Blockchain: Desarrollo de Aplicaciones

Contador con React, Redux y Drizzle
Caso de Estudio #5

BCDA 2018

Versión: 2018-12-08

© Santiago Pavón - UPM-DIT

Índice

- ~~Introducción, documentación.~~
- ~~Ganache~~
- ~~Caso de Estudio #1: SC Contador~~
  - ~~Crear un proyecto que usa un Smart Contract llamado Contador.~~
  - ~~Usando Truffle, crear pruebas, scripts, cmds y una dapp.~~
- ~~Caso de Estudio #2: Webpack~~
  - ~~Añadir Webpack al caso de estudio #1~~
- ~~Caso de Estudio #3: Introducción a React y Redux~~
  - ~~Crear un ejemplo de una SPA con un nuevo contador.~~
    - ~~El contador es diferente al de los casos de estudio anteriores: Sin Blockchain.~~
    - ~~Partiendo de la configuración de Webpack del caso de estudio #2 y ampliarla para soportar react.~~
- ~~Caso de Estudio #4: Smart Contract Contador con React y Redux. Y SIN Drizzle.~~
  - ~~Unir todo lo visto en los casos de estudio anteriores.~~
- **Caso de Estudio #5: SC Contador con React, Redux y CON Drizzle.**
  - **Añadir Drizzle al caso de estudio #4.**

© Santiago Pavón - UPM-DIT

Introducción a Drizzle

- Página oficial:
  `https://truffleframework.com/drizzle`
- Drizzle usa un store de Redux para almacenar el estado de todo lo que pasa en la red Ethereum:
  - Mantiene sincronizado su estado con el estado de los contratos,
  - Gestiona las transacciones y las llamadas que realizamos,
  - etc...
- Puede usarse también todo lo ofrecido por web3.
- Podemos usar nuestro propio store mezclándolo con el que genera drizzle, o usar solo el de Drizzle si no tenemos otros datos, o tener varios stores separados.
- Proporciona un módulo para usarlo facilmente con React.
  - Proporciona un componente provider, una función compose, etc...
- Proporciona una librería de componentes React
  - ContractData, ContractForm, LoadingContainer.

© Santiago Pavón - UPM-DIT

- `drizzle`: The core library responsible for web3, account and contract instantiation; wiring up the necessary synchronizations and providing additional contract functionality.
- `drizzle-react`: Provides a DrizzleProvider component and drizzleConnect helper method to make it easier to connect Drizzle with your React app.
- `drizzle-react-components`: A library of useful components for common dapp functions. Currently includes ContractData, ContractForm and LoadingContainer.

https://truffleframework.com/drizzle

© Santiago Pavón - UPM-DIT

¿Cómo se mantienen los datos actualizados?

- Drizzle instancia web3 y nuestros contratos.
- Drizzle observa la cadena de bloques subscribiéndose para detectar cuándo llegan bloques nuevos.

© Santiago Pavón - UPM-DIT

- Drizzle se apunta cuales son las llamada (call) que le interesan a la dapp para saber qué debe sincronizar.

- Cuando hay un nuevo bloque, Drizzle comprueba que no este pendiente, y entonces examina las transacciones que contiene para ver si alguna de ellas cambia nuestros contratos.

© Santiago Pavón - UPM-DIT

- Si alguna transacción modifica nuestros contratos, entonces se ejecutan otra vez todas las llamadas call para actualizar los valores calculados que le interesaban a la dapp.

© Santiago Pavón - UPM-DIT

Objeto Drizzle

- En nuestros programas hay que crear un objeto Drizzle.
  - Este objeto contiene las instancias de los contratos y el store con el estado.
- Se crea así:

```
const drizzle = new Drizzle(options, drizzleStore);
```

  - Toma como parámetros opciones de configuración y un store de Drizzle.
- El objeto Drizzle crea las instancias de los contratos inteligentes que se especifican en las opciones.
  - Estas instancias se usarán para lanzar nuevas transacciones o hacer llamadas call a los contratos.
  - El objeto drizzle crea una propiedad para cada instancia de contrato creado.
    - El nombre de la propiedad es el nombre del contrato.
- El objeto Drizzle también mantiene un store (Redux) en su propiedad **store**.
  - Este store contiene el estado de Drizzle.
  - El estado contiene información sobre el proceso de inicialización de Drizzle, las cuentas existentes y sus balances, el valor devuelto por las llamadas call, el estado de las transacciones realizadas, etc.

© Santiago Pavón - UPM-DIT

Opciones de Configuración

- Esta configuración especifica qué debe hacer Drizzle.
- Es un objeto JSON con los siguientes campos:
  - **`contracts`**: Un array con los artefactos de los contratos que nos interesan.
  - **`events`**: Nombres de los eventos (agrupados por contrato) que queremos atender para sincronizar el store.
    - Es un objeto clave/valor donde la clave es el nombre de contrato, y el valor asociado es un array con los nombres de los eventos.
  - **`polls`**: Objeto clave/valor para indicar cada cuanto tiempo se sondea la red para detectar cambios en las cuentas (y balances) y la existencia de nuevos bloques.
    - La clave puede ser **`blocks`** o **`accounts`**.
    - El valor es el intervalo de sondeo en milisegundos.
  - **`syncAlways`**: booleano para indicar que se reejecuten todas las llamadas (call) de los contratos cada vez que llega un nuevo bloque. Necesario cuando un contrato proxy ofusca la dirección de nuestro contrato y no se detecta que transacciones afectan a nuestro contrato.
  - **`web3`**: opciones para instanciar web3 cuando no se inyecta por MetaMask, Mist, u otro.
- Nota: en la documentación de Drizzle hay más detalles sobre los campos anteriores.

© Santiago Pavón - UPM-DIT

- Ejemplo:

```
import Contador from './build/contracts/Contador.json';

const drizzleOptions = {
    contracts: [ Contador ],
    events: {
        Contador: [ 'Tic' ]
    },
    polls: {
        blocks: 3000,
        accounts: 3000
    },
    web3: {
       fallback: {
            type: 'ws',
            url: 'ws://127.0.0.1:7545'
        }
    }
};
```

© Santiago Pavón - UPM-DIT

El Estado de Drizzle

- El estado de Drizzle mantiene actualizada información de estado de web3, de las instanciaciones de los contratos, de las llamadas call y transacciones realizadas, de las cuentas y balances, ...
- Es el estado del store del objeto drizzle:

```
const drizzleState = drizzle.store.getState();
```

- Es un objeto JSON con las siguientes claves y valores:
  - **`web3`**: El estado de la instanciación de web3.
    - Valores posibles: "initializing", "initialized" y "failed".
    - Útil para mostrar warnings si falla la instanciación de web3.
  - **`drizzleStatus`**: Objeto con información sobre el status de Drizzle.
    - Es un objeto clave/valor.
    - Clave **`initialized`**: El valor es **false** hasta que web3 está listo, se han guardado todas las direcciones en el estado, y todos los contratos se han instanciado, que pasa a **true**.
  - **`accounts`**: Array con las direcciones proporcionadas por web3.
  - **`accountBalances`**: Objeto clave/valor donde las claves son las direcciones de la cuentas y el valor es su balance en weis.

© Santiago Pavón - UPM-DIT

- **`contracts`**: Objeto clave/valor donde las claves son los nombres de los contratos, y el valor es el estado del contrato.
  - El estado del contrato es un objeto clave/valor, con las siguientes claves:
    - **initialized**: true cuando el contrato se ha instanciado.
    - **synced**: true cuando el contrato está sincronizado. false cuando se produce un cambio de estado en un bloque y drizzle está reejecutando sus llamadas call.
    - **events**: array de objetos evento.
      - Drizzle solo escucha los eventos declarados en las opciones.
    - **<Nombre_de_una_Función>**: Es el nombre de una función del contrato, que es de tipo constante, y que ha sido llamada.
      - Se crea una clave **`<Nombre_de_una_Función>`** para todas las funciones del contrato que se llamen.
      - Para cada llamada a la función se guardan los diferentes argumentos usados en la llamada, y el resultado devuelto por la función.
        - Se calcula el hash de los argumentos usados en la llamada a la función, y se guardan los argumentos y el resultado devuelto por la funcion.
          - Si cambia el estado porque llega un nuevo bloque, se llama otra vez a las funciones para recalcular los resultados devueltos y así mantenerlos siempre actualizados.
        - Si la llamada a la función no tiene argumentos, se usa 0x0 como hash.

© Santiago Pavón - UPM-DIT

- **`transactionStack`**: Pila (es un array) donde se guarda el hash de las transacciones solicitadas.
  - Las transacciones se crean con la función **cacheSend**, que devuelve un **stackId**.
    - **stackId** es el índice o posición en la pila (array) donde se ha guardado el hash de la transacción.
      - Nota: el hash de la transacción no se crea hasta que la transacción se envía a la red, por lo que inicialmente se guarda un string vacío en la pila.
- **`transactions`**: Objetos con el estado de las transacciones creadas, indexadas por el hash de la transacción.
  - Para cada transacción se crea un objeto clave/valor, con las siguientes claves:
    - **receipt**: Primer objeto con la transacción receipt recibido de transacción realizada son éxito.
    - **confirmations**: Array con los siguientes objetos receipt recibidos, hasta el número 24.
    - **error**: Error producido, si se ha producido.
    - **status**:
      - "**pending**" si la transacción se se enviado con éxito a la red, pero no ha sido minada aun.
      - "**success**" cuando se ha recibido un objeto receipt de la transacción.
      - "**error"**: si se ha producido algún error después de enviar la transacción a la red.

© Santiago Pavón - UPM-DIT

Instancia drizzle vs drizzleState

- drizzleState se usa para consultar información sobre el estado de las cosas:
  - Cuál es el valor de retorno de las llamadas call solicitadas previamente.
  - Cuál es el estado de las transacciones realizadas.
  - Qué cuentas hay.
  - Cuál es el balance de las cuentas,
  - etc.
- La instancia drizzle se usa para lanzar los trabajos:
  - Hacer las llamadas call de los métodos de los contratos.
  - Crear nuevas transacciones.
  - Ejecutar las funcionalidades ofrecidas por web3.
  - etc.

© Santiago Pavón - UPM-DIT

Llamadas call

- Antes de poder realizar llamadas call a los métodos de los contratos, Drizzle debe haberse inicializado:

```
let state = drizzle.store.getState();
if (state.drizzleStatus.initialized) { ESTA INICIALIZADO }
```

- Para realizar una llamada call a un método de un contrato se ejecuta:

```
const dataKey = drizzle.contracts.CONTRATO.methods.METODO.cacheCall(ARGS)
```

  - donde **CONTRATO** es el nombre del contrato, **METODO** es el nombre del método a llamar, y **ARGS** son los argumentos de la llamada al método.

- Para obtener el valor devuelto por el método llamado se tiene que usar la clave **dataKey** devuelta por **cacheCall**:

```
let state = drizzle.store.getState();
state.contracts.CONTRATO.methods.METODO[dataKey].value
```

- Drizzle guarda cuales son las llamadas call que se han realizado.
  - Y cuando se reciben nuevos bloques que cambian el contrato, Drizzle actualiza automáticamente el store con los nuevos valores que devuelven ahora las mismas llamadas call.

- Para hacer una llamada call que no se almacene en el store pueden usarse los métodos estándar de web3:

```
drizzle.contracts.CONTRATO.methods.METODO().call(ARGS)
```

© Santiago Pavón - UPM-DIT

Lanzar Transacciones

- Antes de poder realizar llamadas call a los métodos de los contratos, Drizzle debe haberse inicializado:

```
let state = drizzle.store.getState();
if (state.drizzleStatus.initialized) { ESTA INICIALIZADO }
```

- Para realizar una transacción se ejecuta:

```
const stackId = drizzle.contracts.CONTRATO.methods.
    METODO.cacheSend(ARGS, {from: '0x2a..', gas: ...})
```

  - donde **CONTRATO** es el nombre del contrato, **METODO** es el nombre del método a invocar, **ARGS** son los argumentos de la llamada al método, como último argumento opcional estan las opciones típicas de las transacciones.

- **cacheSend** devuelve **stackId**, que es un índice de la pila **state.transactionStack**.
  - Ese índice es la posición de la pila donde se guarda el hash de la transacción.
  - Con el hash se puede acceder al estado de la transacción usando **state.transactions**.
  - Consultar los detalles en la descripción del estado de de Drizzle visto anteriormente

- Para lanzar una transacción que no se almacene en el store pueden usarse los métodos estándar de web3:

```
drizzle.contracts.CONTRATO.methods.METODO(ARGS).send({from:
'0x2a..', ...})
```

© Santiago Pavón - UPM-DIT

Añadir Contratos

- Pueden añadirse nuevos contratos a Drizzle programáticamente.
  - Por ejemplo, cuando se crean nuevas instancias dinámicamente.
- Para añadirlos se puede usar la acción ADD_CONTRACT o el metodo drizzle.addContract().

```
var contractConfig = {
  contractName: "0x12640891234d5Ed1bacdcAA1786bbda351cDaB57",
  web3Contract: new web3.eth.Contract(/* ... */)
}
events = ['Tic']

// Usando una accion:
dispatch({type: 'ADD_CONTRACT', drizzle,
          contractConfig, events, web3})

// Usando drizzle.addContract:
this.context.drizzle.addContract(contractConfig, events)
```

© Santiago Pavón - UPM-DIT

Caso de Estudio #5

Paso 1

© Santiago Pavón - UPM-DIT

Caso de Estudio #5

- Contador usando Truffle y Drizzle.
- Consiste en:
  - Usar Truffle para compilar el smart contract Contador que desarrollamos en el caso de estudio #1 y desplegarlo en Ganache.
  - Crear la aplicación web de cliente con create-react-app.
    - Usaremos los mismos componentes React del paso 1 de caso de estudio #3.
  - Usar Drizzle para gestionar la cadena de bloques:
    - Instanciar contratos, acceder al estado, y realizar llamadas call y transacciones.
  - Hacer que los componentes React usen Drizzle para actualizarse o hacer modificaciones.

© Santiago Pavón - UPM-DIT

Instalaciones

- Ya debemos de tener instalado de los temas anteriores:
  - node y npm
  - truffle
  - ganache, ganache-cli
- Necesitamos instalar Create-React-App
  - Utilidad que instala todo lo necesario para desarrollar aplicaciones React.
    - Principalmente WebPack

```
$ sudo npm install -g create-react-app
```

© Santiago Pavón - UPM-DIT

Crear un Proyecto

- Nota: Partimos de lo que hicimos en los casos de estudio vistos en temas anteriores.

- Crear un proyecto React llamado caso5:
  - Creamos una primera version en paso1:

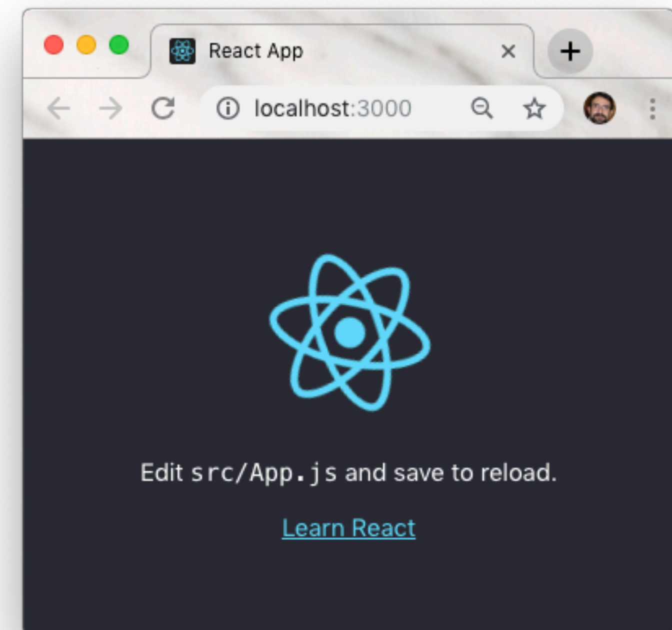
```
$ mkdir caso5 ; cd caso5
$ create-react-app paso1
$ cd paso1
```

- Para lanzar el servidor de desarrollo:

```
$ npm start
```

  - y conectarnos con un navegador a:

```
http://localhost:3000
```



© Santiago Pavón - UPM-DIT

Detalles

La utilidad create-react-app añade los siguientes scripts en package.json:

- npm **start**
  - Lanza el servidor de desarrollo.

- npm run **build**
  - Empaqueta la aplicación en ficheros estáticos listos para poner en producción.

- npm **test**
  - Lanzar los test.

- npm run **eject**
  - Se usa para dejar de usar las facilidades que proporciona create-react-app.
    - El desarrollador continuará usando sus configuraciones y utilidades personales.
  - Se eliminan esta utilidad, y se copian en el directorio de la aplicación las dependencias build, los ficheros de configuración, y los scripts.
    - Para que el desarrollador los cambie a su gusto y continue por su cuenta.
  - Si se ejecuta este script, no se puede dar marcha atrás para deshacer los cambios y volver a la situación inicial.

© Santiago Pavón - UPM-DIT

- Copiamos los contratos, las migraciones y la configuración de truffle que hemos estado usando desde el caso de estudio 1.

```
$ cp -r ....../caso1/contracts .
$ cp -r ....../caso1/migrations .
$ cp ....../caso1/truffle.js .
```

- Lanzamos ganache.
  - Conviene configurarlo para que tarde varios segundos en minar los bloques, y así poder ver más lentamente cómo se realizan las acciones asíncronas (redux-thunk) que crearemos más adelante, y cómo va cambiando el estado del store.
  - También podemos lanzar "**`ganache-cli -b 3`**" para conseguir lo mismo.

- Compilar y desplegar:

```
$ truffle compile
$ truffle migrate
```

- Eliminar la aplicación web creada por create-react-app como ejemplo:
  - Renombramos el directorio src y lo creamos de nuevo:

```
$ mv src src.pslm ; mkdir src ; cd src
```

- Hacer un enlace símbolico desde dentro de `src` hasta `../build/contracts` para que WebPack pueda empaquetar los artefactos de nuestro contrato:

```
$ ln -s ../build/contracts
```

© Santiago Pavón - UPM-DIT

- Instalar Drizzle ejecutando:
  - Ir la raíz del proyecto donde está package.json:

    ```
    $ cd ..
    ```

  - Instalar el paquete drizzle.

    ```
    $ npm install --save drizzle
    ```

    - No es necesario instalar **web3**, ni **truffle-contract**.
      - Drizzle ya contiene todo lo necesario para interactuar con los contratos inteligentes
-

© Santiago Pavón - UPM-DIT

- Copiar los ficheros CSS y Javascript del paso 1 del caso de estudio #3.
  - La primera versión con React, y sin Redux.

```
$ cd src
$ cp -r ....../caso3/paso1/src/css .
$ cp -r ....../caso3/paso1/src/js/* .
```

- Renombramos **`src/app.jsx`** a **`src/index.js`**, que es el nombre que espera WebPack como fichero inicial de entrada:

```
$ mv app.jsx index.js
```

- Hay que editar **`index.js`** para:
  - Cambiar el path del **import** de **`../css/style.css`** por **`./css/style.css`**.
  - Cambiar el valor del **`id`** donde se insertarán los componentes React:
    - Cambiar **`document.getElementById('app')`** por **`document.getElementById('root')`**.

© Santiago Pavón - UPM-DIT

- Creo el fichero **`src/drizzle.js`** para crear el objeto drizzle.
  - Usare los reducers y el store que crea Drizzle por defecto ya que no tenemos datos propios para guardar en el estado, solo los del contrato Contador.
  - Para cambiar algo, podemos copiar el código que devuelve la función generateStore del paquete drizzle, y hacer las adaptaciones que necesitemos.
    - crear mas reducers, añadir middlewares, etc...

© Santiago Pavón - UPM-DIT

```
// importar funciones de drizzle
import { Drizzle, generateStore } from "drizzle";

// Importar la abstraccion del contrato
import Contador from './contracts/Contador.json';

// Opciones de Drizzle:
const options = {
    contracts: [ Contador ],
        web3: {
            fallback: {
                type: "ws",
                url: "ws://127.0.0.1:7545"
            }
        }
    };

// Crear el store de drizzle
const drizzleStore = generateStore(options);

// Crear el objeto drizzle
const drizzle = new Drizzle(options, drizzleStore);

export default drizzle;
```

src/drizzle.js

© Santiago Pavón - UPM-DIT

- Parche: Voy a simplificar **`src/index.js`** para eliminar el anidamiento de componentes **`App`** y **`AppCounter`** ya que App no hace nada:

```
import AppCounter from './components/AppCounter';

const App = () => (
    <AppCounter />
);

ReactDOM.render(
    <App />,
    document.getElementById('root')
);
```

  - y dejar solo:

```
import AppCounter from './components/AppCounter';

ReactDOM.render(
    <AppCounter />,
    document.getElementById('root')
);
```

- *PENDIENTE:*
  - *Aplicar este parche a los casos de estudios ya vistos anteriormente para que todo sea igual.*

© Santiago Pavón - UPM-DIT

- Modificar **`src/index.js`** para importar el objeto **`drizzle`** que hemos creado y pasárselo a **`AppCounter`** como una propiedad:

```
import React from 'react';
import ReactDOM from 'react-dom';

import "./css/style.css";

import drizzle from "./drizzle";

import AppCounter from './components/AppCounter';

ReactDOM.render(
    <AppCounter drizzle={drizzle} />,
    document.getElementById('root')
);
```

src/index.js

© Santiago Pavón - UPM-DIT

- El componente AppCounter:
  - Debe:
    - Esperar hasta que Drizzle se haya terminado de inicializarse (instanciar web3 y los contratos, detectar cuentas disponibles, escuchar o sondear por nuevos bloques).
      - Durante este rato deber mostrar algun componente que informe de que se está inicializando.
    - Cuando haya terminado la inicialización debera mostrar los componentes AppHeader, AppData, AppControls, que deben reflejar siempre que el estado actualizado de Drizzle.
  - Por tanto, crearemos un estado local en el componente AppCounter que tendrá dos valores:
    - loading: un booleano indicando si estamos en la fase de inicialización de Drizzle.
    - drizzleState: el estado de drizzle.
  - Cada vez que cambie algo de este estado local, se repintará el componente.
  - Implementación:
    - La lógica para actualizar el estado la añadiremos en los métodos componentDidMount y componentWillUnmount del ciclo de vida de AppCounter.
    - AppCounter pasará a sus componente hijos el objeto drizzle y el estado drizzleState como propiedades.

© Santiago Pavón - UPM-DIT

```
import React from 'react';

import AppHeader from './AppHeader';
import AppData from './AppData';
import AppControl from './AppControl';

class AppCounter extends React.Component {

    render() {
        return (
            <div className="appCounter">
                <AppHeader />
                <AppData counter={666}
                         updating={'Actualizando'}
                         error={'Algo está mal'} />
                <AppControl onClick={() => false}
                            disabled={false} />
            </div>
        );
    }
}

export default AppCounter;
```

ANTES: src/components/AppCounter.js

© Santiago Pavón - UPM-DIT

```
import React from 'react';
import AppHeader from './AppHeader';
import AppData from './AppData';
import AppControl from './AppControl';

class AppCounter extends React.Component {
    state = { loading: true, drizzleState: null };

    componentDidMount() {
        const { drizzle } = this.props;

        // subscribirse a los cambios en el store:
        this.unsubscribe = drizzle.store.subscribe(() => {

           // Cada vez que cambie el estado del store, si Drizzle ya se ha
           // inicializado, entonces actualizo el estado del componente.
           const drizzleState = drizzle.store.getState();
           if (drizzleState.drizzleStatus.initialized) {
               this.setState({ loading: false, drizzleState });
           }
        });
    }

    compomentWillUnmount() {
        this.unsubscribe();
    }
```

DESPUES: src/components/AppCounter.js (I)

© Santiago Pavón - UPM-DIT

```
    render() {
        if (this.state.loading) return "Cargando Drizzle...";

        return (
            <div className="appCounter">
                <AppHeader />

                <AppData drizzle={this.props.drizzle}
                         drizzleState={this.state.drizzleState} />

                <AppControl drizzle={this.props.drizzle}
                            drizzleState={this.state.drizzleState} />
            </div>
        );
    }
}
export default AppCounter;
```

DESPUES: src/components/AppCounter.js (II)

© Santiago Pavón - UPM-DIT

Modificar AppData

- AppData recibe los objetos **drizzle** y **drizzleState** en sus propiedades.
- La lógica para obtener el resultado de ejecutar el método de acceso **valor()** del contrato inteligente **Contador** es la siguiente:
  - Crear un estado local para guardar el **dataKey** que devolverá **cacheCall**.
  - Cuando se componente se monta,
    - Usar la propiedad **drizzle** para obtener la instancia del contrato Contador.
    - Usar **cacheCall** para registrar que estamos interesados en el resultado del método "**valor()**" del contrato inteligente.
    - Guardar en el estado local el **dataKey** devuelto por **cacheCall**.
      - **dataKey** se usa para recuperar el resultado siempre actualizado que devuelve el método **valor()**.
  - Al renderizar el componente:
    - Acceder al estado del contrato **Contador** y usar **dataKey** para obtener el resultado actualizado que devuelve el método **valor()** .
      - Si el resultado existe, se obtiene usando la propiedad **value**.

© Santiago Pavón - UPM-DIT

- Más cambios:
  - Los subcomponentes **Updating** y **Error** los voy a mover a **AppControl** para que muestren el estado de la transacción **incr**() que lanzará el botón Incrementar.

  - Alternativa: Cuando un subcomponente muestra la información de estado de las llamadas call o transacciones realizadas por otro subcomponente, la lógica y el estado deben subirse hasta un componente padre de los dos subcomponentes anteriores.
    - Por ejemplo:
      - El componente padre es el que guarda en su estado en valor de dataKey.
      - Este valor lo asigna uno de los subcomponentes, y lo usa el otro para acceder la resultado de la llamada call.
      - El padre se comunica con los hijos usando propiedades.

© Santiago Pavón - UPM-DIT

```
import React from 'react';

import PropTypes from 'prop-types';

import Updating from '../common/Info';
import Error from '../common/Info';

const AppData = props => (
    <div className="appCounter-data">
        <p>
            Valor = <span className="appCounter-data-value">
                          {props.counter}
                      </span>
        </p>
        <Updating className="appCounter-data-updating"
                  msg={props.updating}
                  visible={!!props.updating} />
        <Error className="appCounter-data-error"
               msg={props.error}
               visible={!!props.error} />
    </div>
);

// Valores por defecto en caso de que no se pase alguna propiedad.
AppData.defaultProps = { counter: '-1', updating: '', error: '' };

// Validar tipo de las propiedades.
// Indicar que propiedades son obligatorias.
// ...
AppData.propTypes = { counter: PropTypes.number, updating: PropTypes.string, error: PropTypes.string };

export default AppData;
```

ANTES: src/components/AppData/index.js

© Santiago Pavón - UPM-DIT

```
import React from 'react';

import Updating from '../common/Info';
import Error from '../common/Info';

class AppData extends React.Component {

  state = { dataKey: null };

  componentDidMount() {
    const { drizzle } = this.props;

    // Decirle a drizzle que queremos observar el metodo valor().
    const instance = drizzle.contracts.Contador;
    const dataKey = instance.methods["valor"].cacheCall();

    // Guardar `dataKey` en el estado local del componente.
    // Usaremos `dataKey` para recuperar el resultado del metodo.
    this.setState({ dataKey });
  }
```

DESPUES: src/components/AppData/index.js (I)

© Santiago Pavón - UPM-DIT

```
render() {
  // Obtener el estado del contrato desde drizzleState
  const { Contador } = this.props.drizzleState.contracts;

  // Usamos el resultado del metodo "valor()" usando la clave
  // `dataKey` guardada en el estado local del componente.
  const valor = Contador.valor[this.state.dataKey];

  return (
      <div className="appCounter-data">
        <p>
          Valor = <span className="appCounter-data-value">
                    {valor && valor.value }
                  </span>
        </p>
        <Updating className="appCounter-data-updating"
                  msg={"pendiente"}
                  visible={true} />
        <Error className="appCounter-data-error"
               msg={"pendiente"}
               visible={true} />
      </div>
  );
 }
};

export default AppData;
```

DESPUES: src/components/AppData/index.js (II)

© Santiago Pavón - UPM-DIT

Modificar AppControl

- Cambios:
  - El componente **AppControl** tiene un estado local para guardar el **stackId** de la última transacción solicitada.
  - El botón **IncrControl** llama al método **increment** del componente.
  - Se ha añadido el método **increment** en el componente para que contenga el código encargado de llamar a la función **incr** del contrato inteligente.
    - obtiene los objetos **drizzle** y **drizzleState** en sus propiedades.
    - obtiene la referencia a la instancia del contrato.
    - usa **sendCache** para crear la transacción.
      - con la función **incr** y la primera cuenta de **drizzleState**.
    - guarda en el estado local el **stackId** devuelto por **sendCache**.
  - Se han añadido los subcomponentes **Updating** y **Error** para mostrar mensajes de status y error de la última transacción solicitada.
  - Se ha añadido el método **getTxInfo** para obtener la información de **status** y **error** de la última transacción solicitada.
    - se obtiene es hash de la transacción a partir del **stackId**.
    - con el hash podemos acceder a la información de la transacción.

© Santiago Pavón - UPM-DIT

```
import React from 'react';
import PropTypes from 'prop-types';

import IncrControl from './common/Button';

const AppControl = ({ disabled, onClick }) => (
    <IncrControl className="appCounter-control"
                 text="Incrementar"
                 onClick={onClick}
                 disabled={disabled} />
);

// Validar tipo de las propiedades.
// Indicar que propiedades son obligatorias.
// ...
AppControl.propTypes = {
    onClick: PropTypes.func.isRequired,
    disabled: PropTypes.bool,
}

export default AppControl;
```

ANTES: src/components/AppControl.js

© Santiago Pavón - UPM-DIT

```
import React from 'react';

import Updating from './common/Info';
import Error from './common/Info';
import IncrControl from './common/Button';

class AppControl extends React.Component {

  state = { stackId: null };

  increment = () => {
    const { drizzle, drizzleState } = this.props;

    // Usar cacheSend para lanzar una transaccion que
    // ejecutara el metodo  incr del contrato inteligente.
    const instance = drizzle.contracts.Contador;
    const stackId = instance.methods.incr.cacheSend({
      from: drizzleState.accounts[0]
    });

    // Guardar stackId en el estado local
    this.setState({ stackId });
  }
```

DESPUES: src/components/AppControl.js (I)

© Santiago Pavón - UPM-DIT

```
getTxInfo = () => {

  // Si no he pulsado nunca el boton incrementar;
  if (this.state.stackId === null) return {status: null, error: null};;

  // Obtener el estado de las transacciones desde el estado de drizzle
  const { transactions, transactionStack } = this.props.drizzleState;

  // Obtener el hash de la transaccion asociada a stackId.
  // stackId se guardo en el estado local al crear la transaccion.
  const txHash = transactionStack[this.state.stackId];

  // El hash de la transaccion no existe hasta que se envia a la red.
  if (!txHash) return {status: "Pendiente de envio", error: null};

  // Si la transaccion existe, devolvemos su status
  return { status: transactions[txHash].status,
           error: transactions[txHash].error};
};
```

DESPUES: src/components/AppControl.js (II)

© Santiago Pavón - UPM-DIT

```
  render() {
    const {status, error} = this.getTxInfo();

    const errorMsg = error ? `${error.message || error}` : "";

    return (
        <div className="appCounter-control">
            <IncrControl className="appCounter-control-incr"
                         text="Incrementar"
                         onClick={this.increment}
                         disabled={status === 'pending'} />
            <Updating className="appCounter-control-updating"
                      msg={status}
                      visible={true} />
            <Error className="appCounter-control-error"
                   msg={errorMsg}
                   visible={!!error} />
        </div>
    );
  }

}

export default AppControl;
```

DESPUES: src/components/AppControl.js (III)

© Santiago Pavón - UPM-DIT

```
import React from 'react';

class AppData extends React.Component {

  state = { dataKey: null };

  componentDidMount() {
    const { drizzle } = this.props;

    // Decirle a drizzle que queremos observar el metodo valor().
    const instance = drizzle.contracts.Contador;
    const dataKey = instance.methods["valor"].cacheCall();

    // Guardar `dataKey` en el estado local del componente.
    // Usaremos `dataKey` para recuperar el resultado del metodo.
    this.setState({ dataKey });
  }
  render() {
    // Obtener el estado del contrato desde drizzleState
    const { Contador } = this.props.drizzleState.contracts;

    // Usamos el resultado del metodo "valor()" usando la clave
    // `dataKey` guardada en el estado local del componente.
    const valor = Contador.valor[this.state.dataKey];

    return (
        <div className="appCounter-data">
           Valor = <span className="appCounter-data-value">
                      {valor && valor.value }
                   </span>
        </div>
    );
  }
};
export default AppData;
```

Sin Updating y sin Error

DESPUES: src/components/AppData/index.js

© Santiago Pavón - UPM-DIT

Caso de Estudio #5

Paso 2

© Santiago Pavón - UPM-DIT

Usar DrizzleContext

- Para no tener que pasar explícitamente las propiedades drizzle y drizzleState por toda la jerarquía de componentes desde el componente raíz hasta llegar a los subcomponentes hijos interesados .
- El componente raíz es el productor del contexto Drizzle.
- Los subcomponentes hijos que necesiten acceder a la instancia de drizzle o al estado drizzleState serán consumidores del contexto Drizzle.
  - No tienen que subscribirse explícitamente al store para enterarse de los cambios del estado.
  - El hijo del componente consumidor es una función que se invoca pasándola el siguiente objeto como argumento:

```
const drizzleContext = { drizzle, drizzleState, initialized }
```

- Es necesario instalar drizzle-react:

```
$ npm install --save drizzle-react
```

- Documentación:

  https://github.com/trufflesuite/drizzle-react

© Santiago Pavón - UPM-DIT

```
import React from 'react';
import ReactDOM from 'react-dom';

import "./css/style.css";

import drizzle from "./drizzle";

import AppCounter from './components/AppCounter';

ReactDOM.render(
    <AppCounter drizzle={drizzle} />,
    document.getElementById('root')
);
```

ANTES: src/index.js

© Santiago Pavón - UPM-DIT

```
import React from 'react';
import ReactDOM from 'react-dom';

import "./css/style.css";

import drizzle from "./drizzle";

import { DrizzleContext } from "drizzle-react";

import AppCounter from './components/AppCounter';

ReactDOM.render(
    <DrizzleContext.Provider drizzle={drizzle}>
        <AppCounter />
    </DrizzleContext.Provider>,
    document.getElementById('root')
);
```

DESPUES: src/index.js

© Santiago Pavón - UPM-DIT

```
import React from 'react';

import AppHeader from './AppHeader';
import AppData from './AppData';
import AppControl from './AppControl';

class AppCounter extends React.Component {

    state = { loading: true, drizzleState: null };

    componentDidMount() {
    const { drizzle } = this.props;

    // subscribirse a los cambios en el store:
    this.unsubscribe = drizzle.store.subscribe(() => {

       // Cada vez que cambie el estado del store, si Drizzle ya se ha
       // inicializado, entonces actualizo el estado del componente.
       const drizzleState = drizzle.store.getState();
       if (drizzleState.drizzleStatus.initialized) {
           this.setState({ loading: false, drizzleState });
       }
    });
    }

    compomentWillUnmount() { this.unsubscribe(); }

    render() {
        if (this.state.loading) return "Cargando Drizzle...";
        return (
            <div className="appCounter">
                <AppHeader />
                <AppData drizzle={this.props.drizzle}
                         drizzleState={this.state.drizzleState} />
                <AppControl drizzle={this.props.drizzle}
                            drizzleState={this.state.drizzleState} />
            </div>
        );
    }
}

export default AppCounter;
```

ANTES: src/components/AppCounter.js

© Santiago Pavón - UPM-DIT

```
import React from 'react';

import { DrizzleContext } from "drizzle-react";

import AppHeader from './AppHeader';
import AppData from './AppData';
import AppControl from './AppControl';

class AppCounter extends React.Component {

    render() {
      return (
        <DrizzleContext.Consumer>
          {drizzleContext => {
            const { drizzle, drizzleState, initialized } = drizzleContext;

            if (!initialized) return "Cargando...";

            return (
              <div className="appCounter">
                <AppHeader />
                <AppData drizzle={drizzle} drizzleState={drizzleState} />
                <AppControl drizzle={drizzle} drizzleState={drizzleState} />
              </div>
            );
          }}
        </DrizzleContext.Consumer>
      );
    }
}

export default AppCounter;
```

DESPUES: src/components/AppCounter.js

© Santiago Pavón - UPM-DIT