# Actions, Store and Reducer

Creating an app using Redux alongside React to demonstrate the similarities and differences between Flux and Redux.

WE'LL COVER THE FOLLOWING



## Actions

Like Flux, the typical action in Redux is merely an object with a `type` property. Everything else in that object is considered to be context specific data and it is not related to the pattern, but to your application logic. For example:

```
const CHANGE_VISIBILITY = 'CHANGE_VISIBILITY';
const action = {
  type: CHANGE_VISIBILITY,
  visible: false
}
```

It is a good practice that we create constants like `CHANGE_VISIBILITY` for our action types. It happens that there are lots of tools/libraries that support Redux and solve different problems which do require the type of the action only. So it is just a convenient way to transfer this information.

The `visible` property is the meta data that we mentioned above. It has nothing to do with Redux. It means something in the context of the application.

Every time when we want to dispatch a method we have to use such objects. However, it becomes too noisy to write them over and over again. That is why

there is the concept of *action creators*. An action creator is a function that returns an object and may or may not accept an argument which directly relates to the action properties. For example the action creator for the above action looks like this:

```
const changeVisibility = visible => ({
  type: CHANGE_VISIBILITY,
  visible
});

changeVisibility(false);
// { type: CHANGE_VISIBILITY, visible: false }
```

Notice that we pass the value of the `visible` as an argument and we don't have to remember (or import) the exact type of the action. Using such helpers makes the code compact and easy to read.

## Store #

Redux provides a helper `createStore` for creating a store. Its signature is as follows:

```
import { createStore } from 'redux';

createStore([reducer], [initial state], [enhancer]);
```

We already mentioned that the reducer is a function that accepts the current state, action and returns the new state. More about that in a bit. The second argument is the initial state of the store. This comes as a handy instrument to initialize our application with data that we already have. This feature is the essence of processes like server-side rendering or persistent experience. The third parameter, enhancer, provides an API for extending Redux with third party middlewares and basically plugs some functionally which is not baked-in. Like an instrument for handling async processes.

Once created, the store has four methods - `getState`, `dispatch`, `subscribe` and `replaceReducer`. Probably the most important one is `dispatch`:

```
store.dispatch(changeVisibility(false));
```

That is the place where we use our action creators. We pass the result to them or, in other words, action objects to this `dispatch` method. It then gets spread

or, in other words, action objects to this `dispatch` method. It then gets spread to the reducers in our application.

In the typical React application we usually don't use `getState` and `subscribe` directly because there is a helper (we will see it in the next sections) that wires our components with the store and effectively `subscribes` for changes. As part of this subscription we also receive the current state so we don't have to call `getState` ourself. `replaceReducer` is kind of an advanced API and it swaps the reducer currently used by the store. I personally never used this method.

## Reducer #

The reducer function is probably the most *beautiful* part of Redux. There are two characteristics of the reducer that are quite important and without them we basically have a broken pattern.

- It must be a pure function - it means that the function should return the exact same output every time, given the same input.
- It should have no side effects - stuff like accessing a global variable, making an async call or waiting for a promise to resolve have no place in here.

Here is a simple counter reducer:

```
const counterReducer = function (state, action) {
  if (action.type === ADD) {
    return { value: state.value + 1 };
  } else if (action.type === SUBTRACT) {
    return { value: state.value - 1 };
  }
  return { value: 0 };
};
```

There are no side effects and we return a brand new object every time. We accumulate the new value based on the previous state and the incoming action type.

---

In the next lesson, we will see how the react-redux module can be used to connect Redux to React components