Technical approach

1 message

Narendra Bommisetty <narendra.sudheer@gmail.com> Fri, May 9, 2025 at 10:57 AM
To: Narendra Bommisetty <narendra.sudheer@gmail.com>

Okay, here is a structured Technical Approach Document based on our conversation, designed to guide the integration of the bank's React web components into the vendor's Angular web and Angular/Cordova hybrid mobile applications.

Technical Approach Document: Integration of Bank's React Web Components into Vendor Applications

Version: 1.0 **Date:** May 9, 2025

1. Introduction

- **1.1. Purpose:** This document outlines the technical strategy for the bank's development team to build React-based web components (hereinafter "Bank Components") and for these components to be consumed by two distinct vendor-developed applications:
  - An Angular-based online banking web application (hereinafter "Vendor Web App").
  - An Angular and Apache Cordova-based hybrid mobile application (hereinafter "Vendor Hybrid App").
- **1.2. Overview:** The bank aims to develop specialized financial feature components (e.g., Wire Transfers, Direct Deposits, Account Opening) using ReactJS. These components need to be seamlessly integrated into the vendor's existing application ecosystem, ensuring a consistent user experience, robust performance, and high security.
- **1.3. Key Goals:**
  - **Seamless Integration:** Bank Components should appear as native parts of the vendor applications.
  - **Maintainability:** Both bank and vendor teams should be able to update their respective parts with minimal friction.
  - **Performance:** The integration must not degrade the performance of the vendor applications, especially on mobile devices.
  - **Security:** Uphold the stringent security standards expected of banking applications.
  - **Clear Ownership:** Define clear boundaries for development, deployment, and maintenance.
  - **Reusability:** Bank Components should be built once and consumed by both Vendor Web App and Vendor Hybrid App.

2. Guiding Principles

- **2.1. Clear Ownership & Boundaries:** The bank team owns the Bank Components (development, internal testing, versioning, hosting of assets). The vendor team owns the consumption mechanism within their applications.
- **2.2. Decoupling:** Minimize direct dependencies between the vendor applications and Bank Components. Interactions will occur through well-defined interfaces.
- **2.3. Performance:** Prioritize efficient loading, rendering, and resource utilization.
- **2.4. Security:** Adhere to best practices for secure development, data handling, and communication.
- **2.5. Developer Experience:** Provide clear guidelines and tools for both teams.
- **2.6. Maintainability & Upgradability:** Design for ease of updates and version management.
- **2.7. "Build Once, Consume Everywhere":** Bank Components will be designed for consumption by both web and hybrid platforms, with graceful handling of platform-specific capabilities.

3. Core Integration Strategy: Web Components as Custom Elements

- **3.1. Approach:** Bank Components will be developed using ReactJS and then wrapped as standard **Web Components (Custom Elements)**.
- **3.2. Implementation Details:**
  - The bank team will use libraries (e.g., `react-to-webcomponent` or similar, or native browser APIs where feasible) to transform React components into custom elements (e.g., `<bank-wire-transfer>`).
  - These custom elements will expose a defined API through attributes (for passing data/props from the vendor app) and will emit standard DOM Custom Events (for notifying the vendor app of internal events or data changes).
- **3.3. Benefits:**
  - **Framework Agnostic:** Custom elements are a W3C standard, ensuring compatibility with Angular and other web frameworks.
  - **Encapsulation:** Shadow DOM (recommended) will be used to encapsulate the component's styles and structure, preventing conflicts with the vendor application's styling and DOM.
  - **Clear API Contract:** Attributes and events form a well-understood interface.
  - **Independent Deployment:** The bank can update its components and host them independently. As long as the API contract is maintained (or versioned), vendor applications can consume these updates without immediate code changes on their side.

- **3.4. Applicability:** This strategy is equally applicable to the Vendor Web App (running in a standard browser) and the Vendor Hybrid App (running within a Cordova WebView).

4. Build, Deployment, and Dependency Management

- **4.1. Bank's Component Development & Build Process:**
    - React components are developed and tested by the bank.
    - Components are then wrapped as Custom Elements.
    - Build tools (e.g., Webpack, Rollup) will be used to bundle these custom elements into JavaScript files and associated CSS files.
- **4.2. Dependency Management (React/ReactDOM & Other Shared Libraries):**
    - **Externalization (Recommended):** `react`, `react-dom`, and potentially other significant shared libraries (e.g., a common charting library, UI foundation library) will **not** be bundled into each Bank Component.
    - Bank Components will be built to expect these libraries to be available in the global runtime environment (e.g., `window.React`, `window.ReactDOM`). This is achieved using bundler configurations like Webpack's `externals`.
    - This approach minimizes bundle sizes, prevents library duplication, ensures a single instance of React per page, and improves overall performance.
- **4.3. Manifest File (`bank-components-manifest.json`):**
    - The bank will provide and host a versioned manifest file (JSON format). This file is the central point of reference for the vendor applications to discover and load Bank Components and their dependencies.
    - **Contents:**
        - `sharedDependencies`: An object detailing shared libraries. For each library (e.g., `react`, `react-dom`):
            - `version`: The specific version required (e.g., "18.2.0").
            - `url_bank_cdn`: The recommended URL to load the library from the bank's CDN (e.g., `https://cdn.bank.com/libs/react/18.2.0/react.production.min.js`).
            - `integrity_sri`: Subresource Integrity hash for security.
            - `globalVarName`: The global variable name the library exposes (e.g., "React").
        - `components`: An object detailing each available Bank Component. For each component (e.g., `wire-transfer`):
            - `version`: The component's version (e.g., "v1.2.3").
            - `bundleUrl`: URL to the component's JavaScript bundle on the bank's CDN.
            - `cssUrl`: URL to the component's CSS file (if separate).
            - `dependencies`: An array listing its required `sharedDependencies` keys.
- **4.4. Hosting:**
    - All Bank Component bundles (JS, CSS) and the specified versions of shared libraries will be hosted by the bank on a secure, reliable, and versioned Content Delivery Network (CDN).
- **4.5. Vendor's Consumption Logic:**
    - **Initialization:** At application startup (or before any Bank Component is needed), the vendor application will:
        1. Fetch the `bank-components-manifest.json` from the bank's CDN.
        2. Parse the `sharedDependencies` section. For each entry, check if a compatible version is already loaded. If not, dynamically inject a `<script>` tag into the document to load it from the specified `url_bank_cdn`, using the `integrity_sri` attribute. Ensure these scripts are loaded and ready before proceeding.
    - **Component Loading:** When a specific Bank Component is required for a view:
        1. Consult the `components` section of the fetched manifest to get the `bundleUrl` and `cssUrl` for the required version.
        2. Dynamically load the component's CSS (if applicable) and then its JavaScript bundle.
        3. Once the script is loaded, the custom element (e.g., `<bank-wire-transfer>`) can be instantiated and used in the Angular template.

5. Communication Protocol

- **5.1. Input (Vendor App to Bank Component):** Data and configuration will be passed to Bank Components via HTML attributes on the custom element tag. React components will receive these as props.
- **5.2. Output (Bank Component to Vendor App):** Bank Components will communicate outwards by dispatching standard DOM Custom Events. The vendor's Angular components will listen for these events to react accordingly. Event payloads will carry necessary data.
- **5.3. API Calls from Bank Components:** Bank Components should, where possible, encapsulate their own backend API calls (to bank-owned APIs). If interaction with the host application's backend or services is required, it should be done via the defined event mechanism.

6. Styling

- **6.1. Encapsulation:** Shadow DOM will be utilized for Bank Components to encapsulate their styles and structure, preventing CSS conflicts with the vendor applications.
- **6.2. Theming:** If theming is required, CSS Custom Properties can be defined by the Bank Components and overridden by the vendor applications for consistent look and feel.
- **6.3. Responsiveness:** Bank Components must be designed to be responsive and adapt to various screen sizes encountered in both web and mobile environments.

7. Authentication and Authorization (Leveraging Vendor's Okta via OIDC)

- **7.1. Principle: Single Sign-On (SSO):** Users authenticate once via Okta through the vendor application. Bank Components will not initiate separate Okta authentication flows but will rely on the existing authenticated session.
- **7.2. Vendor's Role (Primary OIDC Client):**
  - The Vendor Web App and Vendor Hybrid App will each be configured as OIDC clients in Okta.
  - They will manage the OIDC authentication flow (Authorization Code Flow with PKCE) with Okta.
  - Upon successful authentication, they will securely obtain and manage an ID Token (for user identity) and Access Tokens (for API access). Secure storage (e.g., Cordova secure storage plugins) is mandatory for the hybrid app.
- **7.3. Propagating Authentication Context to Bank Components:**
  - The vendor application will pass necessary user identity information (derived from the ID Token, e.g., user ID, name, authentication status) to Bank Components via attributes/props.
  - If a Bank Component needs to call a bank-owned backend API, the vendor application will provide a relevant Access Token (obtained from Okta, scoped for the bank's API) to the component, typically as a prop.
- **7.4. Bank's Backend APIs:**
  - Configured as Resource Servers in Okta.
  - Will validate Access Tokens issued by Okta that accompany requests from Bank Components (or any client).
- **7.5. Bank Component Behavior:**
  - Use provided authentication information for UI personalization and conditional rendering of features.
  - Include the provided Access Token in `Authorization: Bearer` headers when calling bank backend APIs.
  - Gracefully handle scenarios where authentication context is not available (e.g., user is not logged in).
- **7.6. Logout:** The logout process will be initiated by the vendor application. This involves clearing local tokens, redirecting to Okta's end-session endpoint (for Single Log-Out), and notifying Bank Components (e.g., by updating props or via an event) to clear any user-specific state.

8. Platform-Specific Considerations

- **8.1. Cordova Native Bridge Interaction (Vendor Hybrid App ONLY):**
  - **Detection:** Bank Components that can leverage native device features will first check for the presence of `window.cordova`.
  - **Communication:**
    1. If native functionality is needed (e.g., camera access), the React component will dispatch a specific DOM Custom Event indicating the requested action and any necessary parameters.
    2. The vendor's Angular shell in the hybrid app will listen for these specific events.
    3. Upon catching such an event, the Angular shell will execute the corresponding Cordova plugin.
    4. Results from the Cordova plugin (success, failure, data) will be communicated back to the Bank Component, typically by updating its attributes/props or, if the custom element's design allows, by calling a method on the element instance.
  - **Graceful Degradation:** In the Vendor Web App (where `window.cordova` is undefined), or if a native feature is unavailable, the Bank Component must degrade gracefully (e.g., hide the feature, offer a web-based alternative like `<input type="file">`).
- **8.2. Performance:** Strict attention must be paid to component bundle sizes, lazy loading strategies, and rendering performance, especially for the Vendor Hybrid App running on potentially resource-constrained mobile devices.
- **8.3. Offline Support (Vendor Hybrid App):** If the hybrid app requires offline functionality, considerations for how Bank Components behave offline (e.g., data caching, UI state) must be addressed. Caching of component assets themselves would typically be managed by the vendor's hybrid app PWA/service worker strategy or Cordova's caching capabilities.

9. Error Handling and Logging

- **9.1. Internal Errors:** Bank Components will implement robust internal error handling (e.g., using React Error Boundaries).
- **9.2. Reporting to Host:** Critical errors that affect the component's ability to function or that require the host application's attention will be communicated via specific DOM Custom Events.
- **9.3. Logging:** Bank Components may implement their own logging. For the Vendor Hybrid App, logs from the WebView can be augmented by calls (via the event bridge) to native logging plugins if deeper or persistent logging is required.

10. Collaboration Model and Documentation

- **10.1. Bank-Provided Documentation:** The bank team will provide comprehensive documentation for each Bank Component, including:
  - Custom element tag name.
  - Version history.
  - Required and optional attributes (props), their types, and purpose.
  - DOM Custom Events emitted, including event name and `detail` payload structure.
  - Any Cordova-specific events it may emit and the expected interaction flow.
  - Styling hooks (e.g., CSS Custom Properties).
- **10.2. Manifest File:** The `bank-components-manifest.json` serves as a key part of the technical contract.
- **10.3. Communication:** Regular technical sync-ups between the bank and vendor development teams.
- **10.4. Testing:**
  - **Bank:** Unit tests, integration tests for components in isolation (as custom elements).
  - **Vendor:** Integration tests within their Angular applications.
  - **Joint:** End-to-end (E2E) testing covering features that utilize Bank Components in both vendor applications.

11. Proof of Concept (PoC) and Next Steps

A phased PoC approach is recommended to validate key aspects of this strategy:

- **11.1. PoC Phase 1: Basic Custom Element Integration:**
    - Bank develops a simple, purely informational React component (e.g., "Display User Balance Placeholder") and packages it as a Custom Element.
    - Vendor team attempts to load and display this custom element in test views within both the Angular Web App and the Angular/Cordova Hybrid App. Test basic prop passing and event listening.
- **11.2. PoC Phase 2: Shared Dependency Loading:**
    - Bank externalizes React/ReactDOM from the PoC component.
    - Bank provides a sample `bank-components-manifest.json` specifying React/ReactDOM as shared dependencies.
    - Vendor team implements logic to parse the manifest and dynamically load React/ReactDOM before loading the PoC component in both applications.
- **11.3. PoC Phase 3: Cordova Native Interaction (Hybrid App):**
    - Bank enhances the PoC component (or creates a new one) to request a simple native action (e.g., show a native toast message).
    - Vendor team (Hybrid App) implements the Angular shell listener to call the Cordova plugin and feed results back. Test graceful degradation in the Web App.
- **11.4. PoC Phase 4: Authentication Context Propagation:**
    - Assuming a mock Okta setup or hardcoded tokens in the vendor's PoC app, test passing user identity props and a mock Access Token to a Bank Component.
    - Bank Component displays user info and simulates an authenticated API call.
- **11.5. Define Standards:** Based on PoC learnings, formalize detailed coding standards, API contracts, versioning strategies, and error handling protocols.
- **11.6. Iterative Development:** Proceed with developing and integrating more complex features following the established standards.

This document provides a comprehensive technical foundation. Successful implementation will rely on strong collaboration, clear communication, and iterative refinement between the bank and vendor teams.