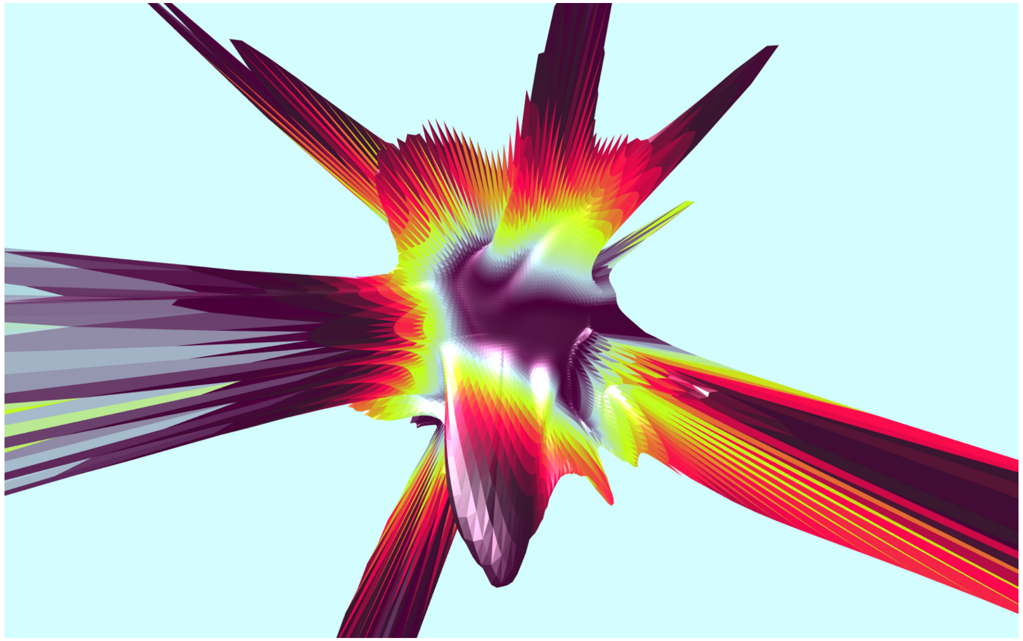
Angular 2—Introduction to Redux

How to use Redux in Angular 2 Applications



FluidScapes by Reza Ali

Redux, now in version 3, has been around less than a year but it has proved very successful. Inspired by Flux and Elm, is used to handle Application state and bind it to the User Interface in a very effective way. Redux also enables cool features like *hot reloading or time travel* with little effort. Redux is usually seen with React but it can be used separately.

> *Redux builds on top of Flux concepts although previous experience is not mandatory.*

If you want to learn more about Flux you can read this post below.

In this article, we are going to explore a **Todo List example** ported from React, from the recent Redux video course by Dan Abramov. Use the links below to hack the final solution:

Demo | Source

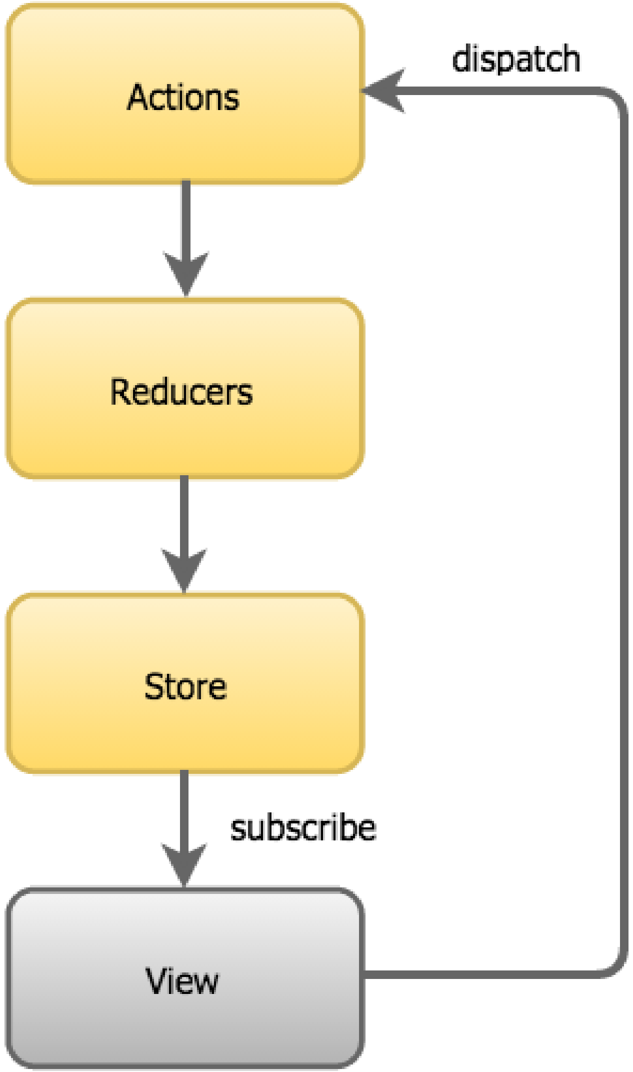
Introduction to Redux

Redux follows three basic principles:

- Single Immutable State Tree
- Uni-directional data-flow
- Changes are made using pure functions (Reducers)

By following these principles We can achieve a predictable and reproducible Application behaviour.

Let’s review the responsibilities for each component on the diagram below.



Redux unidirectional flow

Actions

These are ***Actions*** in our Application. They can be originated by the User or the Server-side. They are the only source of information for the ***Store***. Actions are plain JavaScript objects describing a change and using a ***type*** property as identifier. See an example below:

```
{
  type: 'TOGGLE_TODO',
  id: 0
}
```

actionObject.js hosted with by **GitHub** **view raw**

Action Creators are components containing helper methods that create specific ***Actions*** to be dispatched and run by ***Reducers***.

Reducers

Reducers specify how the state changes in response to ***Actions***. All *Reducers* must be ***pure functions*** meaning:

- they produce the same output given the same input
- they don't produce ***side-effects*** (Eg: mutate state, make calls to backend)

> *Reducers always create a new state to avoid side-effects; a more advanced option is to use a library like immutable.js.*

Reducers can also be composed with other reducers as required with ***combineReducers***. See a basic ***rootReducer*** below.

```
1  function rootReducer(state = initialState, action){
2    switch (action.type) {
3      /* case TodoActions.ADD_TODO:
4          //add logic here returning a new state
5          break; */
6      default:
7        // mandatory for sanity (Eg: initialisation)
8        return state;
9    }
10 };
```

rootReducer.js hosted with by **GitHub** **view raw**

It's a common practice to define the ***initialState*** as a default parameter (line 1) and handle each action with a switch statement.

Application Store

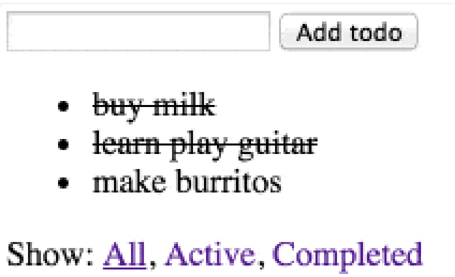
Redux uses a single store containing the *Application State* as a plain JavaScript object.

The ***Application Store*** is central to Redux and offers an API to:

- dispatch actions by ***appStore.dispatch(action)***
- register listeners for change notification: ***appStore.subscribe(callback)***
- read the Application State: ***appStore.getState()***

Todo List example

We are going to explore a **Todo List** Application to learn how we can integrate Redux with Angular 2. This is a basic implementation where we can add new todos, mark them as completed and filter them.



- ~~buy milk~~
- ~~learn play guitar~~
- make burritos

Show: All, Active, Completed

Application Design

In Angular 2 we start designing our applications using a ***component tree*** and starting from the ***root*** component. Find below a schematic pseudo-HTML including all the UI components: add-todo, todo-list (child components: todo), filters (child components: filter-link).

```
<root>
  <add-todo>
    <input><button>Add todo</button></add-todo>
  <todo-list><ul>
      <todo id="0" completed="false"><li>buy milk</li></todo
    </ul></todo-list>
  <filters>
    Show: <filter-link><a>All</a><filter-link> ... </filters
</root>
```

pseudo.html hosted with by **GitHub** **view raw**

Bootstrap Setup

See below the code to setup Redux:

```
 1  // src/main.ts
 2  import {createStore} from 'redux';
 3  import {rootReducer} from './rootReducer';
 4  import {TodoActions} from './todoActions';
 5
 6  const appStore = createStore(rootReducer);
 7
 8  bootstrap(App, [
 9   provide('AppStore', { useValue: appStore }),
10   TodoActions
11  ])
```

main.js hosted with by **GitHub** **view raw**

Angular 2 Applications are bootstrapped passing the ***root*** component and global dependencies. We imported all dependencies (lines 2–4), then instantiated ***appStore*** (line 6) using ***createStore*** and passing the ***rootReducer*** (function). Finally we used ***bootstrap*** with our root component ***App*** (line 8) passing our *Redux Components,* ***appStore*** and ***TodoActions*** (lines 9–10). **TodoActions** (class) will act as an *ActionCreator* with a public method for each action.

> *Note that when using a string token we have to prepend* @Inject('AppStore') *on our components.*

Application State

The *Application Store* (***appStore)*** will hold the Application State. This is: the todos Array and the current filter. We will define the initial state as follows:

```
//src/rootReducer.ts
const initialState = {
 todos: [],
 currentFilter: 'SHOW_ALL'
}
```

initialState.js hosted with by **GitHub** **view raw**

In the next section, we will define the structure for a todo item. This core structure will remain unchanged during the life of the Application.

Adding a New Todo

Let’s see a simplified version of the ***AddTodo*** component that will allow us to add a new todo and take care of user input.

```
1   //src/addTodo.ts
2   @Component({
3    selector: 'add-todo', //matches <add-todo></add-todo>
4    template: `
5     <div>
6      <input #todo>
7      <button (click)=”addTodo(todo)”>Add todo</button>
8     </div>`
9   })
10  export class AddTodo {
11   constructor(
12     @Inject('AppStore') appStore: AppStore,
13     todoActions: TodoActions
14   ){
15     this.appStore = appStore;
16     this.todoActions = todoActions;
17   }
18
19   private addTodo(input) {
20     this.appStore.dispatch(this.todoActions.addTodo(input.va
21     input.value = ‘’;
22   }
23  }
```

addTodo.js hosted with by **GitHub** **view raw**

In the template (lines 4–8), we are using a local template variable ***#todo*** (input HTML element, line 6) and passing its reference on the button click event (line 7). On the constructor, we injected ***appStore*** and ***todoActions*** into the component (lines 11–17) as private properties. When the user enters a description and clicks on '***Add Todo***' this will dispatch an action (line 20) like the one below and clear the input content.

```
{
  type: 'ADD_TODO',
  id: 0,
  text: 'buy milk',
  completed: false
}
```

addTodoAction.js hosted with by **GitHub** **view raw**

To avoid manually creating action objects in our components we created the ***TodoActions*** class as an *ActionCreator*.

```
 1  //src/todoActions.ts
 2  export const ADD_TODO = 'ADD_TODO'; //convenience token
 3  ...
 4
 5  export class TodoActions {
 6   constructor() {
 7     this.nextToDoId = 0; //convenience accumulator
 8   }
 9
10   addTodo(text){
11     return {
12       type: ADD_TODO,
13       id: this.nextToDoId++,
14       text: text,
15       completed: false
16     };
17   };
18  }
```

addTodoActionCreator.js hosted with by **GitHub** **view raw**

We expose the *ADD_TODO* token as an action identifier (line 2). Note how we extended the action object to include the information we require to identify todos and flag them as completed or not (lines 12–15).

After dispatching the action the ***rootReducer*** will be called by the store passing the *currentState* (*initialState* if undefined) and the user action.

```
//src/rootReducer.ts
case TodoActions.ADD_TODO:
  return {
   todos: state.todos.concat({
     id: action.id,
     text: action.text,
     completed: action.completed
   }),
   currentFilter: state.currentFilter
  };
```

addTodoReducer.js hosted with by **GitHub** **view raw**

In order to create the new state we are using ***concat*** (creating a new Array) and maintaining the current filter, initially it shows all todos.

Toggling a Todo

For each todo the user can toggle it as completed clicking over its description. Below you can see a simplified mark-up for an active todo:

```html
<todo-list><ul>
  <todo id="0" completed="false"><li>buy milk</li></todo>
</ul></todo-list>
```

todoList_pseudo.html hosted with by **GitHub** **view raw**

Similar to what we did with add todo, each click event will pass down the todo ***id*** (input attribute, line 6) and dispatch the corresponding action (line 17).

```
1   //src/todo.ts
2   @Component({
3     selector: 'todo',
4     inputs: ['completed', 'id'], //attributes
5     template: `
6      <li (click)=”onTodoClick(id)” ...>
7        <ng-content></ng-content>
8      </li>`
9   })
10  export class Todo {
11   constructor(
12     @Inject(‘AppStore’) private appStore: AppStore,
13     private todoActions: TodoActions
14   ){ }
15
16   private onTodoClick(id){
17     this.appStore.dispatch(this.todoActions.toggleTodo(id));
18   }
19  }
```

todo.js hosted with by GitHub view raw

> *TypeScript tip: using private or public modifiers in the constructor arguments is a shortcut for declaring private or public properties (lines 12–13). See private/public modifiers.*

Toggling the initial example todo would produce the following action:

```
{
  type: 'TOGGLE_TODO',
  id: 0
}
```

toggleTodo.js hosted with by **GitHub** **view raw**

As before, dispatching the action will execute the reducer and create a new state.

```
//src/rootReducer.ts
case TodoActions.TOGGLE_TODO:
 return {
   todos: toggleTodo(state.todos, action),
   currentFilter: state.currentFilter
 };
...
function toggleTodo(todos, action){
  //map returns new array
  return todos.map(todo => {
    //skip other items
    if (todo.id !== action.id)
      return todo;
    //toggle
    return {
      id: todo.id,
      text: todo.text,
      completed: !todo.completed
    };
  });
}
```

toggleReducer.js hosted with by **GitHub** **view raw**

The helper function ***toggleTodo*** creates a new array toggling the todo matching the ***action.id*** being dispatched and maintaining the rest.

Filtering Todos

The ***Filters*** component allows the user to filter: all, only active or only completed todos. We use ***FilterLink*** components to encapsulate each filter passing an identifier through the attribute ***filter***.

```
//src/filters.ts
<filters>Show:
  <filter-link filter="SHOW_ALL"><a>All</a><filter-link>
  <filter-link filter="SHOW_ACTIVE"><a>Active</a><filter-lin
  <filter-link filter="SHOW_COMPLETED"><a>Completed</a><filt
</filters>
```

filters.html hosted with by **GitHub** **view raw**

Within ***FilterLink*** each click event passes down the ***filter*** (input attribute, line 6) and dispatch the corresponding filter action.

```
//src/filterLink.ts
@Component({
 selector: 'filter-link',
 inputs: ['filter'], //attribute
 template:
 `<a href=”#” (click)=”applyFilter(filter);”>` +
   `<ng-content></ng-content>` +
 `</a>`
})
export class FilterLink {

 private applyFilter(filter) {
   this.appStore.dispatch(
     this.todoActions.setCurrentFilter(filter)
   );
 }
}
```

filterLink.js hosted with by **GitHub** **view raw**

Filtering by ***Completed*** will generate the following action

```
{
  type: 'SET_CURRENT_FILTER',
  filter: 'SHOW_COMPLETED'
};
```

currentFilter.js hosted with by **GitHub** **view raw**

As before, dispatching the action will execute the reducer and create a new state. In this case, we keep the same todos and change the current filter with the one dispatched (lines 5).

```
1 //src/rootReducer.ts
2 case TodoActions.SET_CURRENT_FILTER:
3  return {
4    todos: state.todos.map(todo => todo), //map creates a new
5    currentFilter: action.filter
6  };
```

setCurrentFilter.js hosted with by **GitHub** **view raw**

Displaying the Todo List

We will use a child component ***todo*** to encapsulate a single todo passing some properties as attributes (id, completed) and the description (text) as content. This pattern is known as ***Container Component***.

```
1 //src/todoList.ts
2 <ul>
3  <todo *ngFor=”#todo of todos”
4    [completed]=”todo.completed”
5    [id]=”todo.id”
6  >{{todo.text}}</todo>
7 </ul>
```

todolist.js hosted with by GitHub view raw

We are using ***ngFor*** to iterate over the ***todos*** Array (line 3). For each todo we are passing down the todo information using a local template variable ***#todo***.

Following see an extract of the ***TodoList*** component.

```
 1  //src/todoList.ts
 2  export class TodoList implements OnDestroy {
 3    constructor(
 4      @Inject('AppStore') private appStore: AppStore
 5    ){
 6      //subscribe listener to state changes
 7      this.unsubscribe = this.appStore.subscribe(function lis
 8        let state = this.appStore.getState();
 9        this.todos = state.todos;
10      });
11    }
12
13    private ngOnDestroy(){
14      //remove listener
15      this.unsubscribe();
16    }
17  }
```

todoList.js hosted with by **GitHub** **view raw**

Above, we registered a listener using ***appStore.subscribe*** (line 7). Once within our listener, we can easily read the current state using ***appStore.getState*** (line 8). Subscribe returns a function that we can use to unsubscribe. In Angular 2 we use the ***OnDestroy*** event handler for clean up (lines 2, 13–16).

> *Note how we kept all component properties and helper methods as private. We don't want other components accessing them.*

Redux life-cycle review

Let's review how a Redux Application behaves at different stages.

- **On Application bootstrap**: we initialise the ***appStore*** passing the ***rootReducer***. This will trigger ***appStore*** internal initialisation. Usually this sets the ***initialState***.
- **On Component creation**: We inject ***appStore*** and ***TodoActions*** on the constructor as required. Components that display data subscribe to the ***appStore*** and read it by calling ***appStore.getState()***. Components that mutate the state prepare dispatch code for the corresponding action passing any required data.
- **On Component destruction**: Components that display data ***unsubscribe*** to the ***appStore*** to clean up resources.
- **On User interactions**: each user interaction will trigger an underlaying dispatch action. This will execute the ***rootReducer*** producing a new state. The ***appStore*** will then notify all subscribed listeners that will update accordingly.
- **On Server-side initiated actions**: some Applications can dispatch actions in response to server-side initiated events. Eg: WebSockets. These actions once properly setup follow the same flow as User interactions.

We covered how to build a basic Angular 2 Application using Redux. Hope you are now curious about Redux and maybe use it in your next project.

Thanks for reading!

Further Reading

- Watch Getting started with Redux video course, by Dan Abramov @Dan_Abramov
- Smart and Dumb Components, by Dan Abramov
- Building Flux Apps with Redux and Immutable.js, by @JhadesDev
- React Developer Survey Results, by PatrickJS, @AngularClass