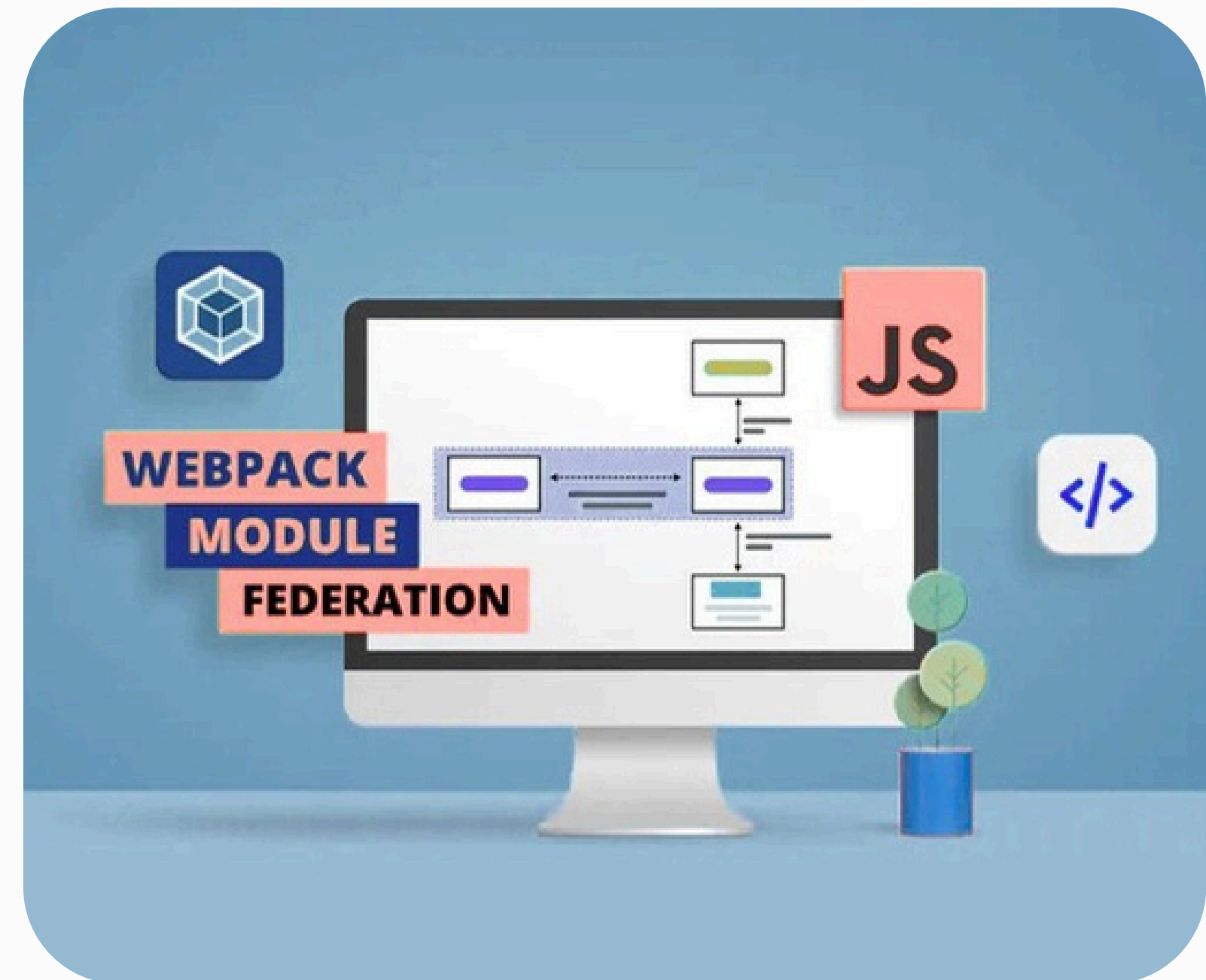
Rôle:

- Permet à une application (Remote) d'exposer du code (Exposes) et à une autre (Host) de l'importer (Remotes).

Avantage Majeur : Shared Dependencies (Dépendances Partagées):

- Évite le chargement multiple de bibliothèques comme React ou Vue, assurant la performance.

L'application Host charge le remoteEntry.js du MFE distant au runtime.



```
const { override, addWebpackPlugin } = require('customize-cra');  
const webpack = require('webpack');
```

```
module.exports = override(  
  (config) => {  
    config.output.publicPath = 'auto';  
    return config;  
  },  
  addWebpackPlugin(  
    new webpack.container.ModuleFederationPlugin({  
      name: 'container',  
      remotes: {  
        catalog_remote: 'catalog_remote@http://localhost:3001/remoteEntry.js',  
        cart_remote: 'cart_remote@http://localhost:3002/remoteEntry.js',  
        checkout_remote: 'checkout_remote@http://localhost:3003/remoteEntry.js',  
      },  
      shared: {  
        react: { singleton: true, eager: true },  
        'react-dom': { singleton: true, eager: true },  
      },  
    })  
  )  
);
```

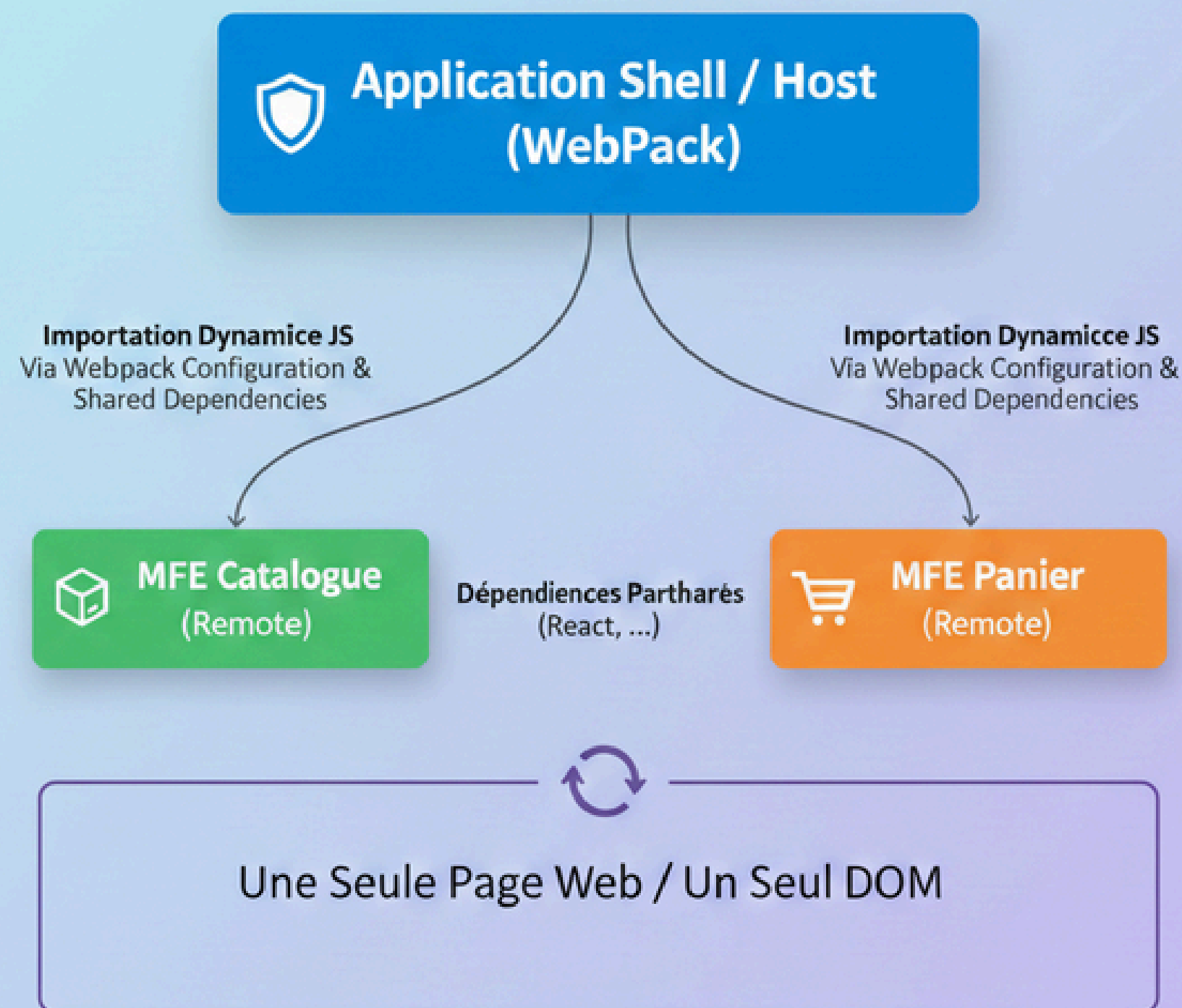


Host (Hôte) : L'application conteneur qui charge les micro-frontends. (container)

Remote (Distant) : Le micro-frontend qui expose ses modules (composants).

Crucial : Section shared: ['react', 'react-dom'] pour l'optimisation des performances.

Partie 3 : Orchestration – Module Federation : Le Liant Technique



L'Indépendance Déploiement (CI/CD)

"Chaque Micro-frontend est un dépôt Git séparé, avec son propre pipeline de CI/CD."

- Intégration (CI) : Les tests et le build sont exécutés uniquement pour le MFE modifié.
- Déploiement (CD) : Le code est déployé uniquement pour le MFE modifié.

Principe Clé

1 MFE = 1 Pipeline

Processus

Isolation : Un commit dans le dépôt cart/ déclenche uniquement le déploiement de ce MFE. Les autres ne sont pas touchés

Avantage

Vélocité et Sécurité : Cycles de release très courts. Mise à jour des fonctionnalités en quelques minutes. Le risque de panne est isolé.

CONCLUSION:

20

LES 3 PILIERS FONDAMENTAUX



Déploiement indépendant:

Chaque micro-frontend est construit et mis en production séparément → on peut déployer le panier 10 fois par jour sans toucher au catalogue.



Autonomie complète des équipes:

Une équipe possède entièrement son micro-frontend (code, tests, CI/CD) → plus de conflits de merge, plus de coordination infinie.



Composition à l'exécution:

Les différentes parties ne sont assemblées qu'au moment où l'utilisateur charge la page → grâce au Container + Module Federation.

→ L'esprit des Microservices appliqué avec succès au Frontend, supporté par Django