Estándar técnico de arquitectura para el desarrollo de MicroFrontends, basado en React, Typescript, Webpack Module Federation, ItauOne y Atomic Design, con ContenStack.   
  
Lineamientos formales usando el lenguaje de requisitos definido por el RFC 2119, con ejemplos técnicos, estructura de carpetas, y buenas prácticas de diseño y despliegue desde el punto de vista de arquitectura de software.

# Estándar Técnico de Arquitectura de MicroFrontends (React, TypeScript, Webpack, ContentStack)

Este documento define los lineamientos obligatorios para el desarrollo de arquitecturas MicroFrontend basados en **React 18.2.0**, **TypeScript 4.9.5**, **Webpack 5 (Module Federation)**, las librerías internas *ItauOne* (@itau-one/core, @itau-one/icons, @itau-one/react), el CMS **ContentStack** y la metodología **Atomic Design**. Los siguientes puntos se establecen con terminología RFC 2119: **DEBE**, **NO DEBE**, **DEBERÍA**, **PUEDE**.

## Tecnologías Fundamentales

* **React 18.2.0:** Se **DEBE** usar React 18.2.0 como librería de UI. Todas las aplicaciones frontend **DEBEN** estar escritas en React, sin usar otros frameworks (Angular, Vue, etc.).
* **TypeScript 4.9.5:** El uso de TypeScript 4.9.5 **DEBE** ser obligatorio para asegurar tipado estático. La integración de TypeScript con React mejora la seguridad y mantenibilidad del código mediante tipado estático y detección temprana de errores en tiempo de compilación.
* **Webpack 5 + Module Federation:** **DEBE** usarse Webpack 5 con el plugin ModuleFederationPlugin para habilitar la arquitectura de MicroFrontends. Webpack 5 incluye Module Federation para cargar módulos compilados por separado, lo que provee una solución oficial para microfrontends.
* **Librerías ItauOne:** **DEBE** incorporarse la librería de componentes corporativos *ItauOne*. En particular:  
  + @itau-one/react: librería de componentes UI React (botones, inputs, layout, etc.).
  + @itau-one/icons: librería de íconos SVG del sistema de diseño.
  + @itau-one/core: estilos globales y tokens de diseño.  
     Se **DEBE** consultar la documentación interna en GitLab de estas librerías para conocer su uso correcto y configuraciones específicas. El uso consistente del sistema de diseño ItauOne promueve interfaces coherentes y escalables.
* **ContentStack (CMS):** **DEBE** usarse ContentStack como CMS Headless. Se **DEBE** integrar mediante su SDK oficial (TypeScript/JavaScript) para recuperar contenido dinámico. Integrar React con un CMS Headless permite construir interfaces de usuario más rápidas y optimizadas.
* **Atomic Design:** **DEBE** organizarse la estructura del proyecto según la metodología Atomic Design. Esto implica clasificar los componentes en los niveles: **Átomos**, **Moléculas**, **Organismos**, **Plantillas/Template** y **Páginas**. Atomic Design facilita la creación de interfaces modulares y consistentes.
* **Otras tecnologías:** **NO SE DEBEN** usar librerías o frameworks fuera de los mencionados. Por ejemplo, no incorporar Angular, Vue, jQuery, Bootstrap (salvo lo que provea ItauOne), ni SDKs no autorizados.

## Arquitectura de MicroFrontends

La arquitectura **DEBE** definirse con un *Shell* (aplicación contenedora o «host») que cargue dinámicamente los MicroFrontends individuales como módulos federados. Cada microfrontend (MFE) es una aplicación React independiente que **DEBE** compilarse y desplegarse por separado. El *Shell* y los MFEs se comunican en tiempo de ejecución mediante URLs externas definidas en Webpack Module Federation. En Webpack, el ModuleFederationPlugin permite exponer y consumir módulos entre builds independientes en tiempo de ejecución.

* El **Shell** (host) **DEBE** declarar en su webpack.config.js los remotos (MFEs) a cargar. Ejemplo mínimo de configuración del Shell:  
    
   const { ModuleFederationPlugin } = require('webpack').container;
* module.exports = {
* entry: './src/index',
* output: { publicPath: 'auto' },
* plugins: [
* new ModuleFederationPlugin({
* name: 'shell',
* remotes: {
* // 'appA' es el nombre del MFE remoto, la URL apunta a su remoteEntry.js
* appA: 'appA@http://localhost:3001/remoteEntry.js'
* },
* shared: ['react', 'react-dom'] // librerías comunes
* })
* ]
* };
* Cada **MicroFrontend (MFE)** **DEBE** exponer sus módulos críticos (por ejemplo, páginas o componentes principales) en su propio webpack.config.js. Ejemplo mínimo de configuración de un MFE:  
    
   const { ModuleFederationPlugin } = require('webpack').container;
* module.exports = {
* entry: './src/index',
* plugins: [
* new ModuleFederationPlugin({
* name: 'appA', // nombre del MFE
* filename: 'remoteEntry.js', // archivo que sirve el MFE
* exposes: {
* './Widget': './src/components/Widget.tsx' // expone el componente Widget
* },
* shared: ['react', 'react-dom'] // librerías compartidas
* })
* ]
* };
* **DEBE** definirse la clave shared en Module Federation para evitar duplicados de bibliotecas comunes (por ejemplo React). Con shared se especifican paquetes compartidos entre builds federadas. Por defecto, Webpack solo comparte el nivel raíz de la librería. Se recomienda compartir al menos react y react-dom, y versionarlos según compatibilidad de semver.
* El **Shell** **DEBE** cargar los MFEs mediante *import* dinámico o componentes personalizados. Por ejemplo, const module = await import('appA/Widget'); cargará el componente remoto. El MFE puede exponer un *Web Component* o un componente React que se integre en el DOM del Shell. Esta integración en tiempo de ejecución permite desarrollo independiente de cada MFE.

## Estructura de Carpeta (Atomic Design)

**DEBE** seguirse la metodología Atomic Design para organizar el código y las carpetas del proyecto. Los componentes se agrupan por nivel de complejidad, facilitando la reutilización:

* **Átomos:** componentes básicos e indivisibles (por ejemplo, Button.tsx, Icon.tsx, etc.).
* **Moléculas:** combinaciones de átomos con funcionalidad específica (por ejemplo, FormField.tsx combina Label + Input).
* **Organismos:** componentes complejos que agrupan moléculas/átomos (por ejemplo, Header.tsx, Footer.tsx, menús completos).
* **Plantillas (Templates):** estructuras de página que disponen organismos (por ejemplo, HomeTemplate.tsx).
* **Páginas:** instancias finales con contenido real (por ejemplo, HomePage.tsx).

Por ejemplo, una posible estructura de carpetas en src/ puede ser:

* src/
* atoms/
* Button.tsx
* Icon.tsx
* Input.tsx
* ...
* molecules/
* FormField.tsx
* SearchBar.tsx
* ...
* organisms/
* Header.tsx
* Footer.tsx
* Sidebar.tsx
* ...
* templates/
* HomeTemplate.tsx
* DashboardTemplate.tsx
* ...
* pages/
* HomePage.tsx
* DashboardPage.tsx
* ...

Esta organización refleja las 5 etapas del diseño atómico (Átomos, Moléculas, Organismos, Plantillas, Páginas). Atomic Design promueve un sistema de diseño consistente que facilita la evolución de la UI. Se **DEBE** adoptar esta estructura en todos los MFEs.

## Librerías ItauOne

Los MicroFrontends **DEBEN** usar los componentes e íconos del sistema de diseño ItauOne para asegurar uniformidad visual. Las recomendaciones son:

* **Componentes UI:** Importar componentes desde @itau-one/react. Por ejemplo, botones, formularios y layouts predefinidos.
* **Íconos:** Usar @itau-one/icons para representar íconos estándar de la marca.
* **Estilos y tokens:** Aplicar los estilos globales y variables definidas en @itau-one/core.

Ejemplo de uso correcto de ItauOne en un componente React:

* import React from 'react';
* import { Button } from '@itau-one/react';
* import { HomeIcon } from '@itau-one/icons';
* export const ExampleComponent: React.FC = () => (
* <div>
* <Button type="primary" onClick={() => alert('Click')}>
* <HomeIcon /> Ir a inicio
* </Button>
* </div>
* );

Se **DEBE** seguir la documentación en GitLab de ItauOne para configuraciones específicas (temas, tipografía, variantes de componentes, etc.). La adopción unificada de ItauOne **permite crear interfaces consistentes y escalables**.

## Integración con ContentStack

La gestión de contenido **DEBE** realizarse a través de ContentStack, siguiendo estas pautas:

* **SDK de ContentStack:** Utilizar el SDK oficial (JavaScript/TypeScript) para inicializar el cliente. Por ejemplo:  
    
   import \* as Contentstack from 'contentstack';
* const Stack = Contentstack.Stack({
* api\_key: 'TU\_API\_KEY',
* delivery\_token: 'TU\_DELIVERY\_TOKEN',
* environment: 'production'
* });
* Este código inicializa el cliente ContentStack con las credenciales correspondientes. Se **DEBE** proteger las credenciales (p.ej. variables de entorno) y no exponerlas en el frontend.
* **Recuperación de contenido:** Definir funciones para consultas de ContentStack. Por ejemplo, para obtener entradas de un tipo de contenido:  
    
   export async function fetchEntries(contentType: string, queryParams = {}) {
* const Query = Stack.ContentType(contentType).Query();
* Object.entries(queryParams).forEach(([key, value]) => Query.where(key, value));
* const result = await Query.includeCount().toJSON().find();
* return result[0];
* }
* Con estas funciones, cada MFE **DEBE** obtener su contenido específico (por ejemplo, entradas de blog, configuraciones, etc.) del CMS en tiempo de ejecución.
* **Sincronización:** Se **DEBE** diseñar el flujo de despliegue de contenido de ContentStack para que, una vez publicado (API pública), los MFEs puedan refrescar o volver a desplegarse si es necesario. ContentStack como CMS Headless mejora la eficiencia al separar contenido de presentación.
* **Buenas prácticas:** Es recomendable utilizar Query.includeCount() o Query.language() según se necesite. También **DEBERÍA** aprovecharse la vista previa en vivo de ContentStack durante el desarrollo (Live Preview) para validar cambios de contenido rápidamente.

## Despliegue y CI/CD

Para mantener la integridad y calidad del sistema, **DEBE** establecerse un pipeline de CI/CD que automatice la construcción, prueba y despliegue de cada MicroFrontend. Las recomendaciones mínimas son:

* **Integración continua (CI):** Cada MFE **DEBE** tener su propio pipeline en GitLab CI/CD (o similar). El pipeline **DEBE** ejecutar *lint*, pruebas unitarias y compilación (build) en cada push. Solo si todas las validaciones pasan, se debe permitir el despliegue. Los pipelines automatizan la construcción, prueba y despliegue del código.
* **Entrega continua (CD):** Los artefactos compilados (por ejemplo, bundles o contenedores Docker) **DEBEN** publicarse de manera independiente. Cada MFE **DEBE** desplegarse a su entorno (desarrollo, QA, producción) sin afectar a otros. Se **DEBE** soportar rollback de versiones en caso de error.
* **Entornos:** **PUEDE** usarse infraestructura de nube corporativa o servidores de aplicaciones para hospedar los bundles. Por ejemplo, servir cada MFE (y el Shell) en un contenedor web con capacidad de servir el *remoteEntry.js* correspondiente.
* **Versionamiento:** **DEBE** aplicarse versionamiento semántico (SemVer) a cada MFE. Se **DEBE** etiquetar releases con versiones claras para seguimiento.
* **Pruebas:** Se **DEBE** incluir pruebas unitarias y de integración en cada MicroFrontend, de modo que cada cambio individual sea validado. Por ejemplo, usar Jest/RTL para React. Estas pruebas se ejecutan en CI antes del despliegue.
* **Revisión de código:** **DEBE** haber revisiones de código (“code review”) obligatorias entre miembros de diferentes células para asegurar calidad uniforme.
* **Seguridad:** **DEBE** verificarse que las dependencias no autorizadas (no listadas en este estándar) no estén presentes (p.ej. mediante herramientas de análisis de seguridad en el pipeline).
* **Observabilidad:** **DEBERÍA** implementarse registro de errores (monitoring) y métricas por MicroFrontend para detectar y solucionar problemas en producción. Cada MFE puede incorporar herramientas internas de telemetry de ItauOne o soluciones existentes.

Cada pipeline **DEBE** estar integrado al sistema de control de versiones (GitLab) y respetar las políticas de la organización. Los pipelines CI/CD automatizados aseguran que el proceso de construcción, prueba y despliegue sea reproducible y confiable.

## Mantenibilidad del Código

Para garantizar la mantenibilidad del frontend modular, **DEBE** adoptarse las siguientes prácticas:

* **Modularidad:** El código **DEBE** ser modular y reutilizable. Cada componente React **DEBE** corresponder a un solo propósito (una vista, un widget, etc.). Los módulos comunes (p.ej. utilidades, hooks, servicios de API) **DEBERÍAN** compartirse a nivel corporativo o mediante paquetes internos cuando aplique.
* **Documentación:** **DEBERÍA** usarse una herramienta de documentación/componentes (como Storybook) para cada librería de componentes. Cada componente **DEBE** tener documentación de uso básico.
* **Estilo y linters:** **DEBE** aplicarse una convención de estilos consistente (p.ej. ESLint, Prettier con configuración unificada). El pipeline debe validar el estilo para prevenir divergencias entre equipos.
* **Control de versiones:** Todas las dependencias **DEBEN** estar declaradas en package.json y **DEBE** evitarse el uso de dependencias globales. Se **DEBE** actualizar regularmente las dependencias de ItauOne y React/TypeScript según compatibilidad, testear y documentar cada actualización.
* **Optimización:** **DEBERÍA** habilitarse en Webpack optimizaciones como *code splitting*, *tree shaking* y modos de producción para minimizar el tamaño de los bundles resultantes.
* **Estados y datos:** Cada MFE **DEBE** gestionar su propio estado (p.ej. con React Context, Redux, Recoil o similares). No **DEBE** compartirse estado mutable entre MFEs directamente; cualquier intercambio de datos **PUEDE** realizarse mediante el Shell o API central.
* **Reutilización de ItauOne:** No **DEBE** duplicarse la implementación de componentes que ya provee ItauOne. Si un componente no existe, **DEBERÍA** evaluarse extender o solicitar su creación en @itau-one/react. La duplicación innecesaria incrementa el coste de mantenimiento.
* **Pruebas de integración:** **DEBERÍA** probarse la integración en conjunto (end-to-end) del Shell con los MFEs, especialmente en cada release mayor, para asegurar que la federación funciona correctamente.

En resumen, el código **DEBE** mantenerse organizado, documentado y revisado en cada cambio. Estas prácticas reducen la complejidad y facilitan la evolución futura del sistema.

## Referencias Internas

Para detalles adicionales sobre las librerías internas ItauOne, consultar la documentación privada en GitLab de cada paquete (@itau-one/core, @itau-one/icons, @itau-one/react). Asimismo, la *documentación oficial* de Webpack Module Federation y ContentStack provee ejemplos útiles (ver referencias). Estas fuentes sirven de soporte a los lineamientos descritos.