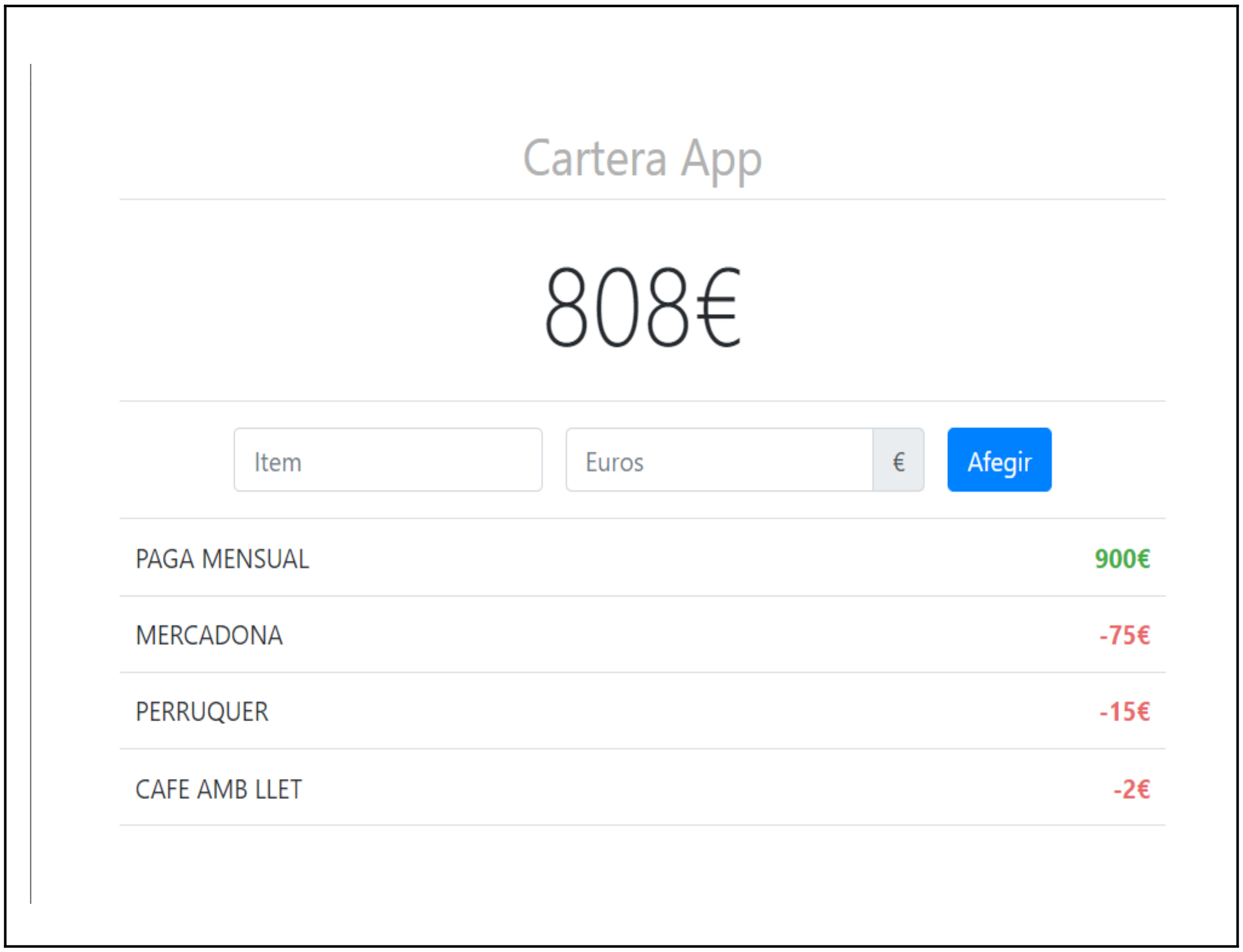
13B. REACT FLUX Example - Wallet.

In this tutorial, you will learn about Facebook's **Flux architecture** and how it's used to handle the data flow in React-based applications. We'll begin by covering the basics of Flux and understanding the motivation behind its development, and then we'll practice what we've learned by building **a simple virtual wallet application**.

Output:



Cartera App

808€

PAGA MENSUAL	900€
MERCADONA	-75€
PERRUQUER	-15€
CAFE AMB LLET	-2€

Throughout the tutorial, I will assume you've used **React** before, but have no experience with Flux. You might get something out of it if you already know the basics of Flux and are looking to gain a deeper understanding.

13B.1. Components Setup.

Start by including the following code inside **js/index.js**, which serves as the application's entry point:

Example: js/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

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

For the main **<App />** component, create a new file inside **js/components** called App.js and add the following code:

Example: js/components/App.js

```
import React from 'react';
import AddNewItem from './AddNewItem';
import ItemsList from './ItemsList';

class App extends React.Component {
    render() {
        return (
            <div className="container">
                <h1 className="app-title">Flux Wallet</h1>
                <AddNewItem />
                <ItemsList />
            </div>
        );
    }
}

export default App;
```

The **<App />** component wraps two other components, one for the form responsible for adding new items and another one for the list of items. To create the **<AddNewItem />** component, create a new file **AddNewItem.js** inside **js/components** and add this code:

Example: js/components/AddNewItem.js

```
import React from 'react';
```

```
class AddNewItem extends React.Component {

    // Set the initial state.
    constructor(props) {
        super(props);

        this._getFreshItem = this._getFreshItem.bind(this);

        this.state = {
            item: this._getFreshItem()
        };
    }

    // Return a fresh item.
    _getFreshItem() {
        return {
            description: '',
            amount: ''
        };
    }

    // Update the state.
    _updateState(event) {
        let field = event.target.name;
        let value = event.target.value;

        // If the amount is changed and it's not a float, return.
        if (value && field === 'amount' && !value.match(/^[a-z0-
9.\+\-]+$/g)) {
            return;
        }

        this.state.item[field] = value;
        this.setState({ item : this.state.item });
    }

    // Add a new item.
    _addNewItem(event) {
        // ...
    }

    render() {
        return (
            <div>
                <h3 className="total-budget">0€</h3>
                <form className="form-inline add-item"
                      onSubmit={this._addNewItem.bind(this)}>
                    <input type="text" className="form-control
description" name="description" value={this.state.item.description}
placeholder="Description" onChange={this._updateState.bind(this)} />
                    <div className="input-group amount">
                        <div className="input-group-addon">$</div>
                        <input type="text" className="form-control"
name="amount" value={this.state.item.amount} placeholder="Amount"
onChange={this._updateState.bind(this)} />
                    </div>
                    <button type="submit" className="btn btn-primary
add">Add</button>
                </form>
            </div>
        )
    }
}

export default AddNewItem;
```

The component bundles some logic for updating the state when the form fields update and also some basic validation. Let's finish off the components setup by creating the last one inside **js/components/ItemsList.js** for the items list, using this code:

Example: js/components/ItemsList.js

```
import React from 'react';

class ItemsList extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
            items: []
        };
    }

    render() {

        let noItemsMessage;

        // Show a friendly message instead if there are no items.
        if (!this.state.items.length) {
            noItemsMessage = (<li className="no-items">
                                  Your wallet is new!
                                </li>);
        }

        return (
            <ul className="items-list">
                {noItemsMessage}
                {this.state.items.map((itemDetails) => {
                    let amountType = parseFloat(itemDetails.amount) >
0 ? 'positive' : 'negative';
                    return (<li key={itemDetails.id}>
                           {itemDetails.description}
                            <span className={amountType}>
                              {itemDetails.amount}
                            </span>
                        </li>
                       );
                })}
            </ul>
        );
    }
}

export default ItemsList;
```

That's it! You're done setting up the project's components. The great part is that they also come with free styling.

Run **npm start** and wait for the bundle to build. If you point your browser to **localhost:3000**, you should see the app without any functionality.

Next, we'll cover what Flux is and how you can use it to add functionality to the virtual wallet application.

13B.2. The Flux Building Blocks

At a high level, Flux breaks down into four major parts: actions, the dispatcher, stores, and views:

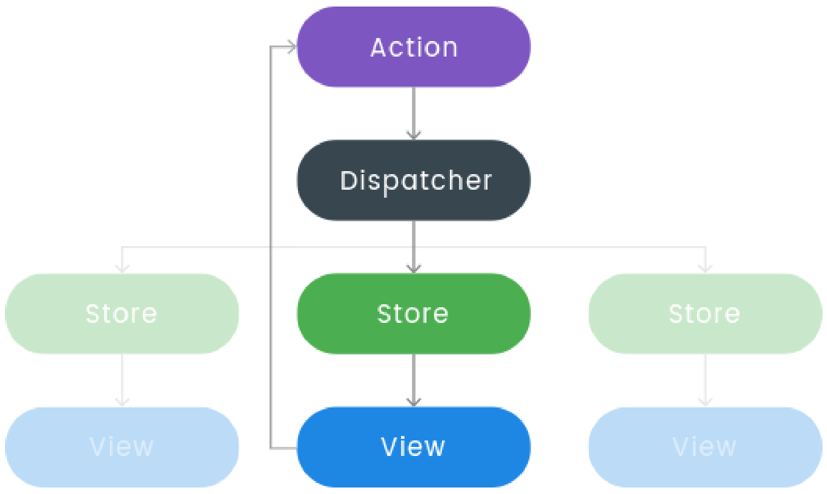
- **Actions** describe an action that took place in the application.
- The **dispatcher** is a singleton registry of callbacks. It acts as a middleman by passing the actions to all the stores that subscribed to it.
- **Stores** manage the state and logic needed to update it for specific parts of the application.
- **Views** are plain old React components.

In Flux, **all data flows in a single direction**:

1. **Actions** are passed to the **dispatcher** using convenience classes called **action creators**.
2. The **dispatcher** sends (is dispatching) the actions to all the **stores** that subscribed to it.

3. Finally, if the **stores** care about a particular action that was received (or more), they update their state and signal the **views** so they can re-render.

Below is a visual representation of this process.



13B.3. Actions

Data is sent "through the wire" in **a single direction** using plain JavaScript objects called **actions**. Their job is to describe an event that took place in the application and to transport the new data to the stores. Each action must have a type and an optional payload key that contains the data. An action looks similar to the one below:

Example:

```
{
    actionType: "UPDATE_TITLE",
    payload: "This is a new title."
}
```

The action's type must be represented by a descriptive and consistent uppercase string—similar to the common convention of defining constants. They serve as unique IDs that stores will use to identify the action and respond accordingly.

A common practice is to **define all action types in a constants** object and reference that object instead across the application to maintain consistency. Our virtual wallet will support a single action, which adds items to the list—both expenses and financial gains will be treated as a single item—so our constants file will be very slim.

Create an **index.js** file in the **js/constants** folder and use the following code to create your first action type:

Example: js/constants/index.js

```
export default {
    ADD_NEW_ITEM: 'ADD_NEW_ITEM'
}
```

Actions are passed to the **dispatcher** using convenience class helpers called **action creators** that handle the simple task of creating and sending the action to the **dispatcher**. Before creating our action creator, let's see what the dispatcher does first and understand its role in Flux.

13B.4. The Dispatcher

The **dispatcher** is used to coordinate the communication between action creators and stores. You can use it to register a store's actions handler callback and also to dispatch actions to the stores that subscribed.

The **dispatcher's API** is simple, and it has only five methods available:

- **register**(): Registers a store's action handler callback.

- **unregister**() : Unregisters a store's callback.

- **waitFor**(): Waits for the specified callback(s) to run first.
- **dispatch**(): Dispatches an action.
- **isDispatching**(): Checks if the dispatcher is currently dispatching an action.

The most important are **register**() and **dispatch**() as they're used to handle most of the core functionality. Let's see how they look and work behind the scenes.

Example:

```
let _callbacks = [];

class Dispatcher {

    // Register a store callback.
    register(callback) {
        let id = 'callback_' + _callbacks.length;
        _callbacks[id] = callback;
        return id;
    }

    // Dispatch an action.
    dispatch(action) {
        for (var id in _callbacks) {
            _callbacks[id](action);
        }
    }
}
```

This is, of course, the basic gist. The **register**() method stores all callbacks in a private **_callbacks** array and **dispatch**() iterates and calls each callback stored using the received action.

For simplicity, we won't write our own dispatcher. Instead, we'll use the one provided in Facebook's library. I encourage you to check out Facebook's GitHub repo and see how it's implemented.

Inside the **js/dispatcher folder**, create a new file **index.js** and add this code snippet:

Example: js/dispatcher/index.js

```
import { Dispatcher } from 'flux';

export default new Dispatcher();
```

It imports the dispatcher from the flux library—which was installed using yarn earlier—and then exports a new instance of it.

Having the dispatcher ready now, we can get back to actions and set up our app's action creator. Inside the **js/actions** folder, create a new file called **walletActions.js** and add the following code:

Example: js/actions/walletActions.js

```
import Dispatcher from '../dispatcher';
import ActionTypes from '../constants';

class WalletActions {

    addNewItem(item) {

        // Note: This is usually a good place to do API calls.
        Dispatcher.dispatch({
            actionType: ActionTypes.ADD_NEW_ITEM,
            payload: item
        });
    }

}

export default new WalletActions();
```

The **WalletActions** class is exposing an **addNewItem()** method that handles three basic tasks:

- It receives an item as an argument.
- It uses the dispatcher to dispatch an action with the **ADD_NEW_ITEM** action type we created earlier.
- It then sends the received item as **payload** along with the action type.

Before putting this action creator to use, let's see what stores are and how they fit in our Flux-powered application.

13B.5. Stores

I know, I said you shouldn't compare Flux with other patterns, but **Flux stores are in a way similar to models in MVC**. Their role is to **handle the logic and store the state for a particular top-level component in your application**.

All Flux stores must define an action handler method that will then be registered with the dispatcher. This callback function mainly consists of a switch statement on the received action type. If a specific action type is met, it acts accordingly and updates the local state. Finally, **the store broadcasts an event to signal the views about the updated state so they can update accordingly.**

In order to broadcast events, stores need to extend an event emitter's logic. There are various event emitter libraries available, but the most common solution is to use Node's **event emitter**. For a simple app like a virtual wallet, there's no need for more than one store.

Inside the **js/stores** folder, create a new file called **walletStore.js** and add the following code for our app's store:

Example: js/actions/walletActions.js

```
import { EventEmitter } from 'events';
import Dispatcher from '../dispatcher';
import ActionTypes from '../constants';

const CHANGE = 'CHANGE';
let _walletState = [];

class WalletStore extends EventEmitter {

    constructor() {
        super();

        // Registers action handler with the Dispatcher.
Dispatcher.register(this._registerToActions.bind(this));
    }

    // Switches over action's type when action is dispatched.
    _registerToActions(action) {
        switch(action.actionType) {
            case ActionTypes.ADD_NEW_ITEM:
                this._addNewItem(action.payload);
            break;
        }
    }

    // Adds a new item to the list and emits a CHANGED event.
    _addNewItem(item) {
        item.id = _walletState.length;
        _walletState.push(item);
        this.emit(CHANGE);
    }

    // Returns the current store's state.
    getAllItems() {
        return _walletState;
    }


    // Calculate the total budget.
    getTotalBudget() {
        let totalBudget = 0;

        _walletState.forEach((item) => {
            totalBudget += parseFloat(item.amount);
        });

        return totalBudget;
    }


    // Hooks a React component's callback to the CHANGED event.
    addChangeListener(callback) {
        this.on(CHANGE, callback);
    }

    // Removes the listener from the CHANGED event.
    removeChangeListener(callback) {
        this.removeListener(CHANGE, callback);
    }
}
```

```
export default new WalletStore();
```

We start by importing the required dependencies needed for the store, beginning with Node's event **emitter**, the dispatcher followed by the **ActionTypes**. You will notice that below it, there is a constant **CHANGE**, similar to the action types you learned about earlier.

It's actually not one, and it shouldn't be confused. It's a constant used for the event trigger when the store's data change. We will keep it in this file as it isn't a value used in other parts of the application.

When initialized, the **WalletStore** class starts by registering the **_registerToAction**() callback with the dispatcher. Behind the scenes, this callback will be added to the dispatcher's **_callbacks** array.

The method has a single switch statement over the action's type received from the dispatcher when an action is dispatched. If it meets the **ADD_NEW_ITEM** action type, it then runs the **_addNewItem**() method and passes along the payload it received.

The **_addNewItem**() function sets an id for the item, pushes it to the list of existing items, and then emits a **CHANGE** event. Next, the **getAllItems**() and **getTotalBudget**() methods are basic getters, which we'll use to retrieve the current store's state and the total budget.

The final two methods, **addChangeListener**() and **removeChangeListener**(), will be used to link the React components to the **WalletStore** so they get notified when the store's data change.

13B.6. Controller Views

Using React allows us to break down parts of the application into various **components**. We can nest them and build interesting hierarchies that form working elements in our page.

In Flux, components located at the top of the chain tend to store most of the logic needed to generate actions and receive new data; therefore, they are called **controller views**. These views are directly hooked into stores and are listening for the change events triggered when the stores are updated.

When this happens, controller views call the **setState** method, which triggers the **render**() method to run and update the view and send data to child components through props. From there, React and the Virtual DOM do their magic and update the DOM as efficiently as possible.

Our app is simple enough and does not respect this rule by the book. However, depending on complexity, larger apps can sometimes require multiple controller views with nested sub-components for the major parts of the application.

13B.7. Fitting It Together

We've finished covering the major parts of Flux, but the virtual wallet app is not yet completed. In this last section, we'll review the entire flow from actions to views and fill in the missing code needed to complete Flux's unidirectional data flow.

13B.7.1. Dispatching an Action

Getting back to the **<AddNewItem />** component, you can now include the **WalletActions** module and use it to generate a new action in the _**addNewItem**() method.

Example: js/components/AddNewItem.js

```
import React from 'react';
import WalletActions from '../actions/walletActions';

// …

_addNewItem(event) {
    event.preventDefault();
    this.state.item.description = this.state.item.description
|| '-';
    this.state.item.amount = this.state.item.amount || '0';
    WalletActions.addNewItem(this.state.item);
    this.setState({ item : this._getFreshItem() });
}

// ...
```

Now, when the form is submitted, an action is dispatched and all stores—one in our case—are notified about the new data.

13B.7.2. Listening for Store Changes

In your **WalletStore**, currently when an item is added to the list its state changes and the **CHANGE** event is triggered, yet no one is listening. Let's close the loop by adding a change listener inside the **<ItemsList** /> component.

Example: js/components/ItemsList.js

```
import React from 'react';
import WalletStore from '../stores/walletStore';

class ItemsList extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
            items: WalletStore.getAllItems()
        };
        this._onChange = this._onChange.bind(this);
    }

    _onChange() {
        this.setState({ items: WalletStore.getAllItems() });
    }

    componentWillMount() {
        WalletStore.addChangeListener(this._onChange);
    }

    componentWillUnmount() {
        WalletStore.removeChangeListener(this._onChange);
    }

    render() {
        // ...
    }

}

export default ItemsList;
```

The updated component closes Flux's unidirectional data flow. Note that I skipped including the entire **render**() method to save some space. Let's go step by step through what's new:

- The **WalletStore** module is included at the top.
- The initial state is updated to use the store's state instead.
- A new _**onChange**() method is used to update the state with the new data from the store.
- Using React's lifecycle hooks, the _**onChange**() callback is added and removed as the store's change listener callback.

13B.8. Conclusion

Congrats! You've finished building a working virtual wallet app powered by Flux. You've learned how all the Flux components interact with each other and how you can add structure to React apps using it.

When you're feeling confident in your Flux skills, make sure you also check out other Flux implementations like **Alt, Delorean, Flummox or Fluxxor** and see which one feels right for you.

- http://alt.js.org/
- https://github.com/f/delorean
- https://github.com/acdlite/flummox
- http://fluxxor.com/