academeez

@ngrx/store

Redux implementation for Angular

@ngrx/*

academeez

- @ngrx/* are a set of libraries that brings us redux state management for an angular application
- The core redux implementation is located at @ngrx/store library
- We will start by going over @ngrx/store and covering the other libraries as well starting with the core @ngrx/store

Redux

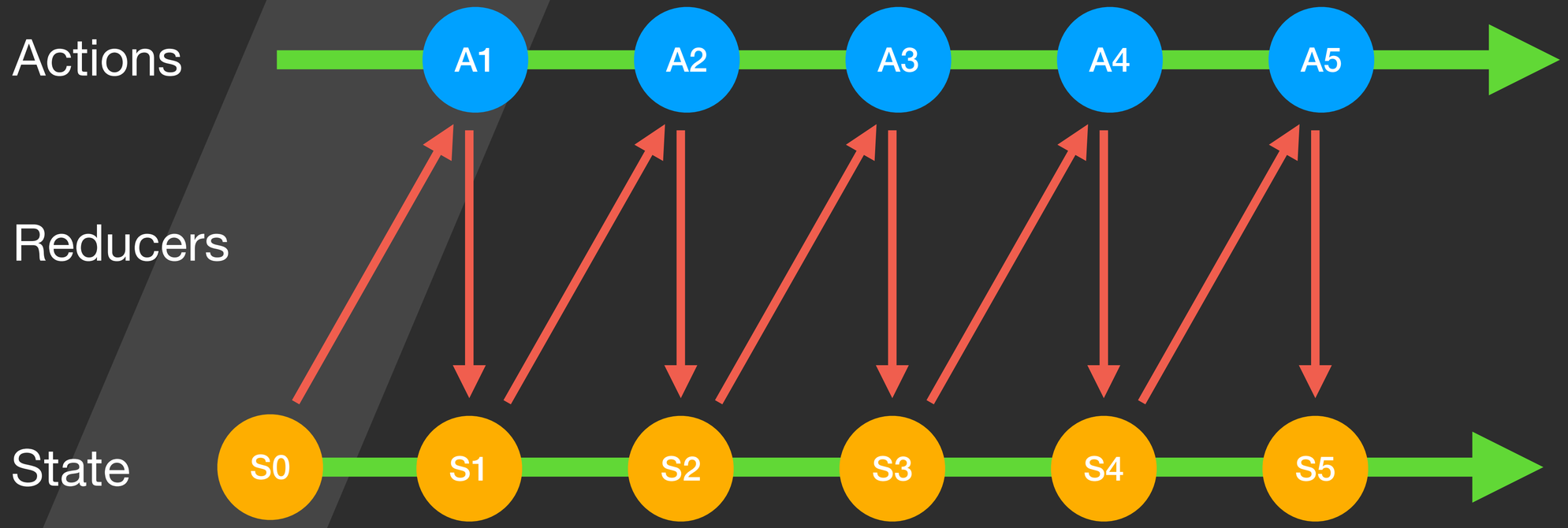
academeez

- In the previous lesson we learned that redux manage our state
- An action can change the state
- And a reducer decides how our state will change

Redux as RXJS

academeez

- The same redux implementation can be done using RXJS and Observables



Redux as RXJS

- @ngrx/store looks at the state as an Observable
- That Observable is implemented in an angular service **Store**
- The **Store.prototype.dispatch** method will accept an Action and will start the process of state change using reducers

@ngrx/store - change/read state

- In this ex. We will try to create 2 components, one is going to change the state and one is going to read the state
- The component that will change the state will contain a form with a text input to send the text to the other component
- The other component will read the text from @ngrx/store state
- Our state will look like this

```
const state = {
  message: 'hello world'
}
```

@ngrx/store flow

academeez

- Reducers determine the state sections and how they will change
- Actions describe a change we want to perform in the state
- The store service dispatch an action with store.dispatch
- The reading of data is done with selectors

@ngrx/store change data

academeez

@ngrx/store read data

academeez

Actions

academeez

- A change in our state must come from an Action
- An action describes the change and pass needed params for the state change
- The action is created using the method **createAction**
- An action has a unique name, and optional params needs for the state change

```
import { createAction, props } from '@ngrx/store';

export const changeMessage = createAction(
  '[message] Change Message',
  props<{message: string}>()
);
```

Reducer

- The action is passed to the reducers via **store.dispatch(action)**
- The reducer will decide if the state will change and how
- We create the reducer with the method: **createReducer**

```
import { createReducer, on, Action } from '@ngrx/store';
import { changeMessage } from '../actions/message.actions';

const initialState = 'hello world'

const featureReducer = createReducer(
  initialState,
  on(changeMessage, (state, action) => action.message ),
);

export function reducer(state: string, action: Action) {
  return featureReducer(state, action);
}
```

Changing the state

- To change the state we ask for the **Store** service and call the **dispatch** method passing the action to change the state.

```
export class SendComponent {
  newMessage: string = '';

  constructor(private _store: Store) {}

  send(event) {
    event.preventDefault();
    this._store.dispatch(changeMessage({message: this.newMessage}));
  }
}
```

Reading the state - Selectors

academeez

- We define Selectors which are functions to select from the state
- These functions are memoized to increase performance
- We create the selectors using **createSelector** method

```
import { createSelector } from '@ngrx/store';

export const selectMessage = createSelector(
  (state: any) => state.message,
  (message) => message
)
```

Reading the state - Component

- To read from the state we grab the **Store** service which is an observable emitting the current state
- We use **pipe** and the **select** operator to grab from the state the part that interests us using **Selectors**
- We get the data wrapped in Observable so we use the async pipe to use the data in the template - which means we can use OnPush

```
export class RecieveComponent {
  message$ = this._store.pipe(
    select(selectMessage)
  );

  constructor(private _store: Store) { }
}
```

Using Service to pass data is much simpler

academeez

- We used to just place the data to a Service and use that service to change the data and read the data, it’s much more complex with NGRX, what is the benefit?
- In ngrx the data Is wrapped in Observable which means we can use OnPush, to achieve the same in a Service we would have to manually wrap it in a Subject or Observable
- We split the change to actions and reducers, this way we can collect the action in an array and see the data change along a timeline
  - We achieve predictability
  - Easy testing
  - Easy undo redo
- We split the Selectors to improve reading performance

Predictability

- The fact that we separate the actions allows us to collect the array of actions and see exactly how the state got to it's current position
- We can examine the actions using a browser extension called **redux dev tools**
- We can install the package **@ngrx/store-devtools** and add the module to the imports array to connect our store to the devtools
- We can now examine our state and the actions that led us to the current state

Summary

academeez

- With @ngrx/store our data is managed using redux
  - Actions change the state
  - The reducer decides how the state will change
  - Components can read from the state using selectors
  - Components can change the state using the **store.dispatch**

academeez

Thank You

Next Lesson: @ngrx/effects