Angular and Redux

@ngrx/store

Our Goal

JOHN BRYCE
Leading in IT Education
a matrix company

- Understand Redux
- Understand how to combine Redux with Angular
- Understand how combining the two can improve performance

Our Goal

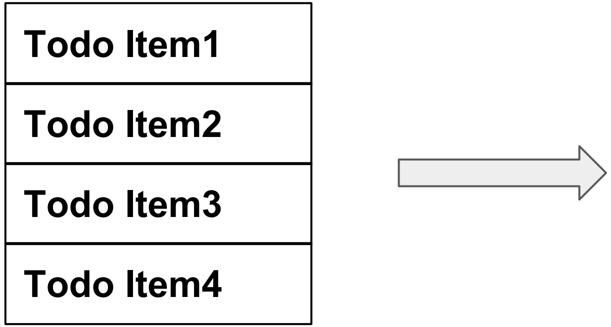
- Following our Redux lesson, our objective:
  - Create a todo list app
  - Entire app with chage detection strategy OnPush

What is a State

- In college at automation course we learned about Finite-state machine (FSA)
- FSA is a mathematical model of computation
- FSA can be in one state out of a group of final states at any given time
- FSA can transition to a new state based on certain change in input
- Can we look at our frontend application as an FSA?
- What can be the cause of a transition in state in a frontend web app?

Todo App State

- In our todo app we have a component that displays the list of todo items
- At the beginning, the list is empty and we query the server for the list of items
- After the server return our response we display the list of tasks
- The todo list component looks like this:



Todo Item1

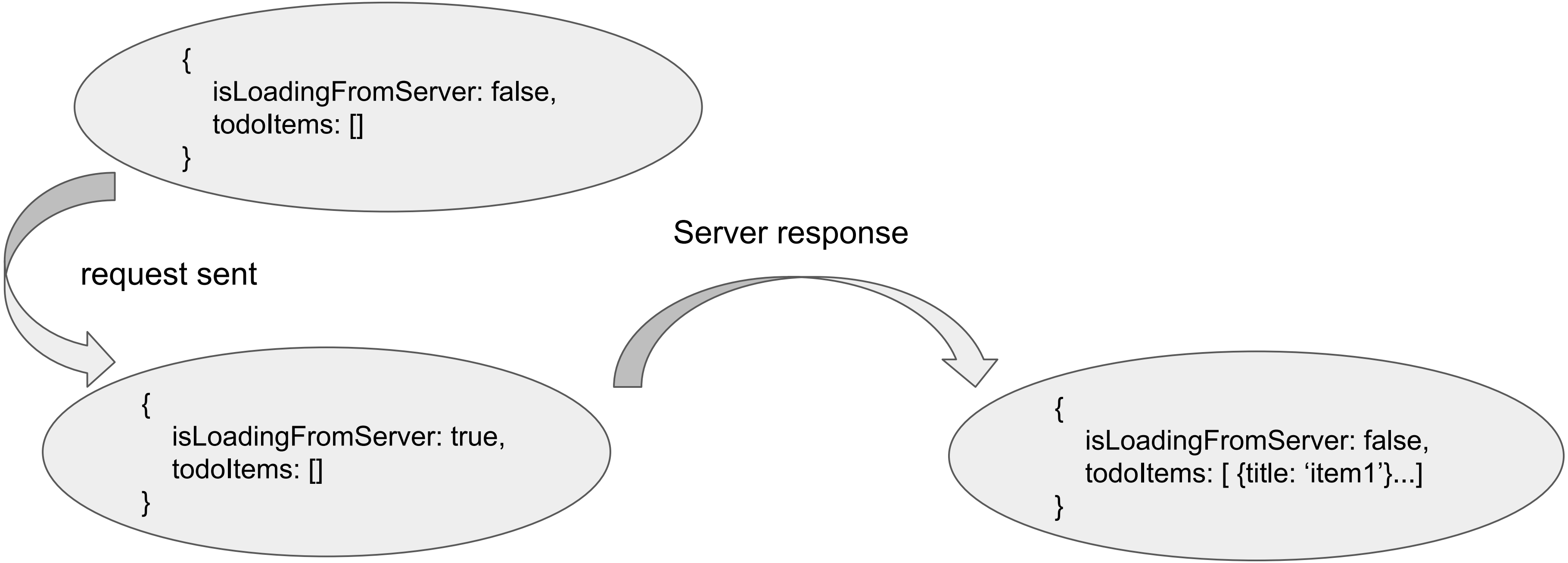
Todo Item2

Todo Item3

Todo Item4

This list is from the server

State for Todo App



```
{
    isLoadingFromServer: false,
    todoItems: []
}
```

request sent

```
{
    isLoadingFromServer: true,
    todoItems: []
}
```

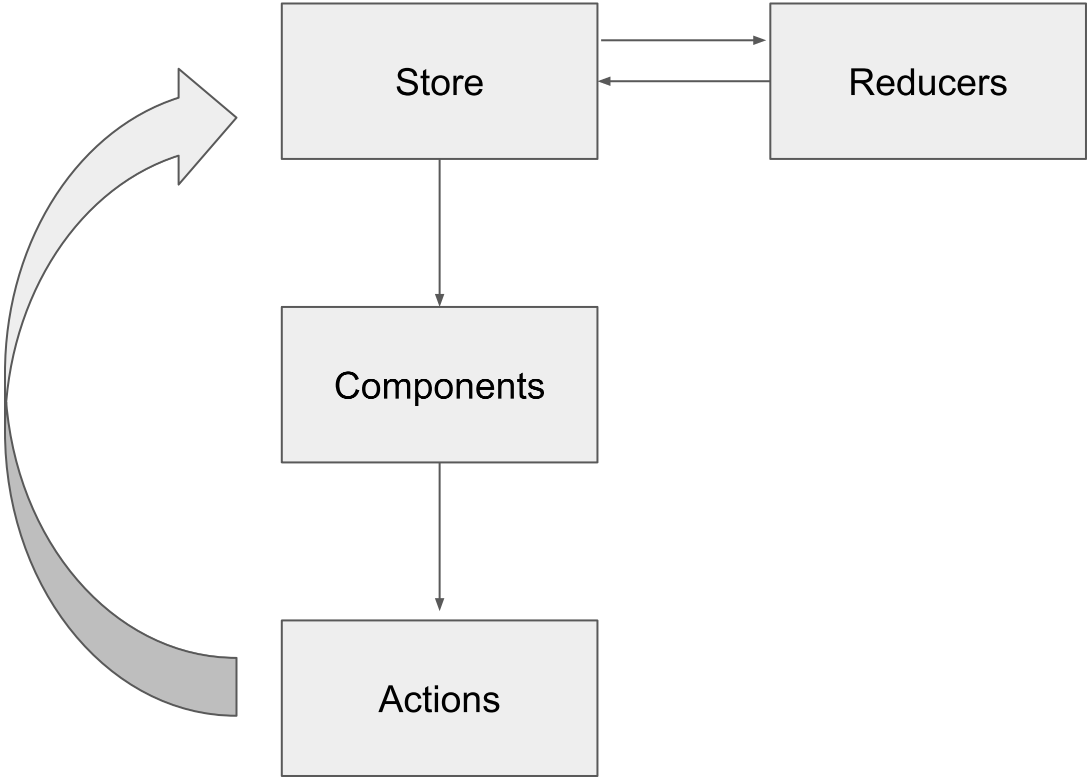
Server response

```
{
    isLoadingFromServer: false,
    todoItems: [ {title: ‘item1’}...]
}
```

What is Redux

- Predictable state container for JS apps
- Using redux we have a single object holding the state of our app which is called **store**
- The state in the store can be an Object, Array, Primitive but it's highly recommended that it will be immutable
- What data will we hold in the state, and what data in the components?
- The only way to change the state is by calling a method on the store called **dispatch**
  - **store.dispatch(action)**
- An action is a simple object describing **what happened**
- The store decides how the state will change using a pure function called reducer
- The reducer answers the questions **How does the state change?**
  - **(state, action) => state**

Redux architecture - Uni Directional Data Flow



Redux Core

Redux core concepts are:

- Actions
- Reducers
- Store

Let’s try to demonstrate redux usage using the state of our todo application

Actions / Action creators

- Action is a simple JS Object
- The action needs to hold the following information
  - identifier of the action (string)
  - data associated with the action
  - example: **{type: ‘SET_LOADING’, payload: true }**
- Action creator is a pure function which returns an action

```
function setIsLoading(isLoading) {

    return {type: ‘SET_LOADING’, payload: isLoading};

}
```

- What action creators do you think our todo app requires?

Reducers

- Reducers are pure functions
- Reducers get the current state and action as arguments
- The reducer need to decide based on the action and the current state what the next state will be
- The state has to be immutable to guarantee that no one will mutate the state in the reducer we will use a library called **immutable.js**
- When our state grows if we use only a single reducer the reducer will be huge so it is better to split the state to sections and also split the reducers
- Redux has a function called **combineReducers** to help us split the app and make each reducer in charge of certain section of the state
- Before we dive into reducers let's do a small intro on **immutable.js**

immutable.js

- immutable data cannot be changed after created
- immutable.js offer us api to change the data but instead of changing the data it will yield a new immutable object
- immutable.js offer us immutable data structures we will cover the popular ones:
  - Map
  - List
- We can install immutable.js with npm
  - **npm install immutable --save**
- let’s try and create a map and a list and see the common api for those data structures

Todo reducer

- Our todo app will contain a single reducer
- When initiated redux will call the reducer with the state undefined so we can utilize this fact to set our initial state
- let’s create the reducer for our todo app

combineReducers

- When our state grows so will our single reducer
- We want to split the state to logical sections
- Let’s say our todo application will need to hold information about the current user
- We will create another reducer that will handle the user info and combine those reducers with the **combineReducers** function

Store

- there is a single store in redux application
- the store holds the state of our app
- to create the store we use **createStore**
    - **createStore(reducers, initialState)**
- we need to pass to the **createStore** the combined reducers and optional initial state
- Let’s create our store

Redux Middlewares

- The only way to change the state is using the **store.dispatch** method
- A middleware will decorate the **dispatch** method and add logic before and after the store calls the reducer
- There is a function in redux called **applyMiddleware** which takes the list of middlewares we want to apply
- The result is enhanced **dispatch** method which we place as the third argument in the **createStore**
- We won’t cover how to create middleware of our own cause in common use cases we will use community middlewares
- Let’s install our first middleware

Redux-thunk - Async Actions and Query Server

- redux-thunk is a redux middleware
- it allows action creators to return a function
- if action creator is returning a function it will be run by thunk middleware
- the function returned will be called with 2 arguments
  - **dispatch**
  - **getState**
- we can use redux thunk to delay store dispatch, condition the dispatch, run async code like server communication
- we can install it with npm: **npm install redux-thunk --save**
- we need to install it as a middleware to our redux store
- let’s create an action which queries the server for todo items

Redux dev tools

- one of the strong features of redux is the easy of testing and easy of development
- you have a state and the app should behave according to the state
- for development purpose it’s better if we can easily examine the current state and all the actions that got us to this state
- there is an easy to use browser extension:
  - ○ https://github.com/zalmoxisus/redux-devtools-extension
- to install the devtool extension: **npm install redux-devtools-extension --save-dev**

```
const store = createStore(reducer, composeWithDevTools(
  applyMiddleware(...middleware),
  // other store enhancers if any
));
```

Combining with Angular

- Now that we know what Redux is, lets try and combine Angular and redux together using **@ngrx/store**

What is @ngrx/store

- state management for Angular applications inspired by redux
- we select items from the state and get them as observables
- we then use async pipe to display them in the components
- async pipe will cause rerender even if component is in change strategy on push
- install with npm:
  - **npm install @ngrx/store --save**

Task Model and Interface

- Let’s start by creating our task model
- We will also create an interface for our model

Todo Service

- Lets create a service with a method to grab all the tasks from the server

Store as a Module

- Similar to how we placed routing in separate modules, we will do the same with our state
- Each module which needs to add items to our state will have that logic in a store module
    - **app-store** - will contain store module for the root module
    - **user-store -** will contain store module for the feature module called user
- Inside each module we can split the reducers based on logical sections

Actions

- Let’s start by implementing our actions
- We will the string types of the actions in a class with static properties called TodoActionsTypes
- Each action will implement Action interface
- We will also use type alias to name the type of actions

Reducer

- We will define an interface that describes the section of the state that this reducer is in charge of
- We will define our initial state
- Create our reducer as a pure function with switch and case
- The pure function will use the types defined in the action file

Combining Reducers

- We will create a file called **reducers**
- In this file we will describe the entire state of the app-store module
- We will create **ActionReduceMap<AppInterface>** which is a dictionary which maps keys to reducers

Store Module

- We will call **StoreModule.forRoot** to create our store
- It will get the reducer map we created before
- For lazy loaded module their state will be lazy loaded as well and we will call the function **StoreModule.forFeature**
- We will add our store module to the app module
- We can now inject the store to our services and components

Modifying the Service

- We can inject in out todo service the store
- In the **map** operator when we get the tasks from the server we can call the **dispatch** method with our action to add the tasks to the state
- In the app component call the service get tasks to initiate the state with the tasks from the server

Todo List Component

- Create a component to display the list of todo tasks
- Inject the store to the component
- We will use the select method on the store to select certain property from the state
- The property will return to us as observable
- We will use the async pipe to display that observable

Summary

- With redux we can manage the state of our app
- The only way to change the state is by calling **store.dispatch(action)**
- Reducers will decide how the state changes
- Combining Angular and redux is a powerful tool which gives us
  - More testable app
  - Better performance
  - Easier management of the data of our app