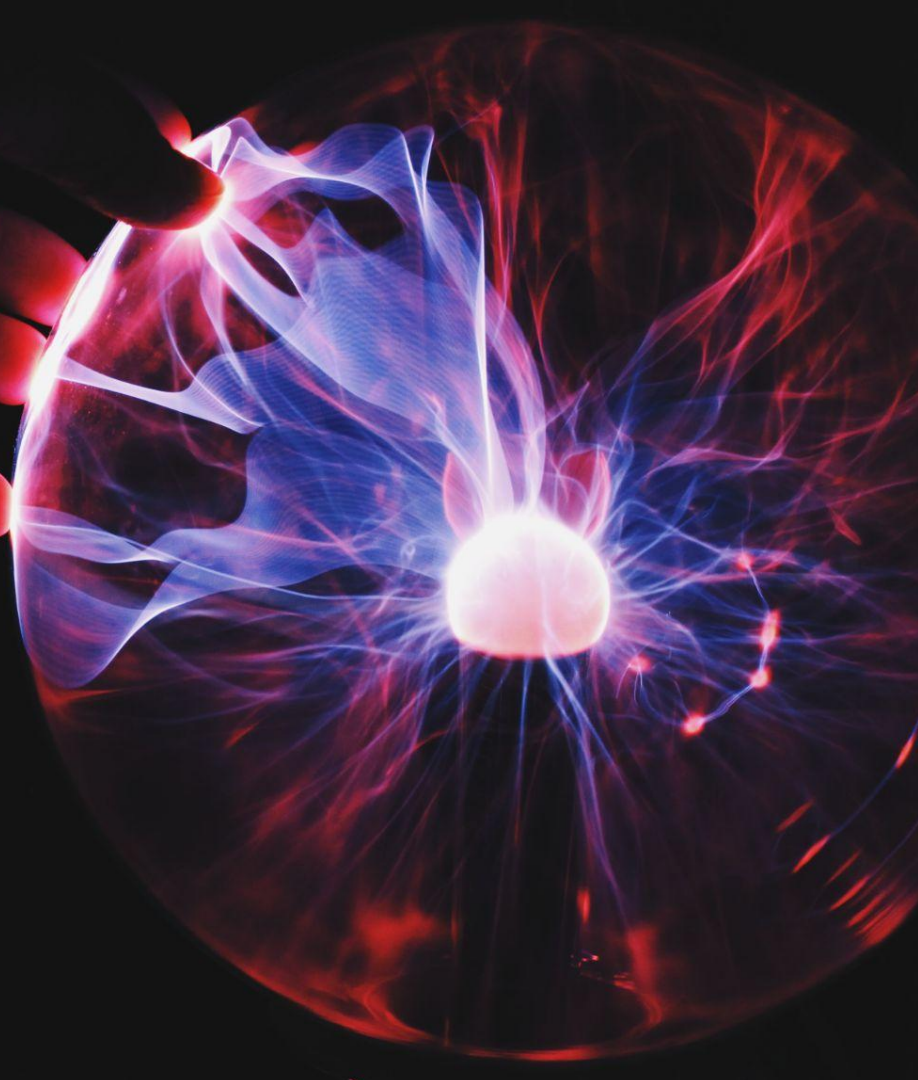
INPUT | OUTPUT

MeshJS y NativeScripts

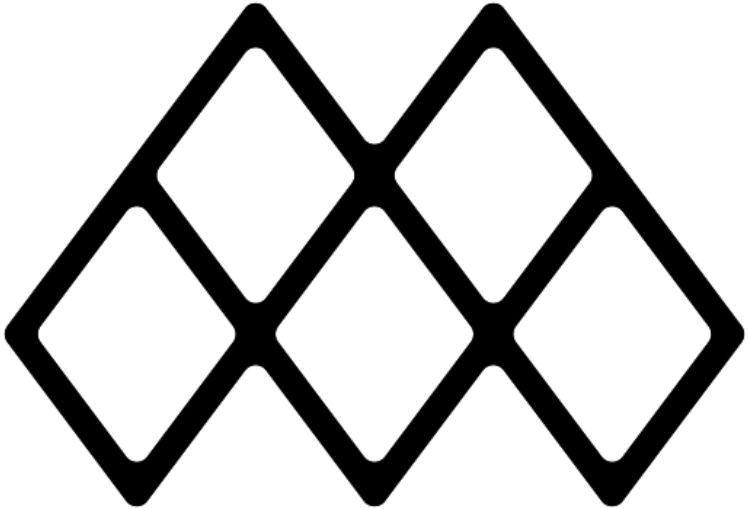


+ MeshJS_

El SDK para el desarrollo web3 en Cardano

MeshJS: Docs

Mesh es una biblioteca de código abierto para el desarrollo de Web3 en Cardano



MeshJS: Compatible con diferentes wallets

También es compatible con wallets como:

- Wallets generadas con cardano-cli
- Wallets con seed phrase
- Clave privada

```
import { AppWallet } from '@meshsdk/core';

const wallet = new AppWallet({
  ...
  key: {
    type: 'cli',
    payment: 'CLIGeneratedPaymentCborHexHere',
    stake: 'optionalCLIGeneratedStakeCborHexHere',
  },
});
```

MeshJS: Componentes de React

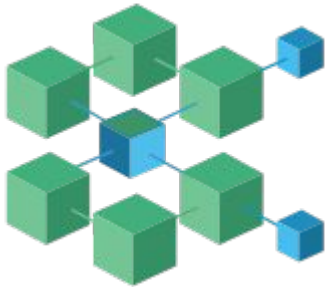
Viene con componentes de ReactJS para facilitar el desarrollo aplicaciones web

```
import { CardanoWallet } from '@meshsdk/react';

export default function Page() {
  return (
    <>
      <CardanoWallet />
    </>
  );
}
```

MeshJS: Proveedores

- Soporta diferentes proveedores para consultar la Blockchain.
- Los proveedores son servicios que facilitan la interacción con Cardano (consultas y envío de transacciones) sin la necesidad de correr un nodo que puede ser costoso.
- Al abstraer la API del proveedor, uno puede cambiar fácil (no vendor lock-in).



Maestro

OGMIOS

BLOCKFROST

Proveedores: Ejemplos

Maestro: `new MaestroProvider(maestro_config);`

Blockfrost: `new BlockfrostProvider('<BLOCKFROST_API_KEY>');`

Koios: `new KoiosProvider(baseUrl: [string](/types/string))`

MeshJS: Docs

LISTO!

Ahora podemos usar funciones y componentes siguiendo los docs en:

https://meshjs.dev/

+ TRANSACCIONES SIMPLES _

Billetera: Construcción

Para construir una billetera en MeshJS se necesita por lo menos un provider:

```
import { MeshWallet } from '@meshsdk/core';

const blockchainProvider = new MaestroProvider(maestro_config);

const wallet = new MeshWallet({
  networkId: 0,
  fetcher: blockchainProvider,
  submitter: blockchainProvider,
  key: {
    type: 'root',
    bech32: 'xprv…eqgu',
  },
});
```

Billetera: Fondear billetera

Para fondear una billetera tenemos que tener una dirección de cardano, lo hacemos de la siguiente manera:

```
Const address  = await wallet.getChangeAddress();
console.log(address);
// addr_test1qpzu…fskn
```

Después copiar la dirección en https://docs.cardano.org/cardano-testnets/tools/faucet/ para poder fondearla

Simples transacciones: Transacción - Importamos `Transaction`

```
import { Transaction } from '@meshsdk/core';
const tx = new Transaction({initiator: wallet})
tx.sendLovelace(“addr1a…”,"1000000")
tx.sendLovelace(“addr1b…”,"1000000")
const unsignedTx = await tx.build();
const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);
```

Simples transacciones: Transacción - Creamos instancia `Transaction`

```
import { Transaction } from '@meshsdk/core';
const tx = new Transaction({initiator: wallet})
tx.sendLovelace(“addr1a…”,"1000000")
tx.sendLovelace(“addr1b…”,"1000000")
const unsignedTx = await tx.build();
const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);
```

Simples transacciones: Transacción - Agregamos instrucción para enviar ADA

```
import { Transaction } from '@meshsdk/core';
const tx = new Transaction({initiator: wallet})
tx.sendLovelace(“addr1a…”,"1000000")
tx.sendLovelace(“addr1b…”,"1000000")
const unsignedTx = await tx.build();
const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);
```

Simples transacciones: Transacción - Se construye y firma la transacción

```
import { Transaction } from '@meshsdk/core';
const tx = new Transaction({initiator: wallet})
tx.sendLovelace("addr1a…","1000000")
tx.sendLovelace("addr1b…","1000000")
const unsignedTx = await tx.build();
const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);
```

Simples transacciones: Transacción - Se manda la transacción

```
import { Transaction } from '@meshsdk/core';
const tx = new Transaction({initiator: wallet})
tx.sendLovelace(“addr1a…”,"1000000")
tx.sendLovelace(“addr1b…”,"1000000")
const unsignedTx = await tx.build();
const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);
```

Simples transacciones: Transacción

```
import { Transaction } from '@meshsdk/core';
const tx = new Transaction({initiator: wallet})
tx.sendLovelace(“addr1a…”,"1000000")
tx.sendLovelace(“addr1b…”,"1000000")
const unsignedTx = await tx.build();
const signedTx = await
wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);
```

Simples transacciones: Transacción con assets

```
tx.sendLovelace("addr_test1b…","1000000")
```



```
tx.sendAssets(
    'addr_test1v',
    [
      {
            unit:
'd9c43…5d06e',
            quantity: '1',
        },
    ]
);
```

+ NATIVE SCRIPTS _

Native scripts: ¿Qué son?

1. **DSL** Turing incompleto con sintaxis basada en **JSON** que define las reglas para gastar un **UTxO** o para minar nuevos tokens.
2. Se pueden combinar cada una de las siguientes reglas en un script: `sig`, `after`, `before`, `atLeast`, `any`, `all`.

Native scripts: ¿Qué son?

`sig`: La transacción tiene que ser firmada por dicha firma para que sea válida.

`atLeast`: La transacción tiene que cumplir con por lo menos una cantidad definida de scripts.

`any`: La transacción tiene que cumplir con por lo menos un script.

`all`: La transacción tiene que cumplir con todos los scripts.

`after`: La transacción tiene que ser ejecutada después de un slot.

`before`: La transacción tiene que ser ejecutada antes de un slot.

Native scripts: sig

Se tiene que definir el **keyHash** de la dirección que deseamos que firme.

```
{
  "type": "sig",
  "keyHash": "e09d36c79dec9bd1b3d9e152247701cd0bb860b5ebfd1de8abb6735a"
}
```

Native scripts: atLeast

```
{
  "type": "atLeast",
  "required": 2,
  "scripts":
  [
    {
      "type": "sig",
      "keyHash": "2f3d4cf10d0471a1db9f2d2907de867968c27bca6272f062cd1c2413"
    },
    {
      "type": "sig",
      "keyHash": "f856c0c5839bab22673747d53f1ae9eed84afafb085f086e8e988614"
    },
    {
      "type": "sig",
      "keyHash": "b275b08c999097247f7c17e77007c7010cd19f20cc086ad99d398538"
    },
  ]
}
```

Native scripts: any

```
{
  "type": "any",
  "scripts":
  [
    {
      "type": "sig",
      "keyHash": "d92b712d1882c3b0f75b6f677e0b2cbef4fbc8b8121bb9dde324ff09"
    },
    {
      "type": "sig",
      "keyHash": "4d780ed1bfc88cbd4da3f48de91fe728c3530d662564bf5a284b5321"
    },
    {
      "type": "sig",
      "keyHash": "3a94d6d4e786a3f5d439939cafc0536f6abc324fb8404084d6034bf8"
    }
  ]
}
```

Native scripts: all

```
{
  "type": "all",
  "scripts":
  [
    {
      "type": "sig",
      "keyHash": "e09d36c79dec9bd1b3d9e152247701cd0bb860b5ebfd1de8abb6735a"
    },
    {
      "type": "sig",
      "keyHash": "a687dcc24e00dd3caafbeb5e68f97ca8ef269cb6fe971345eb951756"
    },
    {
      "type": "sig",
      "keyHash": "0bd1d702b2e6188fe0857a6dc7ffb0675229bab58c86638ffa87ed6d"
    },
  ]
}
```

Native scripts: before and after

```
{
   "type": "all",
   "scripts":
   [
      {
         "type": "after",
         "slot": 1000
      },
      {
         "type": "before",
         "slot": 2000
      }
   ]
 }
```

+ MULTISIGNATURE _

Ejercicios: Requerimientos

Multisignature

- 3 de 5 personas tienen que firmar

Vesting

- La persona X firma después de una fecha dada

Multisignature: Código

```
{
  "type": "atLeast",
  "required": 3,
  "scripts":
  [
    {
      "type": "sig",
      "keyHash": "78489d89f029b893c8d2f2be4152b4a27ba7e6cae9f594b3f073ae09"
    },
    {
      "type": "sig",
      "keyHash": "d3fde93434c1f4f2eb0b6db8719e43df403b3e326e67f3aab62f2dd0"
    },
    {
      "type": "sig",
      "keyHash": "c3a0bd312ba1fc7a4c09ca2e5ee8ea31d2970a12029722bde85d3e45"
    },
    {
      "type": "sig",
      "keyHash": "81a8ec65f2082422cdf7651d52a326c06e2e8af2bd18ba39d7d4731d"
    },
    {
      "type": "sig",
      "keyHash": "131ab636f9a92333fc866a53508c24b81a3f440481f2fcc3b9917736"
    }
  ]
}
```

Vesting: Código

```
{
   "type": "all",
   "scripts":
   [
      {
         "type": "sig",
         "keyHash": “d3fde93434c1f4f2eb0b6db8719e43df403b3e326e67f3aab62f2dd0”
      },
      {
         "type": "after",
         "slot": “52593306”
      }
   ]
 }
```

+ Native tokens_

Native tokens: ¿Qué son los Native tokens?

Son tokens dentro de **Cardano** que pueden representar activos como **Stablecoins**, criptomonedas, activos físicos. Utilizando **políticas monetarias** ya sea con Plutus o **Native Scripts** se pueden definir las reglas para la emisión de estos tokens.

Son nativos porque funcionan de manera orgánica con la red de Cardano ya que en una transacción se envían de la misma manera que **ADA**.

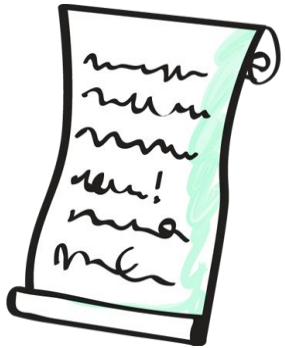
Native tokens: Características

- **Integración nativa:** Los Native tokens pueden ser transferidos de la misma manera en la que las ADAs son transferidas.
- **Sin necesidad de contratos inteligentes:** No se necesita ejecutar un contrato inteligente para poder transferir estos tokens, lo que significa que es más barato para transferir.
- **Diversidad de usos:** Por medio de contratos inteligentes su emisión puede representar una amplia variedad de activos.
- **Soporte de multi activos:** Una sola transacción puede soportar el envío de múltiples tipos de activos, incluyendo ADA.

Native tokens: Emisión

Primero se escriben las reglas de los nuevos tokens:

- ¿Quién puede emitirlos?
- ¿Cuándo se pueden emitir?
- ¿Cuál es el nombre del token?
- ¿Cuántos tokens se van a emitir?



Native tokens: Emisión

Después se crea una transacción que crea los nuevos **tokens**, esta transacción tiene que contener las reglas que ya hemos creado.

Native tokens: Emisión

Tenemos nuevos tokens que pueden ser transferidos entre los usuarios de Cardano.

¿Preguntas?