



Angular and NgRx 16 – Signals Store

we will start in a few minutes

Kobi Hari
(hari@applicolors.com)



Hi, I'm...



Kobi Hari

- Freelancer
- Developer, Instructor and Consultant
- Angular, Async development, .NET core

 hari@applicolors.com

 <https://github.com/kobi2294>

 <https://www.youtube.com/@kobihari-applicolors1464>

We have a GitHub Repository!

[kobi-hari-courses/2306-dev-geek-week-ngrx](https://github.com/kobi-hari-courses/2306-dev-geek-week-ngrx)



Session Summary

Project Code with
Commits Summary

This presentation and other
Useful Links

Commits

#	Link	Description
1	Initial application	Created an empty application using the angular CLI: ng new redux-pop-quiz
2	Added Material Design	Added angular material using the cli: ng add @angular/material
3	Custom Theme	Defined the theme palettes in partial _common.scss and generated the material styles in styles.scss
4	Tested theme	Used some angular material components like icon and button to test the new theme
5	Grid, Flex layout	Created layout surface with material flexbox grid or flex for easier layout management
6	CSS Variables	Added CSS variables to the theme

Building the UI

#	Link	Description
7	Toolbar	Added material toolbar

presentations

projects

LICENSE

README.md

Our Agenda



The Redux Pattern



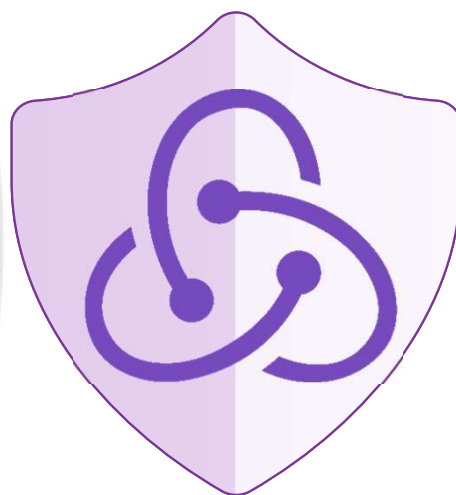
NgRx Store – Selectors, Actions, Reducers



NgRx Effects – Reactivity On Steroids



NgRx Component Store – Local Stores



The “Redux” Pattern

The Redux Pattern



Single Point of Truth



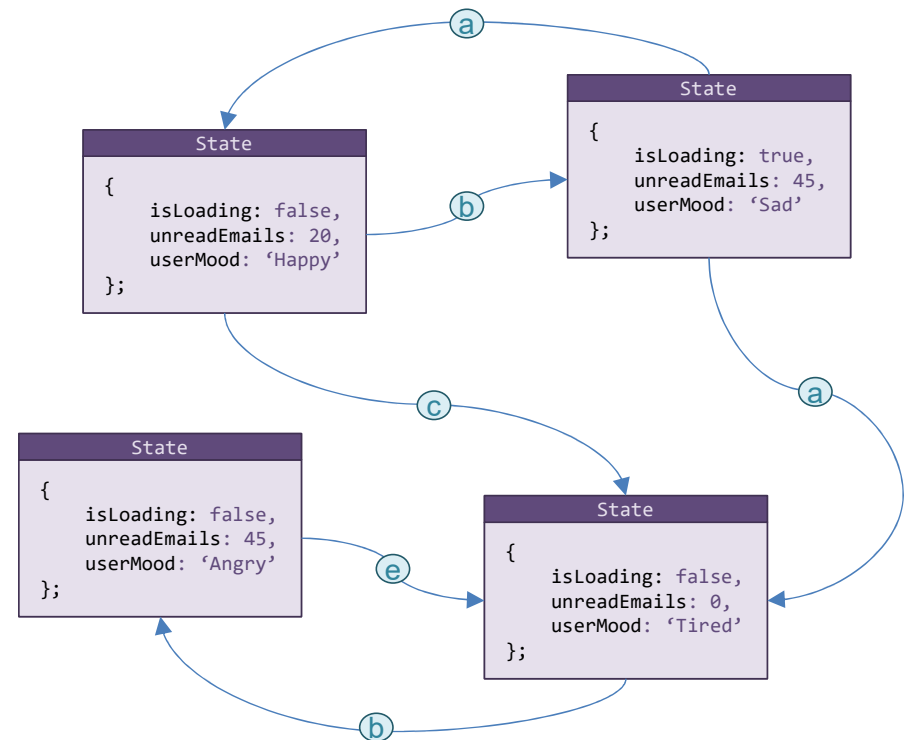
Immutability

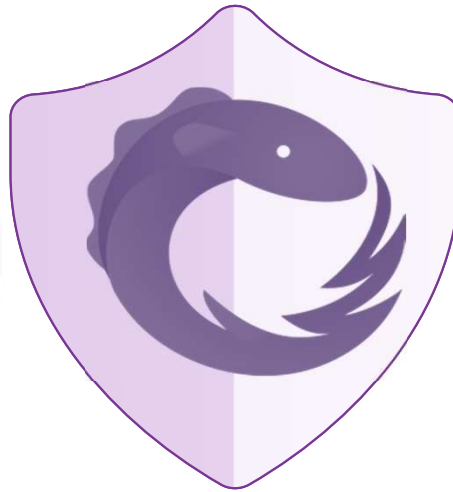


Pure functions - Reducers

What is a “State”

- **State** is an object holding all the changeable data in the application
- Any change to data means that you move from one **state** to another
- **State** is replaced as a result of an Action



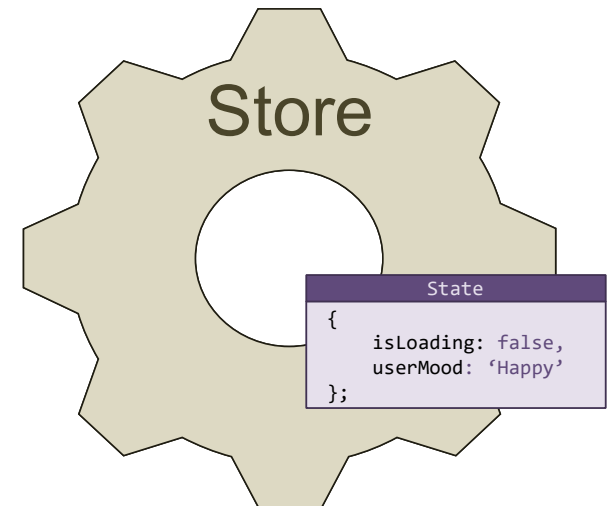


NgRx Store

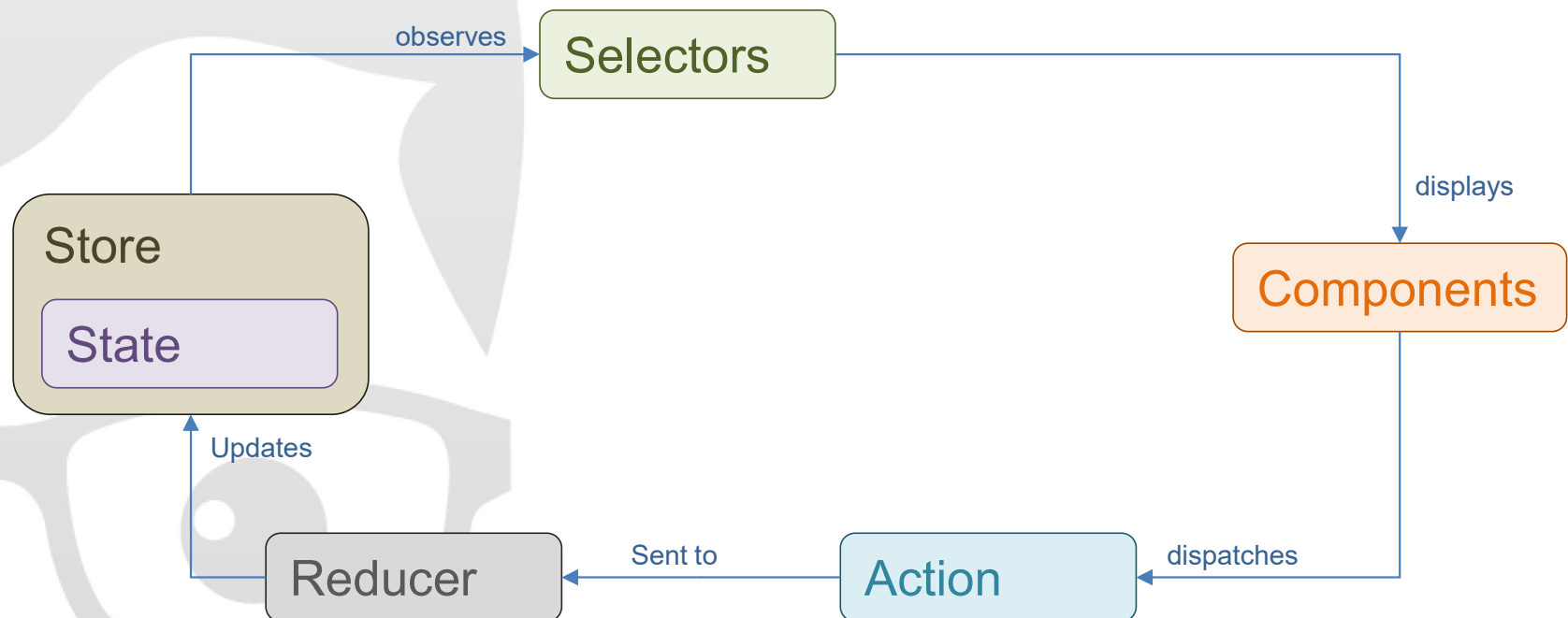
Selectors, Actions and Reducers

The Store

- The store is a service that holds the state
- The structure of the state is private (!)
- Reading or Updating the state is done indirectly using **Selectors** and **Actions**



The Circle of Life



Demo - Setting up the Store



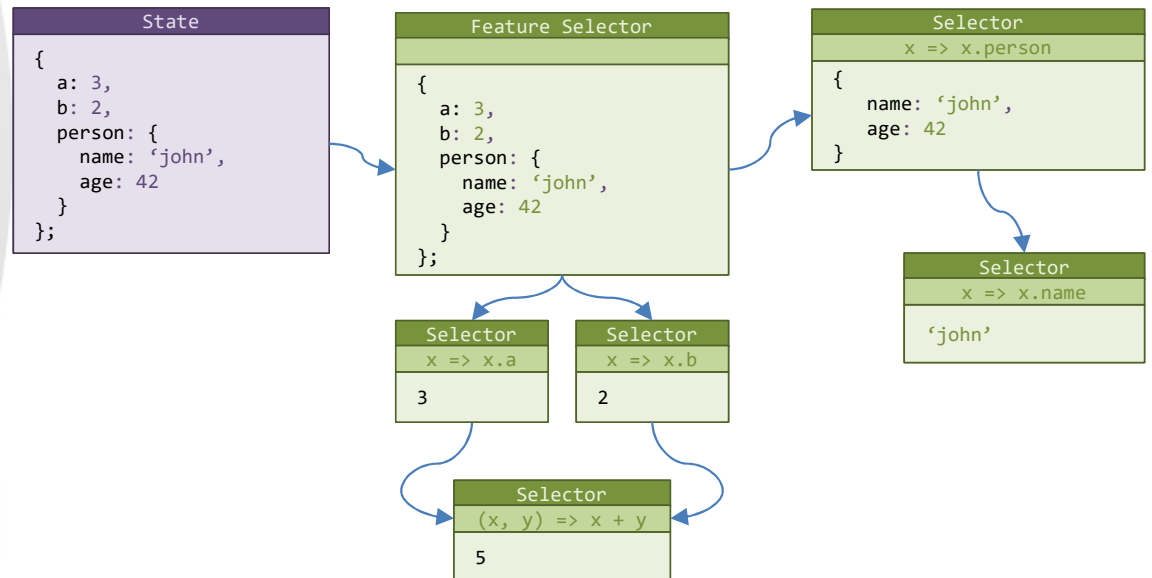
- ✓ Installing the packages
- ✓ Creating State and Store
- ✓ Instrumenting The dev-tools
- ✓ Creating features
- ✓ Providing the feature

Do you want to see
me code?



Selectors

- Selectors select a piece of the state
- A selector is based on:
 - Parent selector (or many)
 - Projection function
- They can also select “calculated expression” of the state



Selecting from the store

- Selectors are used to create observables
- These observables are memoized and optimized

 `store.select(`

Selector
<code>x => x.person.name</code>

`)`



`Observable<string>`



Demo – Using Selectors



- ✓ Atomic Selectors
- ✓ Adding Extra Selectors
- ✓ Deriving Selectors from multiple sources
- ✓ Using selectors in Components

Do you want to see
me code?



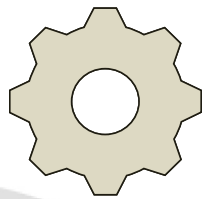
Actions

- Action is a simple JSON
- It must contain a “**type**” property
- It may contain additional properties (called the “**payload**”)
- NgRx provides “Action Creators”
- Actions describe **events**.
- By convention the **type** should look like this:
‘[SOURCE] name’

```
Action
{
  type: '[USER] set loading',
  value: true
};
```


Dispatching Actions

- Use the **Store** to dispatch actions



store.dispatch

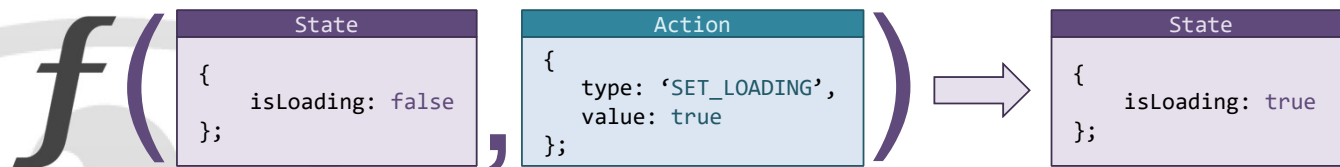
```
Action  
{  
  type: '[USER] set loading',  
  value: true  
};
```

Action Groups

- NgRx 16 now has a new concept for actions:
 Action Group
- Action Groups are used to group together actions from the same source
- Angular uses Typescript Templates to pull some neat compiler tricks

Reducers

- Reducer is a “Pure Function”
- It calculates new state from old state, and an action



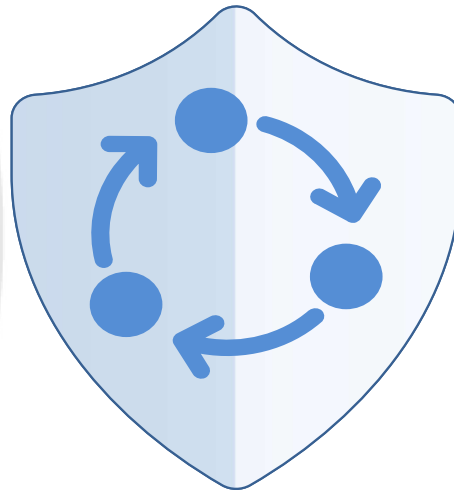
Demo – Using Action Groups and Reducers



- ✓ Creating Action groups
- ✓ Using the `props` functions to define payload
- ✓ Dispatching actions
- ✓ Implementing the Reducers

EUROPE! Do you want to see me code?



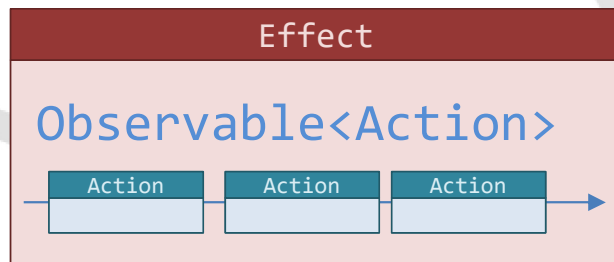


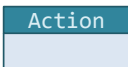
NgRx Effects

Reactivity On Steroids

What are “Effects”


- Effects are Observables
 - That the store subscribes to
 - That yield actions
 - That get dispatched automatically

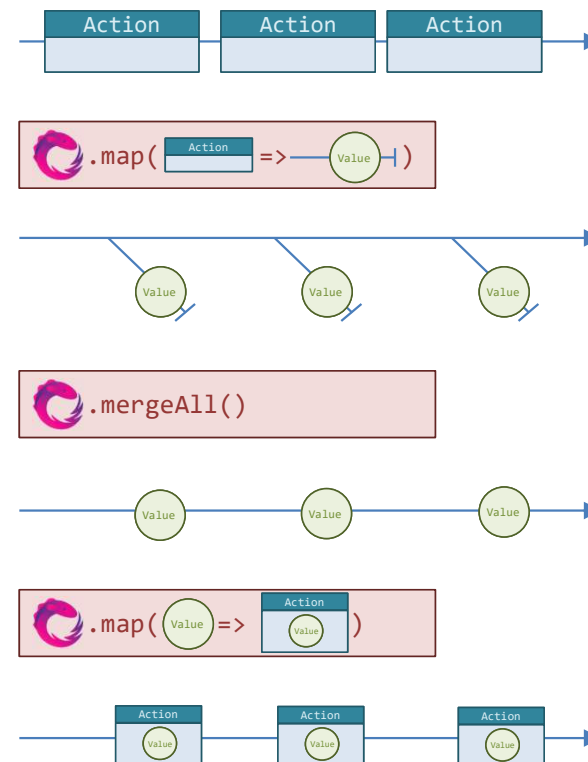


.subscribe( =>

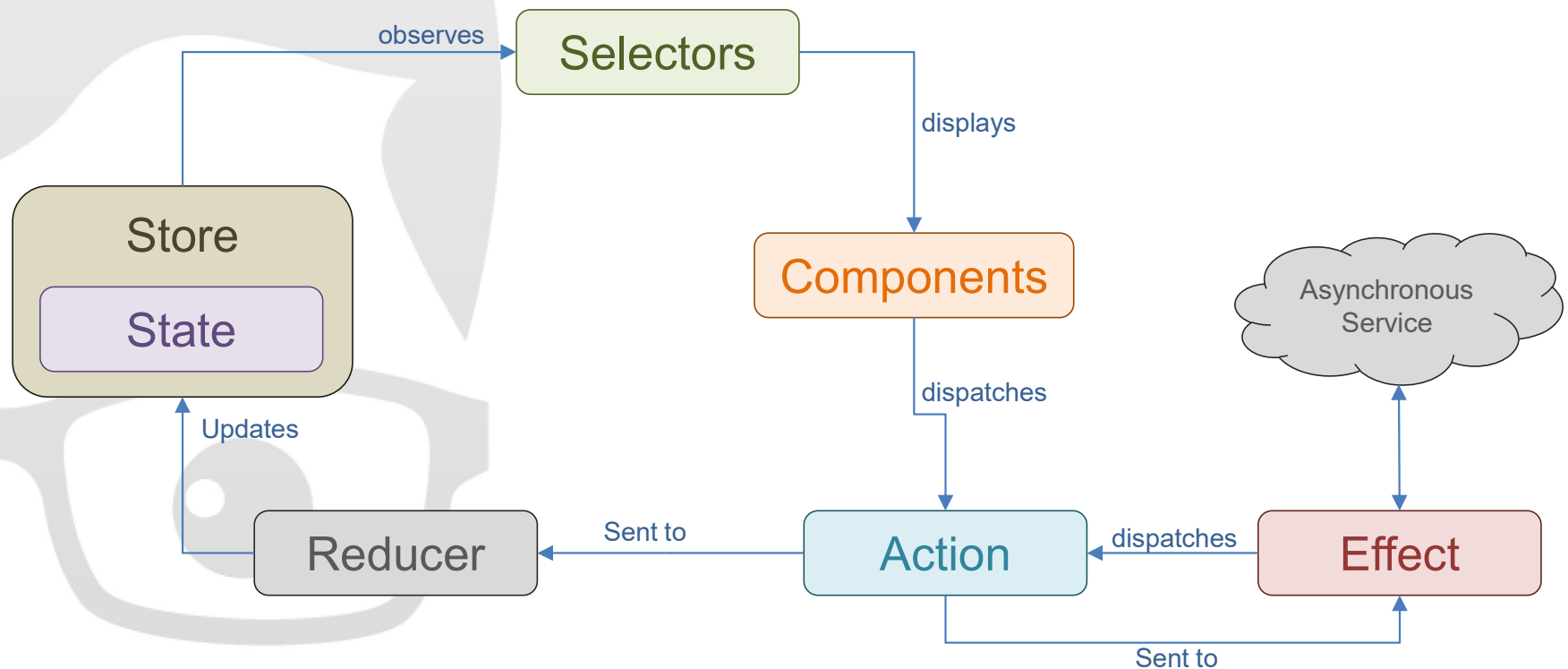
 .dispatch())

Fun facts about Effects

- They are usually created from other observables of **actions**
- They are built using  **RxJS operators**
- They are often mapped to asynchronous **values**
- They often cause.... **Side effects**
(AH HA!!!!)



The Circle of Life – with effects



Actions drive the app

- **Actions** are not just for the store
- In the “Reactive” thinking, everything that can happen in the application is an **action**.
- **Effects** handle **Actions** by performing operations that are translated into other actions
- **Reducers** handle **Actions** by changing the current state of the application

Demo – Using Effects



- ✓ Installing the @ngrx/effects package
- ✓ Writing Effects as functions
- ✓ Reacting to Actions or State
- ✓ Invoking and Flattening async methods
- ✓ Handling Errors
- ✓ Registering the Effects

Watch me!





NgRx Component Store

Local Store as a Service

Component Store

- Component Store is a service
- It is provided by a component
- It is usually tied to the life-cycle of the component
- It is like a “baby” store
 - It has **selectors**
 - It has “reducer-like” methods
 - It has **effects**
 - It has no **actions** (!!!)

Creating a component store

- Define a state: `T`
- Create a service that extends `ComponentStore<T>`
- You may set the initial state in the `constructor` by calling `super`

```
export interface MoviesState {  
  movies: Movie[];  
}  
  
@Injectable()  
export class MoviesStore extends ComponentStore<MoviesState> {  
  
  constructor() {  
    super({movies: []});  
  }  
}
```

Consuming the Store

- In the component, provide it using the “**providers**” array
- In the component or any of its descendants, inject it using **dependency injection**.
- You can create observables of slices of data using the **select** method

```

1. @Component({
2.   template: `
3.     <li *ngFor="let movie of (movies$ | async)">
4.       {{ movie.name }}
5.     </li>
6.   `,
7.   providers: [ComponentStore],
8. })
9. export class MoviesPageComponent {
10.   readonly movies$ = this.componentStore.select(state => state.movies);
11.
12.   constructor(
13.     private readonly componentStore: ComponentStore<{movies: Movie[]}>
14.   ) {}
15.

```


Demo – Using Component-store



- ✓ Installing the @ngrx/component-store package
- ✓ Creating a store service
- ✓ Initializing the state
- ✓ Providing it in the component
- ✓ Injecting it into the component
- ✓ Consuming the state

Phenome
Phenome
Phenomenal



Creating Selectors

- You can define observable properties by using the `select` method.
- You can compose selectors into a new selector
- You can also base selectors on `global store` selectors

```

1. export interface MoviesState {
2.   movies: Movie[];
3.   userPreferredMoviesIds: string[];
4. }
5.
6. @Injectable()
7. export class MoviesStore extends ComponentStore<MoviesState> {
8.
9.   constructor() {
10.     super({movies:[], userPreferredMoviesIds:[]});
11.   }
12.
13.   readonly movies$ = this.select(state => state.movies);
14.   readonly userPreferredMovieIds$ = this.select(state => state.userPreferredMoviesIds);
15.
16.   readonly userPreferredMovies$ = this.select(
17.     this.movies$,
18.     this.userPreferredMovieIds$,
19.     (movies, ids) => movies.filter(movie => ids.includes(movie.id))
20.   );
21. }

```

Demo – Component store **Selectors**



- ✓ Creating Selectors
- ✓ Deriving Selectors from other selectors
- ✓ Combining Global store Selectors
- ✓ Selecting Observables
- ✓ Selecting View Models
- ✓ Selecting Signals

Se-lec-tors!
Se-lec-tors!



Setting the state

- You can use `setState` and `patchState` to modify the current state.
 - `setState` takes a full state as parameter
 - `patchState` takes a partial state as parameter
 - Both can either take the final state, or a function that reduces it from the current state
- It is recommended to only use these in the service implementation

```
1  resetSelectedCharacters() {  
2    this.setState({selectedCharacterIds: []});  
3  }  
4  
5  selectCharacter(characterId: number): void {  
6    this.setState((state) => {  
7      return {  
8        ...state,  
9        selectedCharacterIds: [...state.selectedCharacterIds, characterId],  
10       };  
11     });  
12   }
```

Creating an Updater

- Updater is replacing “**action** + **reducer**”
- The **updater** method creates a function
- Calling the function is like dispatching an **action** that updates the state

```
@Injectable()
export class MoviesStore extends ComponentStore<MoviesState> {

  constructor() {
    super({movies: []});
  }

  readonly addMovie = this.updater((state, movie: Movie) => ({
    movies: [...state.movies, movie],
  }));
}
```

Consuming an Updater

- Updater can be called Imperatively or Declaratively
 - Pass arguments to call it Imperatively
 - Pass an Observable to call it Declaratively

```
this.store.addMovie({movie});  
  
this.store.setMovie({this.form.valueChanges$})
```

Demo – Component store Updaters



- ✓ Using setState and patchState
- ✓ Passing callbacks to setState and patchState
- ✓ Defining an Updater
- ✓ Using Updater imperatively
- ✓ Using Updater Declaratively

Chalas...
Mitzinoo



Creating Effects

- Create effect using the “**effect**” method
 - It consumes an Observable of parameters
 - Use the **tap** operator to create side effect
 - Use **catchError** to handle errors
- Or... you can use **tapResponse** as a shortcut.

```
// Each new call of getMovie(id) pushed that id into movieId$ stream.
readonly getMovie = this.effect((movieId$: Observable<string>) => {
  return movieId$.pipe(
    // 🚀 Handle race condition with the proper choice of the flattening operator.
    switchMap((id) => this.moviesService.fetchMovie(id).pipe(
      // 🚀 Act on the result within inner pipe.
      tap({
        next: (movie) => this.addMovie(movie),
        error: (e) => this.logError(e),
      }),
      // 🚀 Handle potential error within inner pipe.
      catchError(() => EMPTY),
    )),
  );
});
```

```
readonly getMovie = this.effect((movieId$: Observable<string>) => {
  return movieId$.pipe(
    // 🚀 Handle race condition with the proper choice of the flattening operator.
    switchMap((id) => this.moviesService.fetchMovie(id).pipe(
      // 🚀 Act on the result within inner pipe.
      tapResponse(
        (movie) => this.addMovie(movie),
        (error: HttpResponse) => this.logError(error),
      ),
    )),
  );
});
```

Consuming Effects

- An **effect** is a function
- Calling it is like dispatching an action that is handled by an effect
- You can call it imperatively or declaratively

```
this.store.getMovie(movieId);  
  
const movieId$ = this.activatedRoute.params  
    .pipe(map(p => p['id']));  
  
this.store.setMovie(movieId$);
```

Demo – Component store Effects



- ✓ Creating effects
- ✓ Using tapResponse
- ✓ Consuming effects imperatively
- ✓ Consuming effects declaratively

Yalla...
Hayity Kan



Thanks :)

See you next year

