999-999-fr-spl / State management with @ngrx library

The need of a *Shared state service*

Components in applications are usually organized in complex hierarchy structure. Synchronize data between deeply nested components without making redundant requests is important to make the applications persistent. To achieve this, we need to share states across components.

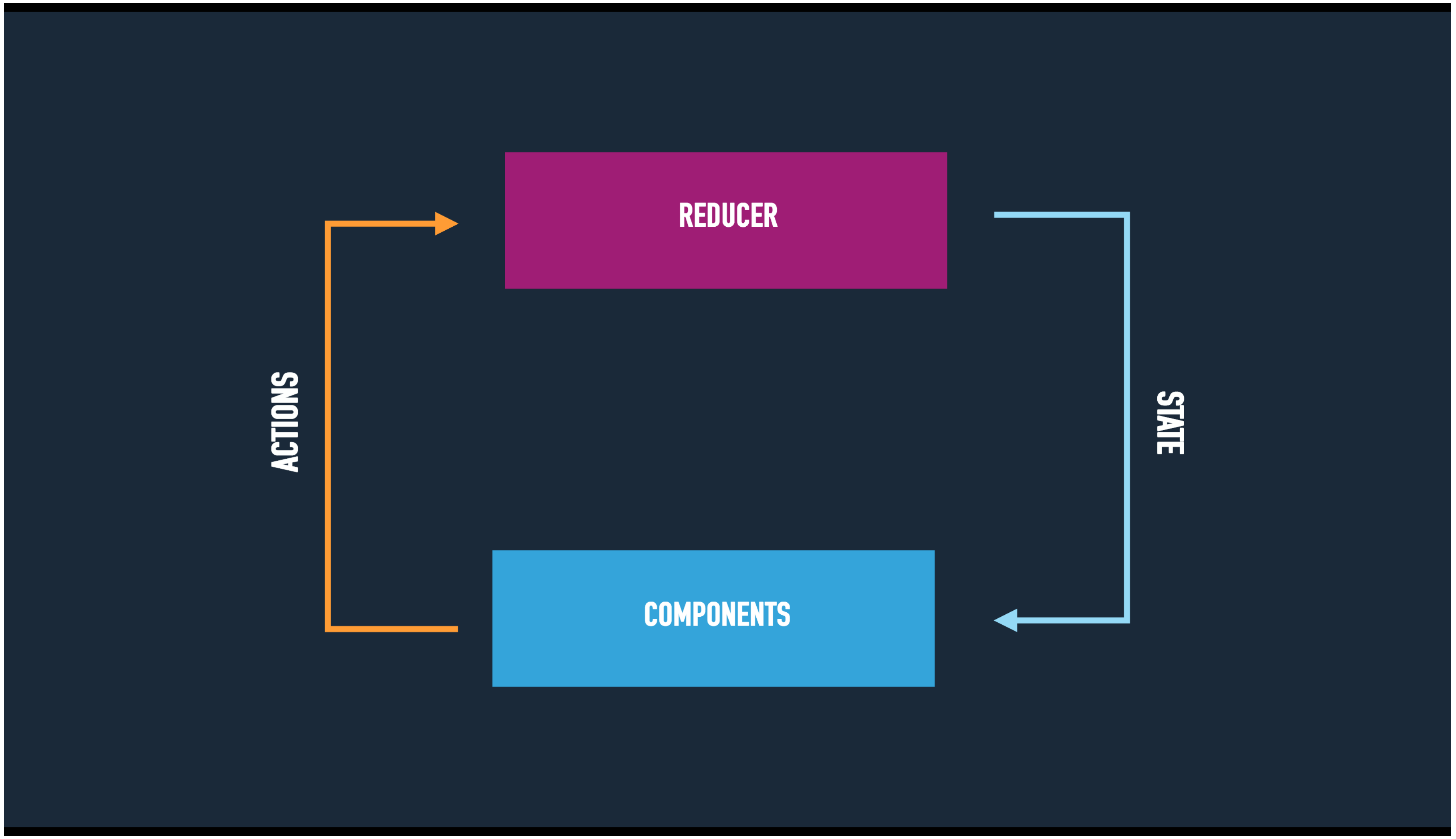
`@ngrx/store` is the library that helps us manage the state of our application. It's a ngrx powered state management container for Angular app.

How `@ngrx/store` works

The idea is that the application will have a `store` which contains the states of our application. Throughout the life cycle circle of the application, we will maintain the states in this store. This store will be **the single source of truth** of our application. This architecture makes data in our application more consistent.

The store holds the entire immutable states of the application. The store in `ngrx/store` is an RxJS observable of state and an observer of actions.

In order to understand how `@ngrx/store` works, consider the following diagram.



Here we have components in our application. When user interacts with `components`, we will take those events and model them in a consistent way using `actions`. Those actions will be reduced into `states` by `reducers`, then delivered to components in an observable service.

Now we look closer at each factor in the above diagram.

Actions

Actions are used to represent all the UI event and HTTP responses in our application. An action usually has `type` property so we can classify it and apply appropriate `reducer` later. Each action type must be unique. An action might contain a payload, which is an extract information it carries. Here's a sample declaration of an action.

```
interface Action {
  type: string;
  payload: any;
}
```

Reducers

Reducer is a special function that takes in the previous state and an action and returns the new state. Reducer must be a pure function, it doesn't mutate the previous instance of the state object it takes in. It always returns a new state object.

```
interface Reducer<State> {
  (state: State, action: Action): State;
}
```

When the application starts up, `state` is going to be `undefined`. The reducers needs to know how to initialize state.

Here's a reducer of our `Counter` app.

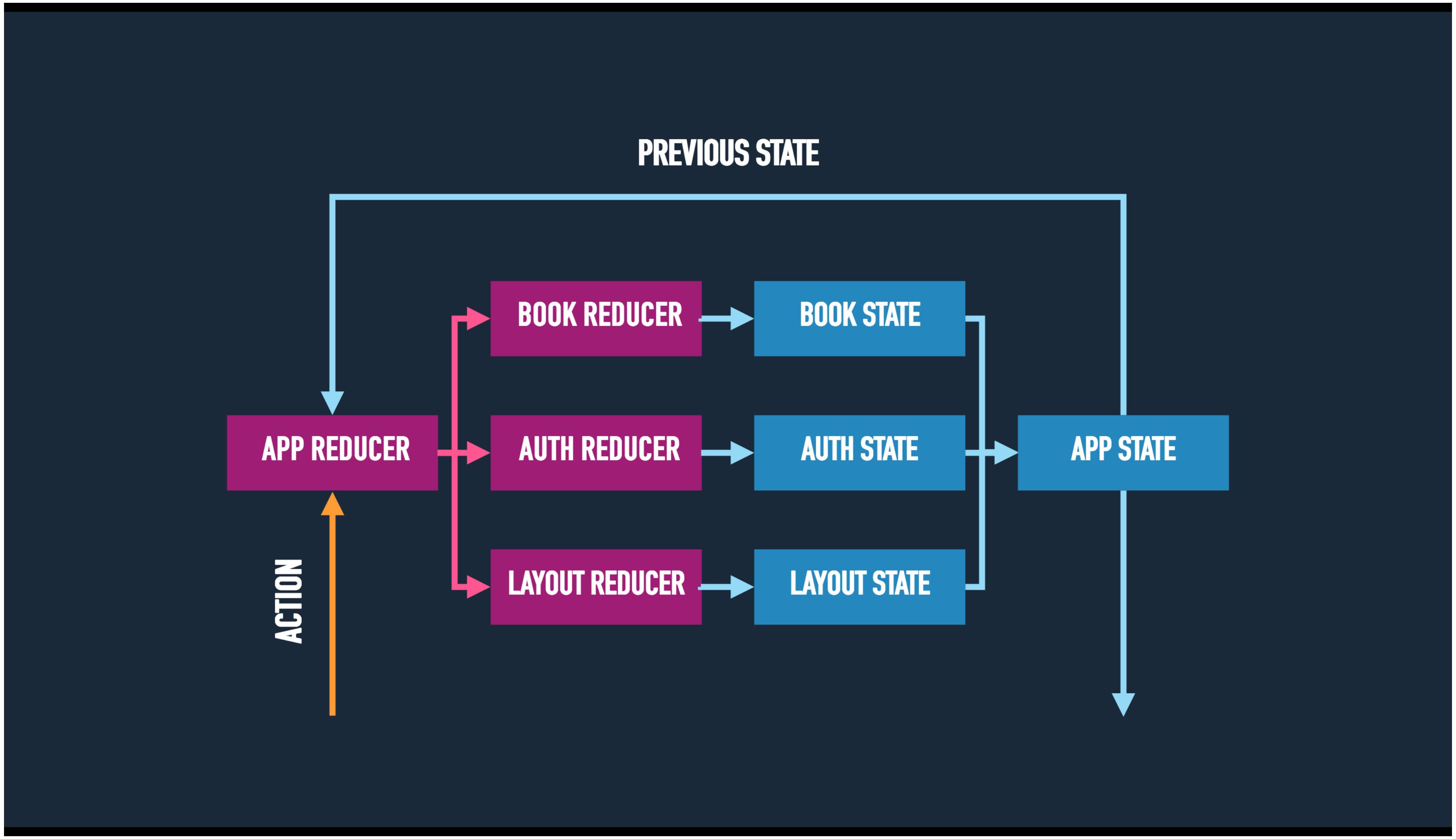
```
export const initialState: State = {
  counter: 0
};

export function counterReducer(state: State = initialState, action: AppAction): State {
  switch (action.type) {
    case actions.INCREASE:
      return { ...{ counter: counter + 1 }, ...action.payload };
    case actions.DECREASE:
      return { ...{ counter: newCounter }, ...action.payload };
    default:
      return state;
  }
}
```

We can see at the initial state the counter is set to `0`.

In real life application, there is a lot of states to manage. Thus it makes more sense to have multiple reducers instead of have one reducer to manage all the states. Each child reducer will handle one piece of the state.

For example, the following diagram shows that this application has a multiple reducers to handle multiple states.



The states and actions are delegated to child reducers, these reducers only handle one piece of state.

Store

Store is a special service. It's an observable of states.

```
class Store<State> extends Observable<State> {
  select<T>(s: (v: State) => T): Observable<T>;
  dispatch(action: Action): void;
}
```

The store has 2 special operators - `select` : as stated above, when the application is complex, the state object will be large. Components usually don't need all information from that large state. Each component only concerns a small aspect of the state. So `select` operator selects the part of the state component needs and returns an observable of changes to that piece of state. - `dispatch` : components uses this operator to dispatch to dispatch actions to the store.

For example, here's the `counter` component. The store is injected into this component. Whenever users interact with the component to increase, decrease or reset the counter, the `dispatch` method is called to dispatch the appropriate `action` .

```
export class CounterComponent {
  counter: Observable<number>;

  constructor(private store: Store<AppState>) {
    this.counter = store.select('app').select('counter');
  }

  increment() {
    this.store.dispatch({ type: INCREMENT });
  }

  decrement() {
    this.store.dispatch({ type: DECREMENT });
  }

  reset() {
    this.store.dispatch({ type: RESET });
  }
}
```

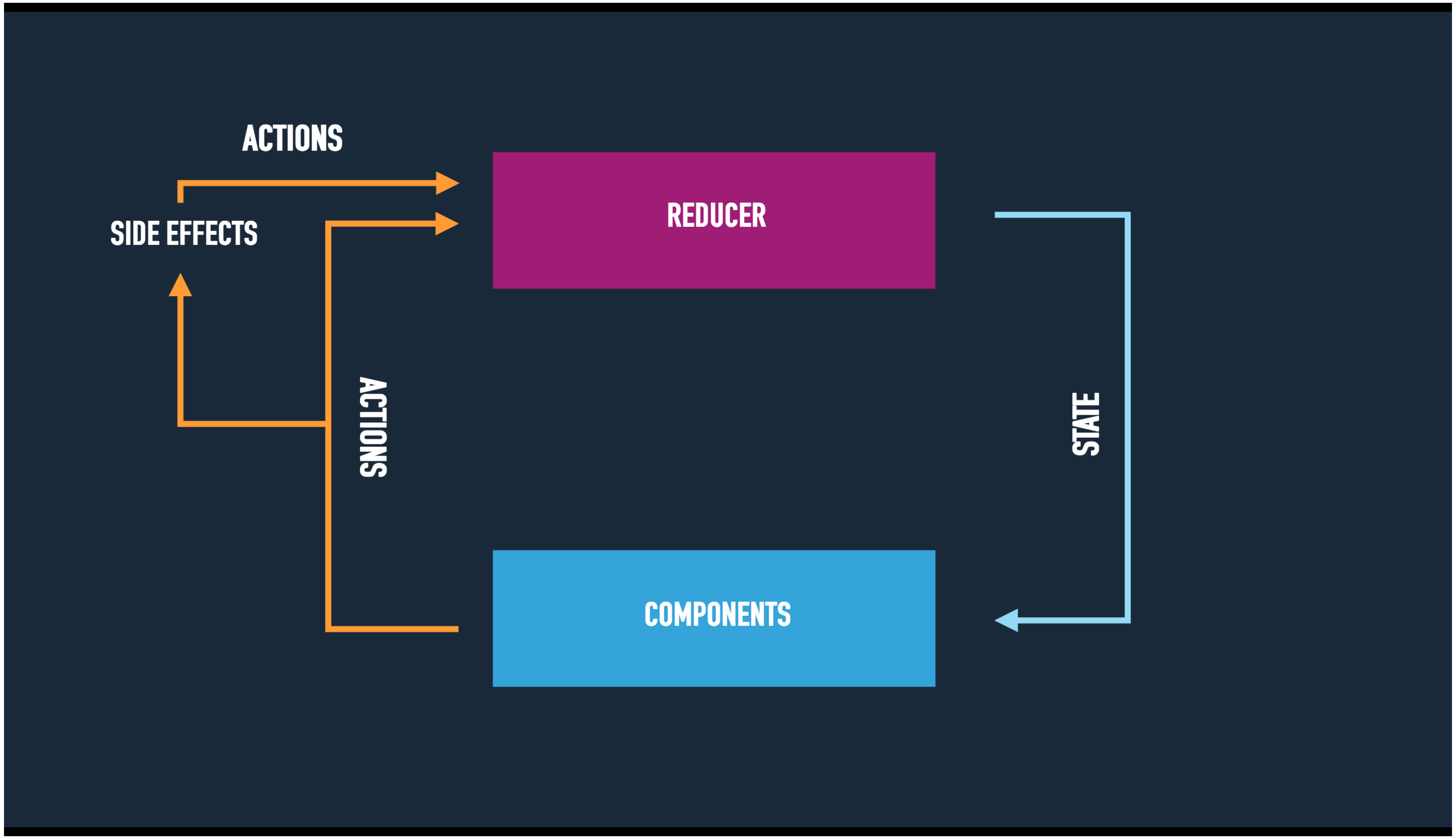
Notice in the constructor, the `counter` property is assigned to an observable (the `select` operator). This means that whenever the `counter` state is updated, the observable emits update so `counter` property always receives the updated value of the `counter` state from the store.

Making side effects

In `@ngrx/store` state management flow, when the action was triggered, a reducers process a data and put into **Store**. **View** will subscribe data from the store and get data to display in an application.

An application may interacts with the outside world by making HTTP requests, store offline data, etc. These are considered as side effects of our application. To make our components *purer*, we need to isolate these side effects. This will make our application easier to test.

`@ngrx/effects` is a side model of the store. Effects subscribe an action, when Action was triggered, an effect function will trigger too. You can do an async task from here, HTTP request, connect API to query data from the database and then trigger another Action to save a response into **Store**.



`@ngrx/effects` provides `Actions` service.

```
class Actions extends Observable<Action> {
  ofType:(type: string): Observable>Action>;
}
```

`Action` is the Observable of all the actions that are dispatched in the application. The special operator `ofType`. It takes an action type and returns an observable of all the time that action is dispatched in our application.

Imagine that, in our `counter` component, when user resets the counter, we would like to store the new value of our counter to local storage. In this case we need to implement an `effect` like this:

```
@Injectable()
export class CounterEffects {
  //..

  @Effect()
  saveCounter = this.actions.ofType(RESET).do((action: ResetCounter) => {
    this.storageService.save(action.payload);
  });

  constructor(private actions: Actions, private storageService: StorageService) {}
}
```

The `StorageService` is just a service to write data to local storage. Writing data to local storage may take time and it's out of scope of our reducer, whose responsibility is just updating the store. Hence writing data to local storage is a side effect and need to be handled by `effects`. Isolating this action makes our reducer purer and easier to test.

Updated 2017-12-19