CLONE AND FOLLOW THE SETUP INSTRUCTIONS

github.com/**CodeSequence/componentsconf2019-ngrx-workshop**

A REACTIVE STATE OF MIND

WITH ANGULAR AND NGRX

Mike Ryan

@MikeRyanDev

Software Architect at Synapse

Google Developer Expert

NgRx Core Team

Dr Phillip Zada

@PhillipZada

Founder and Managing Director @ Z Ware

Chief Solutions Architect

Love NGRX!

Open source libraries for Angular

Built with reactivity in mind

State management and side effects

Community driven

SCHEDULE

- Demystifying NgRx
- Actions
- Reducers
- Store
- Selectors
- Effects

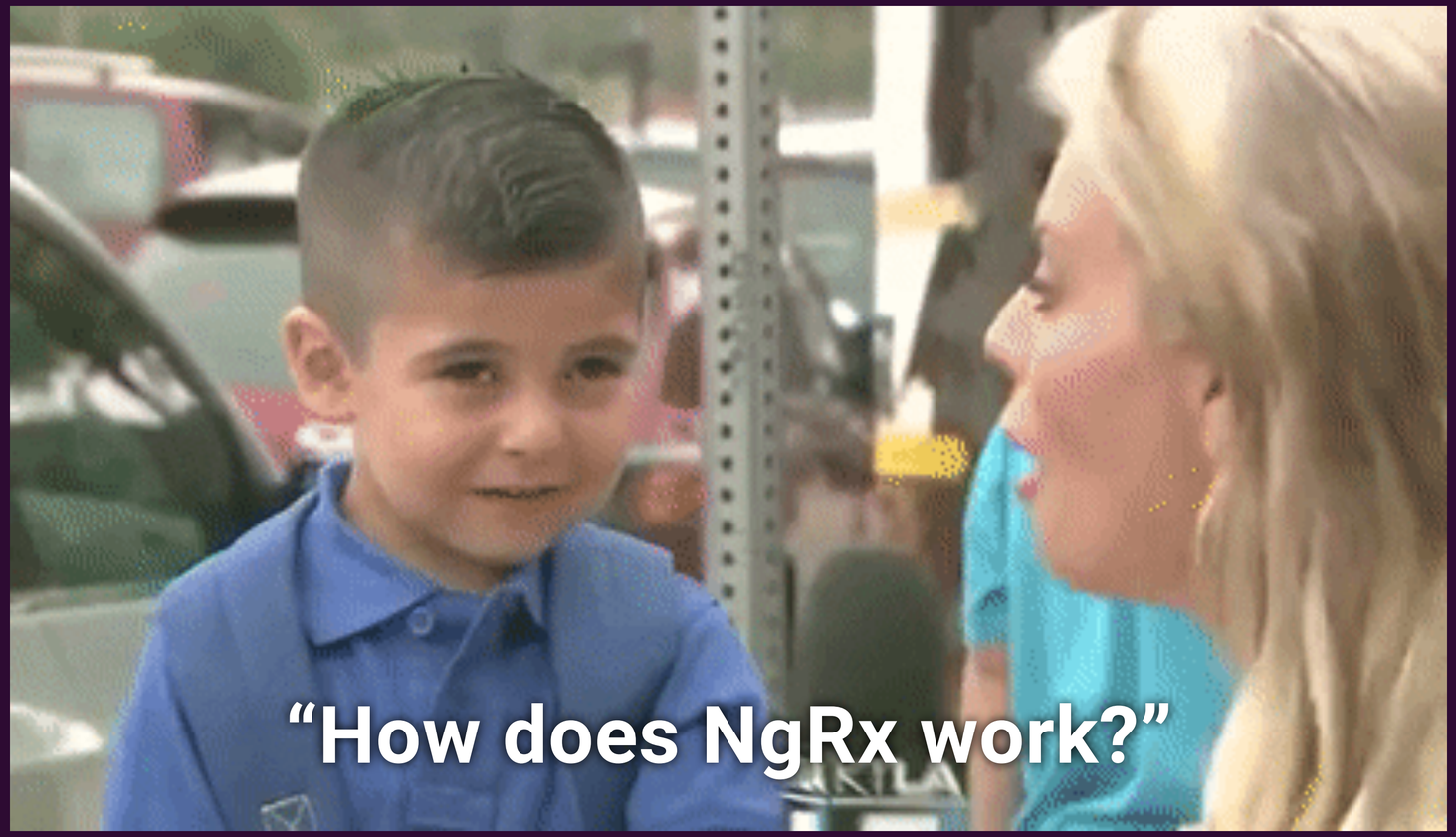
FORMAT

1. Concept Overview
2. Demo
3. Challenge
4. Solution

The Goal

Understand the architectural implications of NgRx and how to build Angular applications with it

DEMYSTIFYING NGRX

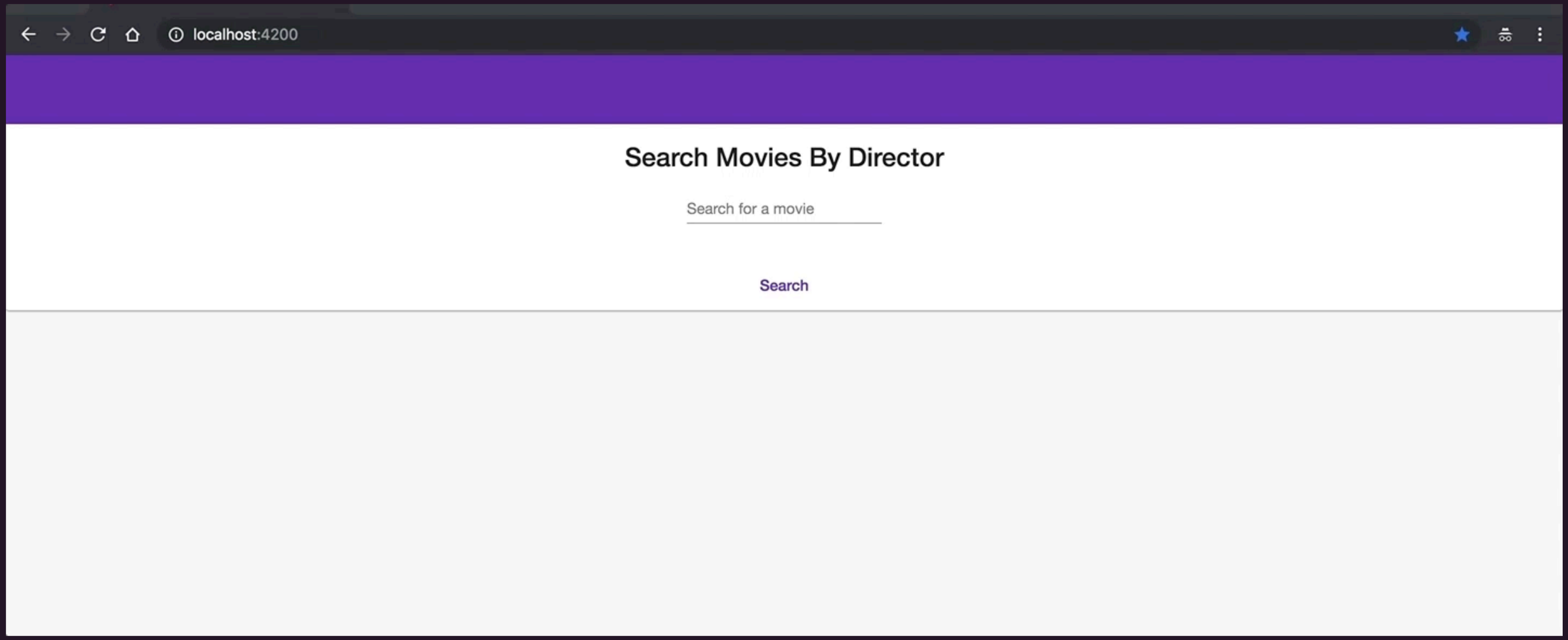

“How does NgRx work?”

- NgRx prescribes an architecture for managing the state and side effects in you Angular application. It works by deriving a stream of updates for your application’s components called the “action stream”.
- You apply a pure function called a “reducer” to the action stream as a means of deriving state in a deterministic way.
- Long running processes called “effects” use RxJS operators to trigger side effects based on these updates and can optionally yield new changes back to the actions stream.

Let’s try this a different way

You already know how NgRx works

COMPONENTS

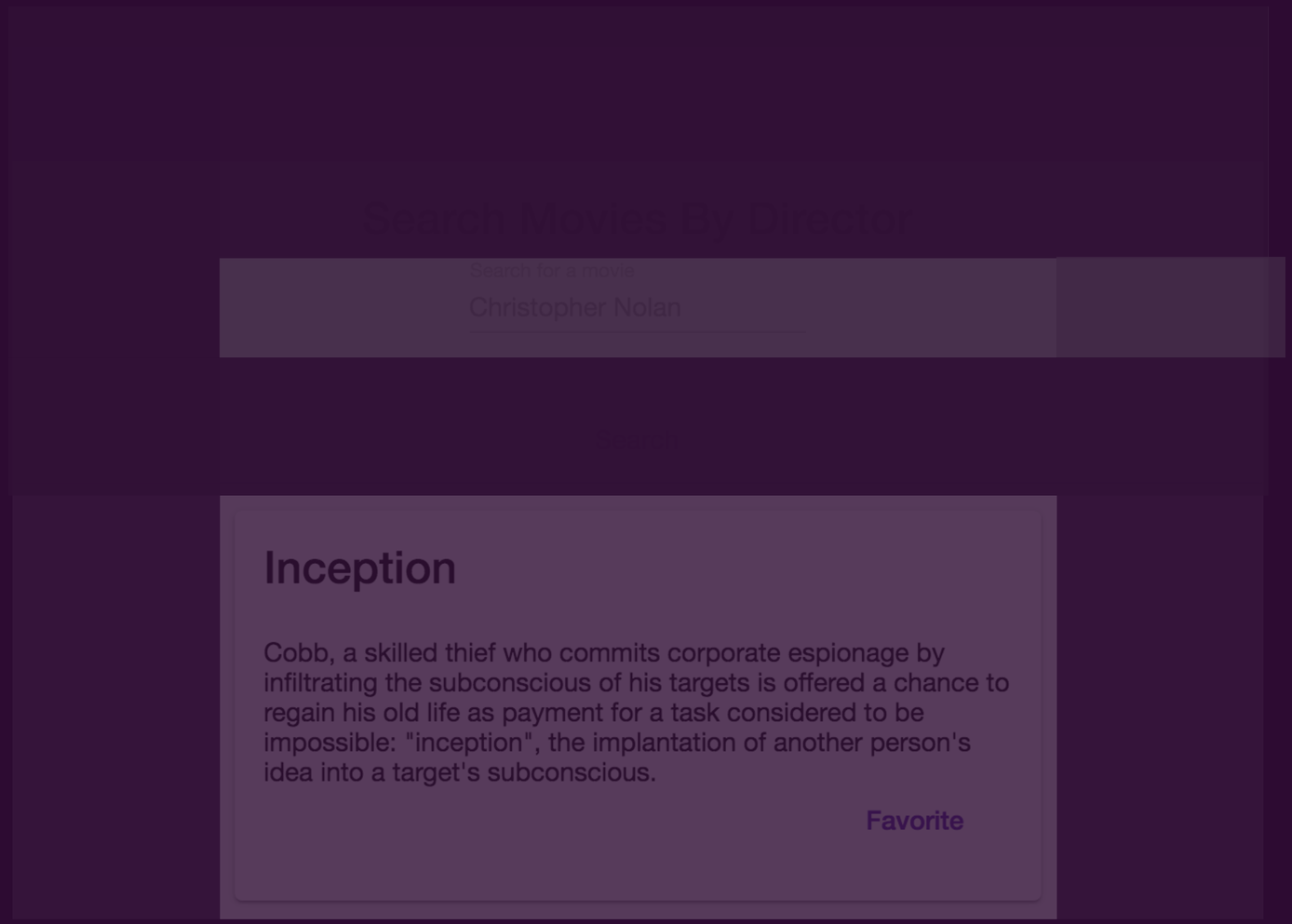

localhost:4200
Search Movies By Director
Search for a movie
Search

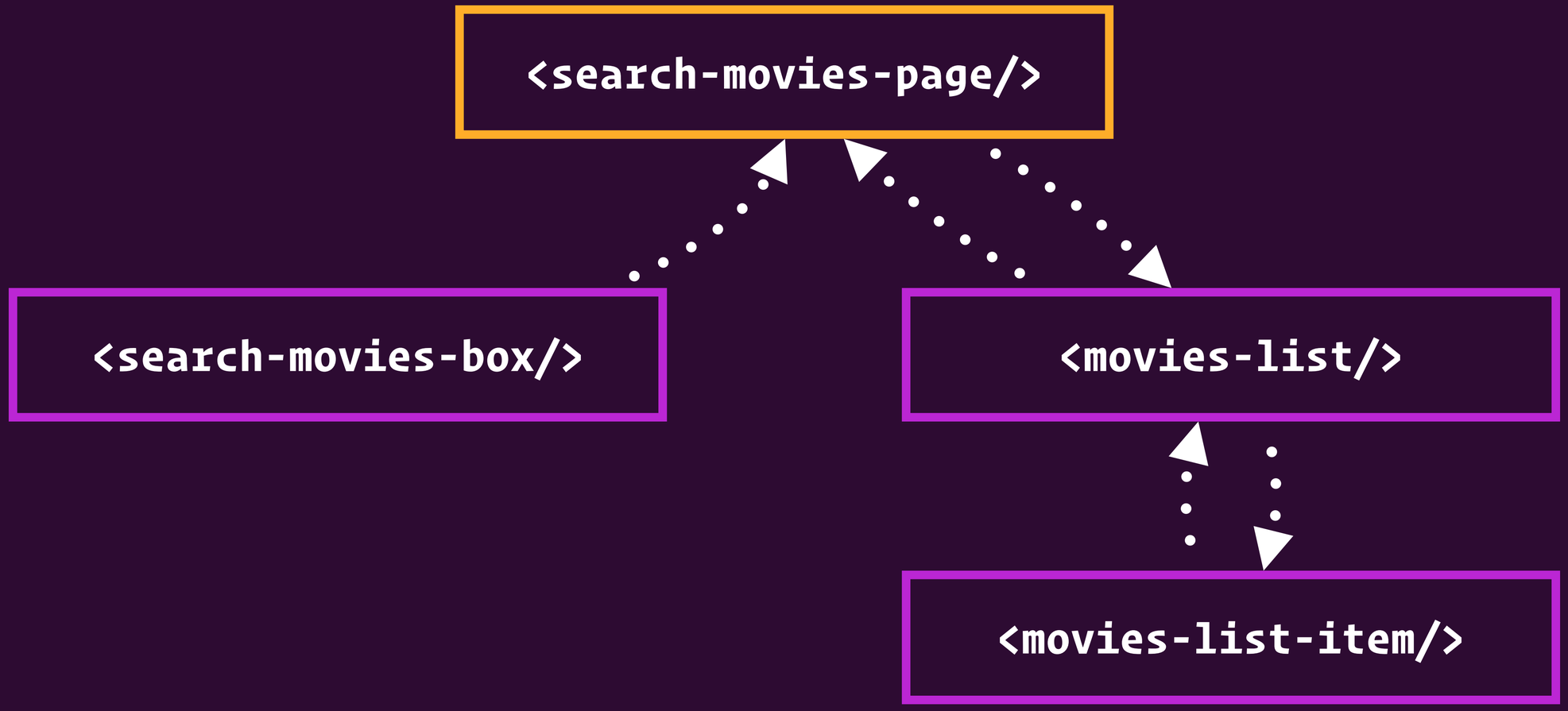
<movies-list-item/>

<movies-list/>

<search-movies-box/>

<search-movies-page/>




<search-movies-page/>
<search-movies-box/>
<movies-list/>
<movies-list-item/>

```
@Component({
  template: `
    <search-movies-box (search)="onSearch($event)"></search-movies-box>
    <movies-list
      [movies]="movies"
      (favoriteMovie)="onFavoriteMovie($event)">
    </movies-list>
  `
})
class SearchMoviesPageComponent {
  movies: Movie[] = [];

  onSearch(searchTerm: string) {
    this.moviesService.findMovies(searchTerm)
      .subscribe(movies => {
        this.movies = movies;
      });
  }
}
```

STATE

```
@Component({
  template: `
    <search-movies-box (search)="onSearch($event)"></search-movies-box>
    <movies-list
      [movies]="movies"
      (favoriteMovie)="onFavoriteMovie($event)">
    </movies-list>
  `
})
class SearchMoviesPageComponent {
  movies: Movie[] = [];

  onSearch(searchTerm: string) {
    this.moviesService.findMovies(searchTerm)
      .subscribe(movies => {
        this.movies = movies;
      });
  }
}
```

SIDE EFFECT

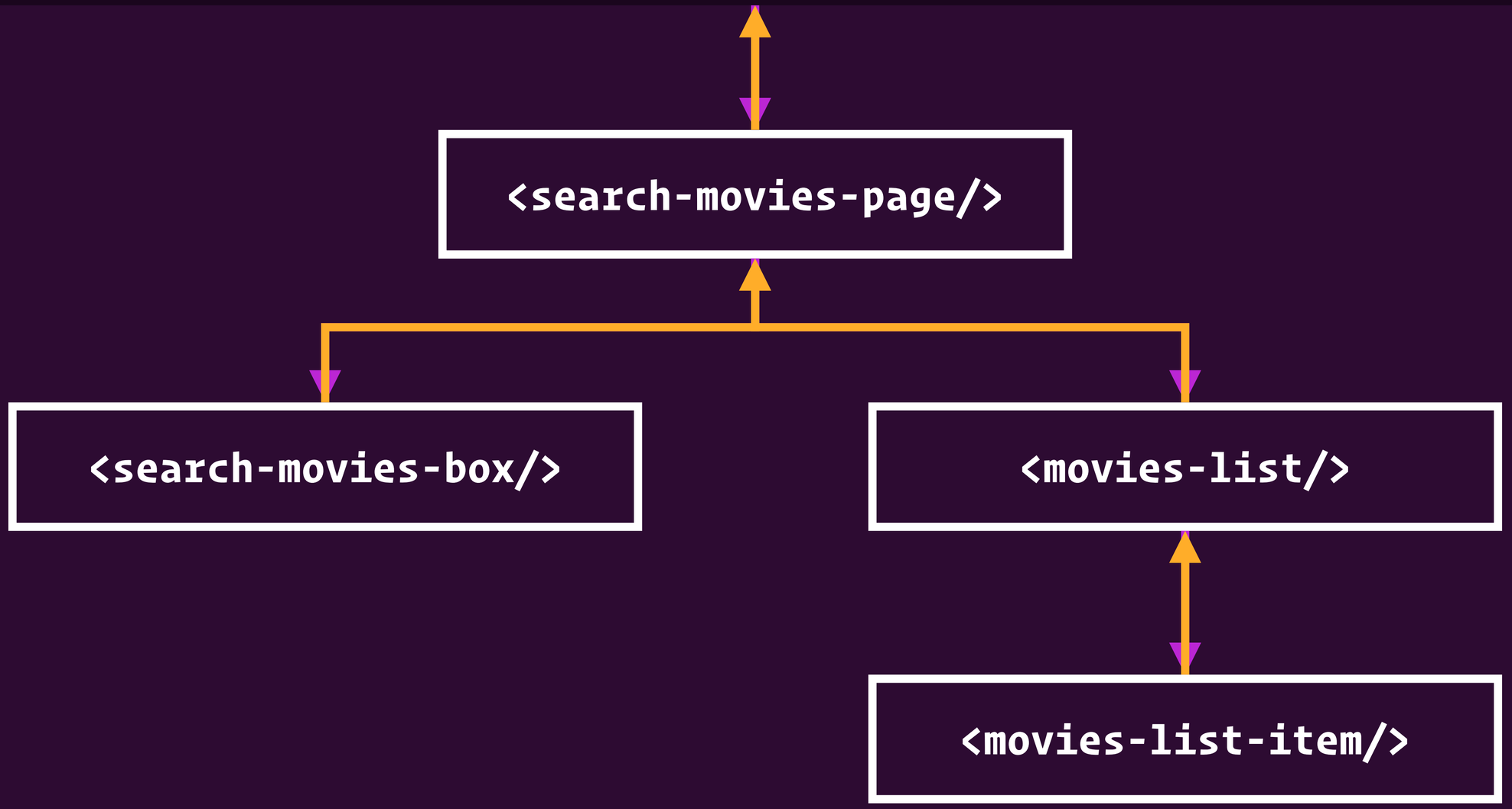
```
@Component({
  template: `
    <search-movies-box (search)="onSearch($event)"></search-movies-box>
    <movies-list
      [movies]="movies"
      (favoriteMovie)="onFavoriteMovie($event)">
    </movies-list>
  `
})
class SearchMoviesPageComponent {
  movies: Movie[] = [];

  onSearch(searchTerm: string) {
    this.moviesService.findMovies(searchTerm)
      .subscribe(movies => {
        this.movies = movies;
      });
  }
}
```

STATE CHANGE

`<search-movies-page/>`

- [x] Connects data to components
- [x] Triggers side effects
- [x] Handles state transitions

OUTSIDE WORLD



NGRX MENTAL MODEL

State flows down, changes flow up

`<search-movies-page/>`

- [x] Connects data to components
- [x] Triggers side effects
- [x] Handles state transitions

Single Responsibility Principle

`<search-movies-page/>`

- [x] Connects data to components

@Input() and @Output()

Does this component know who is binding to its input?

```
@Component({
  selector: 'movies-list-item',
})
export class MoviesListItemComponent {
  @Input() movie: Movie;
  @Output() favorite = new EventEmitter<Movie>();
}
```

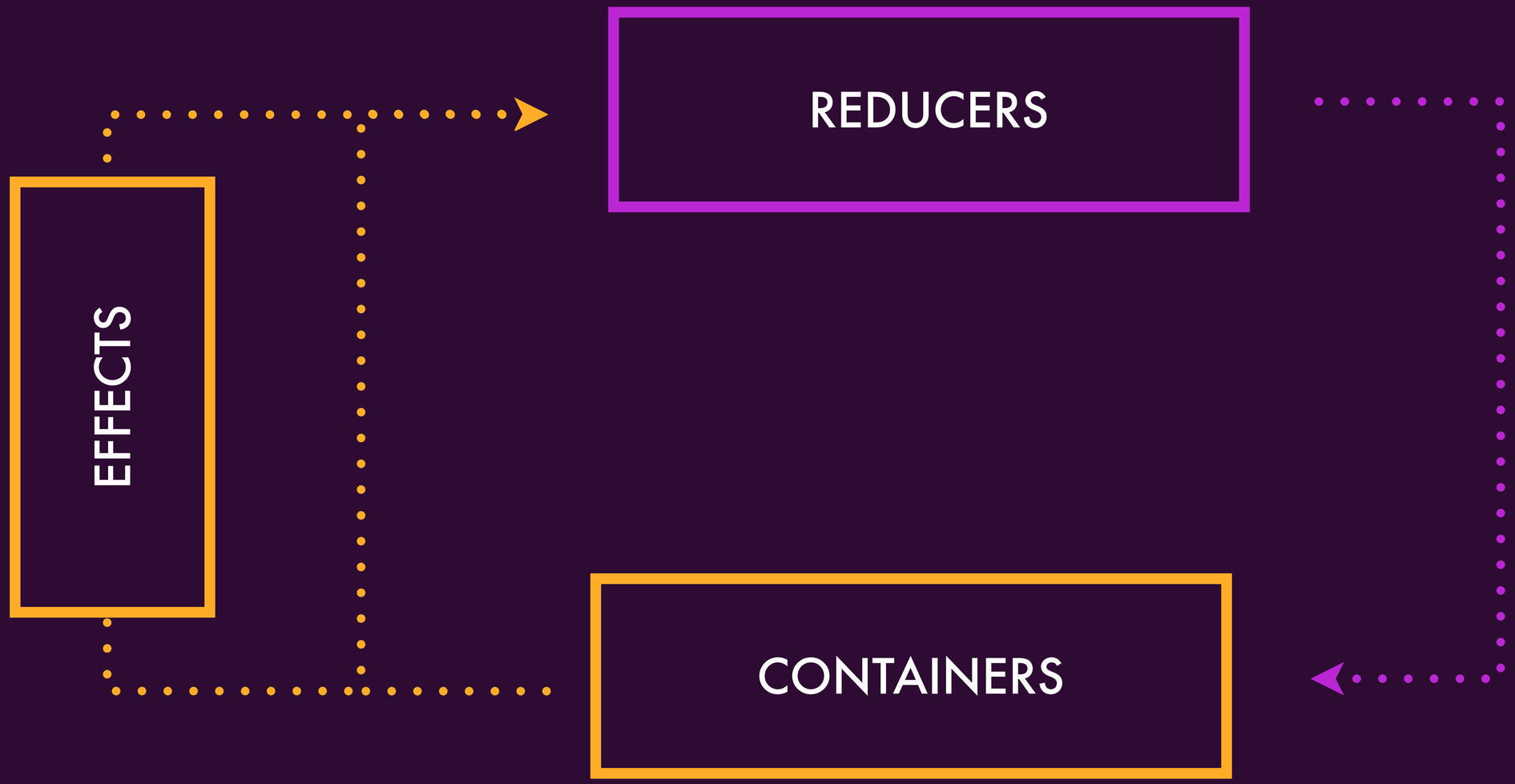
Does this component know who is listening to its output?

```
@Component({
  selector: 'movies-list-item',
})
export class MoviesListItemComponent {
  @Input() movie: Movie;
  @Output() favorite = new EventEmitter<Movie>();
}
```

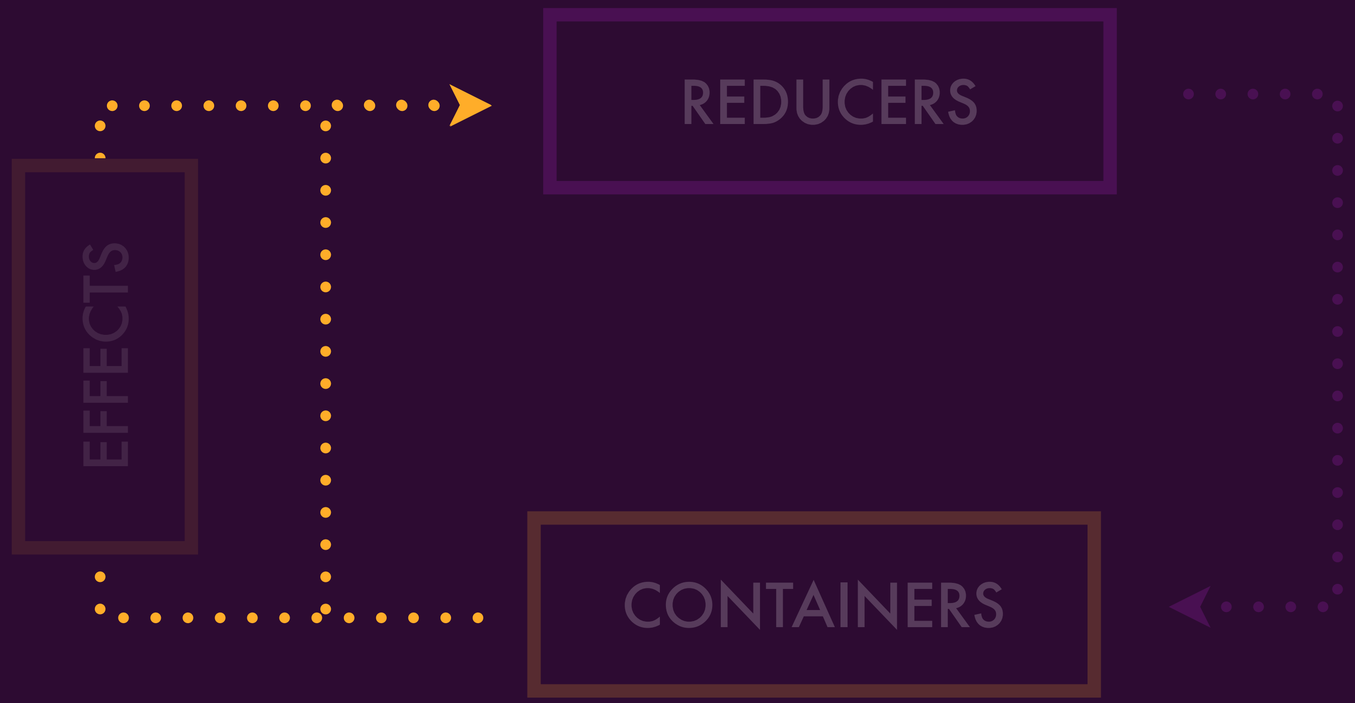
Inputs & Outputs offer Indirection

NGRX MENTAL MODEL

There is indirection between consumer of state, how state changes, and side effects


REDUCERS
EFFECTS
CONTAINERS

ACTIONS

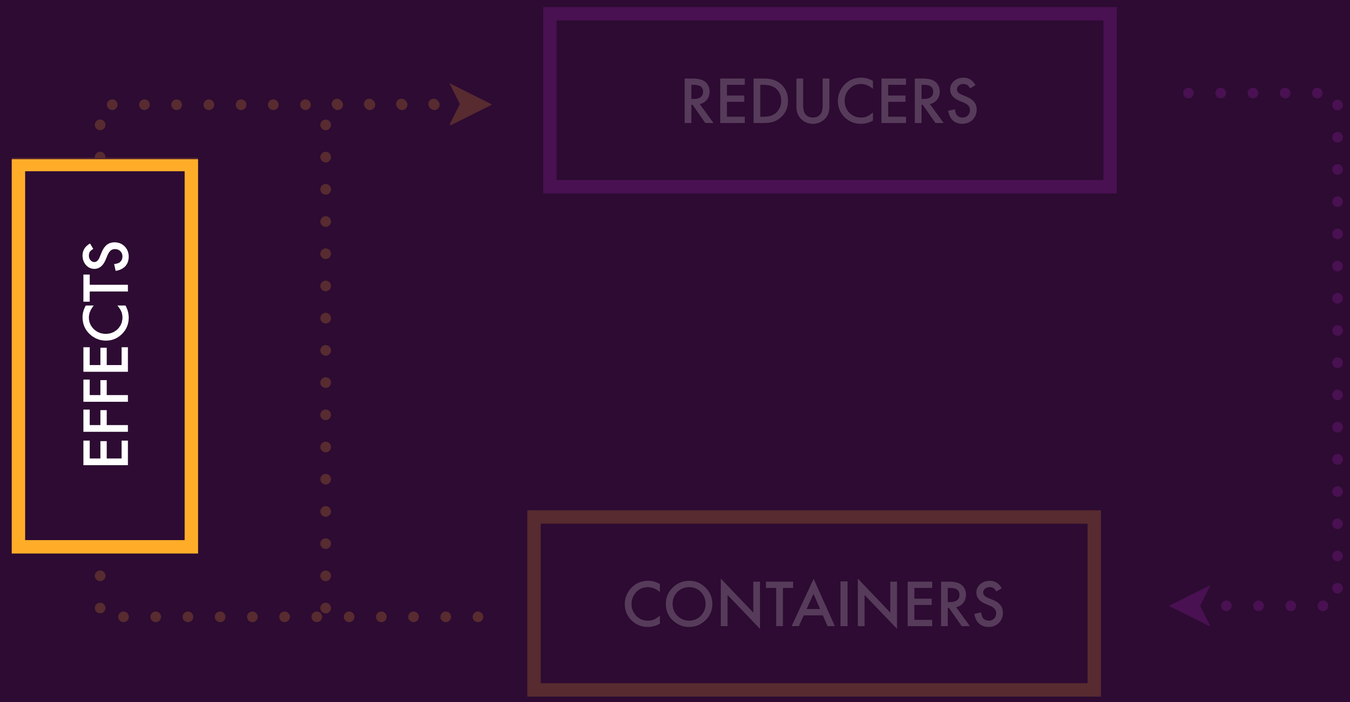


```
interface Action {
  type: string;
}
```

```
this.store.dispatch({
  type: 'MOVIES_LOADED_SUCCESS',
  movies: [{
    id: 1,
    title: 'Enemy',
    director: 'Denis Villeneuve',
  }],
});
```

Global `@Output()` for your whole app

EFFECTS

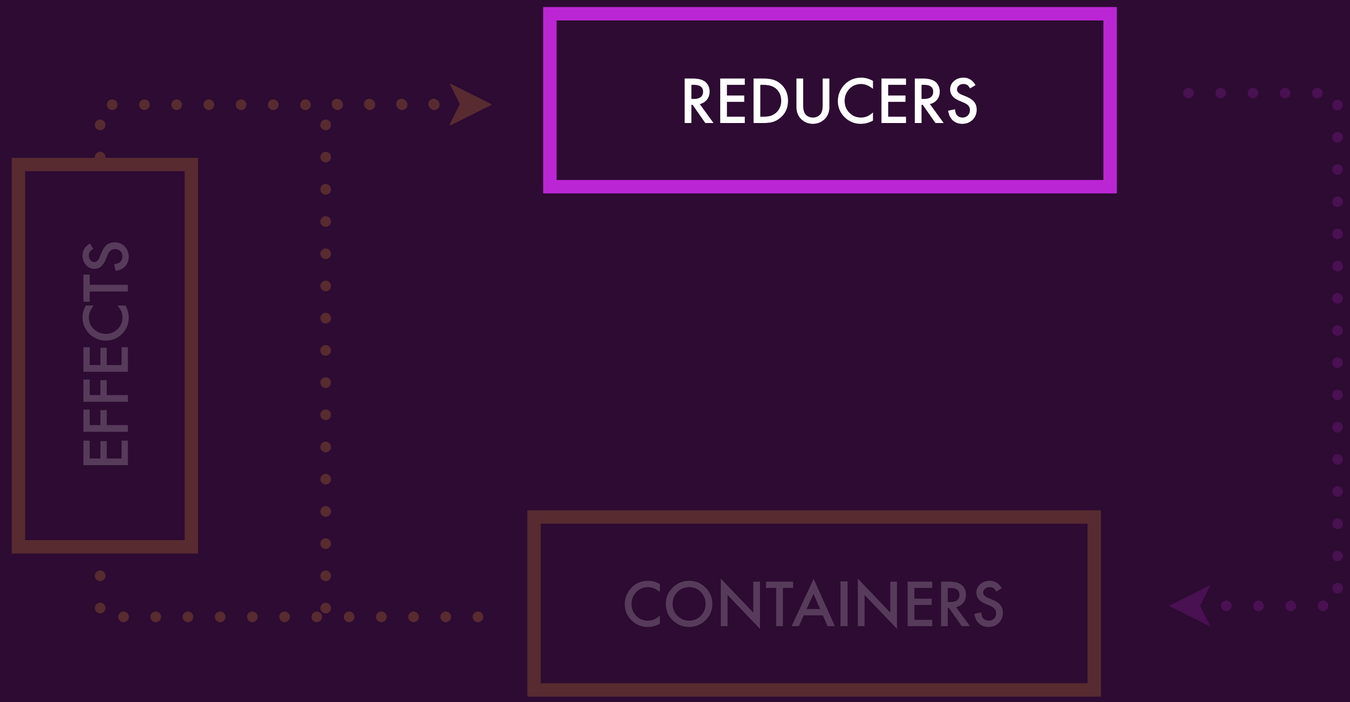


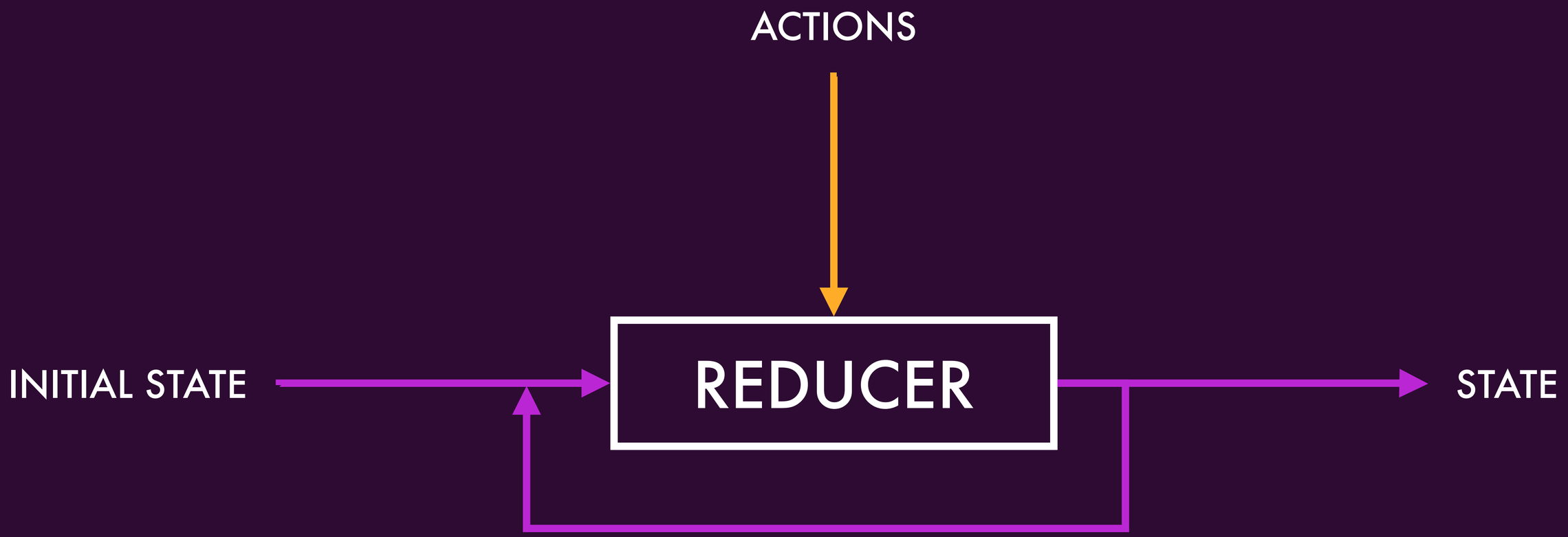
```
@Effect() findMovies$ = this.actions$
  .pipe(
    ofType('SEARCH_MOVIES'),
    switchMap(action => {
      return this.moviesService.findMovies(action.searchTerm)
        .pipe(
          map(movies => {
            return {
              type: 'MOVIES_LOADED_SUCCESS',
              movies,
            };
          })
        )
    }),
  );
```

```
@Effect() findMovies$ = this.actions$
  .pipe(
    ofType('SEARCH_MOVIES'),
    switchMap(action => {
      return this.moviesService.findMovies(action.searchTerm)
        .pipe(
          map(movies => {
            return {
              type: 'MOVIES_LOADED_SUCCESS',
              movies,
            };
          })
        )
    }),
  );
```

```
@Effect() findMovies$ = this.actions$
  .pipe(
    ofType('SEARCH_MOVIES'),
    switchMap(action => {
      return this.moviesService.findMovies(action.searchTerm)
        .pipe(
          map(movies => {
            return {
              type: 'MOVIES_LOADED_SUCCESS',
              movies,
            };
          })
        )
    }),
  );
```

REDUCERS




ACTIONS
INITIAL STATE
REDUCER
STATE

```
function moviesReducer(state = [], action) {
  switch (action.type) {
    case 'MOVIES_LOADED_SUCCESS': {
      return action.movies;
    }

    default: {
      return state;
    }
  }
}
```

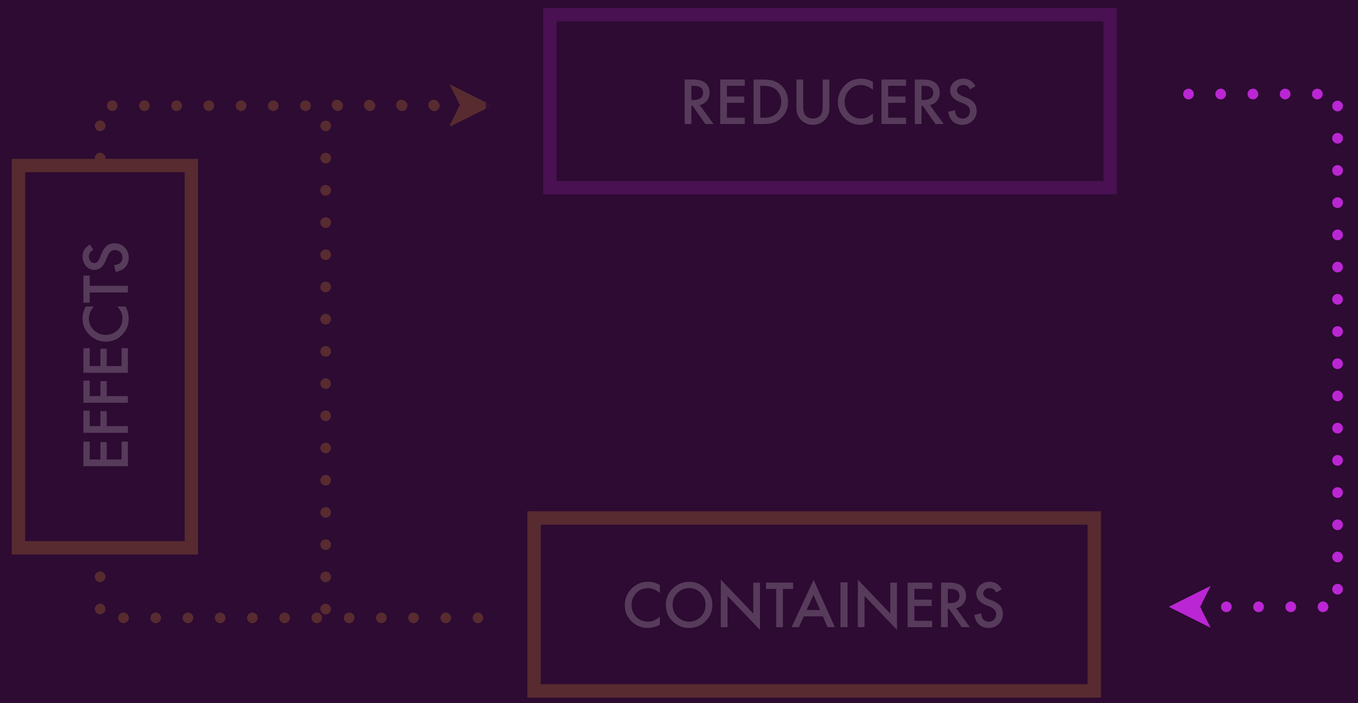
```
function moviesReducer(state = [], action) {
  switch (action.type) {
    case 'MOVIES_LOADED_SUCCESS': {
      return action.movies;
    }

    default: {
      return state;
    }
  }
}
```

```
function moviesReducer(state = [], action) {
  switch (action.type) {
    case 'MOVIES_LOADED_SUCCESS': {
      return action.movies;
    }

    default: {
      return state;
    }
  }
}
```

```
function moviesReducer(state = [], action) {
  switch (action.type) {
    case 'MOVIES_LOADED_SUCCESS': {
      return action.movies;
    }

    default: {
      return state;
    }
  }
}
```

```
function moviesReducer(state = [], action) {
  switch (action.type) {
    case 'MOVIES_LOADED_SUCCESS': {
      return action.movies;
    }

    default: {
      return state;
    }
  }
}
```

SELECTORS




STORE
???
COMPONENTS

```
function selectMovies(state) {
  return state.moviesState.movies;
}
```

Global `@Input()` for your whole app

CONTAINERS

```
@Component({
  template: `
    <search-movies-box (search)="onSearch($event)"></search-movies-box>
    <movies-list
      [movies]="movies$ | async"
      (favoriteMovie)="onFavoriteMovie($event)">
    </movies-list>
  `
})
export class SearchMoviesPageComponent {
  movies$: Observable<Movie[]>

  constructor(private store: Store<AppState>) {
    this.movies$ = store.select(selectMovies);
  }

  onSearch(searchTerm: string) {
    this.store.dispatch({ type: 'SEARCH_MOVIES', searchTerm });
  }
}
```

```
@Component({
  template: `
    <search-movies-box (search)="onSearch($event)"></search-movies-box>
    <movies-list
      [movies]="movies$ | async"
      (favoriteMovie)="onFavoriteMovie($event)">
    </movies-list>
  `
})
export class SearchMoviesPageComponent {
  movies$: Observable<Movie[]>

  constructor(private store: Store<AppState>) {
    this.movies$ = store.select(selectMovies);
  }

  onSearch(searchTerm: string) {
    this.store.dispatch({ type: 'SEARCH_MOVIES', searchTerm });
  }
}
```

```
@Component({
  template: `
    <search-movies-box (search)="onSearch($event)"></search-movies-box>
    <movies-list
      [movies]="movies$ | async"
      (favoriteMovie)="onFavoriteMovie($event)">
    </movies-list>
  `
})
export class SearchMoviesPageComponent {
  movies$: Observable<Movie[]>

  constructor(private store: Store<AppState>) {
    this.movies$ = store.select(selectMovies);
  }

  onSearch(searchTerm: string) {
    this.store.dispatch({ type: 'SEARCH_MOVIES', searchTerm });
  }
}
```

```
@Input() movies: Movie[]
```

```
store.select(selectMovies)
```

```
@Output() search: EventEmitter<string>()
```

```
this.store.dispatch({ type: 'SEARCH_MOVIES', searchTerm });
```

NGRX MENTAL MODEL

Select and Dispatch are special versions of Input and Output

RESPONSIBILITIES

- [x] Containers connect data to components
- [x] Effects triggers side effects
- [x] Reducers handle state transitions

NGRX MENTAL MODEL

Delegate responsibilities to individual modules of code

- [x] State flows down, changes flow up
- [x] Indirection between state & consumer
- [x] Select & Dispatch => Input & Output
- [x] Adhere to single responsibility principle

github.com/**CodeSequence/componentsconf2019-ngrx-workshop**

Demo

Challenge

1. Clone the repo at **github.com/CodeSequence/componentsconf2019-ngrx-workshop**
2. Checkout the **challenge** branch
3. Familiarize yourself with the **file structure**
4. What state does the **BooksPageComponent** have?
5. How do the methods on the **BooksService** get called?
6. How can the user interact with the **Books Page**?

ACTIONS

ACTIONS

- ☑ Unified interface to describe events
- ☑ Just data, no functionality
- ☑ Has at a minimum a type property
- ☑ Strongly typed using classes and enums

```
{
  type: "[Movies Page] Select Movie”;
  book: MovieModel;
}
```

```
class MoviesComponent {
  createMovie(movie) {
    this.store.dispatch({
      type: "[Movies Page] Create Movie",
      movie
    });
  }
}
```

GOOD ACTION HYGIENE

- [x] Unique events get unique actions
- [x] Actions are grouped by their source
- [x] Actions are never reused

"[Movies Page] Select Movie”

"[Movies Page] Add Movie"

"[Movies Page] Update Movie"

"[Movies Page] Delete Movie”

```
export const enter = createAction("[Movies Page] Enter");
```

```
store.dispatch(enter());
```

```
export const createMovie = createAction(
  '[Movies Page] Create Movie',
  props<{ movie: MovieRequiredProps }>()
);
```

```
store.dispatch(createMovie({
  movie: movieRequiredProps
}));
```

```
export const enter = createAction("[Movies Page] Enter");

export const createMovie = createAction(
  "[Movies Page] Create Movie",
  props<{ movie: MovieRequiredProps }>()
);

export const selectMovie = createAction(
  "[Movies Page] Select Movie",
  props<{ movie: MovieModel }>()
);
```

```
import { Store } from "@ngrx/store";
import { State } from "src/app/shared/state";
import { MoviesPageActions } from "../actions";

export class MoviesPageComponent implements OnInit {
  constructor(private store: Store<State>) {}

  ngOnInit() {
    this.store.dispatch(MoviesPageActions.enter());
  }
}
```

EVENT STORMING

EVENT STORMING

1. Using sticky notes, as a group identify all of the events in the system
2. Identify the commands that cause the event to arise
3. Identify the actor in the system that invokes the command
4. Identify the data models attached to each event

Demo

Challenge

1. Open **books-page.actions.ts**
2. Create **strongly typed actions** that adhere to **good action hygiene** for entering the books page, selecting a book, clearing the selection, creating a book, updating a book, and deleting a book.
3. Update **books-page.components.ts** to inject the Store and dispatch the actions


Everything is a stream

```
http.get("/api/v1/movies").subscribe();
```

"[Movies API] Get Movies Success"

"[Movies API] Get Movies Failure"

Demo

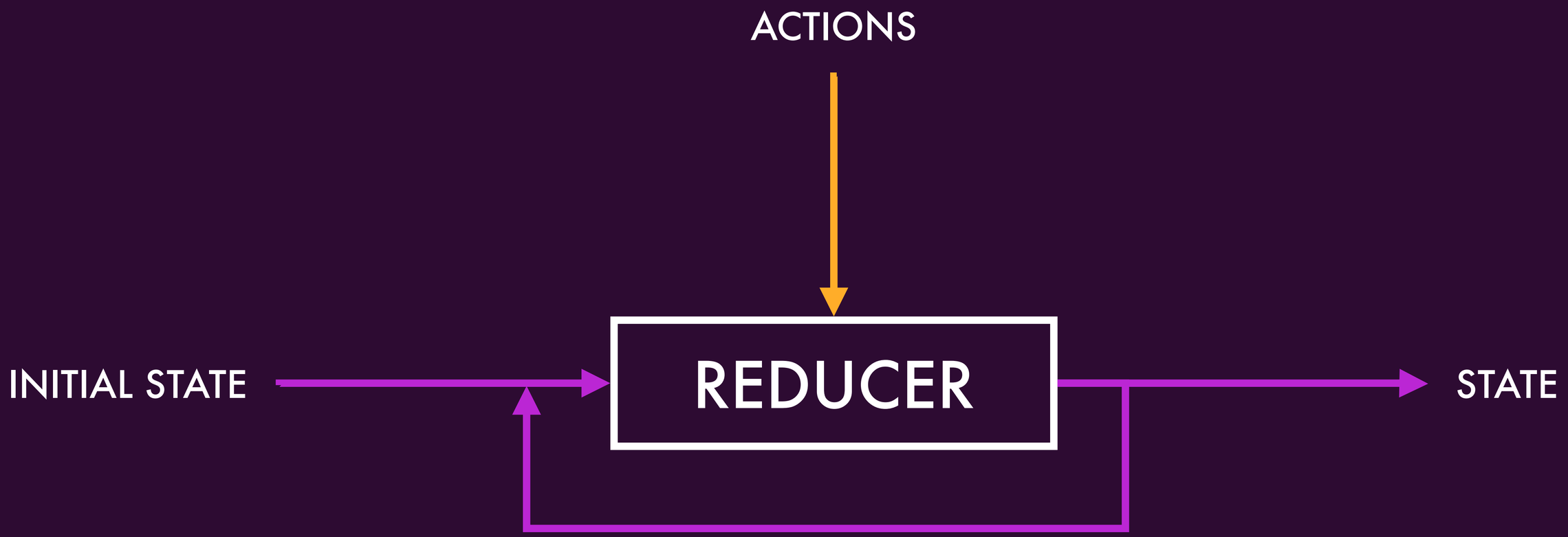
Challenge

1. Open **books-api.actions.ts**
2. Create **strongly typed actions** that adhere to **good action hygiene** for getting all of the books, updating a book, creating a book, and deleting a book. Don’t worry about failure actions for this exercise.
3. Update **books-page.components.ts** to dispatch the actions when the corresponding request completes

REDUCERS

REDUCERS

- [x] Produce new states
- [x] Receive the last state and next action
- [x] Switch on the action type
- [x] Use pure, immutable operations


ACTIONS
INITIAL STATE
REDUCER
STATE

```ts
export interface State {
  collection: MovieModel[];
  activeMovieId: string | null;
}
```

```
export const initialState: State = {
  collection: [],
  activeMovieId: null
};
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeMovieId: null
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeMovieId: null
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeMovieId: null
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeMovieId: null
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeMovieId: null
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeMovieId: null
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.enter, (state, action) => {
    return {
      ...state,
      activeMovieId: null
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.enter, (state, action) => {
    return {
      ...state,
      activeMovieId: null
    };
  }),
  on(
    MoviesPageActions.clearSelectedMovie,
    (state, action) => {
      return {
        ...state,
        activeMovieId: null
      };
    }
  )
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(
    MoviesPageActions.enter,
    MoviesPageActions.clearSelectedMovie,
    (state, action) => {
      return {
        ...state,
        activeMovieId: null
      };
    }
  )
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.selectMovie, (state, action) => {
    return {
      ...state,
      activeMovieId: action.movieId
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.selectMovie, (state, action) => {
    return {
      ...state,
      activeMovieId: action.movieId
    };
  })
);
```

```
export const moviesReducer = createReducer(
  initialState,
  on(MoviesPageActions.selectMovie, (state, action) => {
    return {
      ...state,
      activeMovieId: action.movieId
    };
  })
);
```

Demo

Challenge

1. Open **books.reducer.ts** and define a **State** interface with properties for the **collection** and the **activeBookId**
2. Define an **initialState** object that implements the **State** interface
3. Create a **booksReducer** using **createReducer**
4. Use the **on** function in the reducer to handle the **enter**, **clearSelectedBook**, and **selectBook** actions from **BooksPageActions**

```
const createMovie = (
  movies: MovieModel[],
  movie: MovieModel
) => [...movies, movie];
const updateMovie = (
  movies: MovieModel[],
  changes: MovieModel
) =>
  movies.map(movie => {
    return movie.id === changes.id
      ? Object.assign({}, movie, changes)
      : movie;
  });
const deleteMovie = (
  movies: MovieModel[],
  movieId: string
) => movies.filter(movie => movieId !== movie.id);
```

```
const createMovie = (
  movies: MovieModel[],
  movie: MovieModel
) => [...movies, movie];
```

```
const updateMovie = (
  movies: MovieModel[],
  changes: MovieModel
) =>
  movies.map(movie => {
    return movie.id === changes.id
      ? Object.assign({}, movie, changes)
      : movie;
  });
```

```
const deleteMovie = (
  movies: MovieModel[],
  movieId: string
) => movies.filter(movie => movieId !== movie.id);
```

Demo

Challenge

1. Create a **reducer handler** for the **booksLoaded** action
2. Use the **createBook** helper to create a **reducer handler** for the **bookCreated** action
3. Use the **updateBook** helper to create a **reducer handler** for the **bookUpdated** action
4. Use the **deleteBook** helper to create a **reducer handler** for the **bookDeleted** action

SELECTORS

SELECTORS

- [x] Allow us to query our store for data
- [x] Recompute when their inputs change
- [x] Fully leverage memoization for performance
- [x] Selectors are fully composable

```
export const selectMovies = (state: State) => {
  return state.collection;
};
```

```
export const selectActiveMovieId = state => {
  return state.activeMovieId;
};
```

```
export const selectActiveMovie = (state: State) => {
  const movies = selectMovies(state);
  const activeMovieId = selectActiveMovieId(state);

  return (
    movies.find(movie => movie.id === activeMovieId) || null
  );
};
```

```
export const selectActiveMovie = createSelector(
  selectMovies,
  selectActiveMovieId,
  (movies, activeMovieId) => {
    return (
      movies.find(movie => movie.id === activeMovieId) ||
      null
    );
  }
);
```

```
export const selectEarningsTotals = createSelector(
  selectAll,
  movies => {
    return calculateGrossMoviesEarnings(movies);
  }
);
```

```
export const selectEarningsTotals = createSelector(
  selectAll,
  calculateGrossMoviesEarnings
);
```

Demo

Challenge

1. Open **books.reducer.ts** and define two getter selectors: **selectAll** to select the **collection** and **selectActiveBookId** to select the **activeBookId**
2. Use **createSelector** to define a complex selector called **selectActiveBook** that uses **selectAll** and **selectActiveBookId**
3. Use createSelector to define another complex selector called **selectEarningsTotals** that uses **calculateBooksGrossEarnings** as the **projector function**

SETTING UP THE STORE

STORE

- [x] State contained in a single state tree
- [x] State in the store is immutable
- [x] Slices of state are updated with reducers

```ts
export interface State {
  collection: MovieModel[];
  activeMovieId: string | null;
}
```

```
export const moviesReducer = createReducer(...);
```

```
export const moviesReducer = createReducer(...);
export function reducer(state: State | undefined, action: Action) {
  return moviesReducer(state, action);
}
```

```
import * as fromMovies from "./movies/movies.reducer";

export interface State {
  movies: fromMovies.State;
}

export const reducers: ActionReducerMap<State> = {
  movies: fromMovies.reducer
};
```

```
@NgModule({
  imports: [
    // imports …
    StoreModule.forRoot(reducers),
    StoreDevtoolsModule.instrument({ maxAge: 5 }),
  ],
})
export class AppModule {}
```

Demo

Challenge

1. Open **books.reducer.ts** and create an AOT-compatible wrapper function for the **booksReducer** called **reducer**
2. Import everything from **books.reducer.ts** in **state/index.ts**
3. Add the books **State** interface to the global **State** interface
4. Add the books **reducer** to the application's **reducers map**

USING SELECTORS

```
export const selectActiveMovie = createSelector(
  selectMovies,
  selectActiveMovieId,
  (movies, activeMovieId) => {
    return (
      movies.find(movie => movie.id === activeMovieId) ||
      null
    );
  }
);
```

```
export const selectMoviesState = (state: State) => {
  return state.movies;
};

export const selectActiveMovie = (state: State) => {
  const moviesState = selectMoviesState(state);

  return fromMovies.selectActiveMovie(moviesState);
};
```

```
export const selectMoviesState = (state: State) => {
  return state.movies;
};

export const selectActiveMovie = createSelector(
  selectMoviesState,
  fromMovies.selectActiveMovie
);
```

Demo

Challenge

1. Open **state/index.ts** and add a getter selector at the bottom called **selectBooksState** that selects the **books state**
2. Use **createSelector** and **selectBooksState** to export global selectors for **selectAllBooks**, **selectActiveBook**, and **selectBooksEarningsTotals**

```
class MoviesPageComponent {
  movies: MovieModel[] = [];
}
```

```
class MoviesPageComponent {
  movies$: Observable<MovieModel[]>;

  constructor(store: Store<State>) {
    this.movies$ = store.select(selectAllMovies);
  }
}
```

```
<app-books-list
  [books]="books"
  (select)="onSelect($event)"
  (delete)="onDelete($event)"
></app-books-list>
```

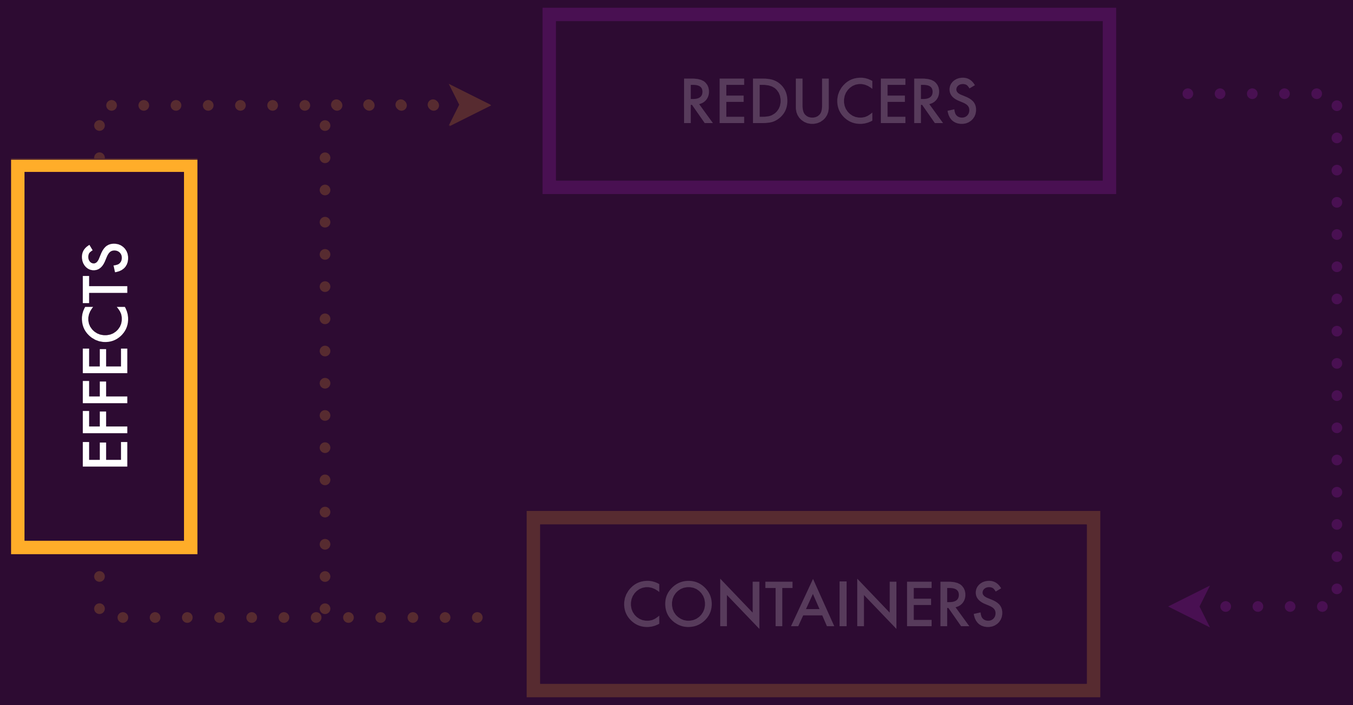
```
<app-books-list
  [books]=“books$ | async”
  (select)="onSelect($event)"
  (delete)="onDelete($event)"
></app-books-list>
```

Demo

Challenge

1. Open **books-page.component.ts** and replace the local state properties - **books**, **activeBook**, and **total** - with observable properties
2. Initialize the observable state properties in the **constructor** using the **Store service** and the **global selectors** exported in **state/index.ts**
3. Update **books-page.component.html** to use the **async pipe** for value bindings
4. Cleanup any remaining references to the local state in **books-page.component.ts**

EFFECTS


REDUCERS
EFFECTS
CONTAINERS

EFFECTS

- [x] Processes that run in the background
- [x] Connect your app to the outside world
- [x] Often used to talk to services
- [x] Written entirely using RxJS streams

```
const BASE_URL = "http://localhost:3000/movies";

@Injectable({ providedIn: "root" })
export class MoviesService {
  constructor(private http: HttpClient) {}

  load(id: string) {
    return this.http.get(`${BASE_URL}/${id}`);
  }
}
```

```
@Injectable()
export class MoviesEffects {
  constructor(
    private actions$: Actions,
    private moviesService: MoviesService
  ) {}
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
EffectsModule.forFeature([MoviesEffects]);
```

Demo

Challenge

1. Create a file at **app/books/books-api.effects.ts** and add an **effect class** to it
2. Define an **effect** called **loadBooks$** that calls **BooksService.all()** and maps the result into a **BooksLoaded** action
3. Register the **effect** using **EffectsModule.forFeature([])** in **books.module.ts**
4. Remove the **getAll() method** in **movies-page.component.ts** that gets all of the books

ADVANCED EFFECTS

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    mergeMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

WHAT MAP OPERATOR SHOULD I USE?

`mergeMap`	Subscribe immediately, never cancel or discard
`concatMap`	Subscribe after the last one finishes
`exhaustMap`	Discard until the last one finishes
`switchMap`	Cancel the last one if it has not completed

RACE CONDITIONS!

`mergeMap`	Subscribe immediately, never cancel or discard
`exhaustMap`	Discard until the last one finishes
`switchMap`	Cancel the last one if it has not completed

CONCATMAP IS THE SAFEST OPERATOR

…but there is a risk of back pressure

BACKPRESSURE DEMO

https://stackblitz.com/edit/angular-kbvxzz

`mergeMap`	Deleting items
`concatMap`	Updating or creating items
`exhaustMap`	Non-parameterized queries
`switchMap`	Parameterized queries

```
class MoviesEffects {
  @Effect() loadMovies$ = this.actions$.pipe(
    ofType(MovesPageActions.enter),
    exhaustMap(() =>
      this.moviesService.all().pipe(
        map(movies =>
          MoviesApiActions.moviesLoaded({ movies })
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

Demo

Challenge

1. Open **books-api.effects.ts** and update the **loadBooks$** effect to use the **exhaustMap** operator
2. Add an effect for **creating** a book using the **BooksService.create()** method and the **concatMap** operator
3. Add an effect for **updating** a book using the **BooksService.update()** method and the **concatMap** operator
4. Add an effect for **deleting** a book using the **BooksService.delete()** method and the **mergeMap** operator
5. Remove the BooksService from BooksPageComponent

EFFECTS EXAMPLES

```
@Effect() tick$ = interval(/* Every minute */ 60 * 1000).pipe(
  map(() => Clock.tickAction(new Date()))
);
```

```
@Effect() = fromWebSocket(“/ws").pipe(map(message => {
  switch (message.kind) {
    case “book_created”: {
      return WebSocketActions.bookCreated(message.book);
    }

    case “book_updated”: {
      return WebSocketActions.bookUpdated(message.book);
    }

    case “book_deleted”: {
      return WebSocketActions.bookDeleted(message.book);
    }
  }
}))
```

```
@Effect()
createBook$ = this.actions$.pipe(
  ofType(BooksPageActions.createBook.type),
  mergeMap(action =>
    this.booksService.create(action.book).pipe(
      map(book => BooksApiActions.bookCreated({ book })),
      catchError(error => of(BooksApiActions.createFailure({
        error,
        book: action.book,
      })))
    )
  )
);
```

```
@Effect() promptToRetry$ = this.actions$.pipe(
  ofType(BooksApiActions.createFailure),
  mergeMap(action =>
    this.snackBar
      .open("Failed to save book.", "Try Again", {
        duration: /* 12 seconds */ 12 * 1000
      })
      .onAction()
      .pipe(
        map(() => BooksApiActions.retryCreate(action.book))
      )
  )
);
```

```
@Effect()
createBook$ = this.actions$.pipe(
  ofType(
    BooksPageActions.createBook,
    BooksApiActions.retryCreate,
  ),
  mergeMap(action =>
    this.booksService.create(action.book).pipe(
      map(book => BooksApiActions.bookCreated({ book })),
      catchError(error => of(BooksApiActions.createFailure({
        error,
        book: action.book,
      })))
    )
  )
);
```

```
@Effect({ dispatch: false })
openUploadModal$ = this.actions$.pipe(
  ofType(BooksPageActions.openUploadModal),
  tap(() => {
    this.dialog.open(BooksCoverUploadModalComponent);
  })
);
```

```
@Effect() uploadCover$ = this.actions$.pipe(
  ofType(BooksPageActions.uploadCover),
  concatMap(action =>
    this.booksService.uploadCover(action.cover).pipe(
      map(result => BooksApiActions.uploadComplete(result)),
      takeUntil(
        this.actions$.pipe(
          ofType(BooksPageActions.cancelUpload)
        )
      )
    )
  )
);
```

ENTITIES

ENTITY

- [x] Working with collections should be fast
- [x] Collections are very common
- [x] Common set of basic state operations
- [x] Common set of basic state derivations

```ts
interface EntityState<Model> {
  ids: string[] | number[];
  entities: { [id: string | number]: Model };
}
```

```ts
export interface MoviesState extends EntityState<Movie> {
  activeMovieId: string | null;
}
export const adapter = createEntityAdapter<Movie>();
export const initialState: Movie = adapter.getInitialState(
  {
    activeMovieId: null
  }
);
```

```ts
export interface MoviesState extends EntityState<Movie> {
  activeMovieId: string | null;
}
export const adapter = createEntityAdapter<Movie>();
export const initialState: Movie = adapter.getInitialState(
  {
    activeMovieId: null
  }
);
```

```ts
export interface MoviesState extends EntityState<Movie> {
  activeMovieId: string | null;
}
export const adapter = createEntityAdapter<Movie>();
export const initialState: Movie = adapter.getInitialState(
  {
    activeMovieId: null
  }
);
```

```ts
export interface MoviesState extends EntityState<Movie> {
  activeMovieId: string | null;
}
export const adapter = createEntityAdapter<Movie>();
export const initialState: Movie = adapter.getInitialState(
  {
    activeMovieId: null
  }
);
```

```
export const {
  selectIds,
  selectEntities,
  selectAll,
  selectTotal
} = adapter.getSelectors();
```

```
export const booksReducer = createReducer(
  initialState,
  on(BooksApiActions.booksLoaded, (state, action) => {
    return {
      ...state,
      collection: action.books
    };
  })
);
```

```
export const booksReducer = createReducer(
  initialState,
  on(BooksApiActions.booksLoaded, (state, action) => {
    return adapter.addAll(action.books, state);
  })
);
```

```
export const selectActiveBook = createSelector(
  selectAll,
  selectActiveBookId,
  (books, activeBookId) => {
    return (
      books.find(book => book.id === activeBookId) || null
    );
  }
);
```

```
export const selectActiveBook = createSelector(
  selectEntities,
  selectActiveBookId,
  (entities, activeBookId) => {
    return activeBookId ? entities[activeBookId] : null;
  }
);
```

Demo

Challenge

1. Update **books.reducer.ts** to use **EntityState** to define **State**
2. Create an **unsorted entity adapter** for **State** and use it to initialize **initialState**
3. Update the **reducer** to use the **adapter** methods
4. Use the adapter to replace the **selectAll** selector and to create a **selectEntities** selector
5. Update the **selectActiveBook** selector to use the **selectEntities** selector instead of the **selectAll** selector

TESTING REDUCERS

```
it("should return the initial state when initialized", () => {
  const state = reducer(undefined, {
    type: "@@init"
  } as any);

  expect(state).toBe(initialState);
});
```

```
const movies: Movie[] = [
  { id: "1", name: "Green Lantern", earnings: 0 }
];
const action = MovieApiActions.loadMoviesSuccess({
  movies
});

const state = reducer(initialState, action);
```

```
const movie: Movie = {
  id: "1",
  name: "mother!",
  earnings: 1000
};
const firstAction = MovieApiActions.createMovieSuccess({ movie });
const secondAction = MoviesPageActions.deleteMovie({ movie });

const state = [firstAction, secondAction].reduce(
  reducer,
  initialState
);
```

```
const movies: Movie[] = [
  { id: "1", name: "Green Lantern", earnings: 0 }
];
const action = MovieApiActions.loadMoviesSuccess({
  movies
});

const state = reducer(initialState, action);

expect(selectAllMovies(state)).toEqual(movies);
```

```
expect(state).toEqual({
  ids: ["1"],
  entities: {
    "1": { id: "1", name: "Green Lantern", earnings: 0 }
  }
});
```

SNAPSHOT TESTING

```
expect(state).toMatchSnapshot();
```

in: **shared/state/__snapshots__/movie.reducer.spec.ts.snap**

```
exports[
  `Movie Reducer should load all movies when the API loads them all successfully 1`
] = `
Object {
  "activeMovieId": null,
  "entities": Object {
    "1": Object {
      "earnings": 0,
      "id": "1",
      "name": "Green Lantern",
    },
  },
  "ids": Array [
    "1",
  ],
}
`;
```

SNAPSHOT TESTING

- [x] Avoid writing out manual assertions
- [x] Verify how state transitions impact state
- [x] Can be used with components
- [x] Creates snap files that get checked in

Demo

Challenge

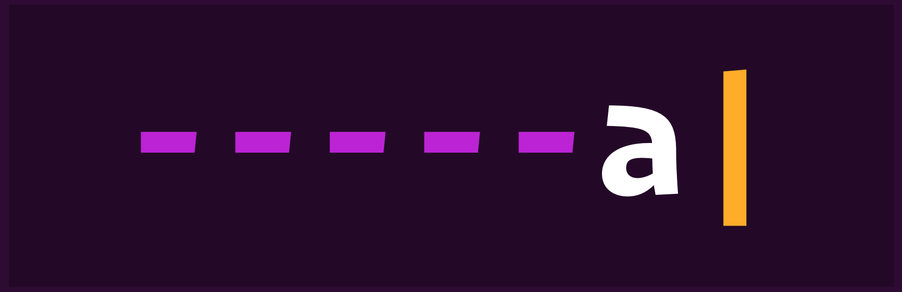
1. Write a test that verifies the **books reducer** returns the initial state when no state is provided using the **toBe** matcher
2. Write tests that verifies the **books reducer** correctly transitions state for **loading all books**, **creating a book**, and **deleting a book** using the **toMatchSnapshot** matcher
3. Write tests that verifies the behavior of the **selectActiveBook** and **selectAll** selectors

TESTING EFFECTS

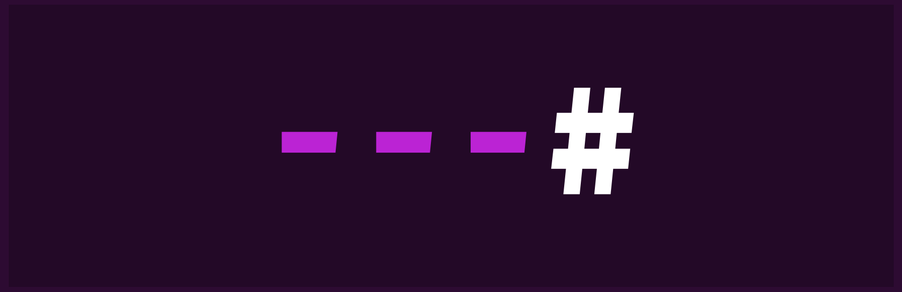
OBSERVABLE TIMELINES

```
import { timer } from "rxjs";
import { mapTo } from "rxjs/operators";

timer(50).pipe(mapTo("a"));
```

```
timer(50).pipe(mapTo("a"));
```



```
timer(30).pipe(mergeMap(() => throwError('Error!')))
```



```
const source$ = timer(50).pipe(mapTo("a"));
const expected$ = cold("-----a|");

expect(source$).toBeObservable(expected$);
```

```
const source$ = timer(30).pipe(
  mergeMap(() => throwError("Error!"))
);
const expected$ = cold("---#", {}, "Error!");

expect(source$).toBeObservable(expected$);
```

-	10ms of time
a b c ...	Emission of any value
#	Error
|	Completion

COLD AND HOT OBSERVABLES

COLD

NETFLIX

HOT

Cable TV

`Actions`	Hot Observable
`HttpClient`	Cold Observables
`Store`	Hot Observable
`fromWebSocket`	Cold Observable

```
let actions$: Observable<any>;

beforeEach(() => {
  TestBed.configureTestingModule({
    providers: [provideMockActions(() => actions$)]
  });
});

actions$ = hot("---a---", {
  a: BooksPageActions.enter
});
```

```
const inputAction = MoviesPageActions.createMovie({
  movie: {
    name: mockMovie.name,
    earnings: 25
  }
});
const outputAction = MovieApiActions.createMovieSuccess({
  movie: mockMovie
});

actions$ = hot("--a---", { a: inputAction });
const response$ = cold("--b|", { b: mockMovie });
const expected$ = cold("----c--", { c: outputAction });
mockMovieService.create.mockReturnValue(response$);

expect(effects.createMovie$).toBeObservable(expected$);
```

JASMINE MARBLES

- [x] Make assertions about time
- [x] Describe Rx behavior with diagrams
- [x] Verify observables behave as described
- [x] Works with hot and cold observables

Demo

Challenge

1. Open **books-api.effects.spec.ts** and declare variables for the **actions$**, instance of the **effects**, and a mock **bookService**
2. Use the **TestBed** to setup **providers** for the effects, actions, and the book service
3. Verify the behavior of the **createBook$** effect using mock actions and test observables

FOLDER LAYOUT

LIFT

- [x] Locating our code is easy
- [x] Identify code at a glance
- [x] Flat file structure for as long as possible
- [x] Try to stay DRY - don’t repeat yourself

```
src/
  shared/
    // Shared code

  modules/
    ${feature}/
    // Feature code
```

```
modules/
  ${feature}/
    actions/
      ${action-category}.actions.ts
      index.ts

    components/
      ${component-name}/
        ${component-name}.component.ts
        ${component-name}.component.spec.ts

    services/
      ${service-name}.service.ts
      ${service-name}.service.spec.ts

    effects/
      ${effect-name}.effects.ts
      ${effect-name}.effects.spec.ts

    ${feature}.module.ts
```

```
modules/
 book-collection/
    actions/
      books-page.actions.ts
      index.ts

    components/
      books-page/
        books-page.component.ts
        books-page.component.spec.ts

    services/
      books.service.ts
      books.service.spec.ts

    effects/
      books.effects.ts
      books.effects.spec.ts

    book-collection.module.ts
```

ACTION BARRELS

```
import * as BooksPageActions from "./books-page.actions";
import * as BooksApiActions from "./books-api.actions";

export { BooksPageActions, BooksApiActions };
```

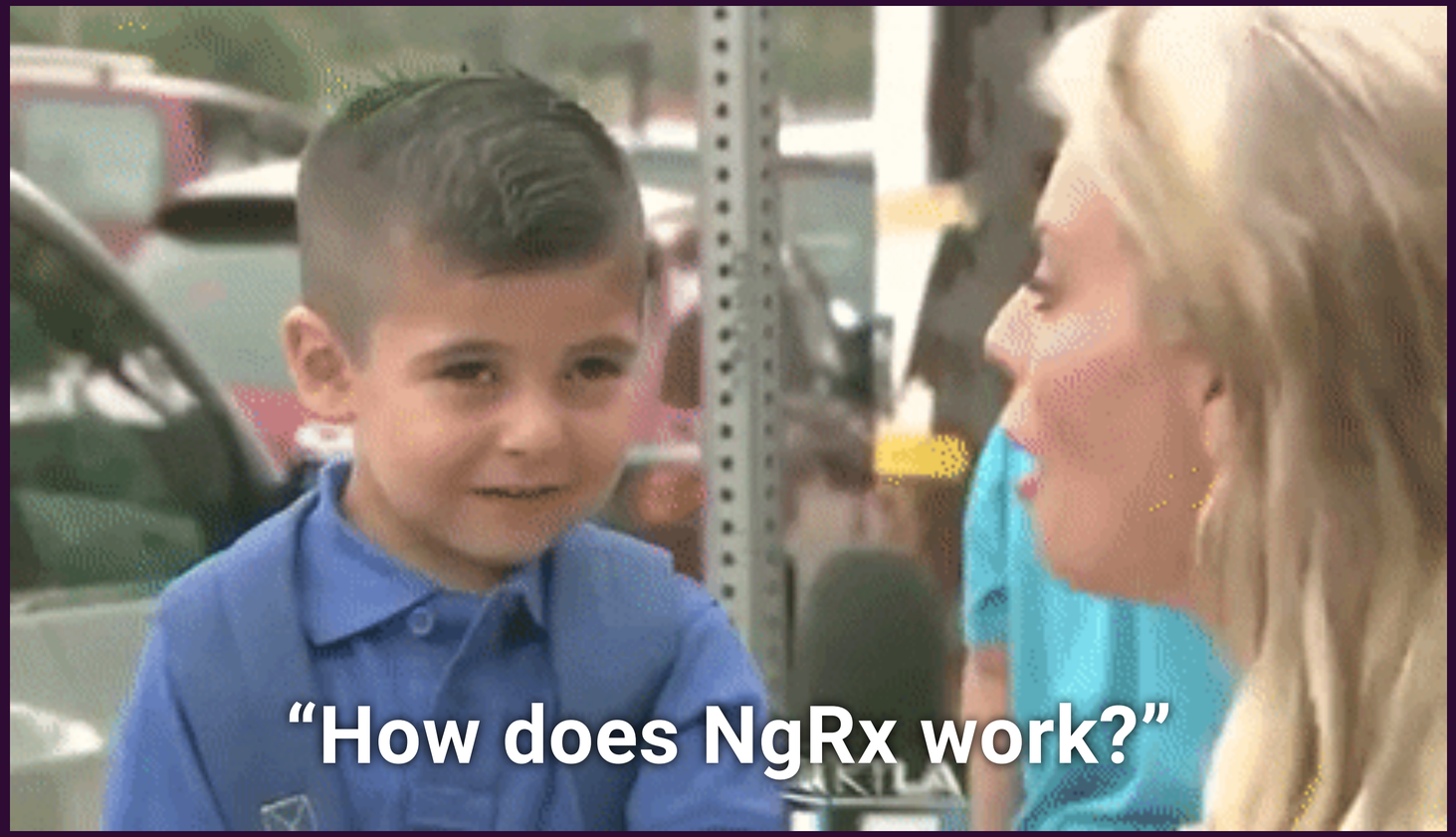
```
import { BooksPageActions } from "app/modules/book-collection/actions";
```

```
src/
  shared/
    state/
      ${state-name}/
        ${state-key}/
          ${state-key}.reducer.ts
          ${state-key}.spec.ts
          index.ts

          ${feature-name}.state.ts
          ${feature-name}.state.spec.ts
          ${feature-name}.state.module.ts
          index.ts

      effects/
        ${effect-name}/
          ${effect-name}.effects.ts
          ${effect-name}.effects.spec.ts
          ${effect-name}.actions.ts
          ${effect-name}.module.ts
          index.ts
```

```
src/
  shared/
    state/
      core/
        books/
          books.reducer.ts
          books.spec.ts

        core.state.ts
        core.state.spec.ts
        core.state.module.ts
        index.ts

      effects/
        clock/
          clock.effects.ts
          clock.effects.spec.ts
          clock.actions.ts
          clock.module.ts
```

FOLDER STRUCTURE

- [x] Put state in a shared place separate from features
- [x] Effects, components, and actions belong to features
- [x] Some effects can be shared
- [x] Reducers reach into modules’ action barrels


“How does NgRx work?”

- [x] State flows down, changes flow up
- [x] Indirection between state & consumer
- [x] Select & Dispatch => Input & Output
- [x] Adhere to single responsibility principle

ACTIONS

- ☑ Unified interface to describe events
- ☑ Just data, no functionality
- ☑ Has at a minimum a type property
- ☑ Strongly typed using classes and enums

REDUCERS

- [x] Produce new states
- [x] Receive the last state and next action
- [x] Switch on the action type
- [x] Use pure, immutable operations

STORE

- [x] State contained in a single state tree
- [x] State in the store is immutable
- [x] Slices of state are updated with reducers

SELECTORS

- [x] Allow us to query our store for data
- [x] Recompute when their inputs change
- [x] Fully leverage memoization for performance
- [x] Selectors are fully composable

EFFECTS

- [x] Processes that run in the background
- [x] Connect your app to the outside world
- [x] Often used to talk to services
- [x] Written entirely using RxJS streams

“How does NgRx work?”

HELP US IMPROVE

https://bit.ly/2ROXvn0

FOLLOW ON TALKS

“Good Action Hygiene” by Mike Ryan

https://youtu.be/JmnsEvoy-gY

“Reactive Testing Strategies with NgRx” by Brandon Roberts & Mike Ryan

https://youtu.be/MTZprd9tI6c

“Authentication with NgRx” by Brandon Roberts

https://youtu.be/46IRQgNtCGw

“You Might Not Need NgRx” by Mike Ryan

https://youtu.be/omnwu_etHTY

“Just Another Marble Monday” by Sam Brennan & Keith Stewart

https://youtu.be/dwDtMs4mN48

@ngrx/schematics

@ngrx/router-store

@ngrx/data

ngrx.io

QUESTIONS?

@MikeRyanDev

THANK YOU

Z WARE M1

AngularRox5!