

REACTME PLUGIN ARCHITECTURE

Un'architettura modulare per micro-frontend scalabili
via Web Components e Dynamic Imports.

L'Obiettivo: Disaccoppiare Host e Plugin

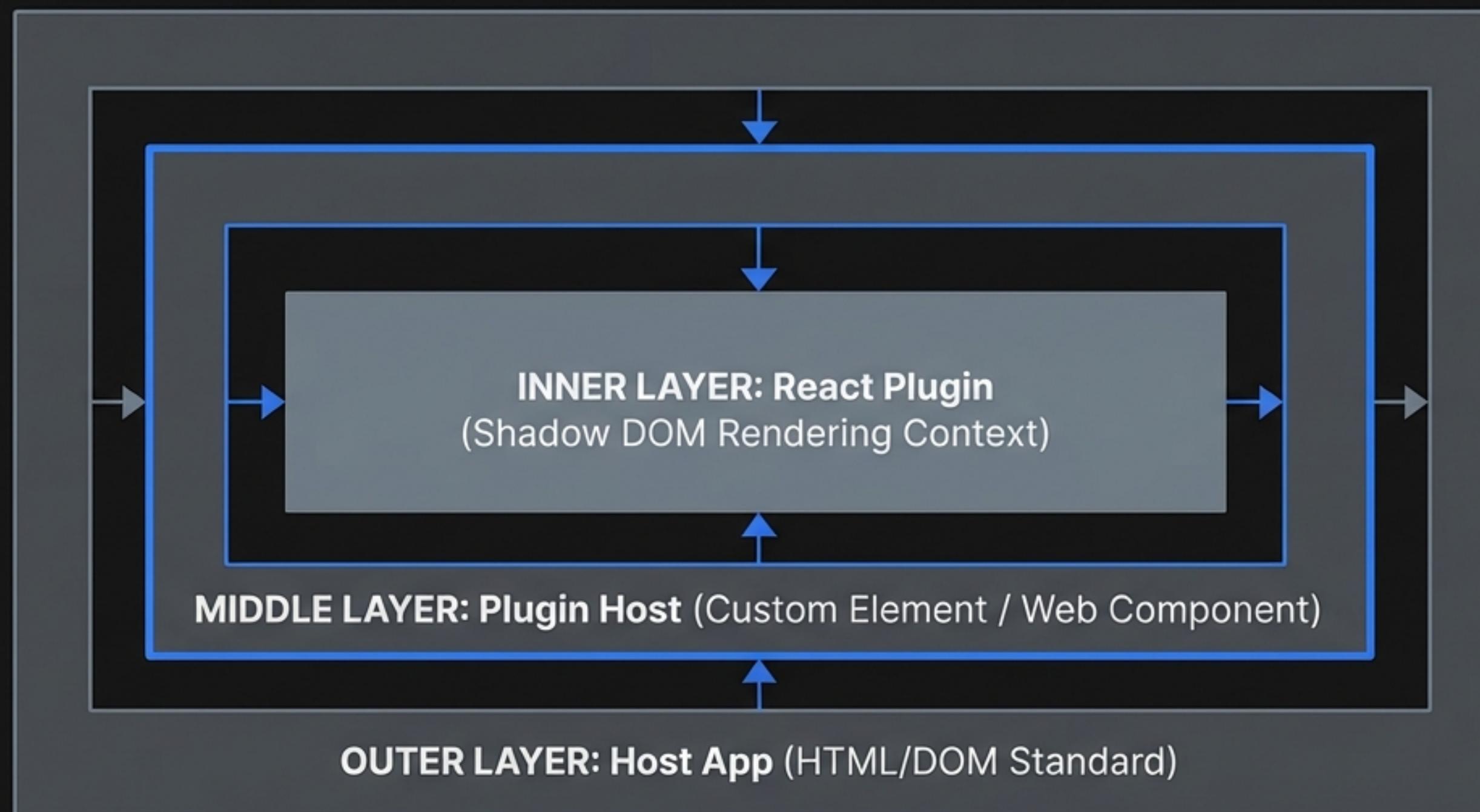


Indipendenza Totale. I plugin sono sviluppati, versionati e distribuiti separatamente dall'applicazione principale.

Caricamento a Runtime (Lazy Loading). L'Host decide quando e quale plugin caricare, senza legami a compile-time.

Isolamento Tecnico & Contratti Tipizzati. Utilizzo di Shadow DOM per evitare conflitti CSS e boundary rigorosa per garantire la Type-safety.

Architettura di Alto Livello: Il Modello a Strati



Insight Chiave: L'Host non collega i plugin in fase di build: li scopre e li compone a runtime. Web Components come unità di isolamento framework-agnostic.

I 5 Pilastri del Sistema

01

Il Contratto.

@acme/plugin-contracts.

Il legislatore che definisce le interfacce.

02

React SDK.

@acme/plugin-react.

Il toolkit per lo sviluppatore.

03

Il Runtime.

@acme/plugin-runtime.

Il ponte logico tra Web Components e React.

04

Host Loader.

L'orchestratore che carica e inietta il contesto.

05

Il Plugin Concreto.

L'implementazione finale della UI.

Attore 1: Il Contratto Condiviso

Definire il confine stabile tra Host e Plugin

Il pacchetto **plugin-contracts** stabilisce i tipi supportati e la struttura del Manifest. È la “verità condivisa” che permette evoluzione indipendente.

```
// packages/plugin-contracts/src/index.ts
export const PLUGIN_TYPES = {
  ROUTE: 'ROUTE',
  WIDGET: 'WIDGET',
  COMMAND: 'COMMAND',
} as const;

export type PluginManifest = {
  type: PT;
  id: string;
  contractVersion: ContractVersion;
  entry: string;
};
```

Attore 2: React SDK per Sviluppatori

API tipizzate per la creazione di plugin.

Offre astrazioni React (Provider/Hooks) per consumare i servizi dell'Host senza dover conoscere la logica di basso livello del DOM.

```
1 // packages/plugin-react/src/index.ts
2 export type PluginDefinition<PT = string> = {
3   type: PT;
4   id: string;
5   Root: ComponentType;
6   activate?: (ctx: PluginContext) => void;
7 };
8
9 export function definePlugin<PT>(def: PluginDefinition<PT>) {
10   return def;
11 }
12 // Hooks: usePluginContext, useServices
```

Attore 3: Il Runtime (The Bridge)

Connessione tra Web Components e React.

Utilizza un GLOBAL_PLUGIN_REGISTRY su globalThis per supportare bundle multipli e risolvere i plugin per ID.

```
1 // packages/plugin-runtime/src/index.ts
2 class PluginElement extends HTMLElement {
3   connectedCallback() {
4     const pluginId = this.getAttribute('plugin-id');
5     // Logica di mount:
6     this.attachShadow({ mode: 'open' });
7     // Mount React nello Shadow DOM
8   }
9 }
10
11 export function registerReactPluginWebComponent({ plugin }) {
12   // Registra il custom element nel browser
13   customElements.define(tag, PluginElement);
14 }
```

Attore 4: Il Caricatore Host

Fetching, Import Dinamico e
Iniezione Context.

L'Host agisce da coordinatore,
scaricando il JSON del manifest e
istanziando il componente corretto nel
DOM.

```
// host/src/PluginLoader.ts
const manifest = await response.json();
// 1. Import dinamico del modulo entry
await import(manifest.entry);

// 2. Creazione del Custom Element
const pluginEl = document.createElement(
  PLUGIN_TAGS[manifest.type]);
pluginEl.setAttribute('plugin-id', manifest.id);

// 3. Iniezione del Contesto (User, Services)
pluginEl.ctx = { user: options.user, services:
  options.services };
```

Attore 5: Il Plugin Concreto

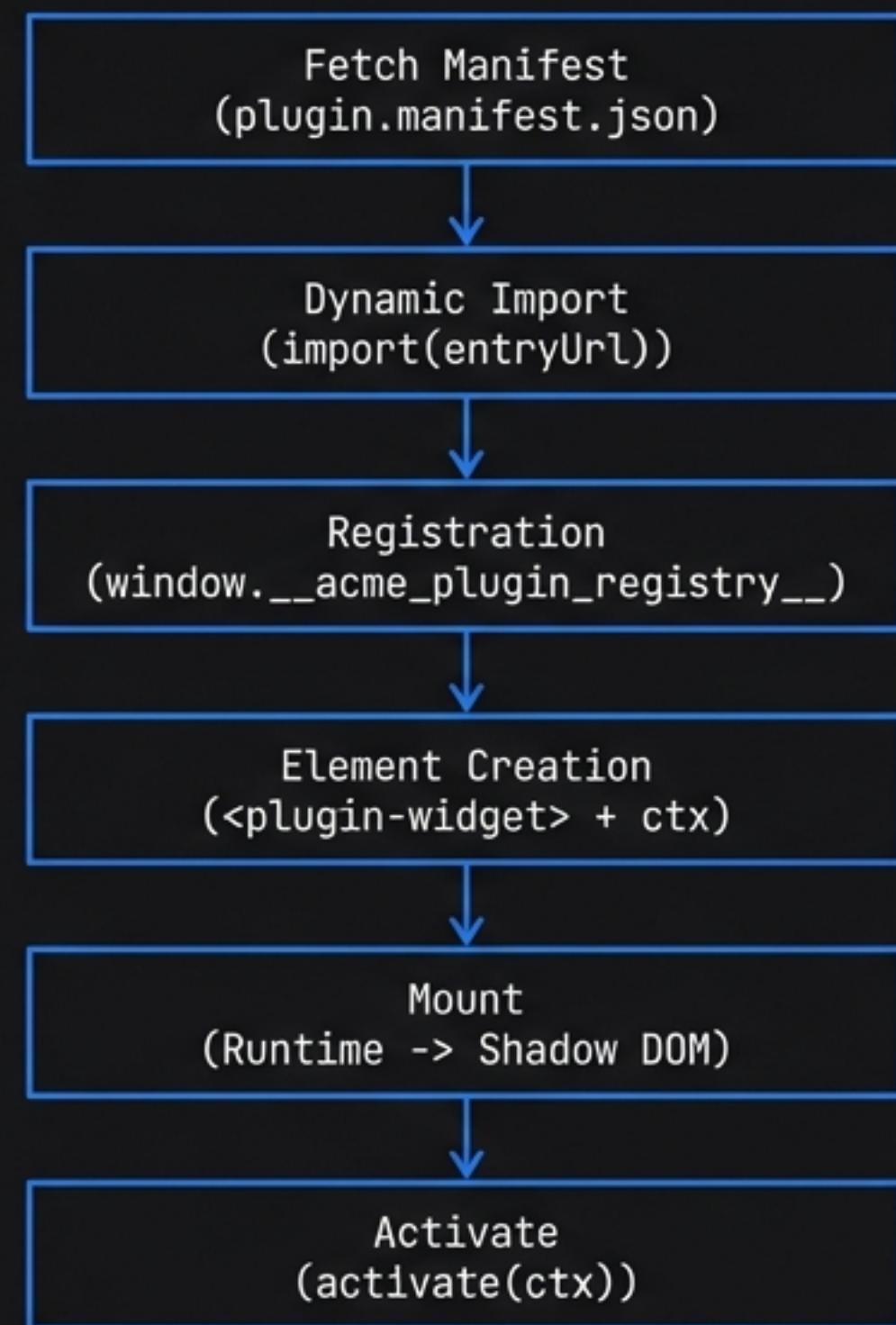
Implementazione della UI e consumo dei servizi.

Il plugin è una normale componente React che “vive” isolata ma può interagire con l’Host tramite i servizi iniettati nel contesto.

```
// plugins/example-plugin/src/PluginRoot.tsx
export function PluginRoot() {
  const services = useServices();
  const user = useUser();

  return (
    <div>
      <h3>Ciao, {user.displayName}</h3>
      <button onClick={() => services.toast.show('Hello!')}>
        Invia Notifica
      </button>
    </div>
  );
}
```

Il Ciclo di Vita (Lifecycle Flow)



Modelli di Distribuzione: Bundled vs Peer

Option A: Self-Contained

945kb

React incluso nel bundle.

- Pro: Zero conflitti.
- Contro: Pesante, duplicazione risorse.

Option B: Peer Dependency

7kb

React esternalizzato.

- Pro: Ultra-leggero, runtime unificato.
- Contro: Configurazione Host (Import Maps).

La Tecnologia Abilitante: Import Maps

Risoluzione nativa dei moduli “Bare Imports”
come fa il browser a capire import React from “react”?

```
<script type='importmap'>
{
  'imports': {
    'react': 'https://cdn.xyz/react.js',
    'react-dom': 'https://cdn.xyz/react-dom.js'
  }
}</script>
```

Mapping name ← → Physical URL

Analisi Tecnica: Vantaggi e Compromessi

Vantaggi

-  **Scalabilità:** Sviluppo distribuito su team diversi.
-  **Performance:** Payload ridotto del **99%** (**Peer dependencies**).
-  **Isolamento:** CSS in Shadow DOM previene regressioni visuali.

Compromessi

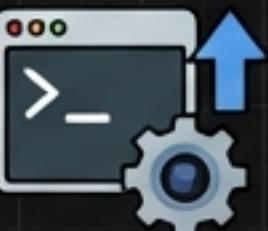
-  **Complessità:** Gestione rigorosa di **Import Maps** e percorsi.
-  **Versionamento:** Necessaria strategia per dipendenze condivise.
-  **Setup:** Configurazione iniziale dell'Host non banale.

Roadmap e Prossimi Passi



Validazione Runtime.

Controllo schema JSON del manifest con errori esplicativi.

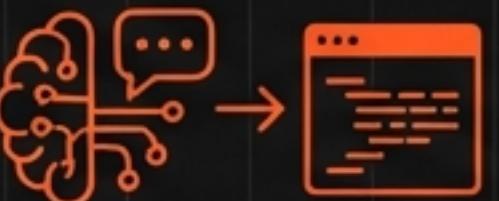


CLI & Scaffolding.

Tooling per generare template di plugin automaticamente.

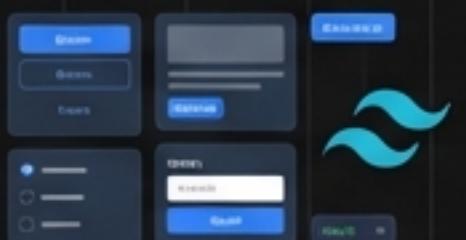
Supporto Multi-Framework.

Integrazione futura con interfacce plugin AngularJS.



AI Generation (MCP).

Creazione automatica di plugin tramite chatbot e Model Context Protocol.



UI Library.

Fornitura di stili e componenti base condivisi (Tailwind).

Conclusione: Un approccio ‘Contract-First’

La stabilità è garantita dal contratto, non dalla co-locazione del codice.

Reacme offre il bilanciamento ideale tra indipendenza di sviluppo e integrazione runtime, sfruttando gli standard moderni (Web Components + Import Maps).

Esplora i pacchetti [@acme/plugin-contracts](#),
[react-sdk](#) e [runtime](#) nel repository.