A REACTIVE STATE OF MIND

WITH ANGULAR AND NGRX

Mike Ryan

@MikeRyanDev

Software Engineer at Synapse

Google Developer Expert

NgRx Core Team

Brandon Roberts

@brandontroberts

Developer/Technical Writer

Angular Team

NgRx Core Team

Alex Okrushko

@AlexOkrushko

Special Guest

Firebase Team / Googler

NgRx Core Team

Open source libraries for Angular

Built with reactivity in mind

State management and side effects

Community driven

SCHEDULE

- Demystifying NgRx
- Setting up the Store
- Reducers
- Actions
- Entities
- Effects

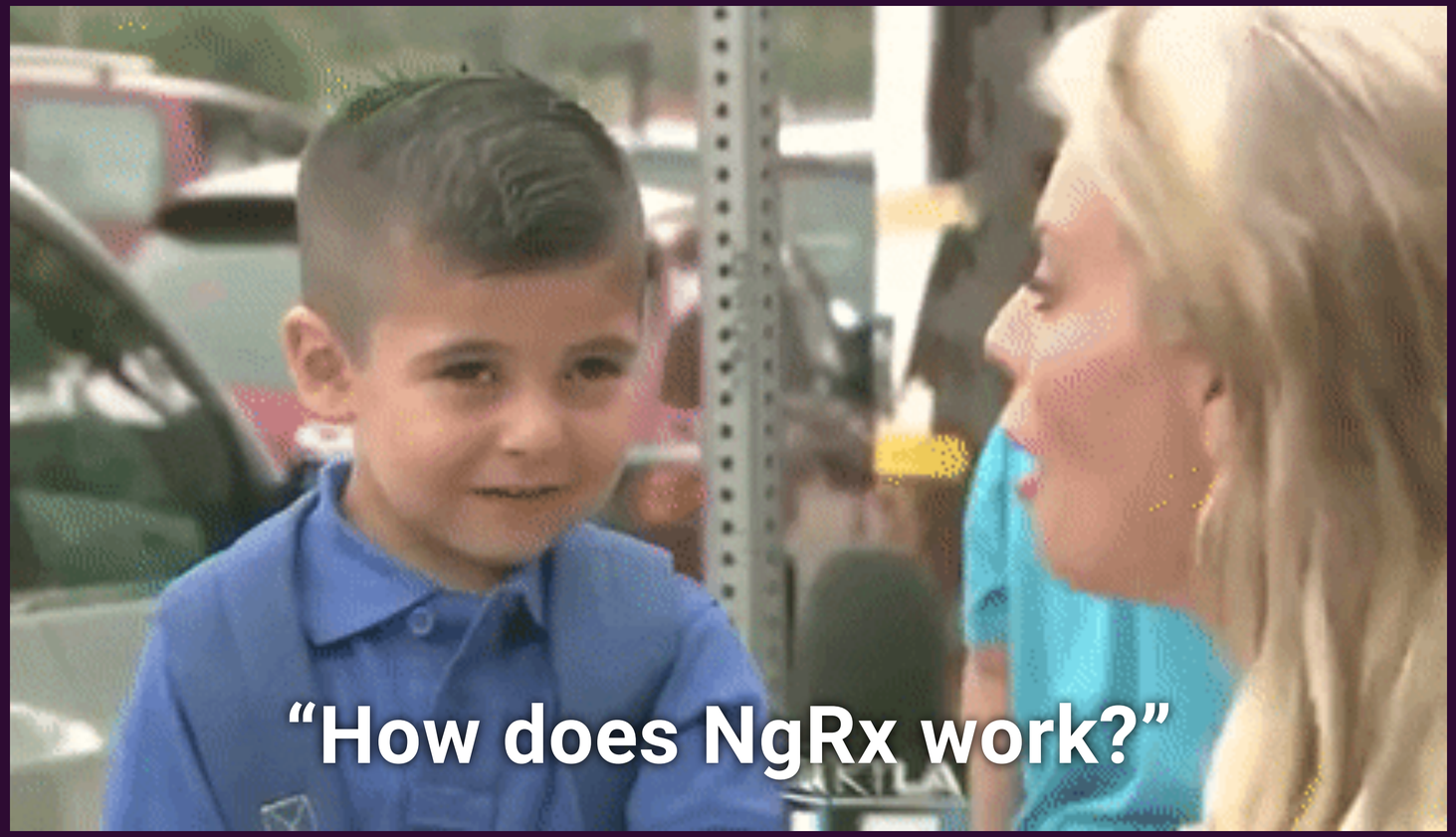
FORMAT

1. Concept Overview
2. Demo
3. Challenge

The Goal

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

DEMYSTIFYING NGRX


“How does NgRx work?”

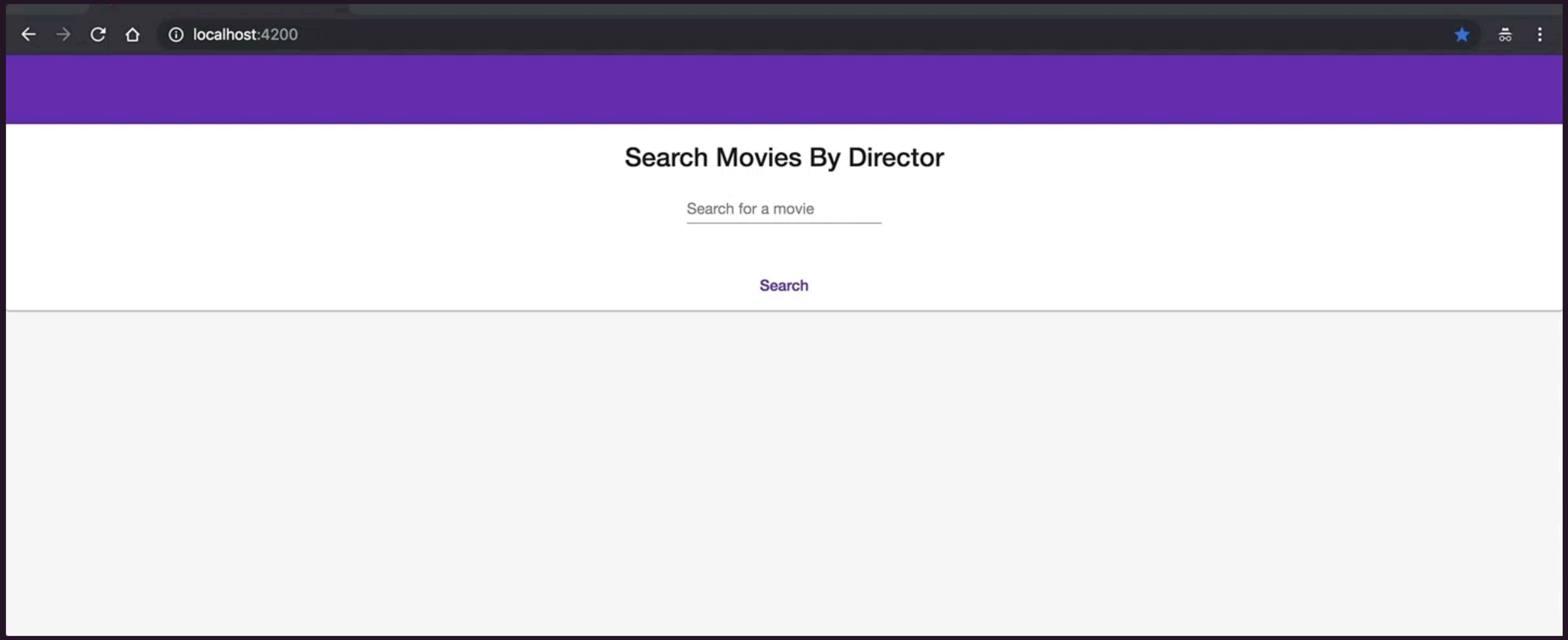
- NgRx prescribes an architecture for managing the state and side effects in you Angular application. It works by deriving a stream of updates for your application’s components called the “action stream”.
- You apply a pure function called a “reducer” to the action stream as a means of deriving state 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.


BOO THIS MAN! BOO!

Let’s try this a different way

You already know how NgRx works

COMPONENTS

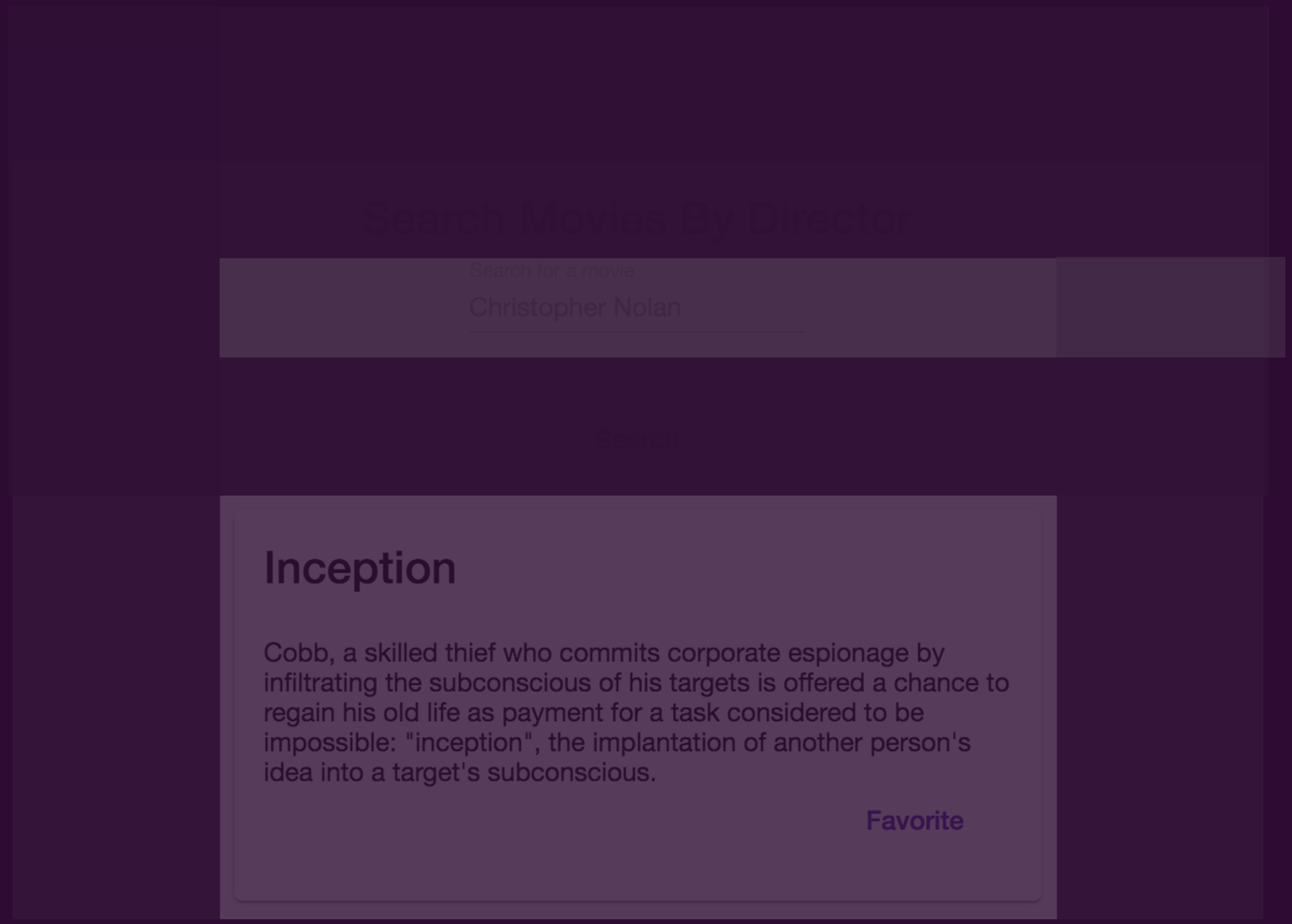

localhost:4200
Search Movies By Director
Search for a movie
Search

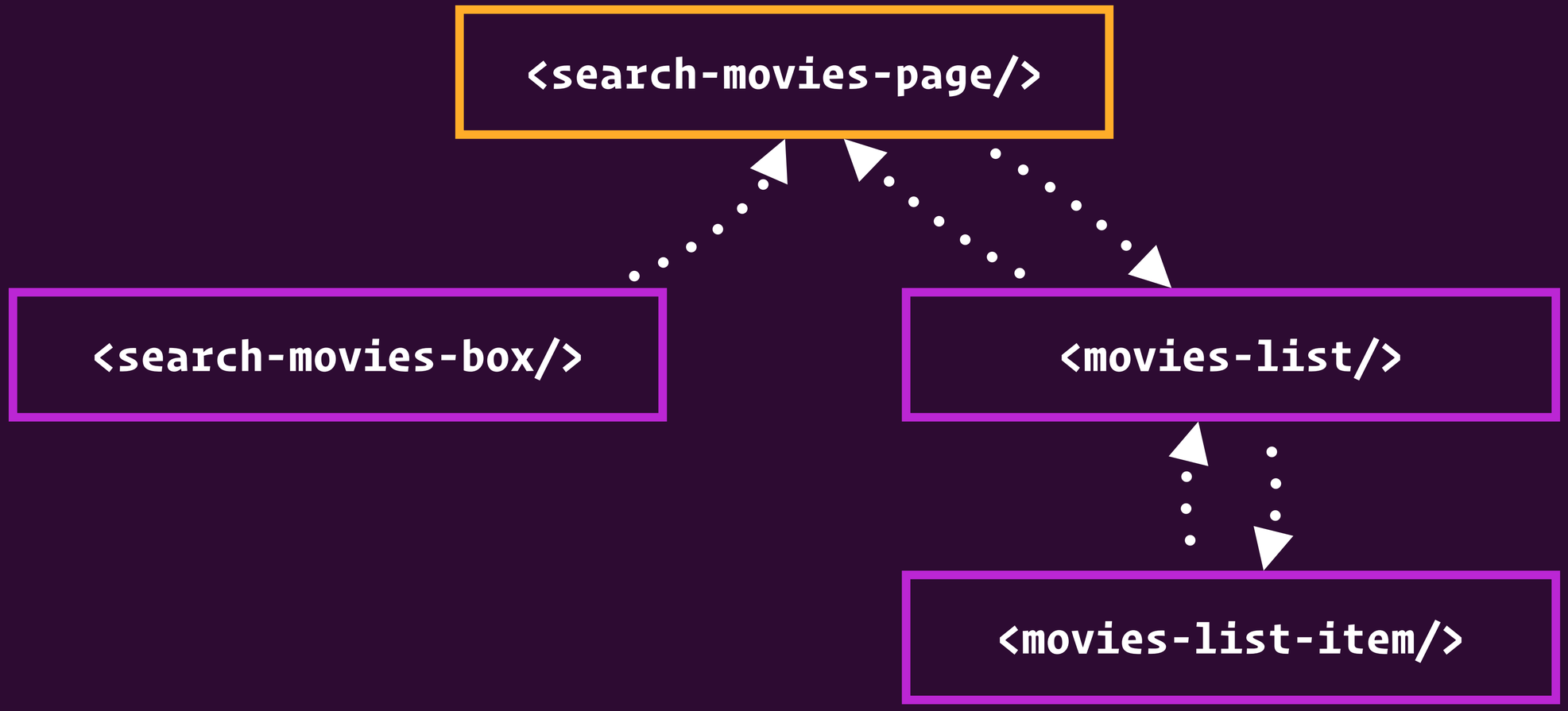
<movies-list-item/>

<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

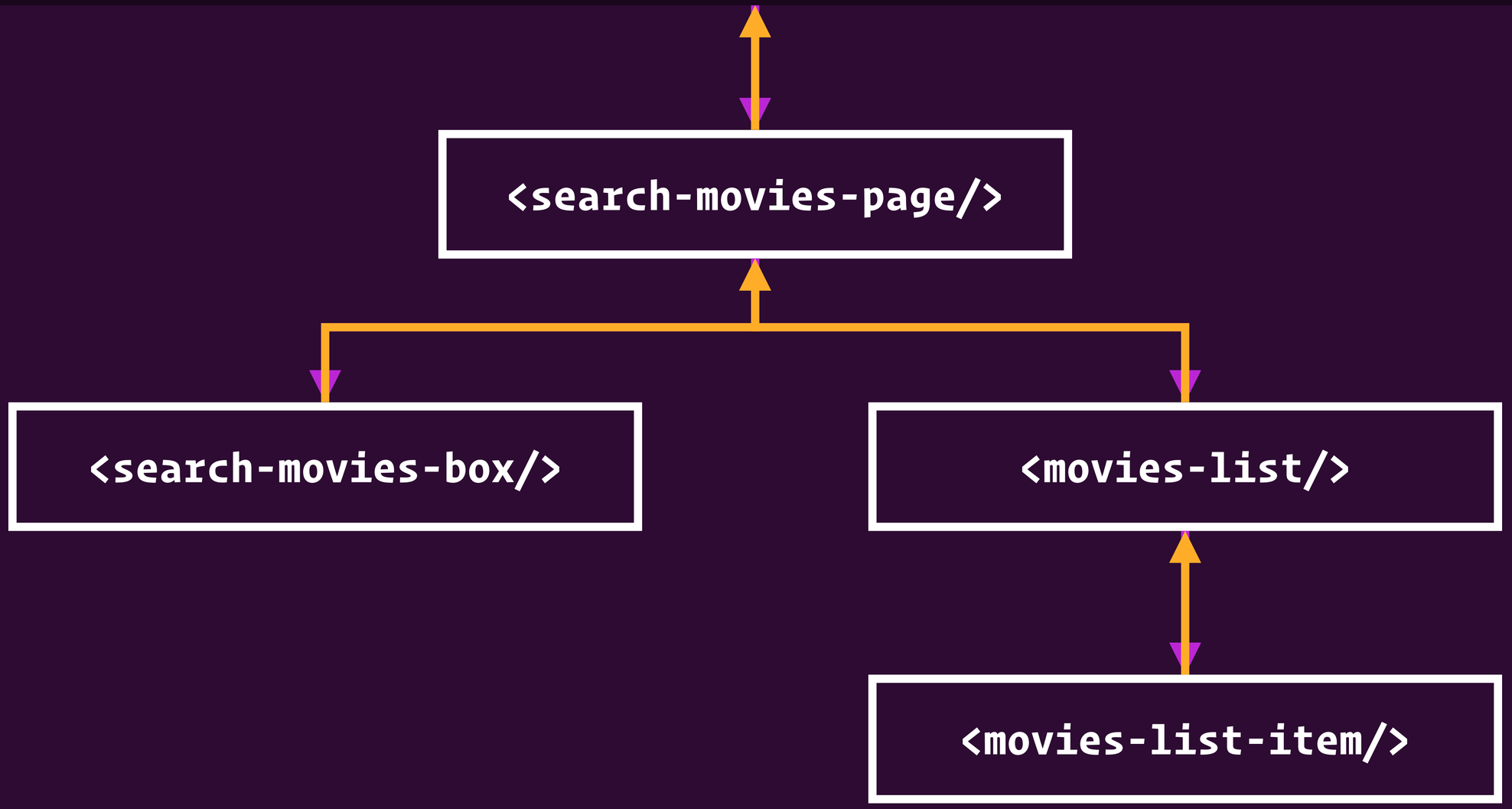
```
@Component({
  template: `
    <search-movies-box (search)="onSearch($event)"></search-movies-box>
    <movies-list
      [movies]="movies"
      (favoriteMovie)="onFavoriteMovie($event)">
    </movies-list>
  `
})
class SearchMoviesPageComponent {
  movies: Movie[] = [];

  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>();
}
```

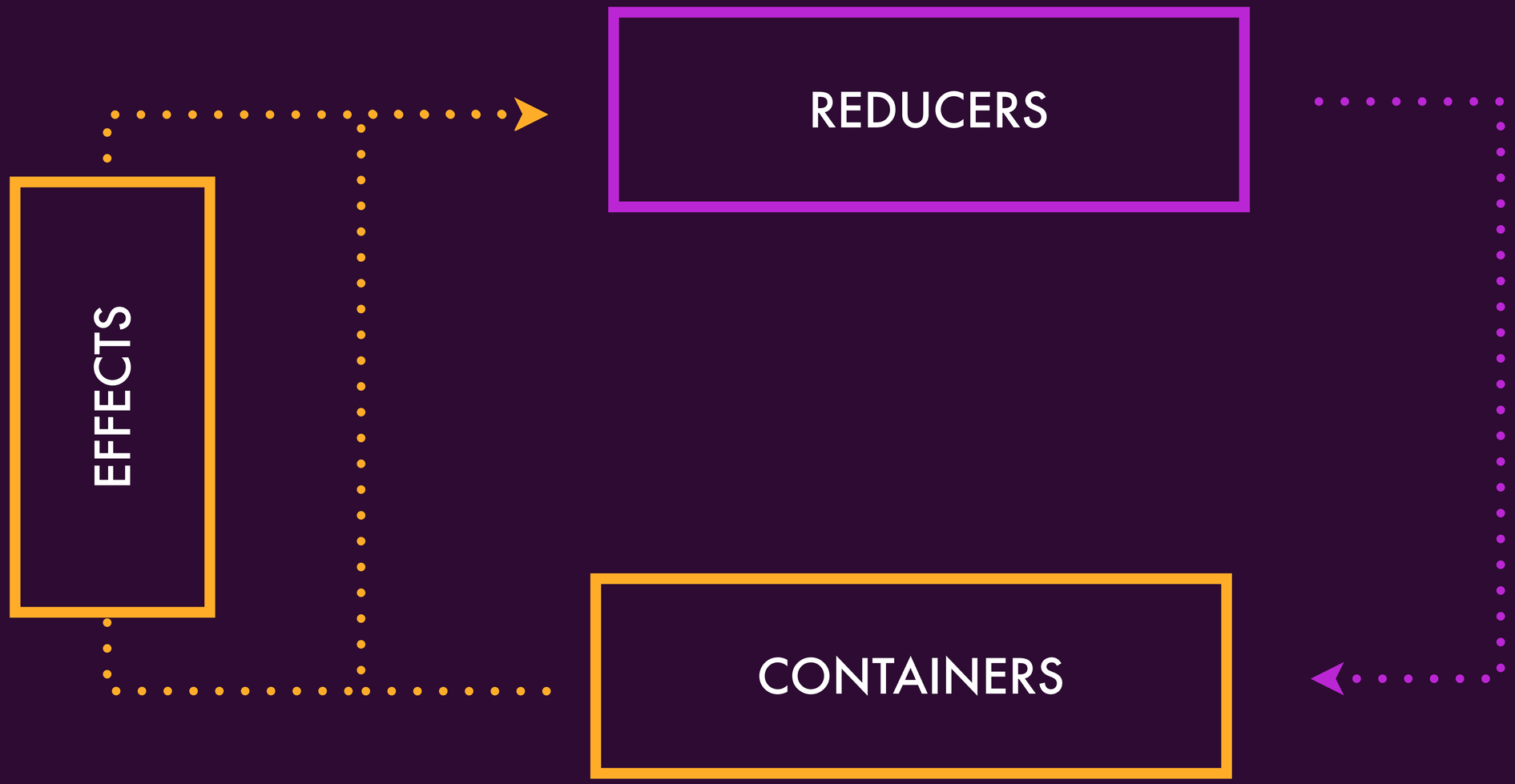
Does this component know who is listening to its output?

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

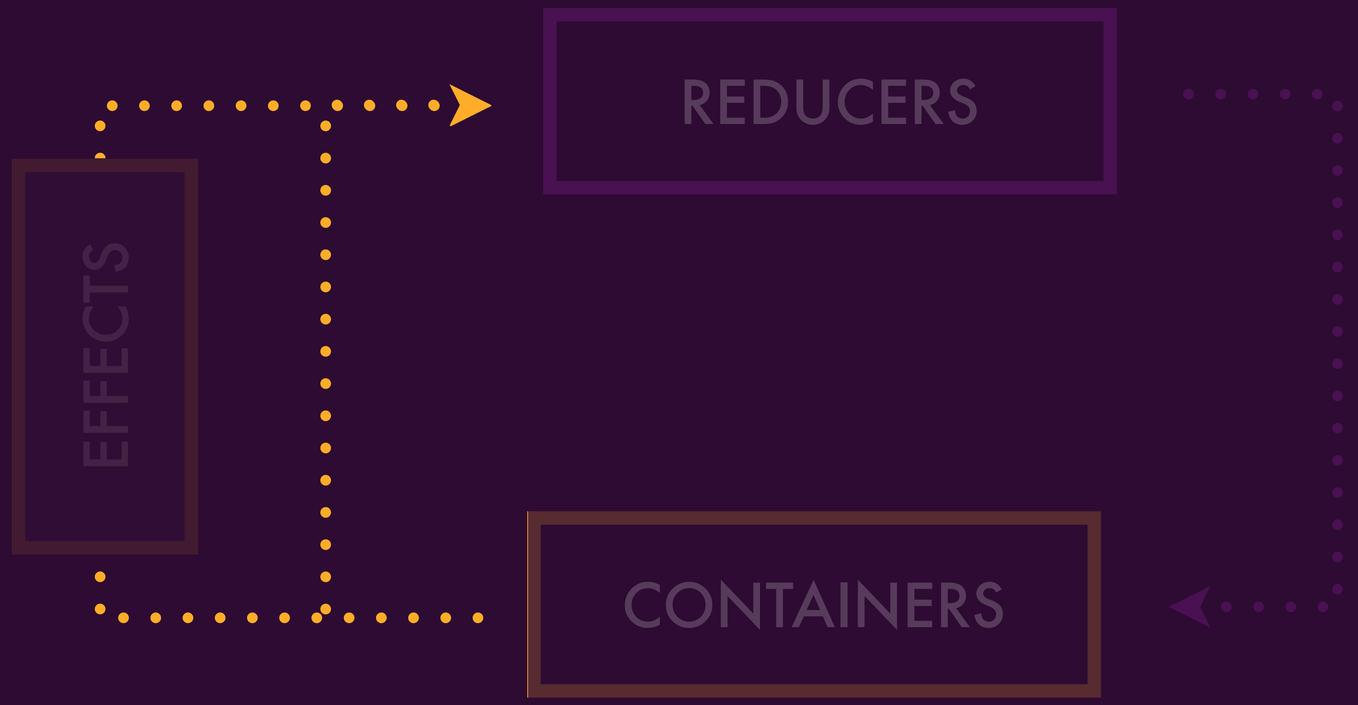
Inputs & Outputs offer Indirection

NGRX MENTAL MODEL

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


REDUCERS
EFFECTS
CONTAINERS

ACTIONS

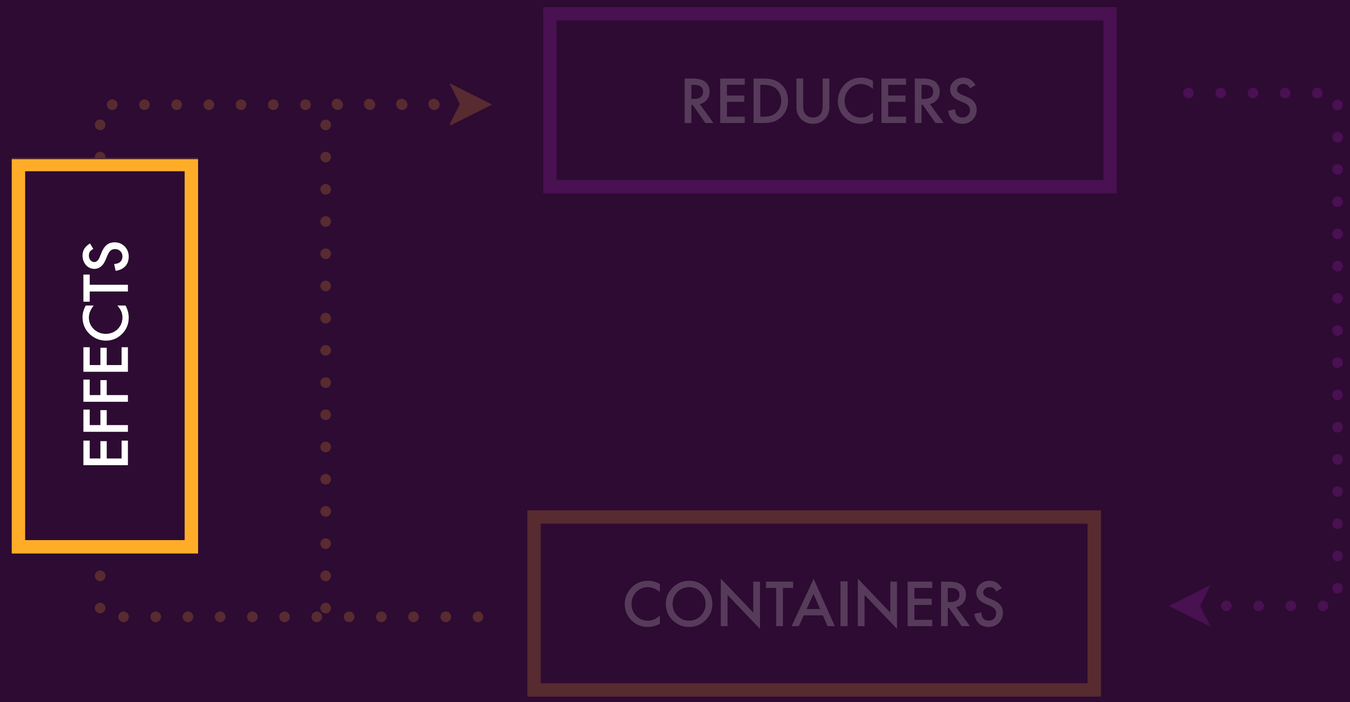


```
interface Action {
  type: string;
}
```

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

Global `@Output()` for your whole app

EFFECTS

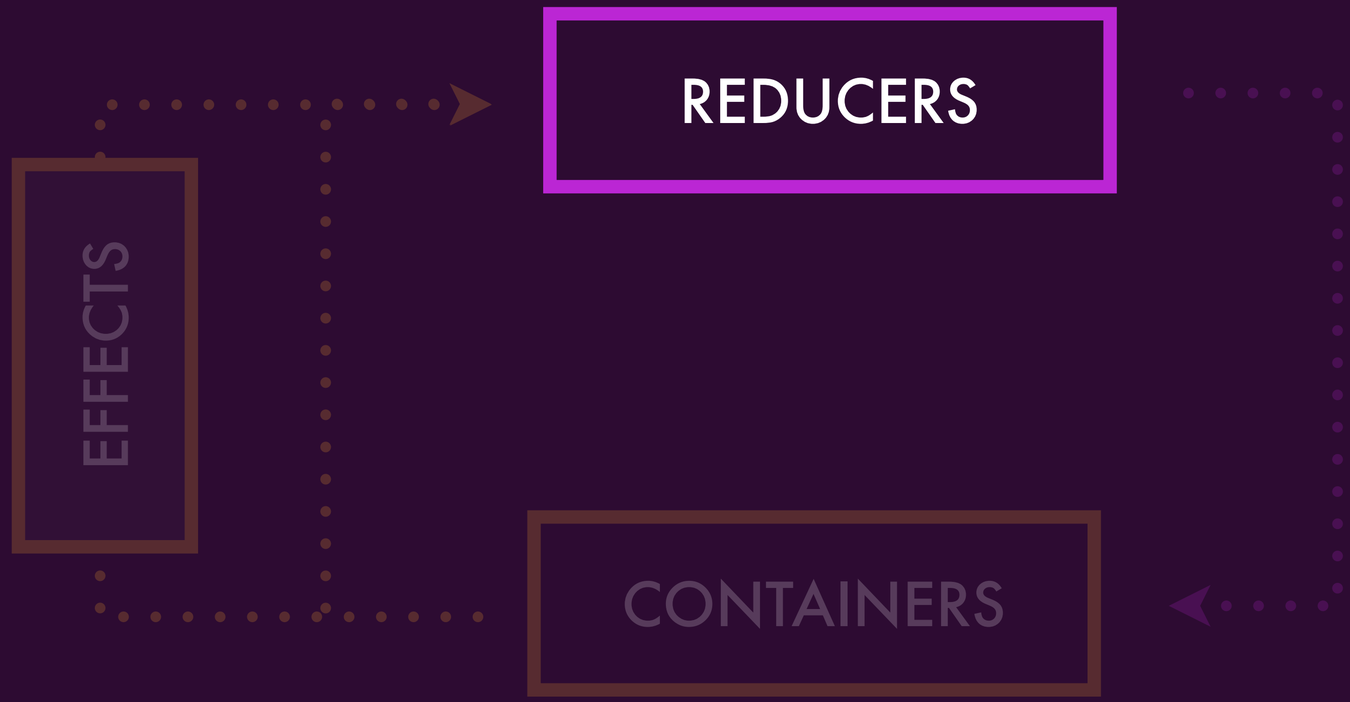


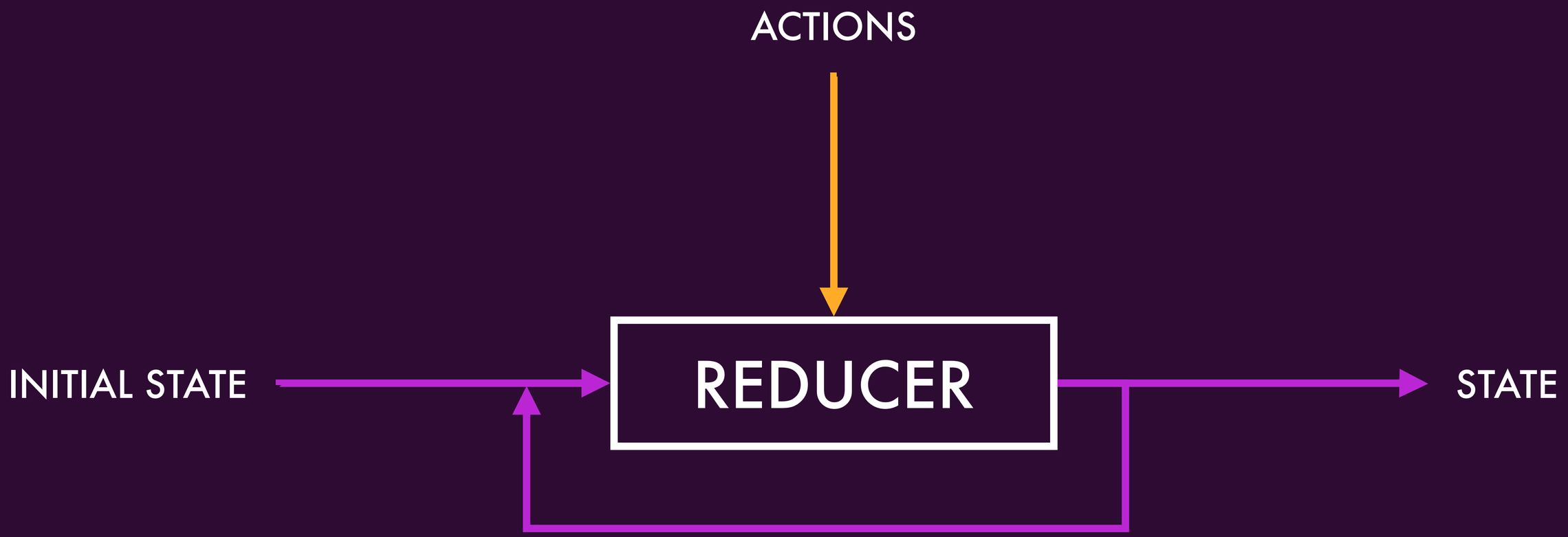
```
@Effect() findMovies$ = this.actions$
  .pipe(
    ofType('SEARCH_MOVIES'),
    switchMap(action => {
      return this.moviesService.findMovies(action.searchTerm)
        .pipe(
          map(movies => {
            return {
              type: 'MOVIES_LOADED_SUCCESS',
              movies,
   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

```
@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;
    }
  }
}
```

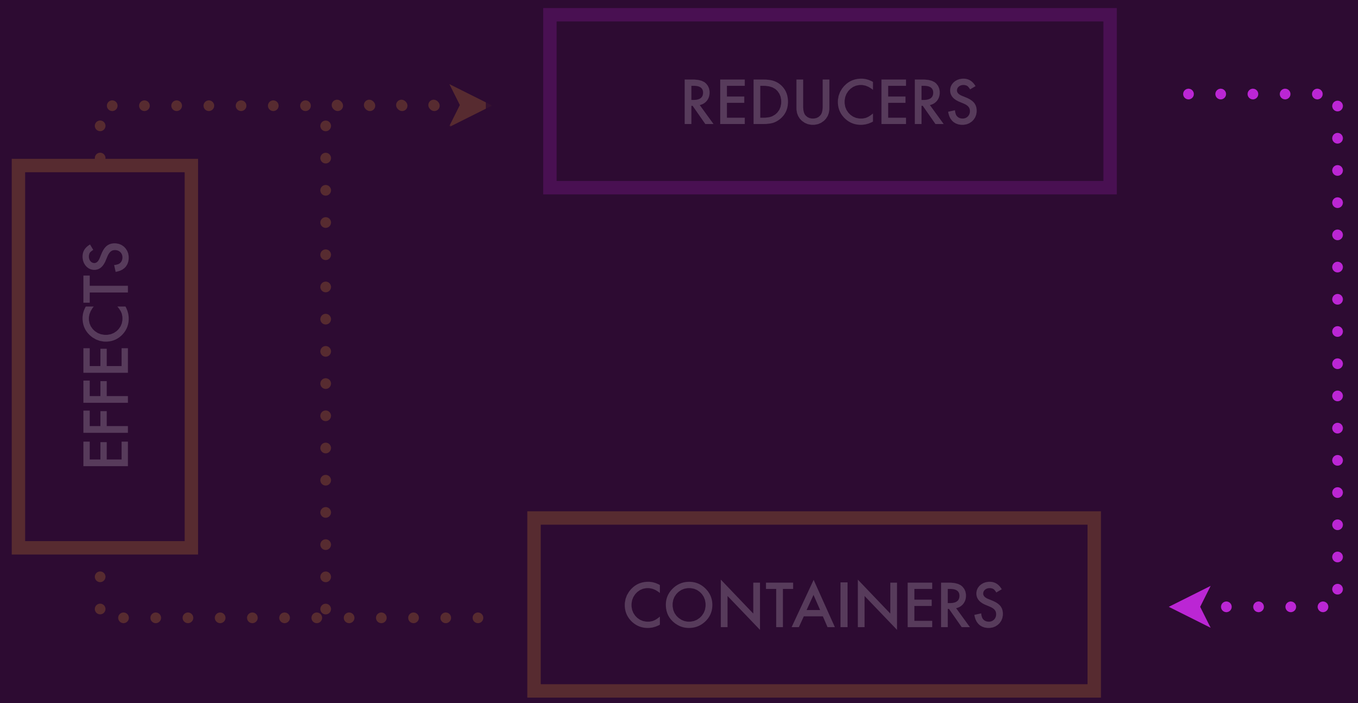
```
function moviesReducer(state = [], action) {
  switch (action.type) {
    case 'MOVIES_LOADED_SUCCESS': {
      return action.movies;
    }

    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

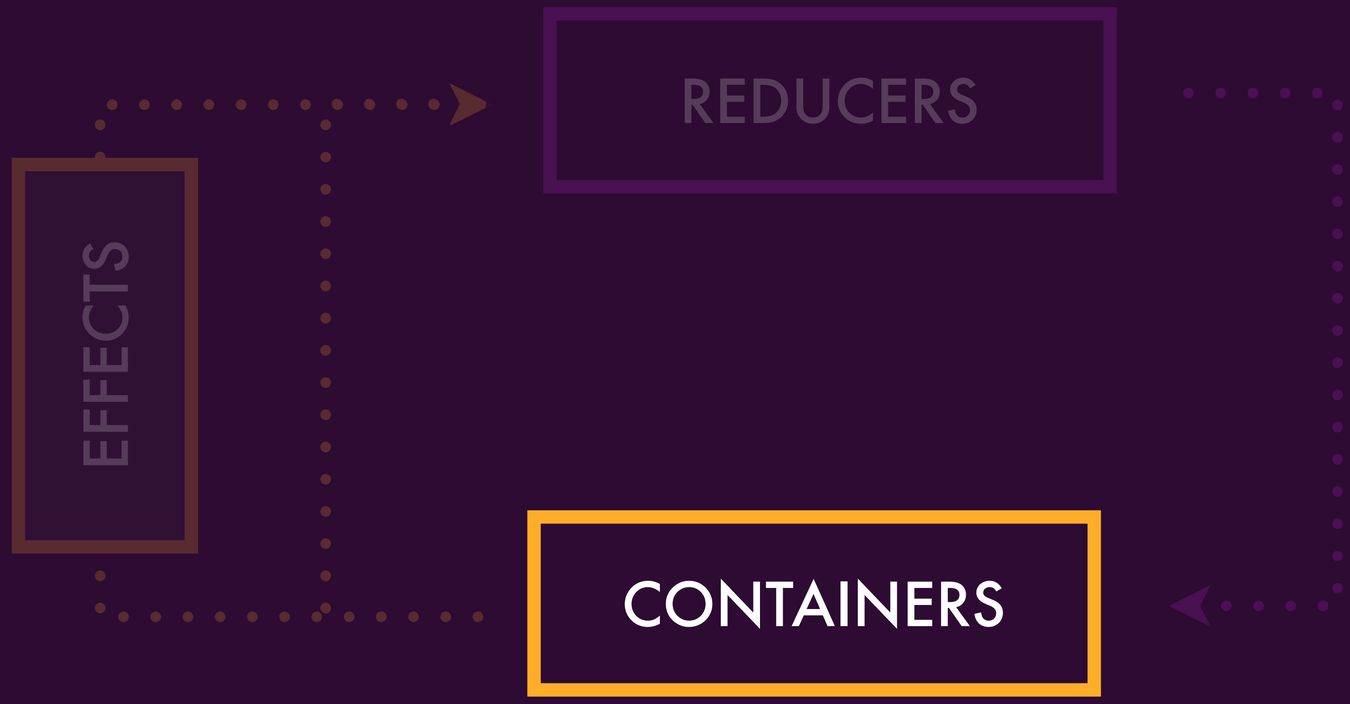



STORE
???
COMPONENTS

```
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($event)">
    </movies-list>
  `
})
export class MoviesListItemComponent {
  movies$: 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($event)">
    </movies-list>
  `
})
export class MoviesListItemComponent {
  movies$: 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

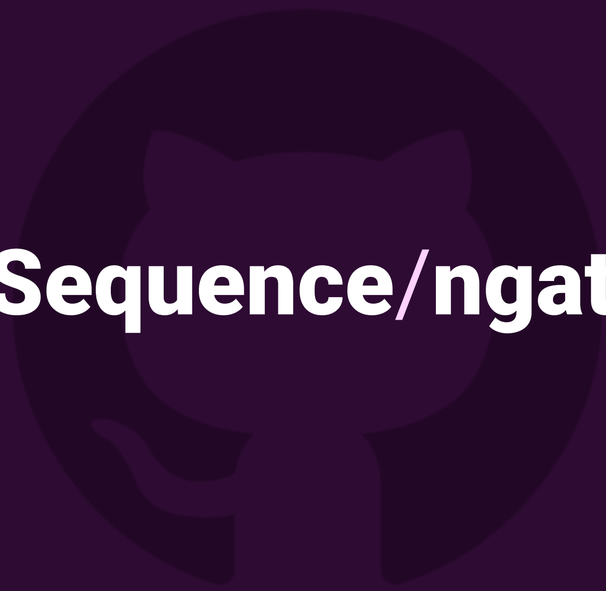
RESPONSIBILITIES

- [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/ngatl-ngrx-workshop**

Demo

Challenge

1. Checkout the **challenge** branch
2. Familiarize yourself with the **file structure**
3. Where is **items state** handled?
4. Where are the **items actions** located?
5. How does the **items state** flow into the items component?
6. How are **events** in the **items component** going to the **items reducer**?

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 WidgetsState {
  selectedWidgetId: string | null;
  widgets: Widget[];
}
```

```
export const initialState: WidgetsState = {
  selectedWidgetId: null,
  widgets: initialWidgets,
};
```

```
export function widgetsReducer(
  state = initialState,
  action: Action
): WidgetsState {
  switch (action.type) {
    default:
      return state;
  }
}
```

```
import * as fromItems from "./items/items.reducer";
import * as fromWidgets from "./widgets/widgets.reducer";

export interface AppState {
  items: fromItems.ItemsState;
  widgets: fromWidgets.WidgetsState;
}

export const reducers: ActionReducerMap<AppState> = {
  items: fromItems.itemsReducer,
  widgets: fromWidgets.widgetsReducer
};
```

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

```
export class WidgetsComponent implements OnInit {
  widgets$: Observable<Widget[]>;
  constructor(private store: Store<WidgetsState>) {
    this.widgets$ = store.select(
      (state: AppState) => state.widgets
    );
  }
}
```

```
<app-widgets-total [widgets]="widgets$ | async">
</app-widgets-total>
<app-widgets-list
  [widgets]="widgets$ | async"
  (selected)="selectWidget($event)"
  (deleted)="deleteWidget($event)"
>
</app-widgets-list>
```

STATE FLOWS DOWN

Demo

Challenge

1. Open **widgets.reducer.ts**
2. Define an interface for **WidgetsState** that has **selectedWidgetId** and **widgets** properties
3. Define an **initialState** object that implements the **WidgetsState** interface
4. Create a **widgetsReducer** that defaults to **initialState** with a **default** case in a switch statement that returns **state**

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

```
export function reducer(state = initialState, action: Action): WidgetsState {
  switch (action.type) {
    case "select":
      return {
        selectedWidgetId: action.payload.id,
        widgets: state.widgets
      };
    case "create":
      return {
        selectedWidgetId: state.selectedWidgetId,
        widgets: createWidget(state.widgets, action.payload)
      };
    default:
      return state;
  }
}
```

```
const createWidget = (widgets, widget) => [
  ...widgets,
  widget
];
const updateWidget = (widgets, widget) =>
  widgets.map(w => {
    return w.id === widget.id
      ? Object.assign({}, widget)
      : w;
  });
const deleteWidget = (widgets, widget) =>
  widgets.filter(w => widget.id !== w.id);
```

```
class WidgetsComponent {
  createWidget(widget) {
    this.store.dispatch({
      type: "create",
      payload: widget
    });
  }
}
```

Demo

Challenge

1. Update widgetsReducer to handle “**create**”, “**update**”, and “**delete**” actions
2. Use the helper functions already in **widgets.reducer.ts**
3. Update **widgets.component.ts** to dispatch actions to the component

ACTIONS

ACTIONS

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

GOOD ACTION HYGIENE

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

```
class WidgetsComponent {
  createWidget(widget) {
    this.store.dispatch({
      type: "create",
      payload: widget
    });
  }
}
```

```
export enum WidgetsActionTypes {
  WidgetSelected = "[Widgets Page] Widget Selected",
  AddWidget = "[Widgets Page] Add Widget",
  UpdateWidget = "[Widgets Page] Update Widget",
  DeleteWidget = "[Widgets Page] Delete Widget"
}
```

```
export class SelectWidget implements Action {
  readonly type = WidgetsActionTypes.WidgetSelected;

  constructor(public payload) {}
}
```

```
export class AddWidget implements Action {
  readonly type = WidgetsActionTypes.AddWidget;
  constructor(public payload: Widget) {}
}

export class UpdateWidget implements Action {
  readonly type = WidgetsActionTypes.UpdateWidget;
  constructor(public payload: Widget) {}
}

export class DeleteWidget implements Action {
  readonly type = WidgetsActionTypes.DeleteWidget;
  constructor(public payload: Widget) {}
}
```

```
export type WidgetsActions =
   | SelectWidget
   | AddWidget
   | UpdateWidget
   | DeleteWidget;
```

```
export function widgetsReducer(
  state = initialState,
  action: WidgetsActions
): WidgetsState {
  switch (action.type) {
    case WidgetsActionTypes.WidgetSelected:
      ...
    case WidgetsActionTypes.AddWidget:
      ...
    case WidgetsActionTypes.UpdateWidget:
      ...
    case WidgetsActionTypes.DeleteWidget:
      ...
    default:
      return state;
  }
}
```

```
createWidget(widget) {
  this.store.dispatch({
    type: "create",
    payload: widget
  });
}
```

```
createWidget(widget) {
  this.store.dispatch(new WidgetActions.AddWidget(widget));
}
```

Demo

Challenge

1. Open **widgets.actions.ts** and create an **enum** to hold the various action types
2. Create **strongly typed actions** that adhere to **good action hygiene** for add, update, delete, and select
3. Export actions as a **union type**
4. Update **widgets.components.ts** and **widgets.reducer.ts** to use the new actions

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 WidgetsState extends EntityState<Widget> {
  selectedWidgetId: string | null;
}
export const adapter: EntityAdapter<Widget> = createEntityAdapter<Widget>();
export const initialState: WidgetsState = adapter.getInitialState(
  {
    selectedWidgetId: null
  }
);
```

```ts
export interface WidgetsState extends EntityState<Widget> {
  selectedWidgetId: string | null;
}
export const adapter: EntityAdapter<Widget> = createEntityAdapter<Widget>();
export const initialState: WidgetsState = adapter.getInitialState(
  {
    selectedWidgetId: null
  }
);
```

```ts
export interface WidgetsState extends EntityState<Widget> {
  selectedWidgetId: string | null;
}
export const adapter: EntityAdapter<Widget> = createEntityAdapter<Widget>();
export const initialState: WidgetsState = adapter.getInitialState(
  {
    selectedWidgetId: null
  }
);
```

```ts
export interface WidgetsState extends EntityState<Widget> {
  selectedWidgetId: string | null;
}
export const adapter: EntityAdapter<Widget> = createEntityAdapter<Widget>();
export const initialState: WidgetsState = adapter.getInitialState(
  {
    selectedWidgetId: null
  }
);
```

```
case WidgetsActionTypes.AddWidget:
  return {
    selectedWidgetId: state.selectedWidgetId,
    widgets: createWidget(state.widgets, action.payload)
  };
```

```
case WidgetsActionTypes.AddWidget:
  return adapter.addOne(action.payload, state);
```

```
export const { selectAll } = adapter.createSelectors();
```

```ts
export const selectAllWidgets = createSelector(
  (state: State) => state.widgets,
  fromWidgets.selectAll,
);
```

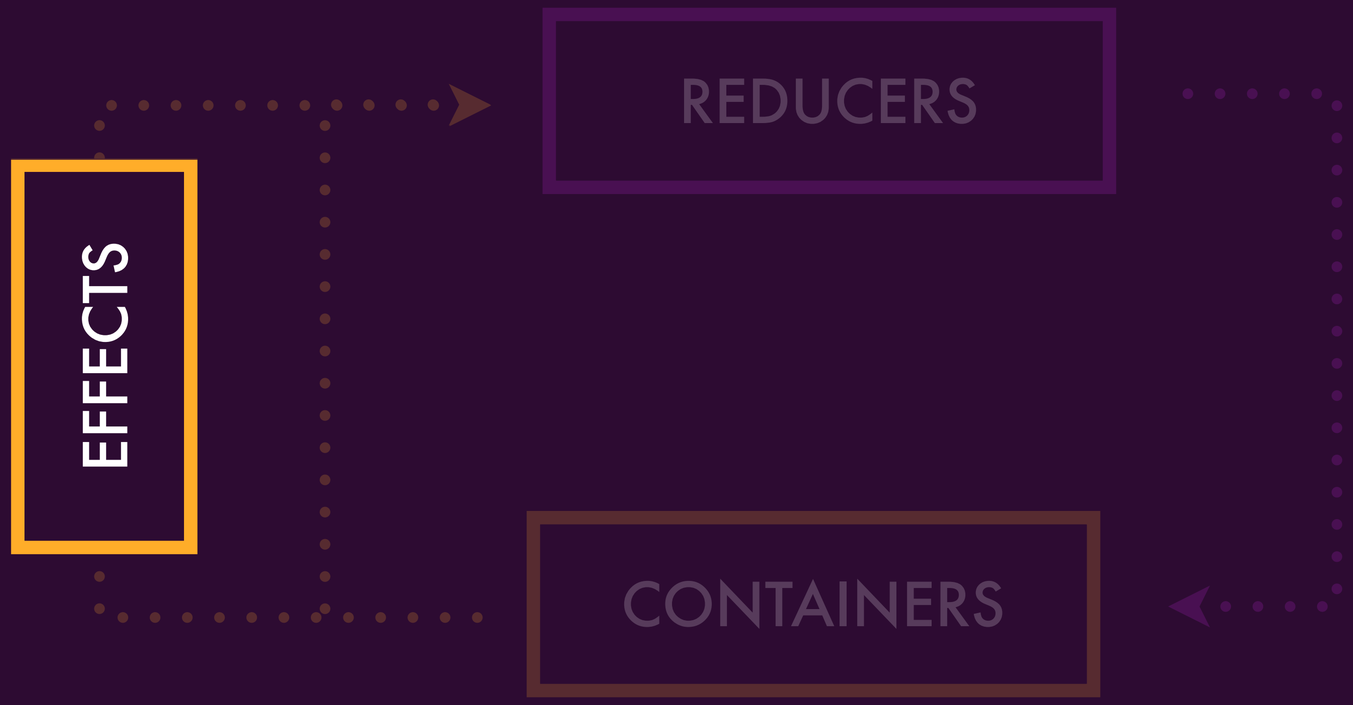
```
this.widgets$ = store.select(selectAllWidgets);
```

Demo

Challenge

1. Update **widgets.reducer.ts** to use **EntityState** to define **WidgetsState**
2. Create an **unsorted entity adapter** for WidgetsState
3. Use the **adapter** to initialize initialState
4. Update the reducer to use the **adapter methods**
5. Create, export, and compose a **selectAllWidgets** selector
6. Use **selectAllWidgets** in **widgets.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

```
export enum WidgetsActionTypes {
  WidgetsLoaded = '[Widgets/API] Widgets Loaded',
  WidgetAdded = '[Widgets/API] Widget Added',
  WidgetUpdated = '[Widgets/API] Widget Updated',
  WidgetDeleted = '[Widgets/API] Widget Deleted'
}
```

```
export class WidgetsEffects {
  @Effect() loadWidgets$ = this.actions$.pipe(
    ofType(WidgetsActionTypes.LoadWidgets),
    mergeMap(() =>
      this.widgetsService.all().pipe(
        map(
          (res: Widget[]) =>
            new WidgetActions.WidgetsLoaded(res)
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
export class WidgetsEffects {
  @Effect() loadWidgets$ = this.actions$.pipe(
    ofType(WidgetsActionTypes.LoadWidgets),
    mergeMap(() =>
      this.widgetsService.all().pipe(
        map(
          (res: Widget[]) =>
            new WidgetActions.WidgetsLoaded(res)
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
export class WidgetsEffects {
  @Effect() loadWidgets$ = this.actions$.pipe(
    ofType(WidgetsActionTypes.LoadWidgets),
    mergeMap(() =>
      this.widgetsService.all().pipe(
        map(
          (res: Widget[]) =>
            new WidgetActions.WidgetsLoaded(res)
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

```
export class WidgetsEffects {
  @Effect() loadWidgets$ = this.actions$.pipe(
    ofType(WidgetsActionTypes.LoadWidgets),
    mergeMap(() =>
      this.widgetsService.all().pipe(
        map(
          (res: Widget[]) =>
            new WidgetActions.WidgetsLoaded(res)
        ),
        catchError(() => EMPTY)
      )
    )
  );
}
```

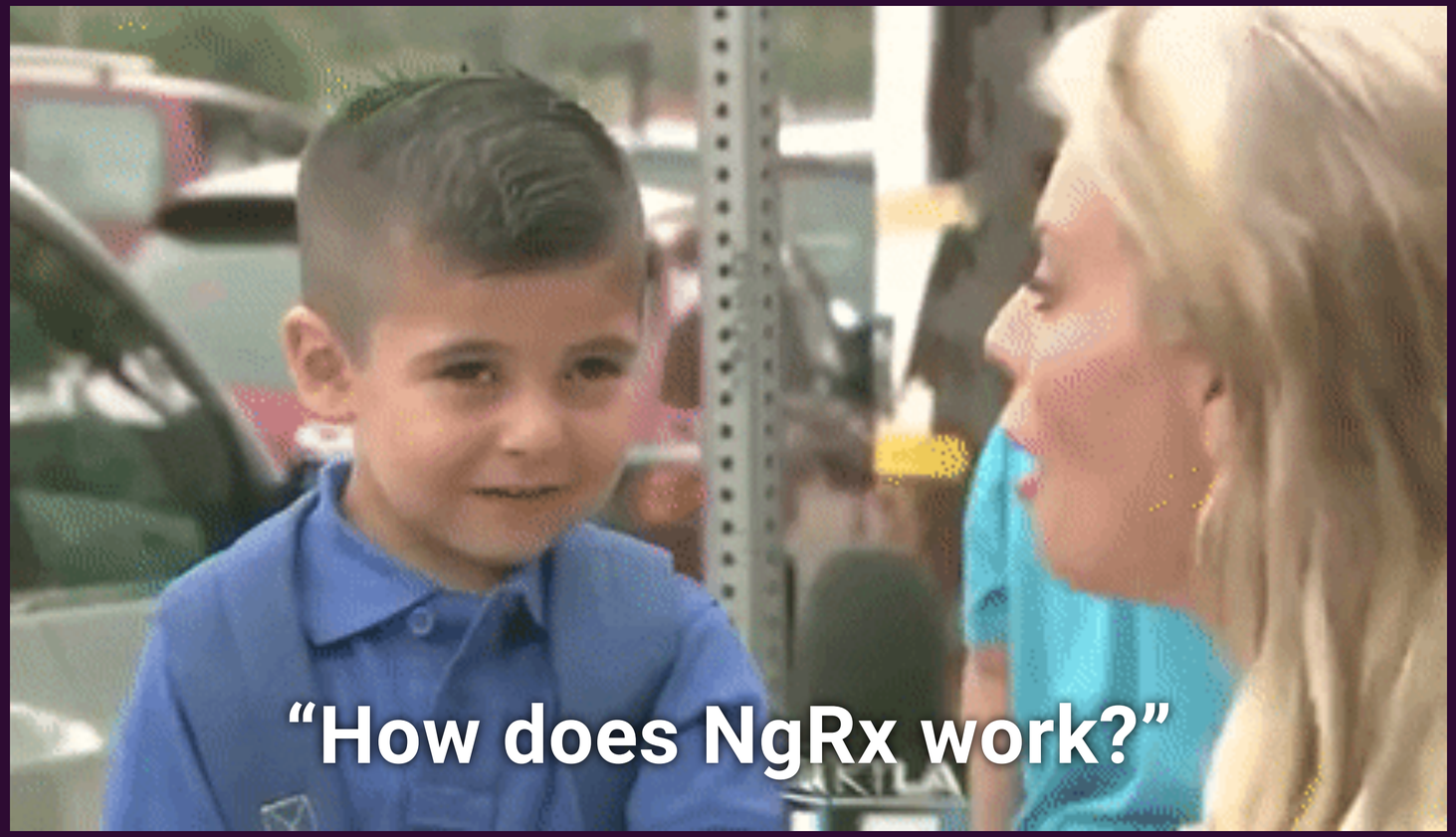
```
const BASE_URL = "http://localhost:3000/widgets/";

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

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

```
export function widgetsReducer(state = initialState, action: WidgetsActions): WidgetsState {
  switch (action.type) {
    case WidgetsActionTypes.WidgetSelected:
      return { ...state, selectedWidgetId: action.payload };
    case WidgetsActionTypes.WidgetsLoaded:
      return adapter.addAll(action.payload, state);
    case WidgetsActionTypes.WidgetAdded:
      return adapter.addOne(action.payload, state);
    case WidgetsActionTypes.UpdateWidget:
      return adapter.upsertOne(action.payload, state);
    case WidgetsActionTypes.DeleteWidget:
      return adapter.removeOne(action.payload.id, state);
    default:
      return state;
  }
}
```

Demo

Challenge

1. Update **widgets.actions.ts** to have actions for **LoadWidgets** and **WidgetsLoaded**
2. Create a **loadWidgets$** effect that calls **WidgetsService.all** and maps the result into a **WidgetsLoaded** action
3. Update **widgetsReducer** to handle the **WidgetsLoaded** action
4. Update the **getWidgets** method in **widgets.component.ts** to dispatch the **LoadWidgets** action


“How does NgRx work?”

“How does NgRx work?”

HELP US IMPROVE

https://bit.ly/2ROXvn0

@MikeRyanDev

@brandontroberts

THANK YOU