



Guía Paso a Paso: Crear una dApp con Vite, TypeScript y Sui Move

En esta guía detallaremos cómo crear una dApp **React + TypeScript** utilizando **Vite**, e integrar en ella un contrato inteligente **Move** ya desplegado en la blockchain de **Sui**. Seguiremos un enfoque paso a paso, desde la configuración inicial del proyecto hasta la llamada a funciones del contrato (por ejemplo, funciones del módulo `theron_token` mostrado). Asegúrese de tener las IDs de paquete y objetos de su contrato desplegado, ya que las utilizaremos en la integración.

Paso 1: Inicializar el proyecto con Vite y TypeScript

1. **Crear el proyecto Vite:** Si aún no lo tiene, genere un nuevo proyecto React con Vite. Puede hacerlo desde la terminal con el comando de creación interactiva de Vite. Por ejemplo, ejecute: `npm init vite@latest` y elija **React** como framework y **TypeScript** como variante ¹. Esto creará una plantilla de proyecto básica con React + TS.
2. **Instalar dependencias Sui:** Dentro del directorio del proyecto, instale los paquetes necesarios para interactuar con Sui. En particular usaremos el SDK de Sui TypeScript (`@mysten/sui`) y el kit de desarrollo para dApps (`@mysten/dapp-kit`). También agregaremos React Query para manejo de datos asíncronos. Ejecute:

```
npm install @mysten/sui @mysten/dapp-kit @tanstack/react-query
```

Esto descargará las librerías necesarias ².

1. **Revisar la estructura:** Asegúrese de tener un archivo de entrada (por ejemplo `src/main.tsx` si es React) y un componente principal (`src/App.tsx`). En los siguientes pasos editaremos estos archivos.

Paso 2: Configurar el proveedor de Sui y la red en `main.tsx`

Antes de poder usar la blockchain de Sui, debemos configurar un **cliente Sui** y envolver la aplicación con los **providers** adecuados. Esto nos permitirá conectar wallets y realizar consultas/transacciones.

- **Importar providers:** Abra el archivo de arranque (como `src/main.tsx`). Importe los providers necesarios de dApp Kit y React Query, así como utilidades del SDK de Sui:

```
import { SuiClientProvider, WalletProvider } from '@mysten/dapp-kit';
import { getFullnodeUrl } from '@mysten/sui/client';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import ReactDOM from 'react-dom/client';
```

```
import App from './App';
import React from 'react';
import '@mysten/dapp-kit/dist/index.css'; // estilos por defecto de dApp Kit
```

- **Configurar redes:** Defina qué red de Sui usará su dApp. Por ejemplo, si su contrato está en **Devnet** o **Testnet**, configure esas URLs. Puede crear un objeto de redes, por ejemplo:

```
const queryClient = new QueryClient();
const networks = {
  devnet: { url: getFullnodeUrl('devnet') },
  testnet: { url: getFullnodeUrl('testnet') },
  mainnet: { url: getFullnodeUrl('mainnet') }
};
```

En este objeto hemos establecido las URLs RPC para Devnet, Testnet y Mainnet usando `getFullnodeUrl(...)`. Asegúrese de poner como **red por defecto** aquella donde esté desplegado su contrato (por ejemplo, "testnet" si sus IDs corresponden a Sui Testnet).

- **Envolver la aplicación con los providers:** Utilice `ReactDOM.createRoot(...).render()` para renderizar su aplicación envuelta primero por `QueryClientProvider`, luego por `SuiClientProvider` (indicando las redes y la red por defecto) y finalmente por `WalletProvider`. Esto podría verse así:

```
ReactDOM.createRoot(document.getElementById('root')!).render(
<React.StrictMode>
  <QueryClientProvider client={queryClient}>
    <SuiClientProvider networks={networks} defaultNetwork="testnet">
      <WalletProvider>
        <App />
      </WalletProvider>
    </SuiClientProvider>
  </QueryClientProvider>
</React.StrictMode>
);
```

En el código anterior, `SuiClientProvider` configura el cliente JSON-RPC de Sui apuntando a la red seleccionada (ejemplo: testnet), y `WalletProvider` habilita la conexión con wallets compatibles 3 4.

Paso 3: Conectar un monedero Sui en la dApp

Con la configuración anterior, ya podemos integrar un **wallet** de Sui para que el usuario firme transacciones. El kit de dApp nos proporciona componentes y hooks listos para esto.

- **Agregar botón de conexión:** En su componente principal (por ejemplo `App.tsx`), importe el componente `ConnectButton` de `@mysten/dapp-kit`. Este botón muestra la opción de conectar

un monedero (como Sui Wallet, Ethos, etc.) y, tras la conexión, muestra la dirección del usuario conectado ⁵. Por ejemplo, en `App.tsx`:

```
import { ConnectButton } from '@mysten/dapp-kit';

function App() {
  return (
    <div>
      <header>
        <ConnectButton />
      </header>
      {/* ... el resto de su UI ... */}
    </div>
  );
}
```

Al hacer clic en **Connect Wallet**, se abrirá una ventana/modal para elegir y autorizar un wallet. Una vez conectado, `ConnectButton` automáticamente mostrará la dirección Sui del usuario y permitirá desconectar ⁵.

- **Obtener la cuenta conectada (opcional):** El hook `useCurrentAccount` de dApp Kit permite obtener la dirección actualmente conectada, en caso de necesitar mostrarla o usarla en lógica de la UI ⁶. Ejemplo:

```
import { useCurrentAccount } from '@mysten/dapp-kit';
const account = useCurrentAccount();
if (account) {
  console.log("Cuenta conectada:", account.address);
}
```

Nota: Este paso es opcional si solo necesita la dirección para cosas simples (pues el `ConnectButton` ya la muestra). Sin embargo, es útil para lógicas condicionales o para mostrar objetos del usuario.

Paso 4: Preparar las llamadas a funciones Move del contrato

Ahora viene la parte central: **invocar las funciones** de su contrato Move desde el frontend. Para ello utilizaremos el SDK de Sui para construir un **Programmable Transaction Block** que llame a la función deseada, y luego lo firmaremos con el wallet del usuario.

- **Configurar constantes del contrato:** Dado que el contrato ya está desplegado, use las IDs proporcionadas. Por ejemplo, en un archivo de constantes o en el componente correspondiente, defina:

```
const ONECHAIN_PACKAGE_ID =
"0xee46771b757523af06d19cff029366b81b6716715bea7bb58d0d5013b0e5c73d";
```

```

const THERON_TREASURY_ID =
"0x7a3a35803966fc82d77c5a1f9dd02859b32321a7131a4f7db8dd5542227c00d2";
const THERON_STATS_ID      =
"0x9c97a8b23df2254b648851b30f322571fee31fd346f827c8b2340b3152b41cf8";
// (THERON_METADATA_ID si se necesita, etc.)

```

Use las IDs reales de **su** despliegue (las anteriores son de ejemplo según lo que proporcionó). Estas identifican el paquete principal y los objetos `TreasuryCapHolder` (tesoro) y `TheronStats` necesarios para las llamadas.

- **Construir una transacción MoveCall:** Para invocar una función Move, construimos un objeto `Transaction` (provisto por `@mysten/sui`) y añadimos un comando `tx.moveCall`. Este comando necesita identificar la función a llamar y sus argumentos ⁷.

Supongamos que queremos usar la función de compra `burn_for_purchase` (quema de tokens para comprar un ítem, como *land*). La función Move correspondiente tiene la firma:

```

burn_for_purchase(&mut TreasuryCapHolder, &mut TheronStats, Coin<THERON_TOKEN>,
vector<u8> item_type, &TxContext)

```

Para llamarla desde TypeScript, haremos lo siguiente en una función de nuestro componente (por ejemplo, al hacer clic en un botón "Comprar"):

```

import { Transaction } from '@mysten/sui/transactions';
import { useSignAndExecuteTransaction } from '@mysten/dapp-kit';

const { mutate: signAndExecuteTransaction } = useSignAndExecuteTransaction();

async function comprarLand(coinId: string) {
    // coinId es el ID del objeto Coin<THERON_TOKEN> que el usuario desea gastar
    const tx = new Transaction();
    tx.moveCall({
        target: `${ONECHAIN_PACKAGE_ID}::theron_token::burn_for_purchase`, // Paquete::Módulo::Función
        arguments: [
            tx.object(THERON_TREASURY_ID), // &mut TreasuryCapHolder
            tx.object(THERON_STATS_ID), // &mut TheronStats
            tx.object(coinId), // Coin<THERON_TOKEN> a quemar
            tx.pure.string("land") // vector<u8> indicando el tipo de ítem
        ("land")
    ]
    // typeArguments: [] // (si la función Move tuviera tipos genéricos, aquí se indicarían)
});

```

```
// Enviar la transacción para que el usuario la firme y ejecute
signAndExecuteTransaction(
  { transaction: tx },
  {
    onSuccess: (result) => {
      console.log('Transacción ejecutada. Digest:', result.digest);
    },
    onError: (error) => {
      console.error('Error ejecutando transacción:', error);
    }
  }
);
```

En el código anterior: - Usamos `tx.object(...)` para pasar los **Object IDs** de los objetos on-chain que requiere la función (tal como el tesoro, las estadísticas, y la **moneda del usuario** que será quemada) ⁸. Cada `tx.object(id)` le indica a Sui que incluya ese objeto en la transacción. - Utilizamos `tx.pure.string("land")` para pasar el texto "land" como un vector de bytes (`vector<u8>`). El SDK simplifica la conversión de *strings* comunes a `vector<u8>` mediante este método auxiliar ⁷. (En otras situaciones, puede usar `tx.pure.u64(valor)` para `u64`, `tx.pure.address(addr)` para direcciones, etc.) - La propiedad `target` se construye con el **Package ID**, nombre del módulo Move y nombre de la función, separados por `::`. En este ejemplo, asumimos que el módulo se llama `theron_token` (según el código Move dado) y llamamos a su función `burn_for_purchase`. **Asegúrese** de usar exactamente el nombre del módulo y función tal como están en su paquete Move desplegado. - No pasamos manualmente el `TxContext` - Sui lo gestiona automáticamente al invocar una entry function.

- **Firmar y ejecutar la transacción:** Usamos el hook `useSignAndExecuteTransaction` para enviar el `Transaction` al wallet conectado y ejecutar la llamada. Este hook devuelve una función (`signAndExecuteTransaction`) que al invocarla abrirá el wallet para pedir la firma al usuario y luego transmitirá la transacción a la red ⁹. En el ejemplo arriba, lo llamamos con `{ transaction: tx }` y definimos callbacks de éxito o error. En `onSuccess` podemos obtener el `digest` (ID) de la transacción confirmada, y en caso de error lo manejamos en `onError`.

Nota: Si configuró `defaultNetwork` correctamente en el provider, no es necesario especificar la red (`chain`) en cada llamada de `signAndExecuteTransaction`. Usará la red activa (por ejemplo, Testnet). Si necesitara especificarlo, podría pasar `{ transaction: tx, chain: 'sui:testnet' }` por ejemplo ¹⁰.

- **Invocar la función en la interfaz:** Finalmente, puede conectar esta función a la interfaz de usuario. Por ejemplo, si tiene un botón "Comprar Land", podría hacer:

```
<button onClick={() => comprarLand(idDeMiCoin)}>
  Comprar Land
</button>
```

Donde `idDeMiCoin` es el Object ID de la moneda THERON que el usuario quiere utilizar. (Este ID podrías obtenerlo listando los objetos que posee el usuario con `client.getOwnedObjects` u otro mecanismo, pero suponiendo que ya lo tienes disponible en contexto).

Paso 5: Verificar la ejecución y siguientes pasos

Una vez que el usuario confirme la transacción en su wallet, Sui ejecutará la llamada a la función Move en on-chain. Al completarse, puedes verificar los **efectos** de la transacción: por ejemplo, que el coin se quemó (ya no existirá) y que en el objeto `TheronStats` se haya incrementado el contador correspondiente (`lands_purchased` en este caso). Incluso se habrá emitido un evento `PurchaseMade` según el contrato Move.

Para inspeccionar cambios, puedes:

- Usar los **resultados devueltos** por `signAndExecuteTransaction` (si solicitaste `showObjectChanges` u otras opciones avanzadas en el hook, podrías obtener información detallada de objetos cambiados 11).
- Consultar manualmente la red: por ejemplo, con el SDK (`suiClient.getObject({ id: THERON_STATS_ID })`) para ver los nuevos campos, o usar un explorador de bloques de Sui para ver la transacción y eventos.

Con estos pasos, has construido una dApp básica en Vite/React que se conecta a un wallet Sui y llama a funciones de un contrato Move ya desplegado. *Desde aquí puedes ampliar la lógica de tu frontend para cubrir otras funciones (por ejemplo, `mint_from_hex_burn`, `split_coin`, etc., siguiendo el mismo patrón de `tx.moveCall` con sus respectivos argumentos) e implementar la interfaz de usuario que necesites.*

Referencias utilizadas: La configuración y métodos mostrados siguen las prácticas recomendadas de la documentación oficial de Sui 1 2 3 5 7 9 para asegurar una integración correcta con contratos Move en la blockchain de Sui. ¡Buena suerte con tu desarrollo!

1 2 3 4 5 6 Client App with Sui TypeScript SDK | Sui Documentation

<https://docs.sui.io/guides/developer/sui-101/client-tssdk>

7 8 Sui TypeScript SDK

<https://blockeden.xyz/docs/sui/sui-typescript-sdk/>

9 10 11 useSignAndExecuteTransaction | Myster Labs TypeScript SDK Docs

<https://sdk.mysterlabs.com/dapp-kit/wallet-hooks/useSignAndExecuteTransaction>