FORK, CLONE AND FOLLOW THE SETUP INSTRUCTIONS

github.com/**LiveLoveApp/mosaic-ngrx-workshop-2021**

A REACTIVE STATE OF MIND

WITH ANGULAR AND NGRX

Mike Ryan

@MikeRyanDev

Mike Ryan

@MikeRyanDev

Principal Architect at LiveLoveApp

Mike Ryan

@MikeRyanDev

Principal Architect at LiveLoveApp

Google Developer Expert

Mike Ryan

@MikeRyanDev

Principal Architect at LiveLoveApp

Google Developer Expert

NgRx Core Team

Brian Love

@brian_love

Brian Love

@brian_love

Principal Architect at LiveLoveApp

Brian Love

@brian_love

Principal Architect at LiveLoveApp

Google Developer Expert

Brian Love

@brian_love

Principal Architect at LiveLoveApp

Google Developer Expert

NgRx Fanboy

Open source libraries for Angular

Open source libraries for Angular

Built with reactivity in mind

Open source libraries for Angular

Built with reactivity in mind

State management and side effects

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
- Entity
- Meta-reducers

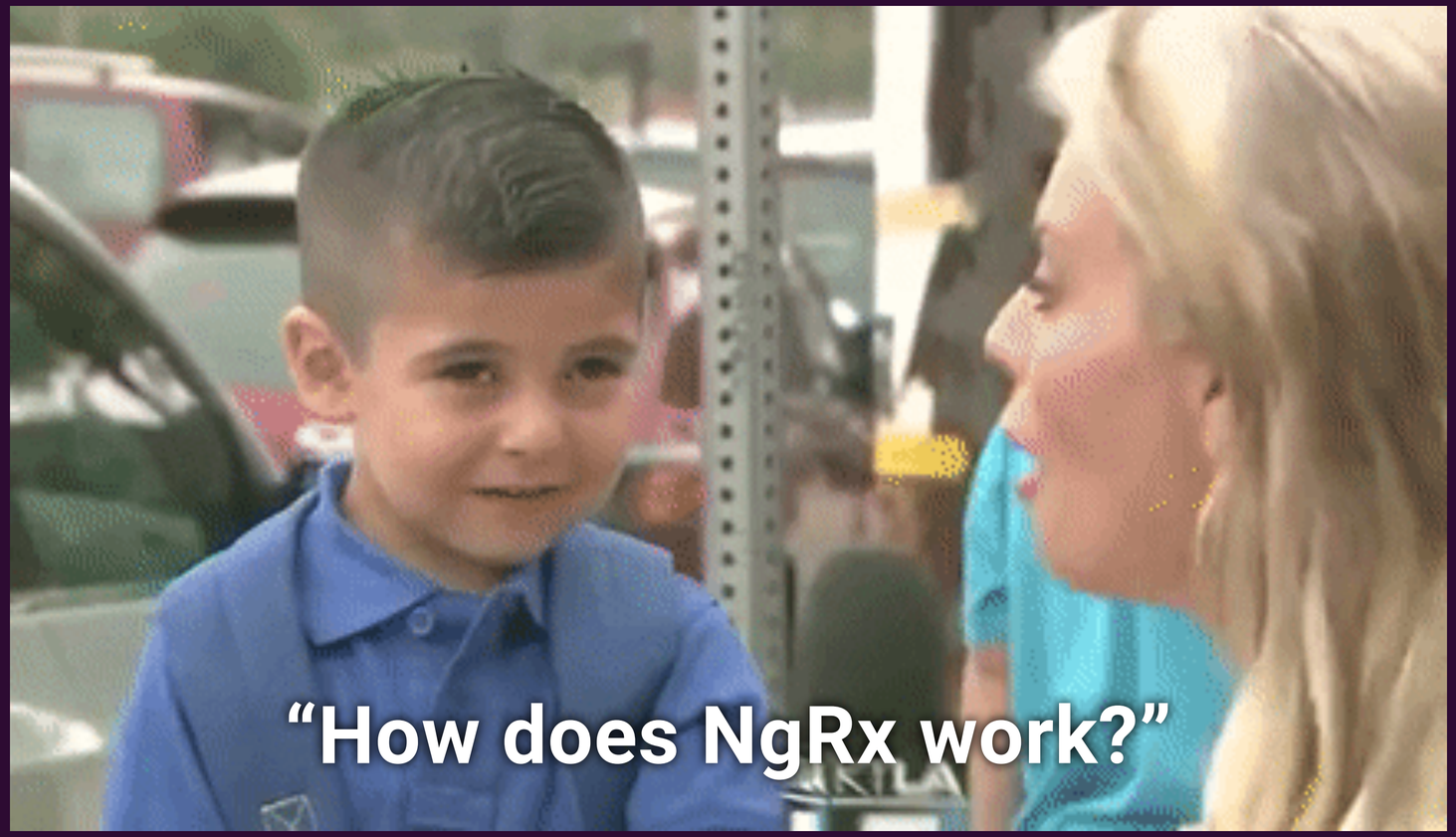
FORMAT

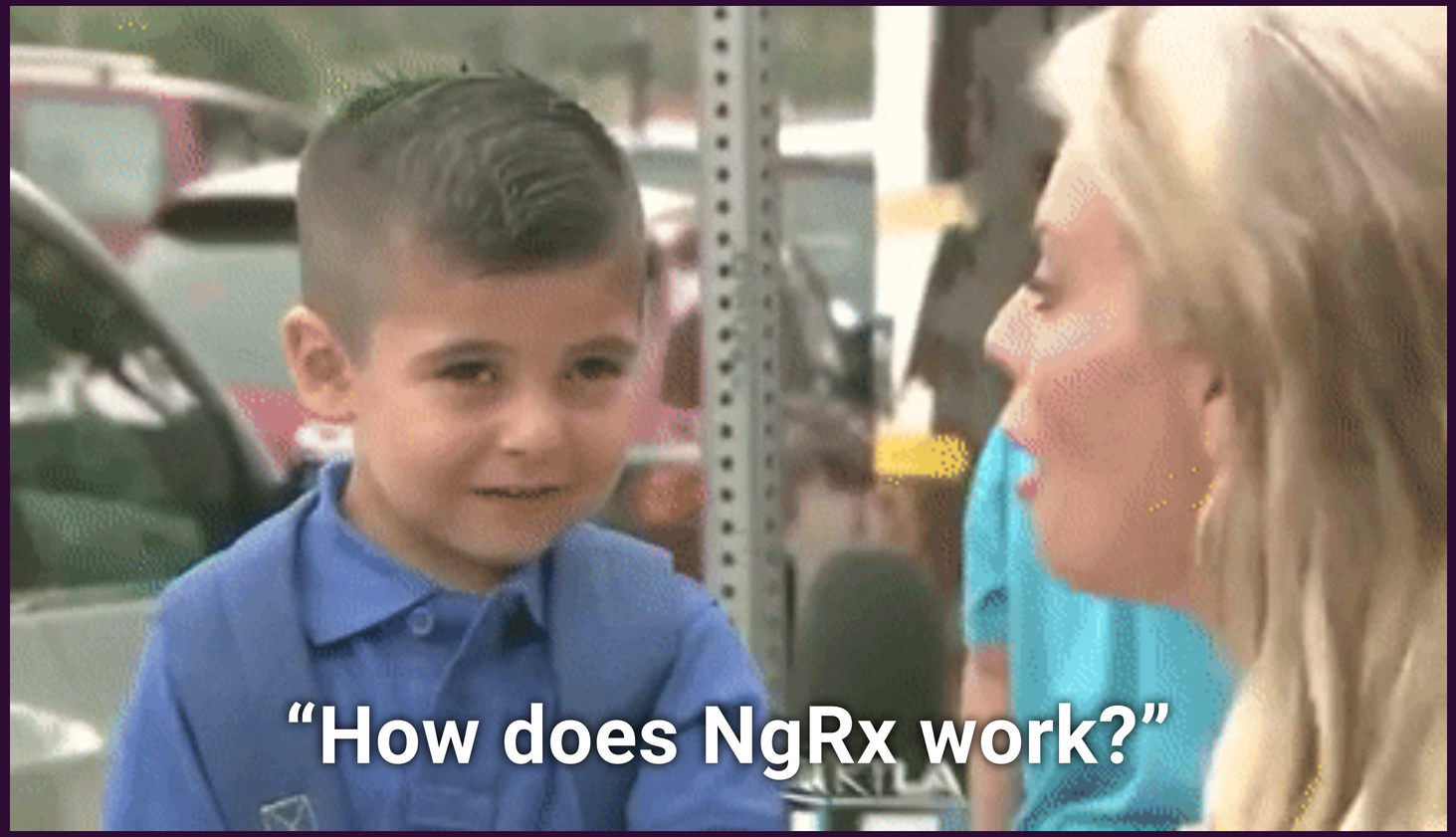
1. Concept Overview
2. Demo
3. Challenge
4. Solution

The Goal

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

DEMYSTIFYING NGRX


“How does NgRx work?”


“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.

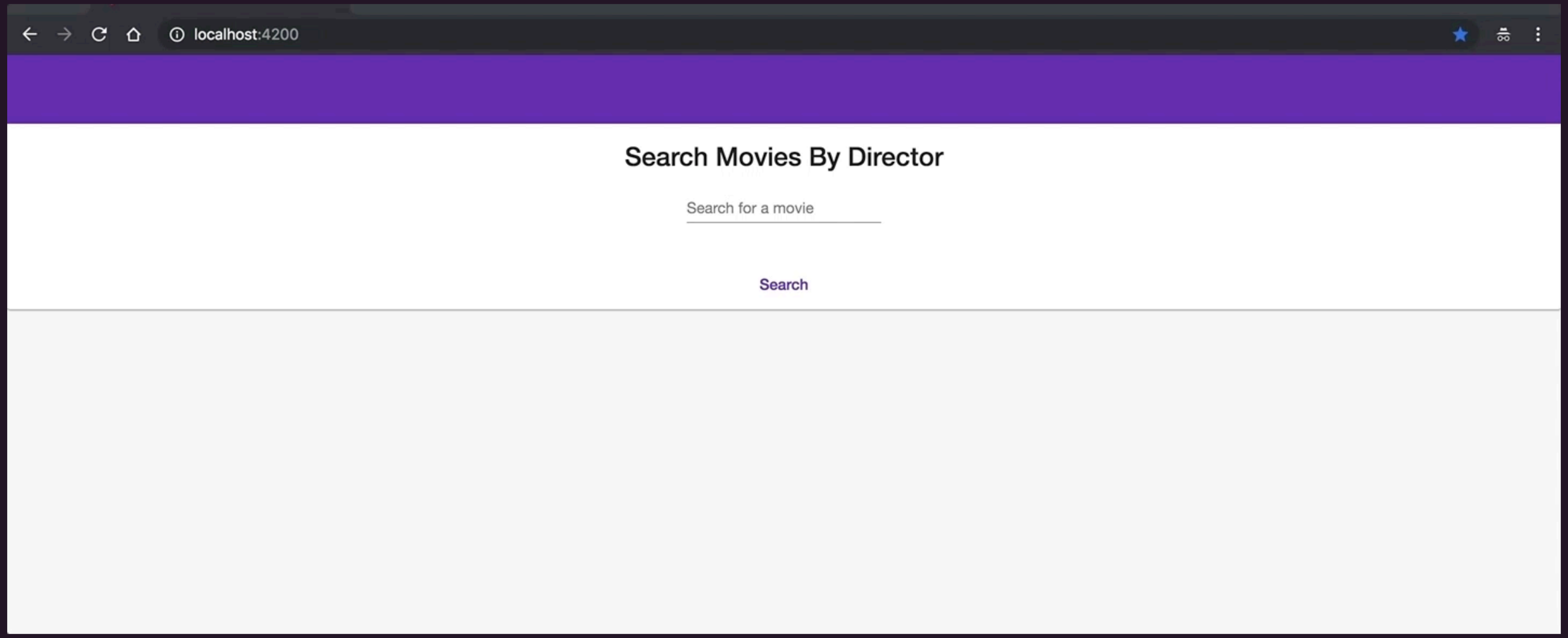
BOO THIS MAN! BOO!

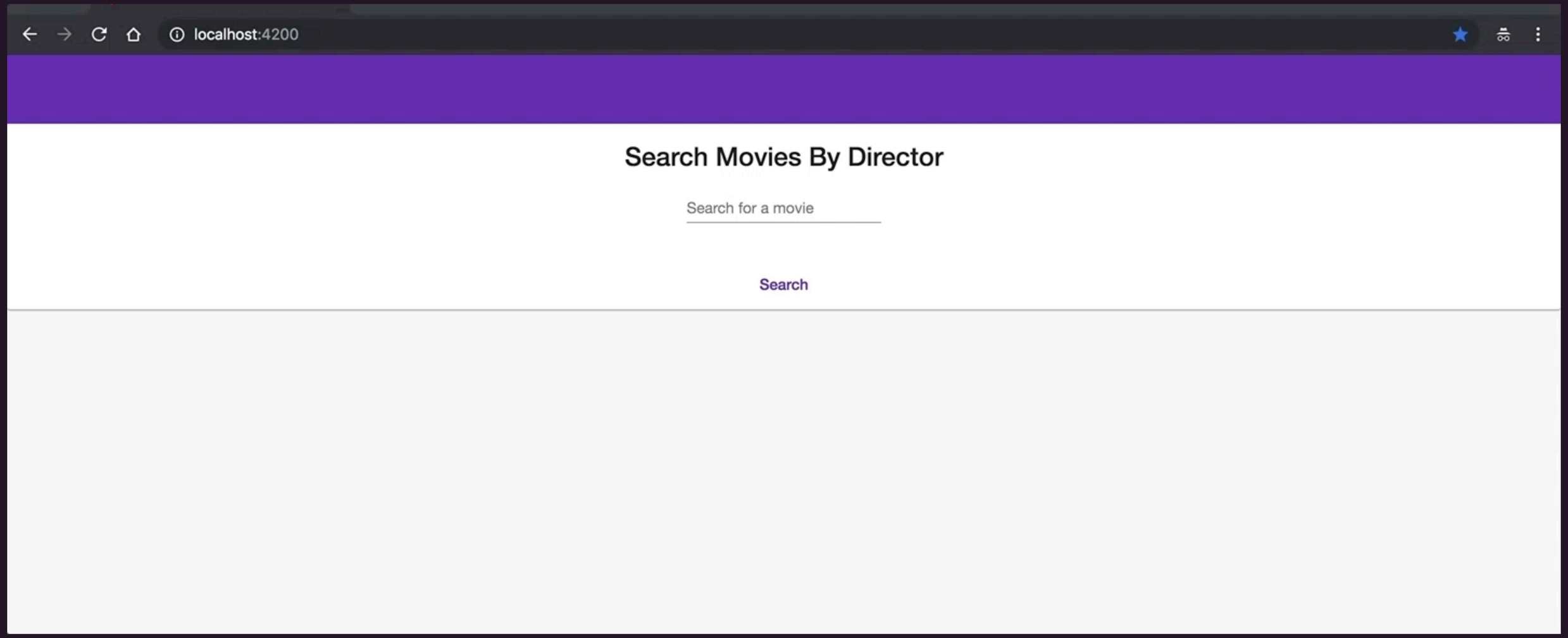
BOO THIS MAN! BOO!

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


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

Search Movies By Director

Search for a movie

Christopher Nolan

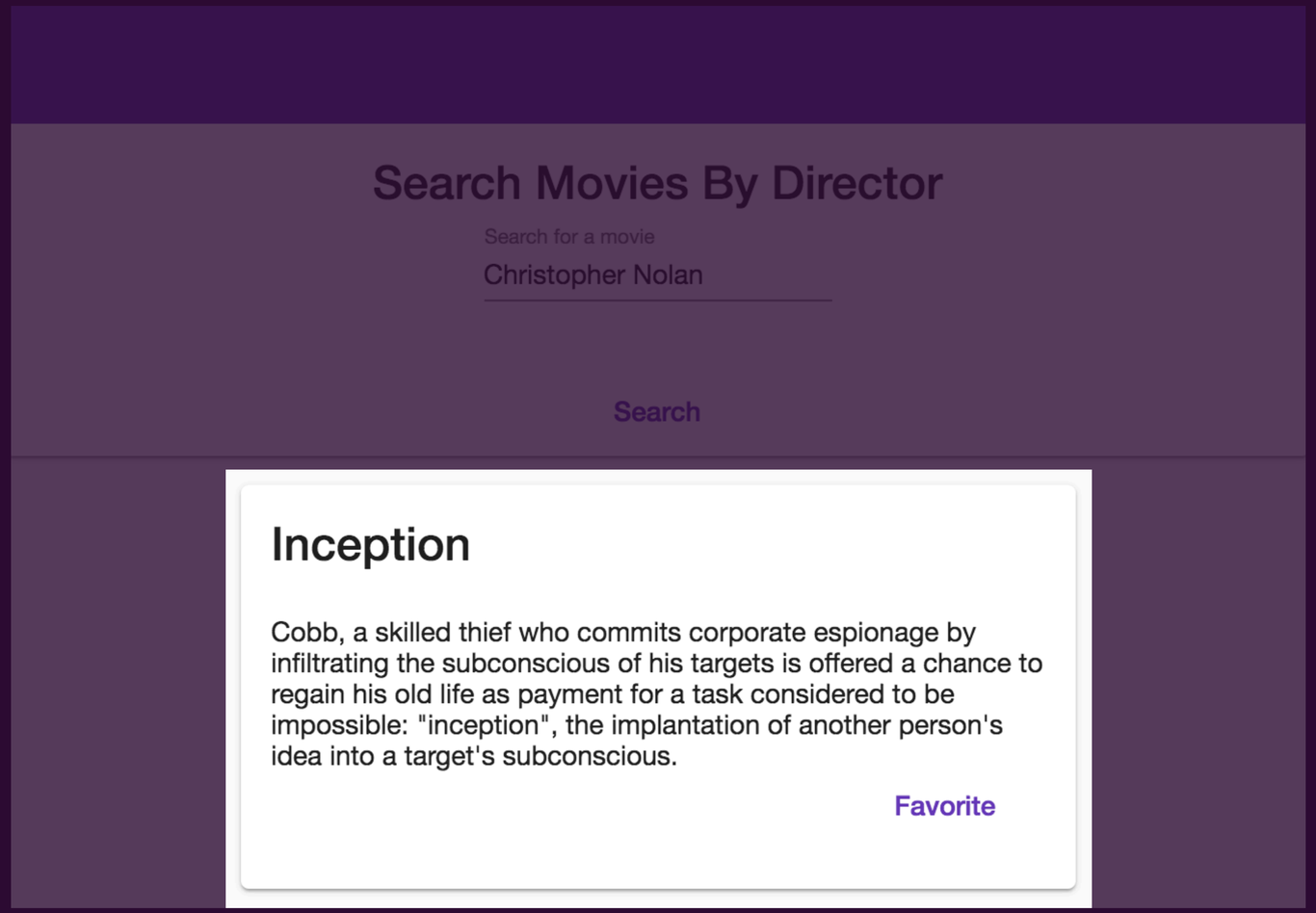
Search

Inception

Cobb, a skilled thief who commits corporate espionage by infiltrating the subconscious of his targets is offered a chance to regain his old life as payment for a task considered to be impossible: "inception", the implantation of another person's idea into a target's subconscious.

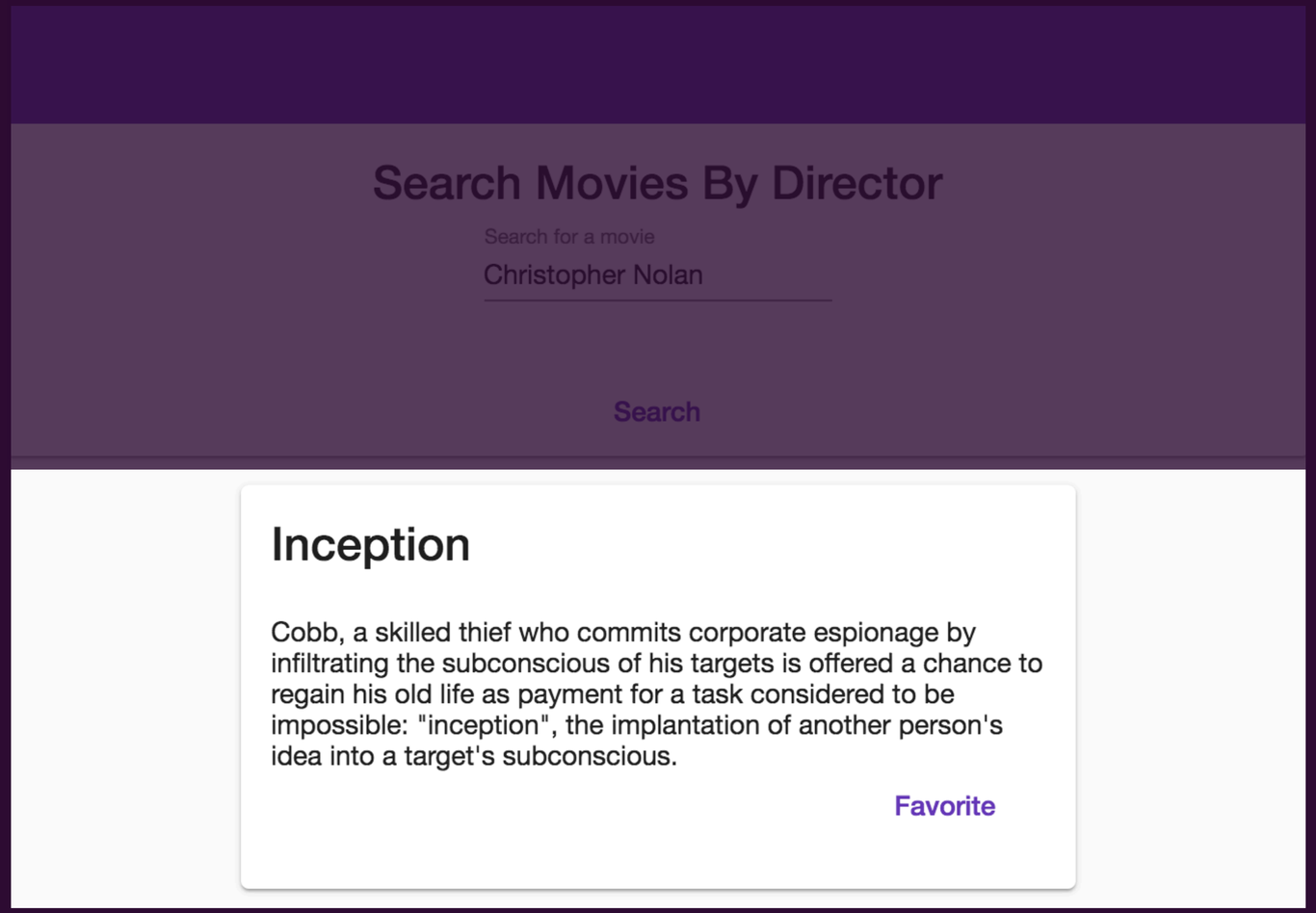
Favorite

```
<movies-list-item/>
```



```
<movies-list-item/>
```

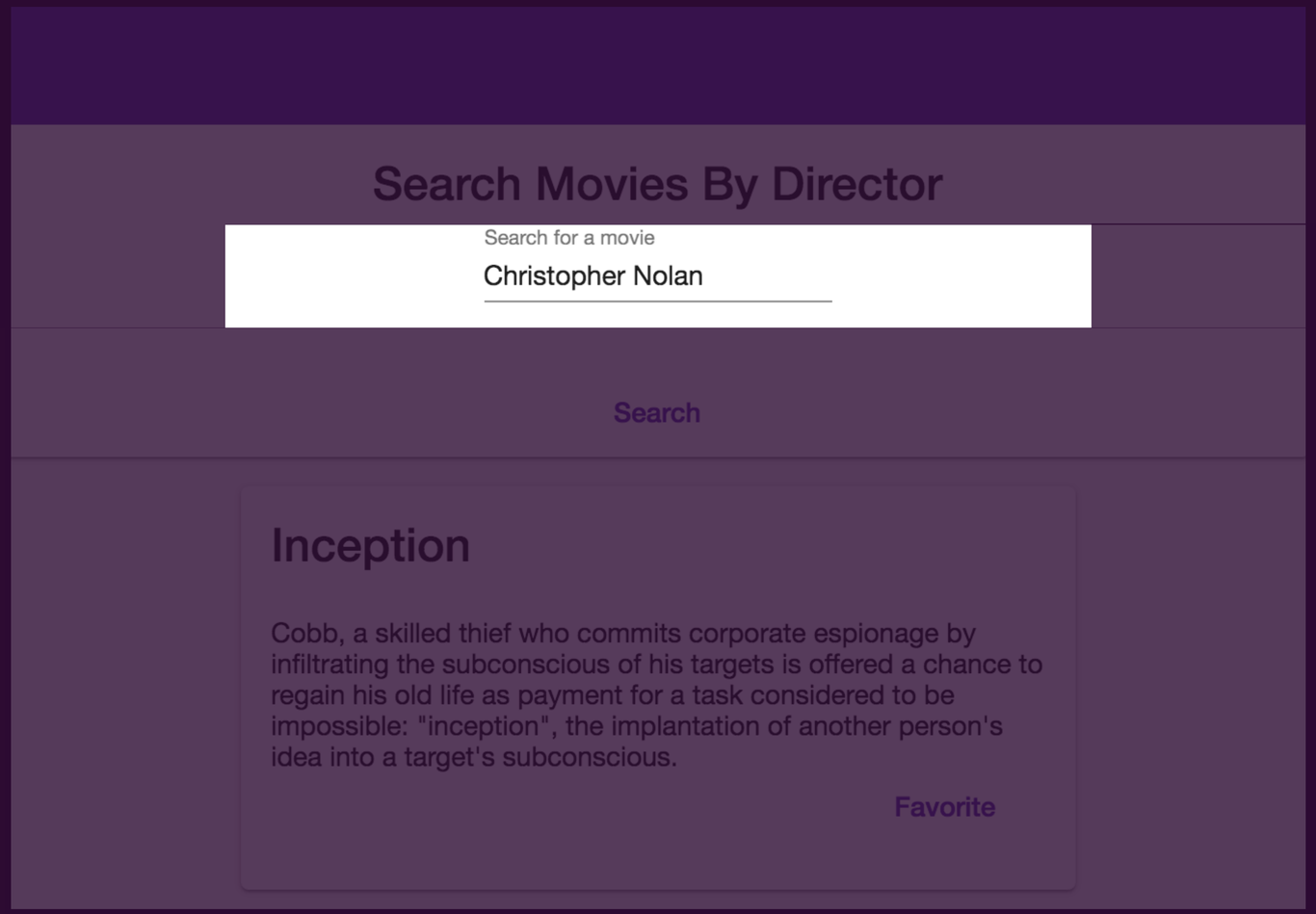
```
<movies-list/>
```



<movies-list-item/>

<movies-list/>

<search-movies-box/>

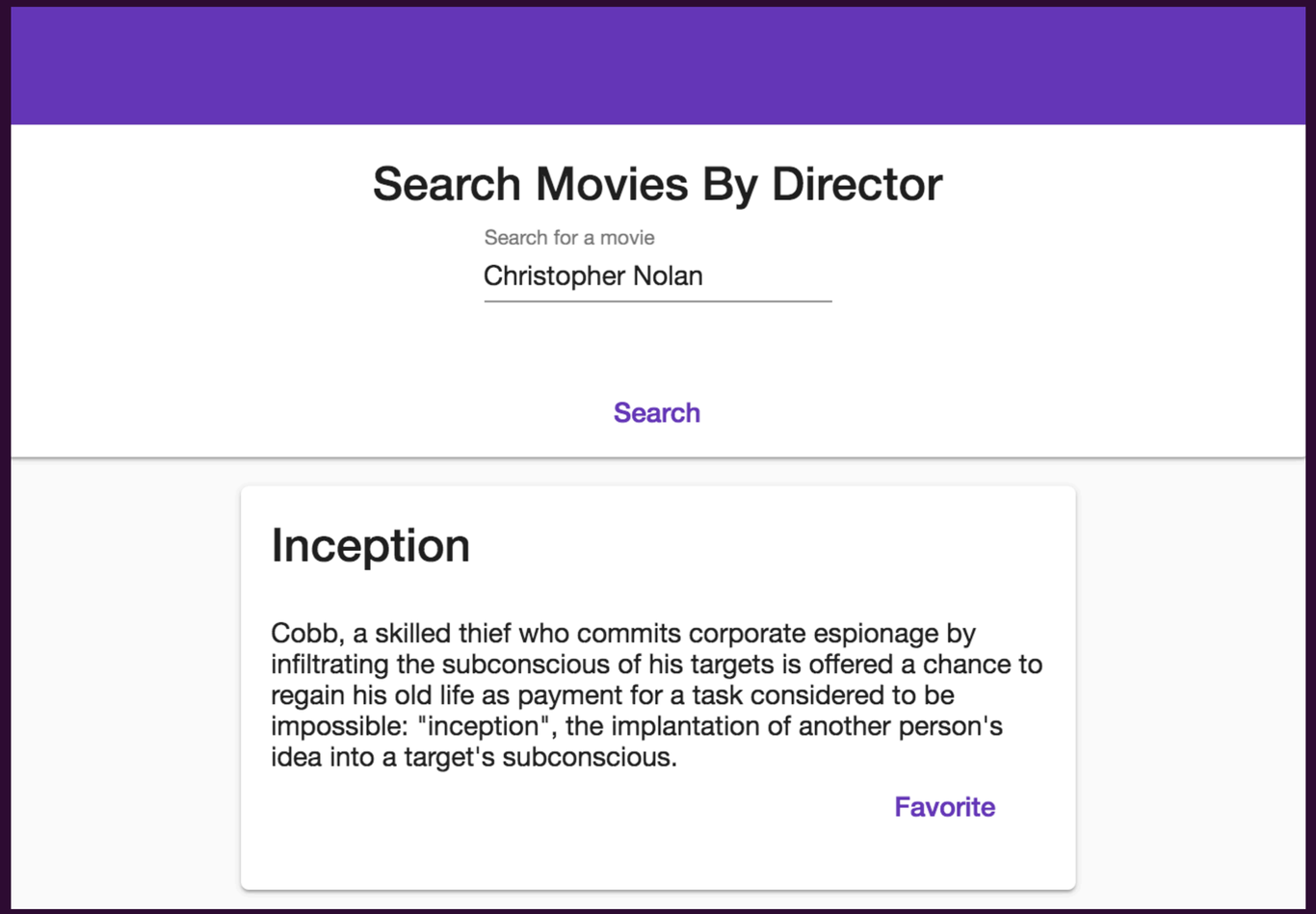


<movies-list-item/>

<movies-list/>

<search-movies-box/>

<search-movies-page/>

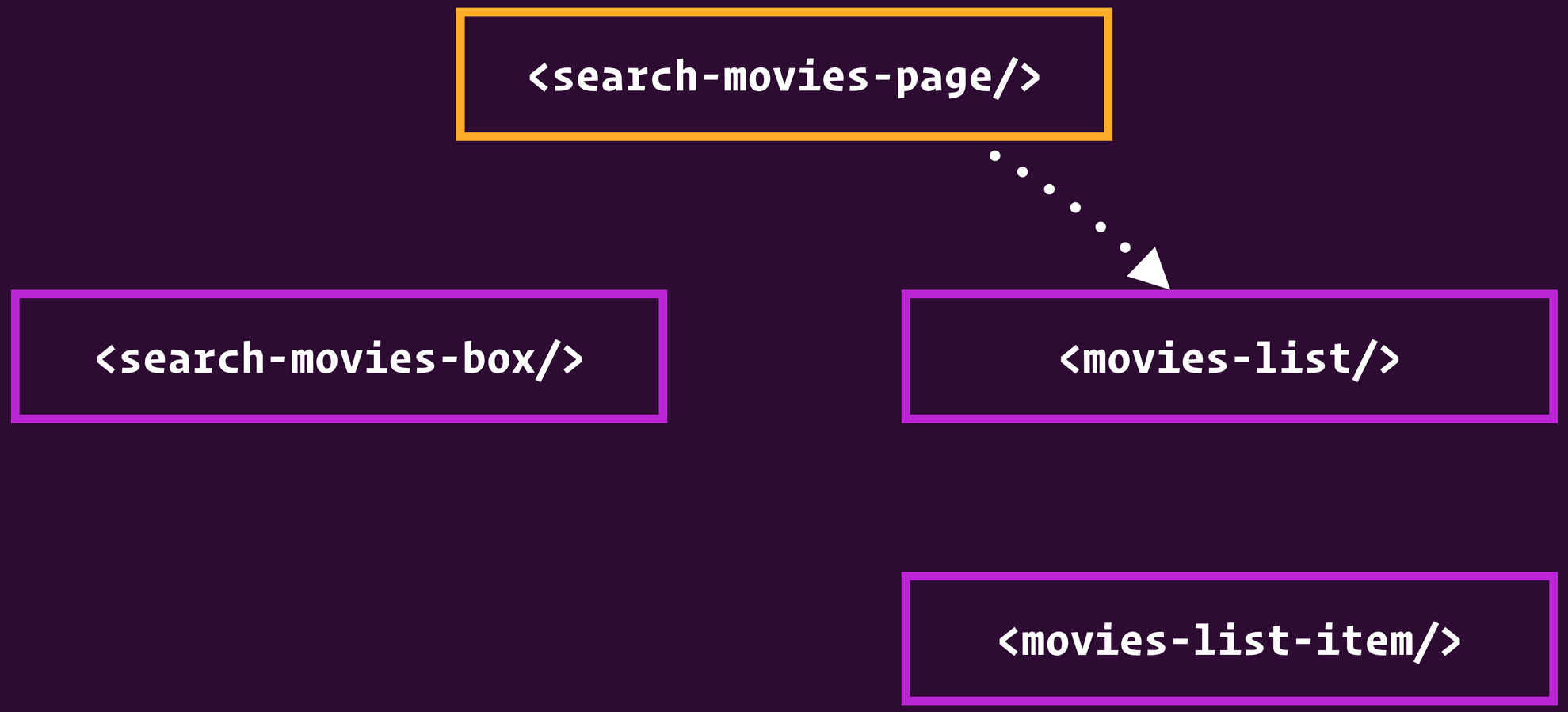


<search-movies-page/>

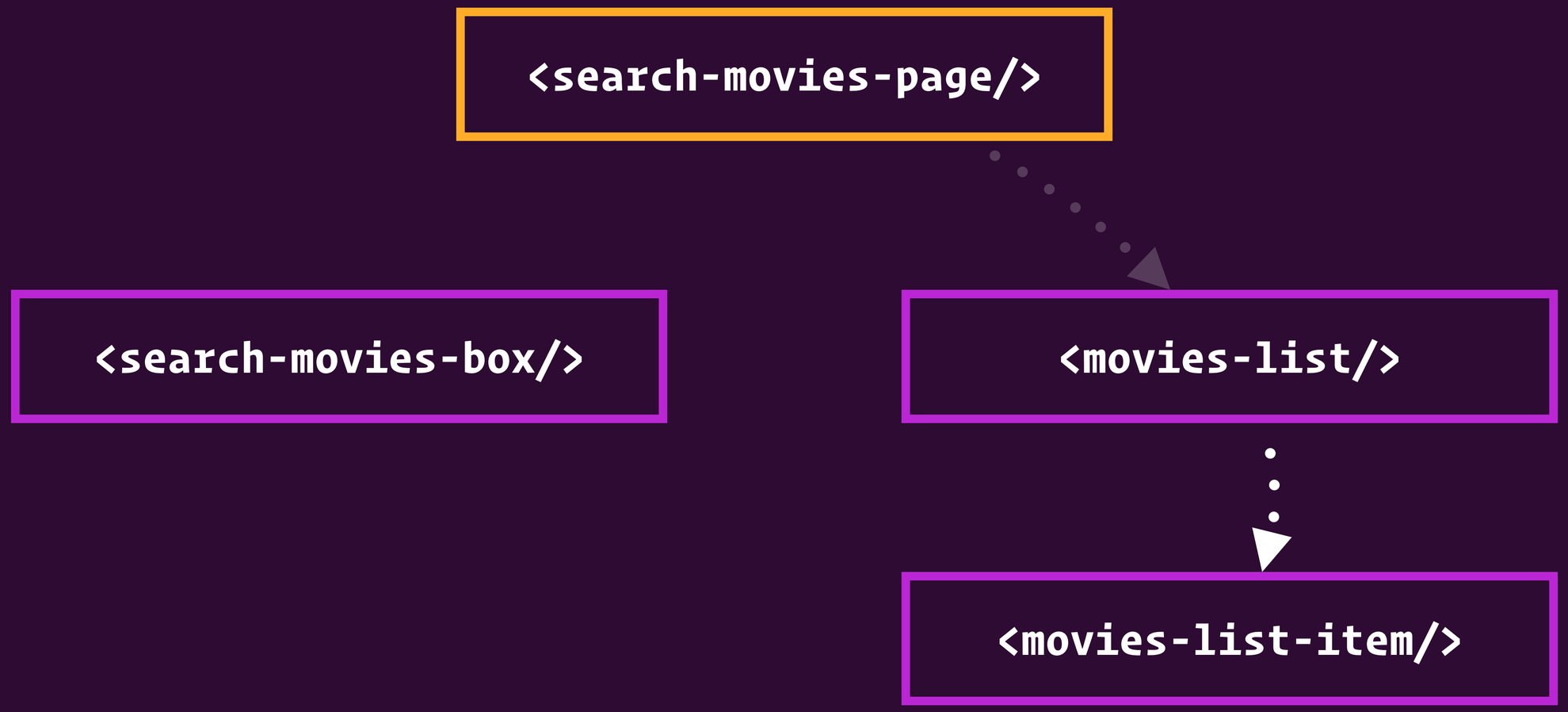
<search-movies-box/>

<movies-list/>

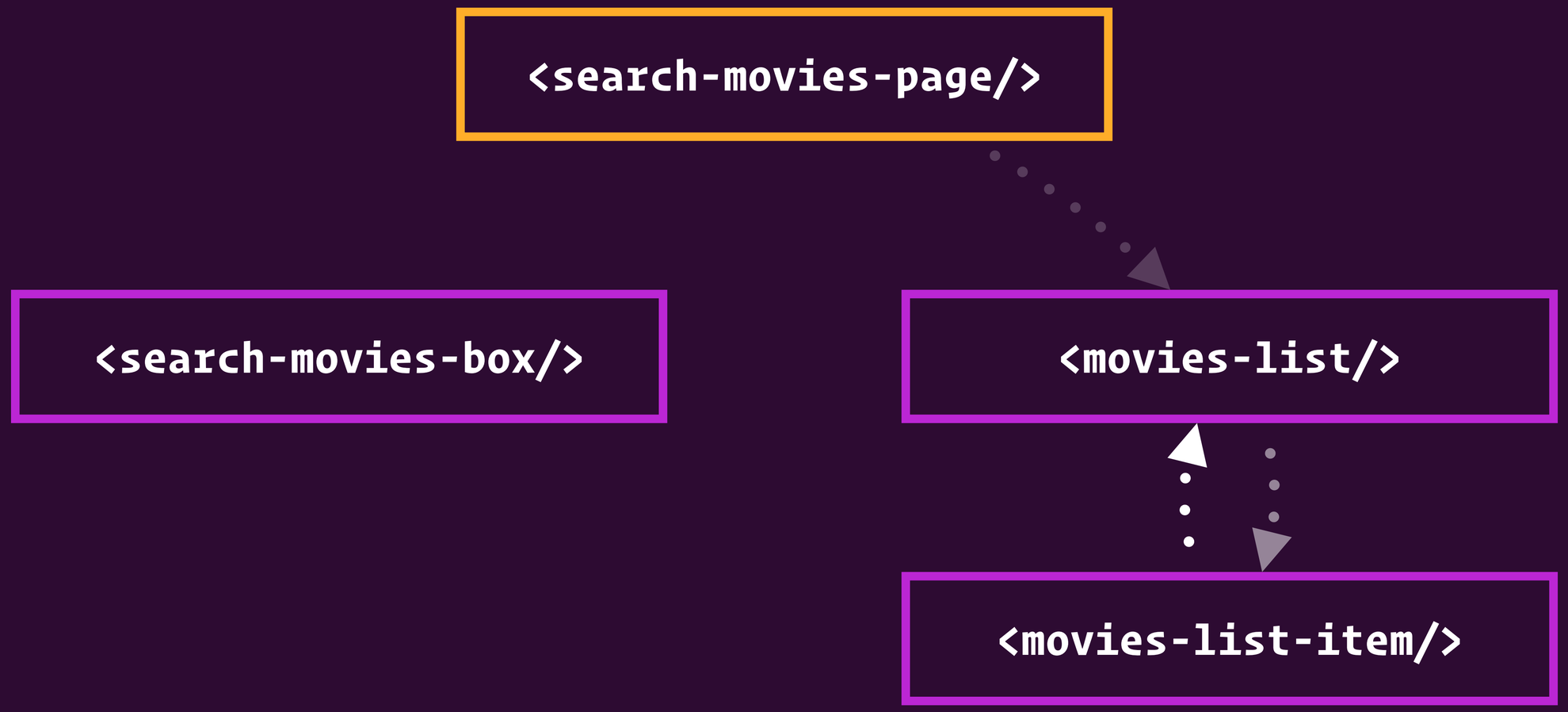
<movies-list-item/>



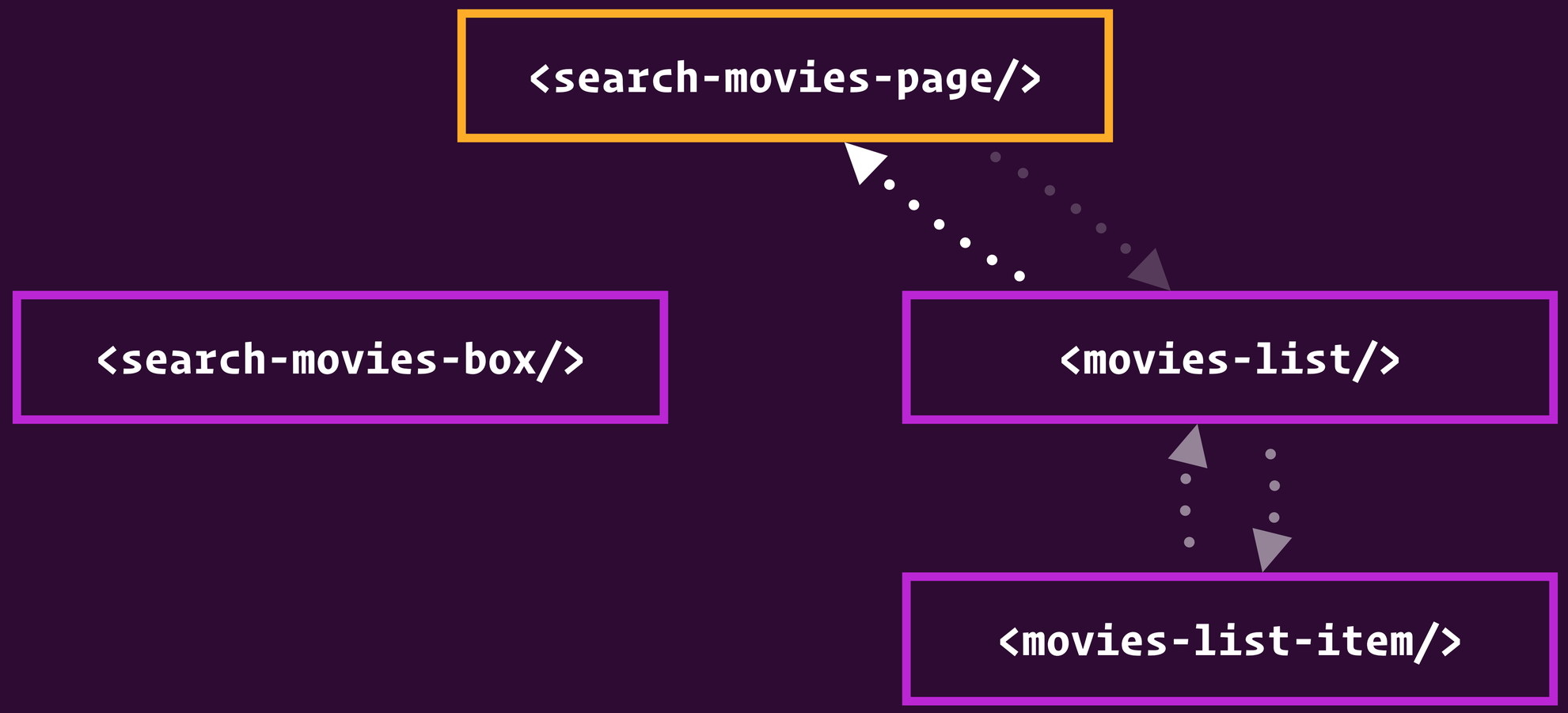

```
@Input() movies: Movie[]
```



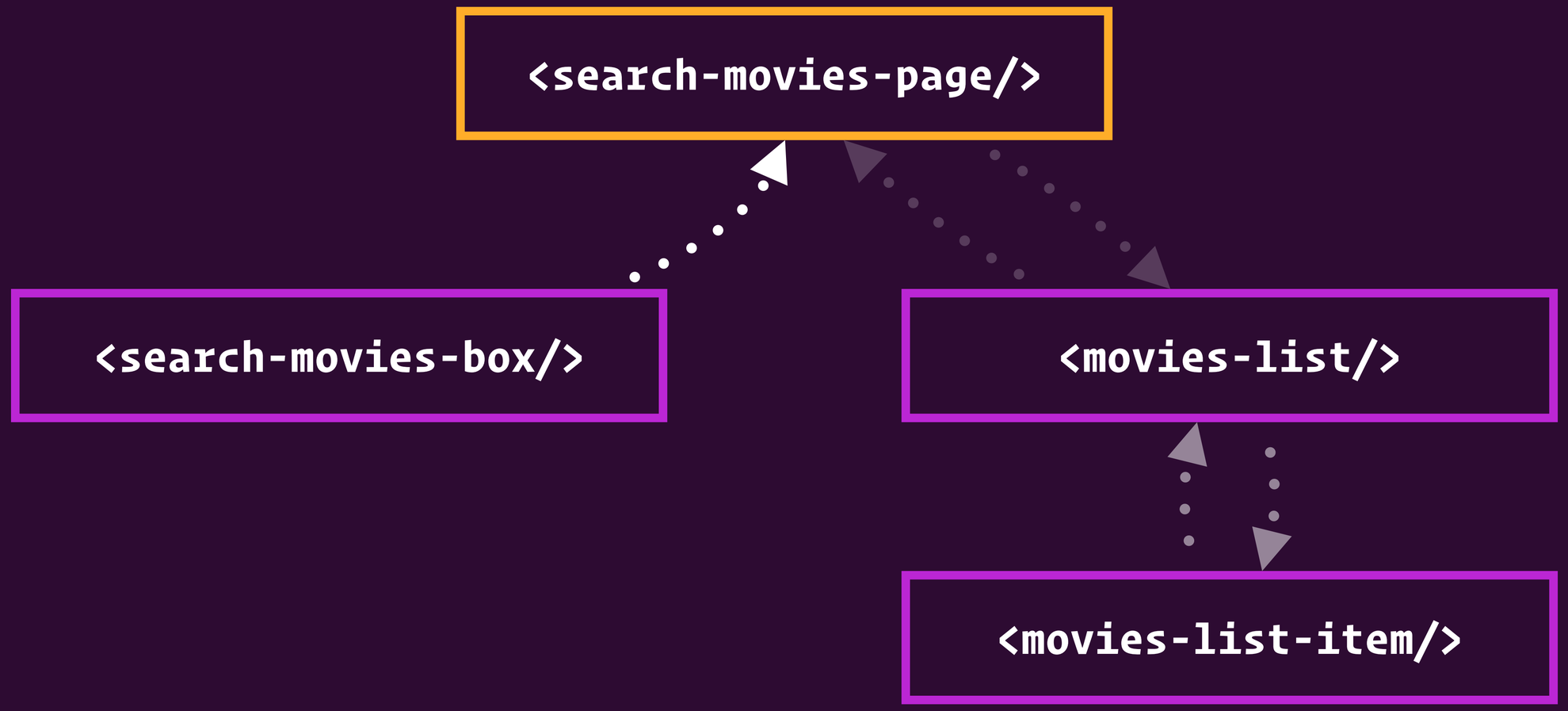

```
@Input() movie: Movie
```




```
@Output() favorite: EventEmitter<Movie>
```




```
@Output() favoriteMovie: EventEmitter<Movie>
```




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

```
@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;
      });
  }
}
```

```
@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;
      });
  }
}
```

```
@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;
      });
  }
}
```

```
@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/>

```
<search-movies-page/>
```

- ☑ Connects data to components

```
<search-movies-page/>
```

- ☑ Connects data to components
- ☑ Triggers side effects

```
<search-movies-page/>
```

- ☑ Connects data to components
- ☑ Triggers side effects
- ☑ Handles state transitions

OUTSIDE WORLD

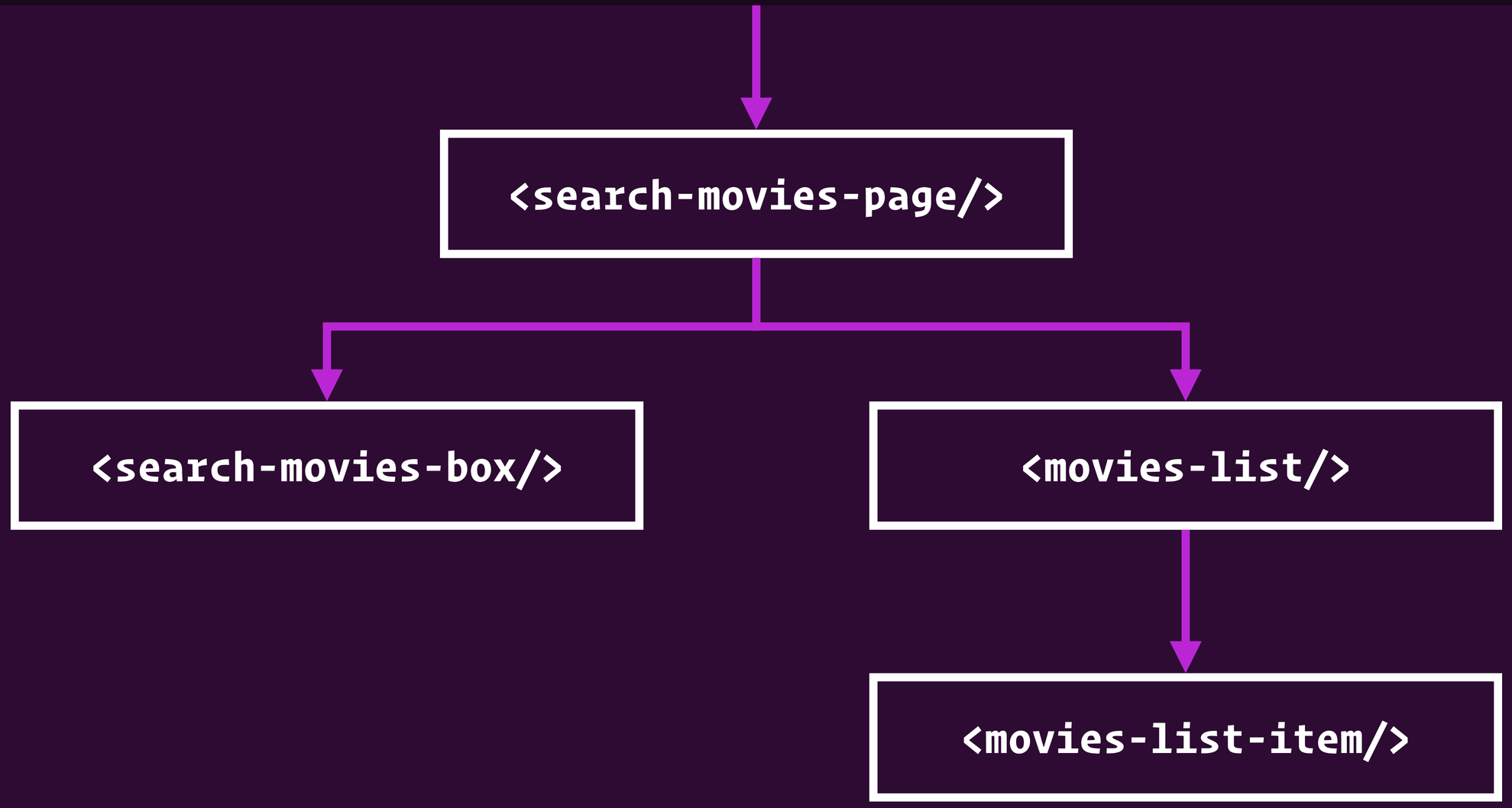
<search-movies-page/>

<search-movies-box/>

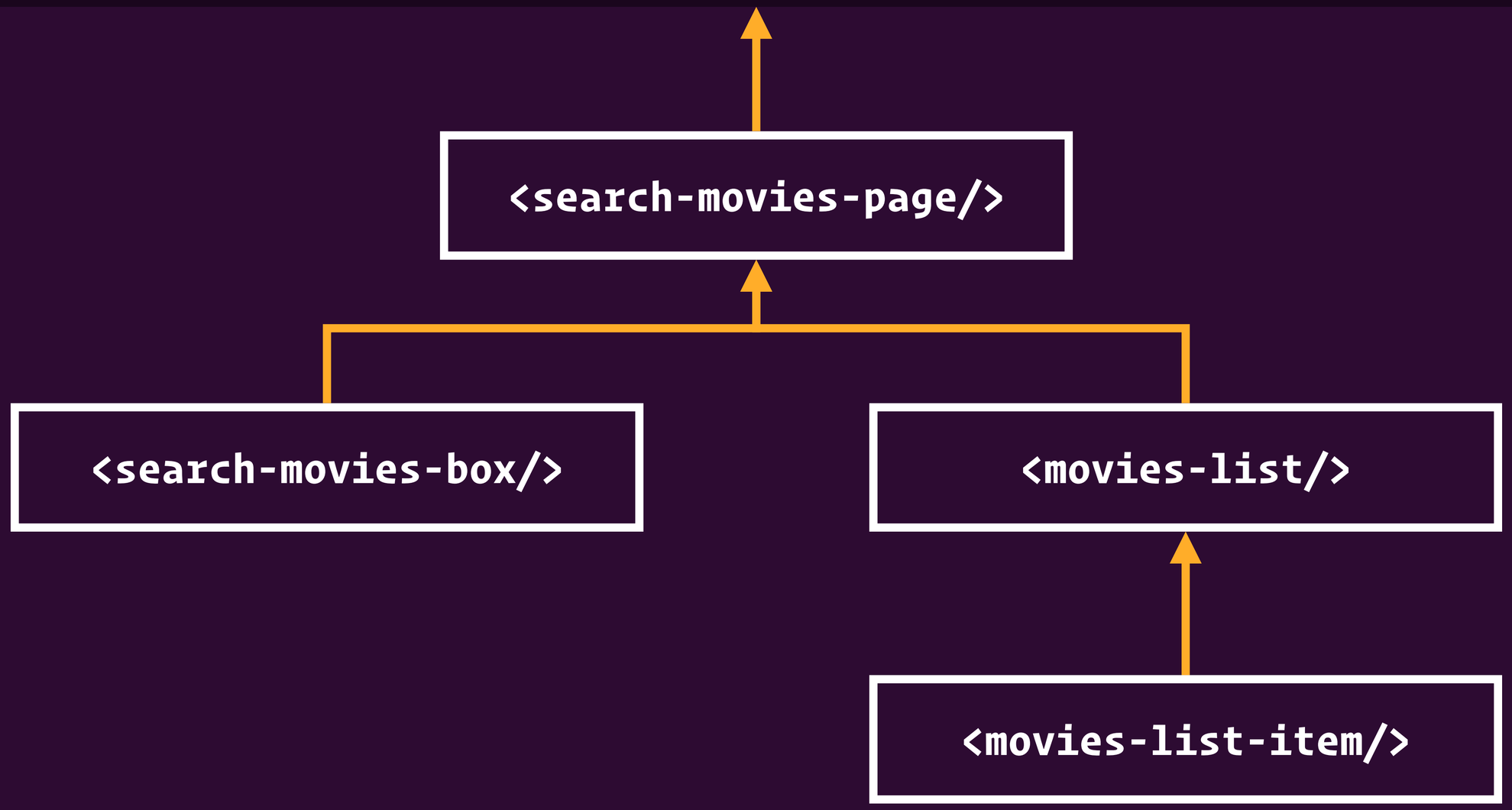
<movies-list/>

<movies-list-item/>

OUTSIDE WORLD



OUTSIDE WORLD



NGRX MENTAL MODEL

State flows down, changes flow up

```
<search-movies-page/>
```

- ☑ Connects data to components
- ☑ Triggers side effects
- ☑ Handles state transitions

Single Responsibility Principle

<search-movies-page/>

```
<search-movies-page/>
```

- ☑ Connects data to components

@Input() and @Output()

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

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>();
}
```

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

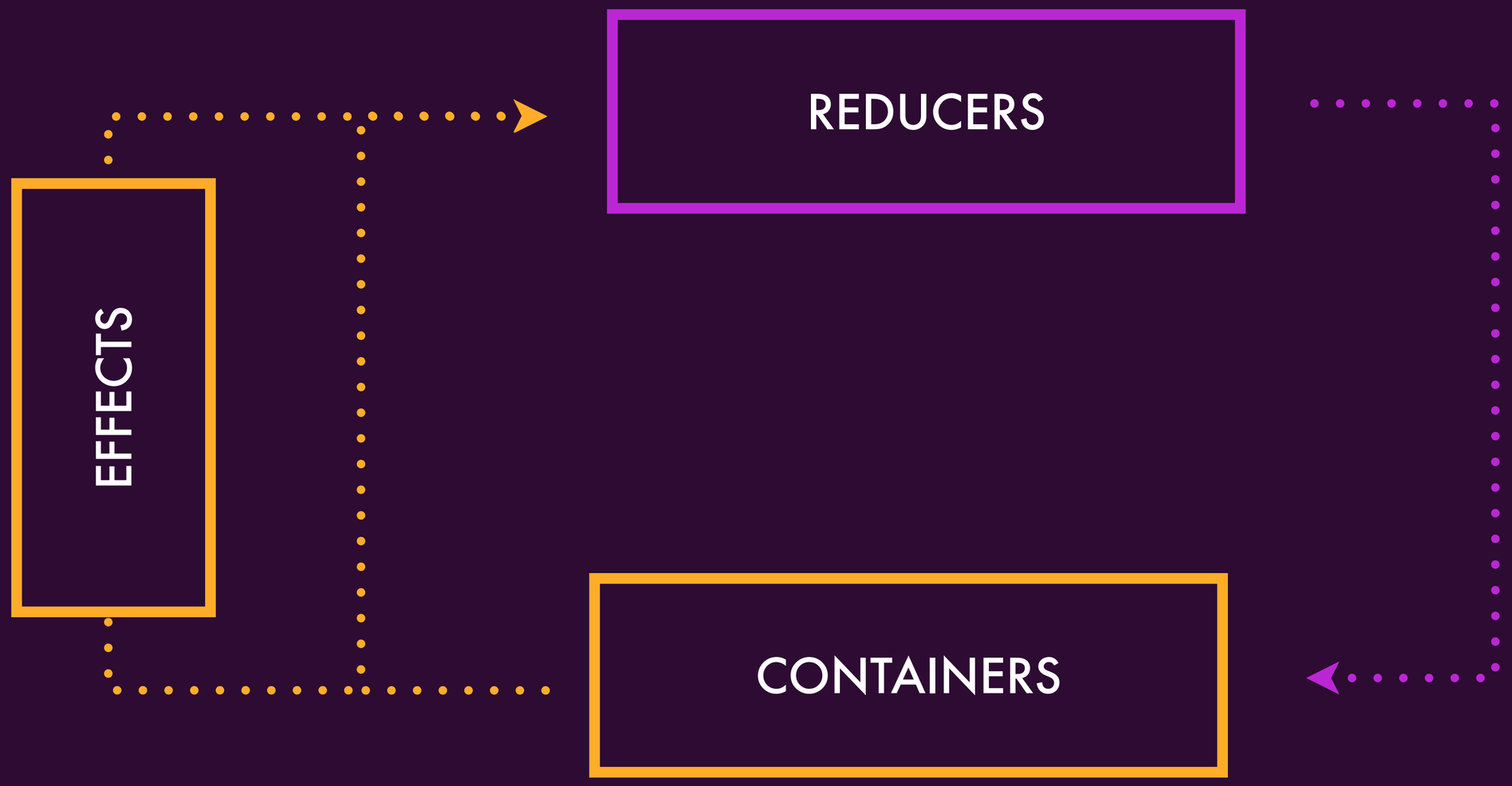
Does this component know who is listening to its output?

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

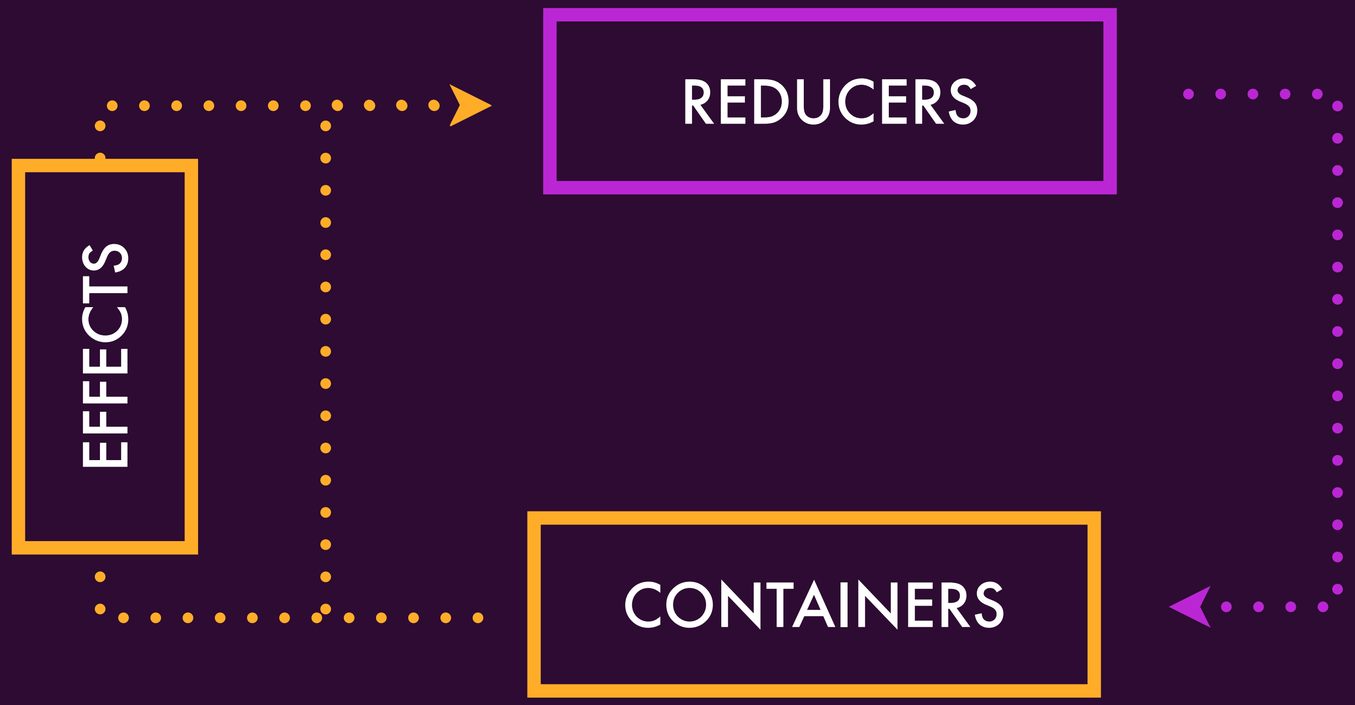
Inputs & Outputs offer Indirection

NGRX MENTAL MODEL

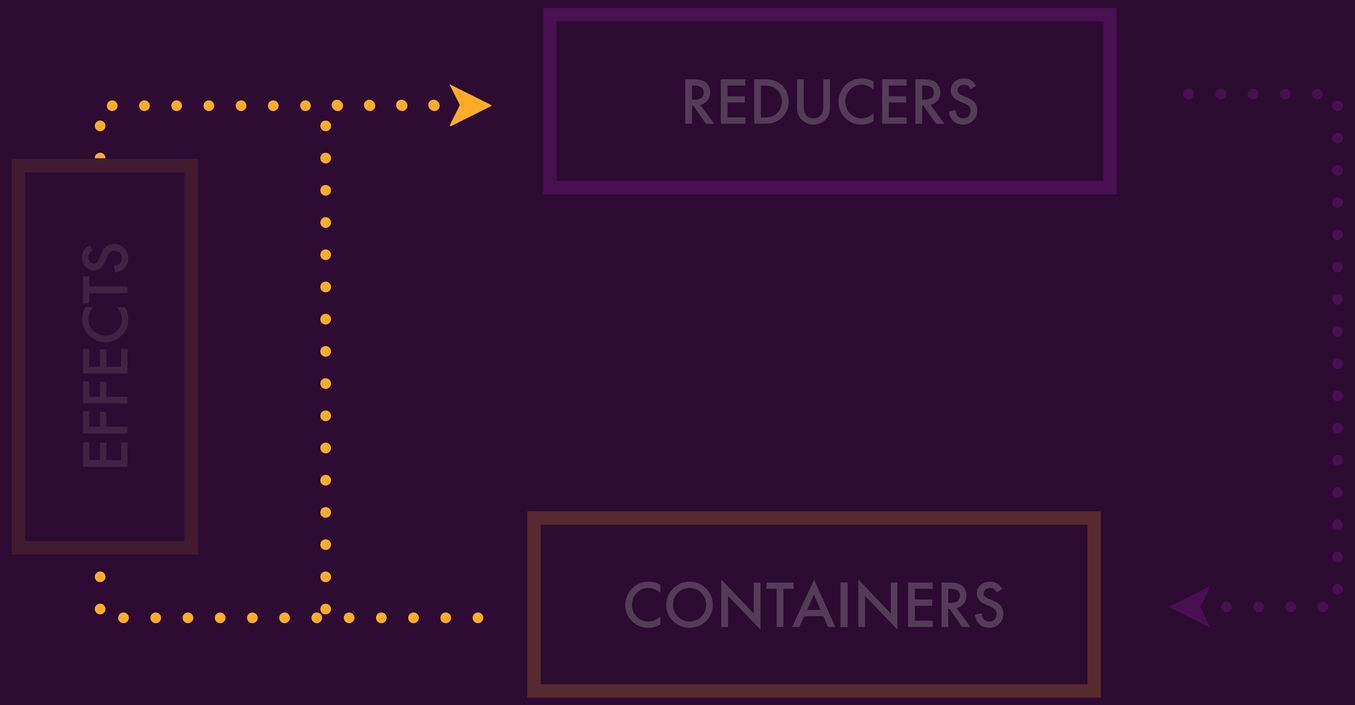
There is indirection between consumer of state, how state changes, and side effects


REDUCERS
EFFECTS
CONTAINERS

ACTIONS



ACTIONS

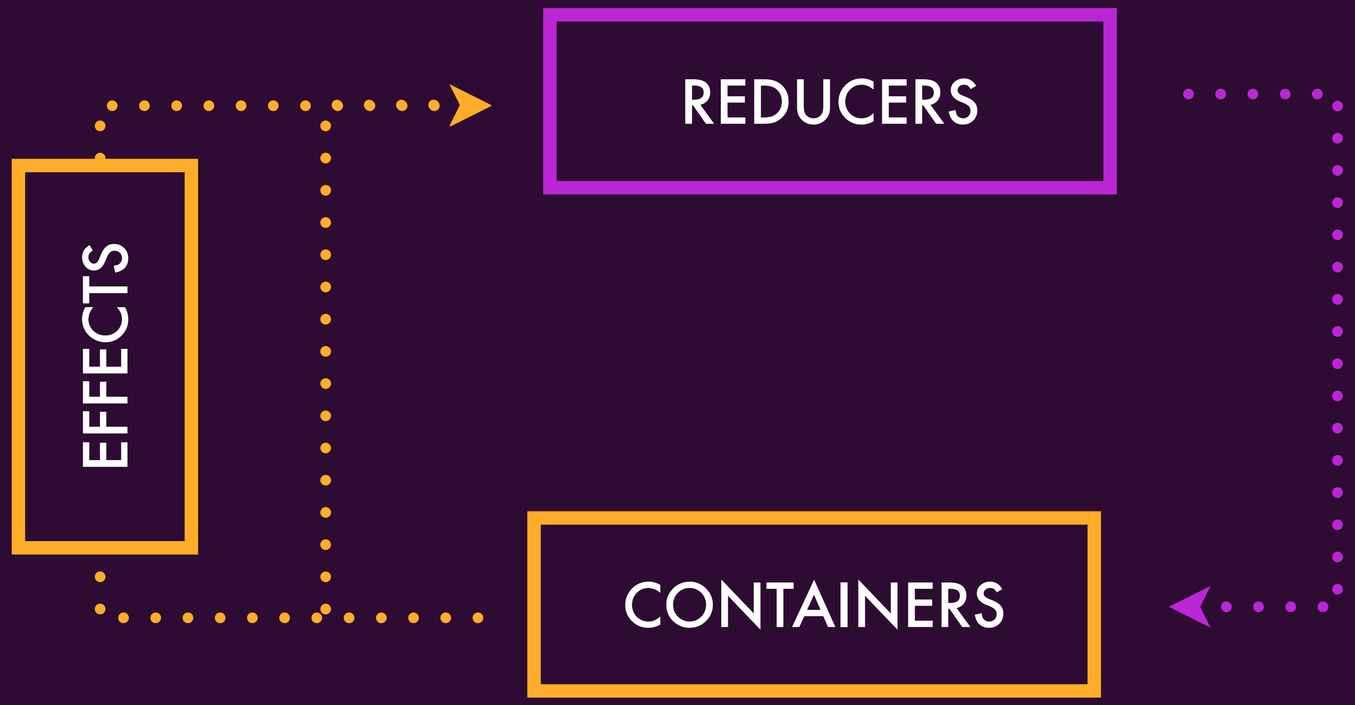


```typescript
interface Action {
  type: string;
}
```

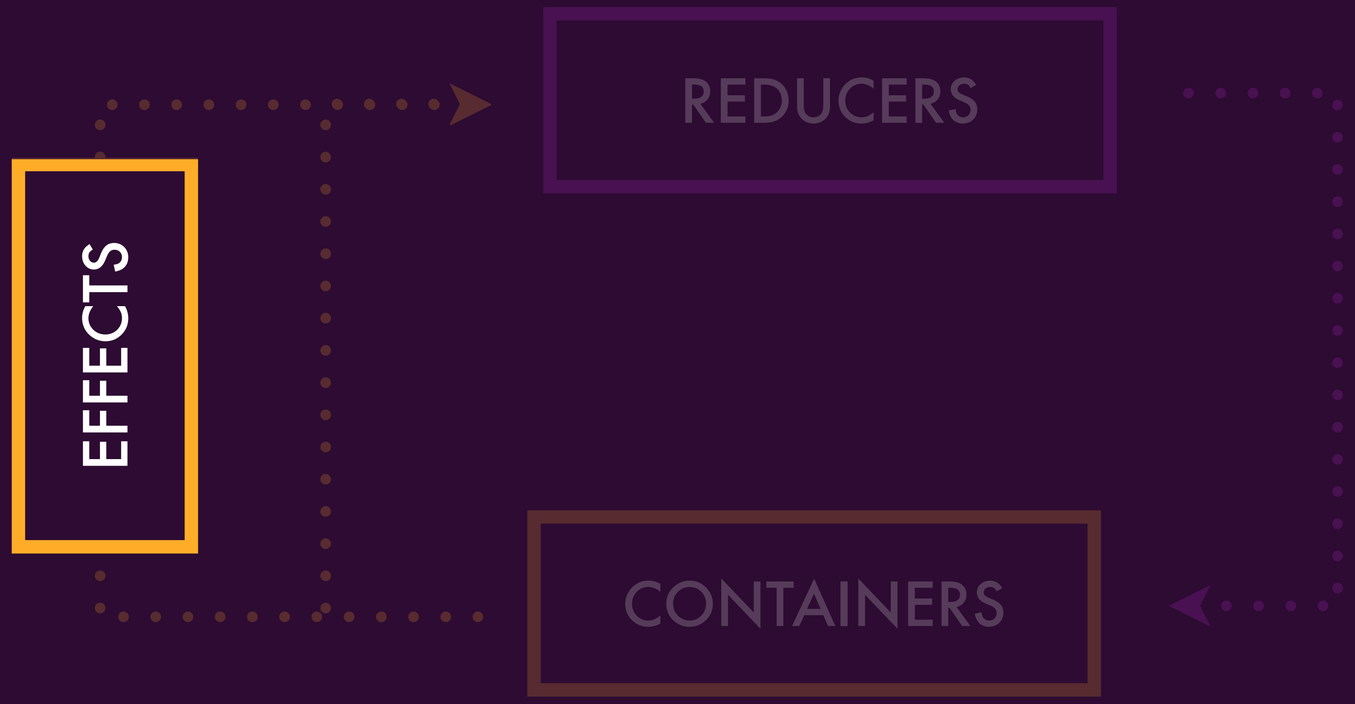
```
this.store.dispatch({
  type: 'MOVIES_LOADED_SUCCESS',
  movies: [{
    id: 1,
    title: 'Enemy',
    director: 'Denis Villeneuve',
  }],
});
```

Global `@Output()` for your whole app

EFFECTS



EFFECTS



```
createEffect(() =>
  this.actions$.pipe(
    ofType("SEARCH_MOVIES"),
    switchMap(action =>
      this.moviesService
        .findMovies(action.searchTerm)
        .pipe(
          map(movies => moviesLoadedSuccessAction(movies))
        )
    )
  )
);
```

```
createEffect(() =>
  this.actions$.pipe(
    ofType("SEARCH_MOVIES"),
    switchMap(action =>
      this.moviesService
        .findMovies(action.searchTerm)
        .pipe(
          map(movies => moviesLoadedSuccessAction(movies))
        )
    )
  )
);
```

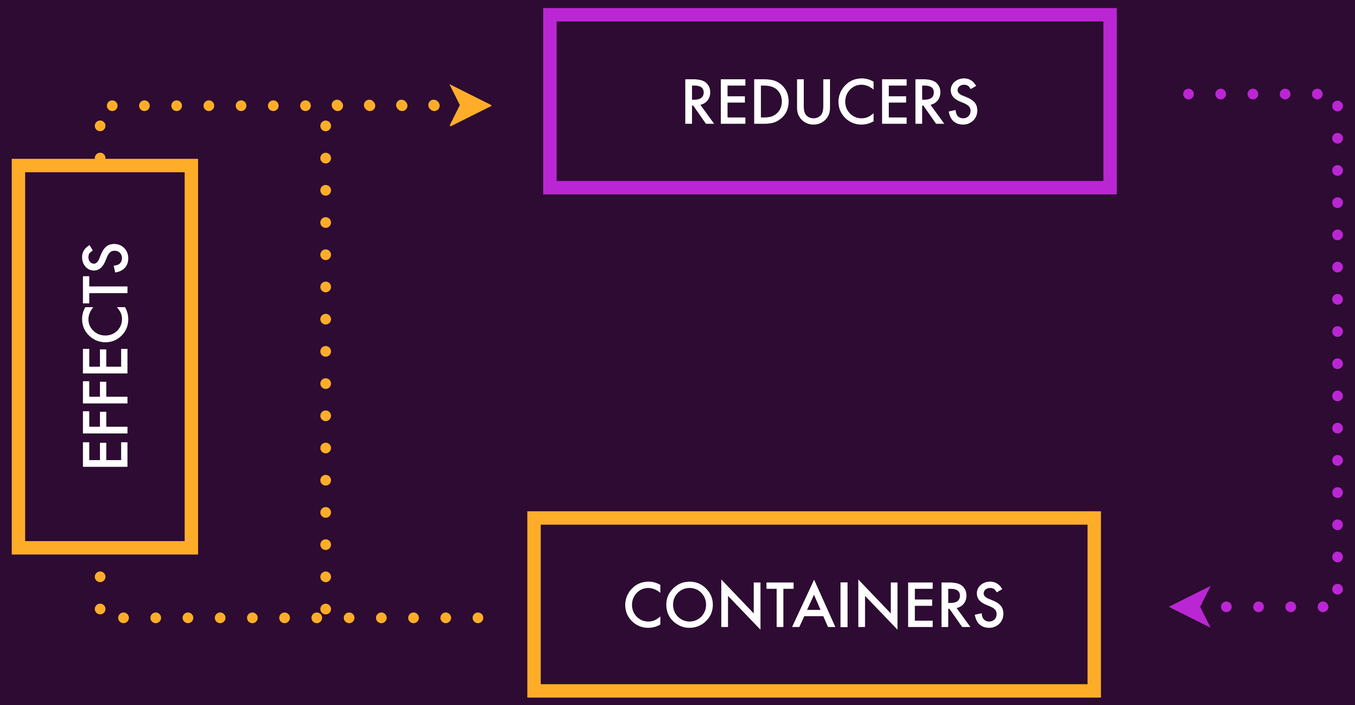
```
createEffect(() =>
  this.actions$.pipe(
    ofType("SEARCH_MOVIES"),
    switchMap(action =>
      this.moviesService
        .findMovies(action.searchTerm)
        .pipe(
          map(movies => moviesLoadedSuccessAction(movies))
        )
    )
  )
);
```

```
createEffect(() =>
  this.actions$.pipe(
    ofType("SEARCH_MOVIES"),
    switchMap(action =>
      this.moviesService
        .findMovies(action.searchTerm)
        .pipe(
          map(movies => moviesLoadedSuccessAction(movies))
        )
    )
  )
);
```

```
createEffect(() =>
  this.actions$.pipe(
    ofType("SEARCH_MOVIES"),
    switchMap(action =>
      this.moviesService
        .findMovies(action.searchTerm)
        .pipe(
          map(movies => moviesLoadedSuccessAction(movies))
        )
    )
  )
);
```

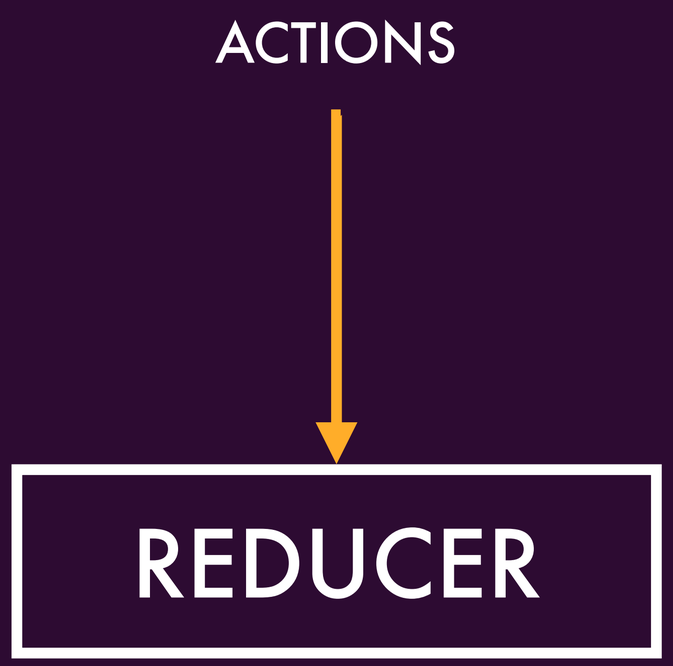
```
createEffect(() =>
  this.actions$.pipe(
    ofType("SEARCH_MOVIES"),
    switchMap(action =>
      this.moviesService
        .findMovies(action.searchTerm)
        .pipe(
          map(movies => moviesLoadedSuccessAction(movies))
        )
    )
  )
);
```

REDUCERS

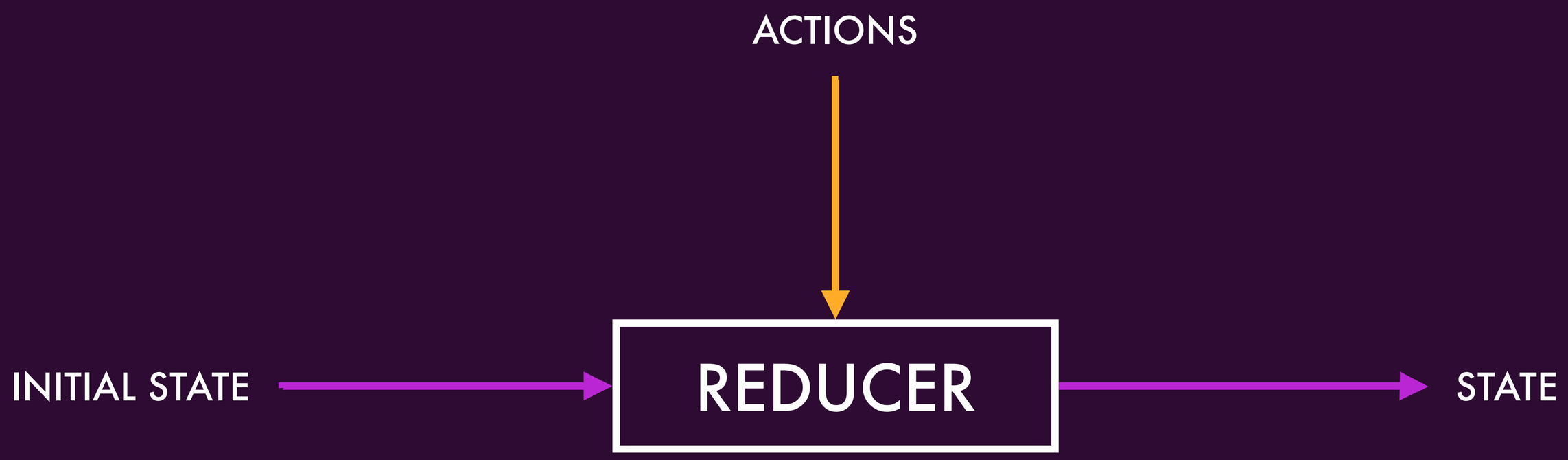


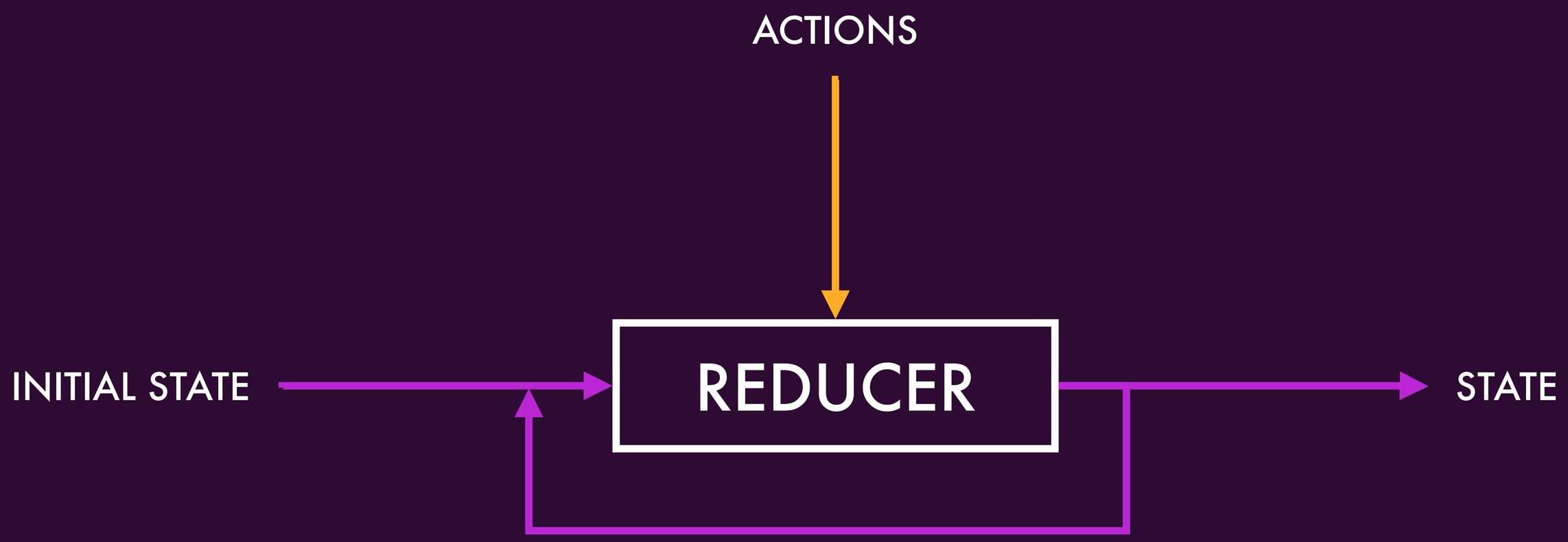
REDUCERS

REDUCER


ACTIONS
REDUCER

ACTIONS
INITIAL STATE
REDUCER


ACTIONS
INITIAL STATE
REDUCER
STATE


ACTIONS
INITIAL STATE
REDUCER
STATE

```
const initialState: Movie[] = [];

createReducer(
  initialState,
  on(moviesLoadedSuccessAction, (state, action) => {
    return action.movies;
  })
);
```

```
const initialState: Movie[] = [];

createReducer(
  initialState,
  on(moviesLoadedSuccessAction, (state, action) => {
    return action.movies;
  })
);
```

```
const initialState: Movie[] = [];

createReducer(
  initialState,
  on(moviesLoadedSuccessAction, (state, action) => {
    return action.movies;
  })
);
```

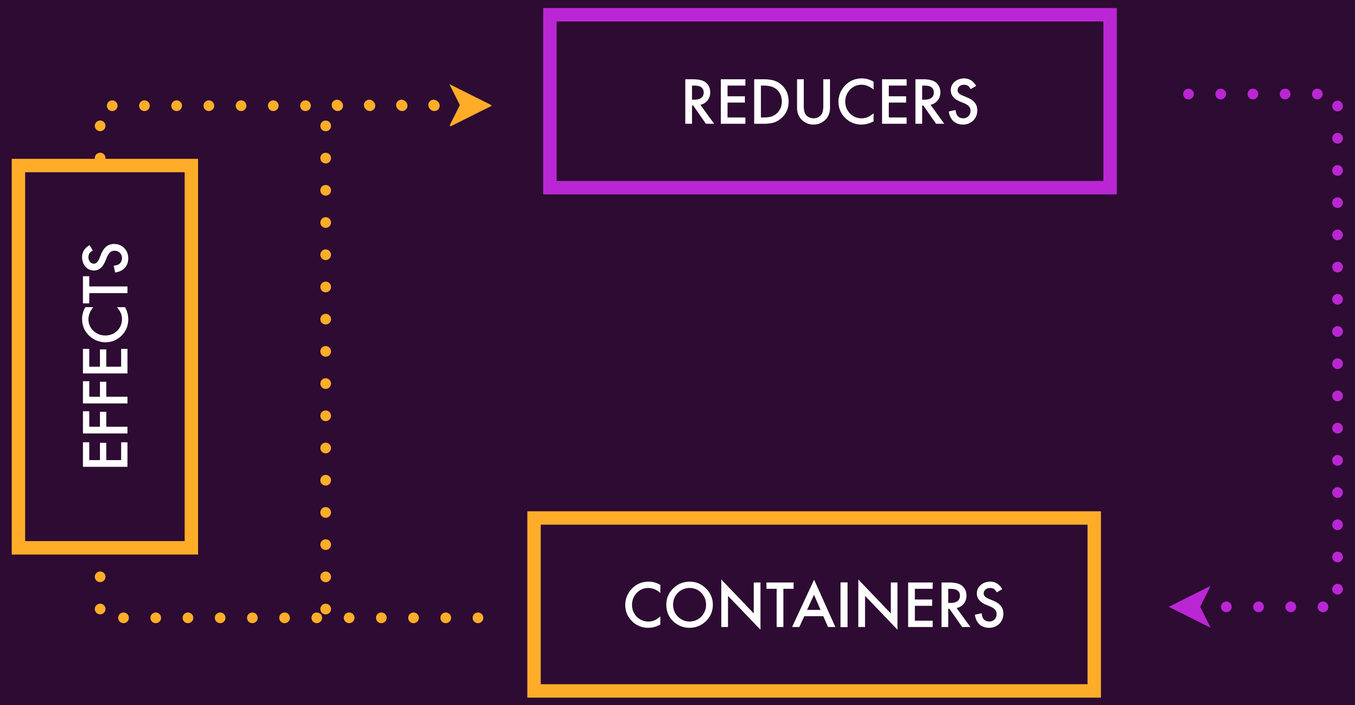
```
const initialState: Movie[] = [];

createReducer(
  initialState,
  on(moviesLoadedSuccessAction, (state, action) => {
    return action.movies;
  })
);
```

```
const initialState: Movie[] = [];

createReducer(
  initialState,
  on(moviesLoadedSuccessAction, (state, action) => {
    return action.movies;
  })
);
```

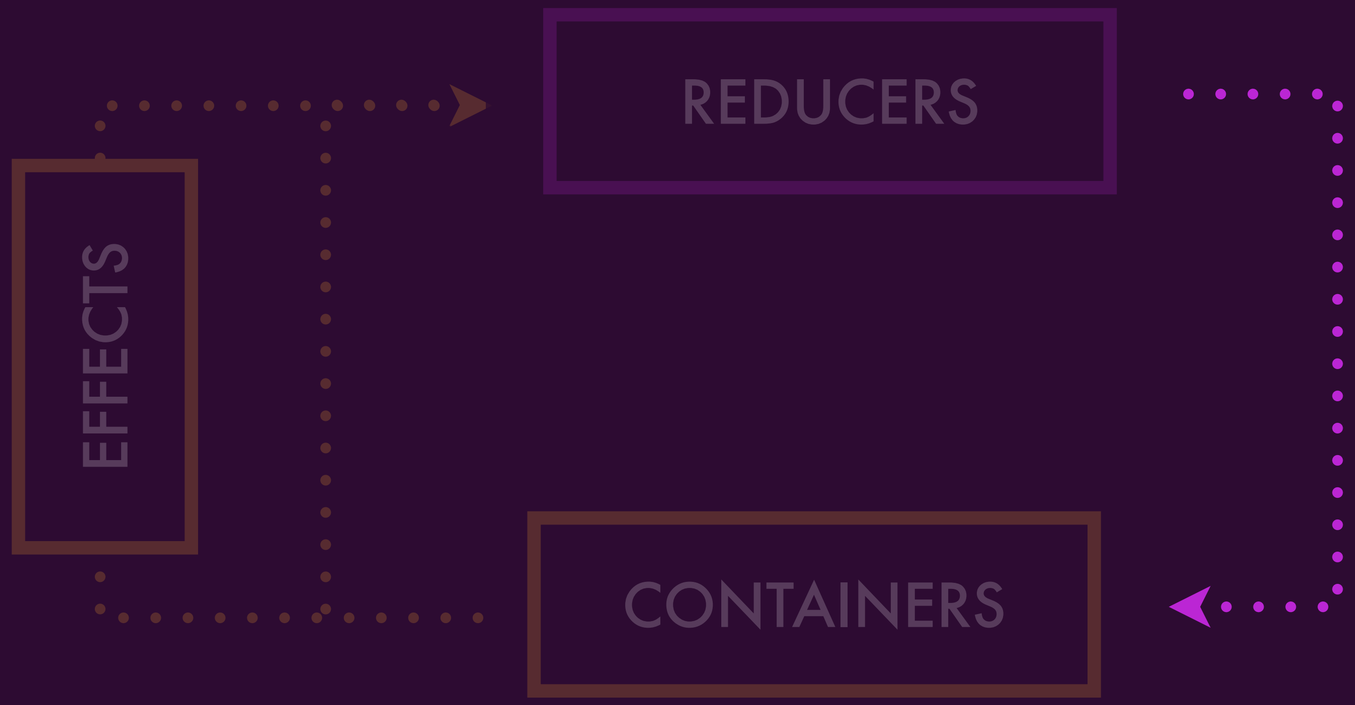
```
const initialState: Movie[] = [];

createReducer(
  initialState,
  on(moviesLoadedSuccessAction, (state, action) => {
    return action.movies;
  })
);
```

```
const initialState: Movie[] = [];

createReducer(
  initialState,
  on(moviesLoadedSuccessAction, (state, action) => {
    return action.movies;
  })
);
```

SELECTORS



SELECTORS

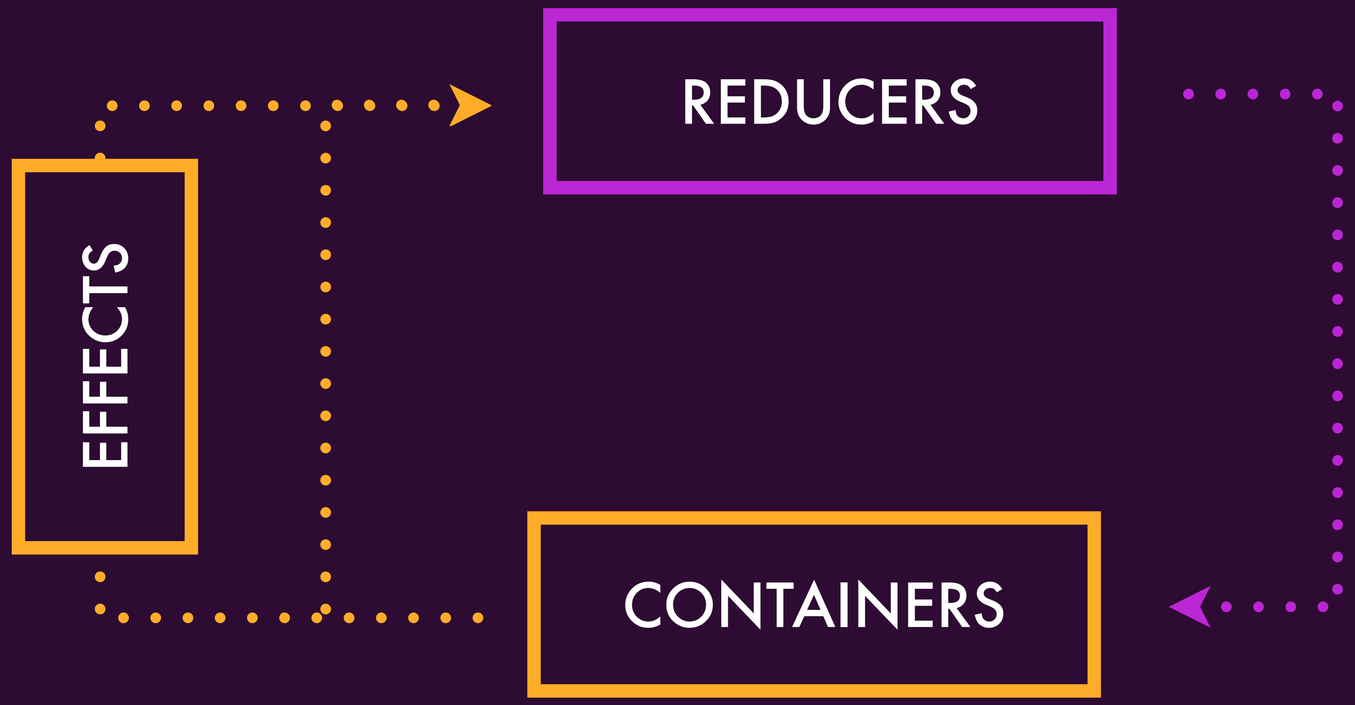


STORE
???
COMPONENTS

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

Global `@Input()` for your whole app

CONTAINERS



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(searchMoviesAction(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(searchMoviesAction(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(searchMoviesAction(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(searchMoviesAction(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(searchMoviesAction(searchTerm));
  }
}
```

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

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

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

```
this.store.dispatch(searchMoviesAction(searchTerm));
```

NGRX MENTAL MODEL

Select and Dispatch are special versions of Input and Output

RESPONSIBILITIES

RESPONSIBILITIES

- ☑ Containers connect data to components

RESPONSIBILITIES

- ☑ Containers connect data to components
- ☑ Effects triggers side effects

RESPONSIBILITIES

- ☑ Containers connect data to components
- ☑ Effects triggers side effects
- ☑ Reducers handle state transitions

NGRX MENTAL MODEL

Delegate responsibilities to individual modules of code

- [x] State flows down, changes flow up

- ☑ State flows down, changes flow up
- ☑ Indirection between state & consumer

- ✓ State flows down, changes flow up
- ✓ Indirection between state & consumer
- ✓ Select & Dispatch => Input & Output

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

github.com/**LiveLoveApp/mosaic-ngrx-workshop-2021**

Demo

Challenge

1. Clone the repo at **github.com/liveloveapp/mosaic-ngrx-workshop-2021**
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

ACTIONS

- [x] Unified interface to describe events

ACTIONS

- ☑ Unified interface to describe events
- ☑ Just data, no functionality

ACTIONS

- ☑ Unified interface to describe events
- ☑ Just data, no functionality
- ☑ Has at a minimum a type property

```
{
  type: "[Books Page] Select Book”;
  book: BookModel;
}
```

```
class BooksComponent {
  createBook(book) {
    this.store.dispatch({
      type: "[Books Page] Create Book",
      book
    });
  }
}
```

GOOD ACTION HYGIENE

GOOD ACTION HYGIENE

- ☑ Unique events get unique actions

GOOD ACTION HYGIENE

- ☑ Unique events get unique actions
- ☑ Actions are grouped by their source

GOOD ACTION HYGIENE

- ☑ Unique events get unique actions
- ☑ Actions are grouped by their source
- ☑ Actions are never reused

"[Books Page] Select Book”

"[Books Page] Add Book"

"[Books Page] Update Book"

"[Books Page] Delete Book”

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

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

```
export const createBook = createAction(
  '[Books Page] Create Book',
  props<{ book: BookRequiredProps }>()
);
```

```
store.dispatch(createBook({
  book: bookRequiredProps
}));
```

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

export const createBook = createAction(
  "[Books Page] Create Book",
  props<{ book: BookRequiredProps }>()
);

export const selectBook = createAction(
  "[Books Page] Select Book",
  props<{ book: BookModel }>()
);
```

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

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

  ngOnInit() {
    this.store.dispatch(BooksPageActions.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

Keep actions close to where they are dispatched

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.component.ts** to inject the Store and dispatch the actions

Everything is a stream

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

"[Books API] Get Books Success"

"[Books API] Get Books Failure"

Demo

Challenge

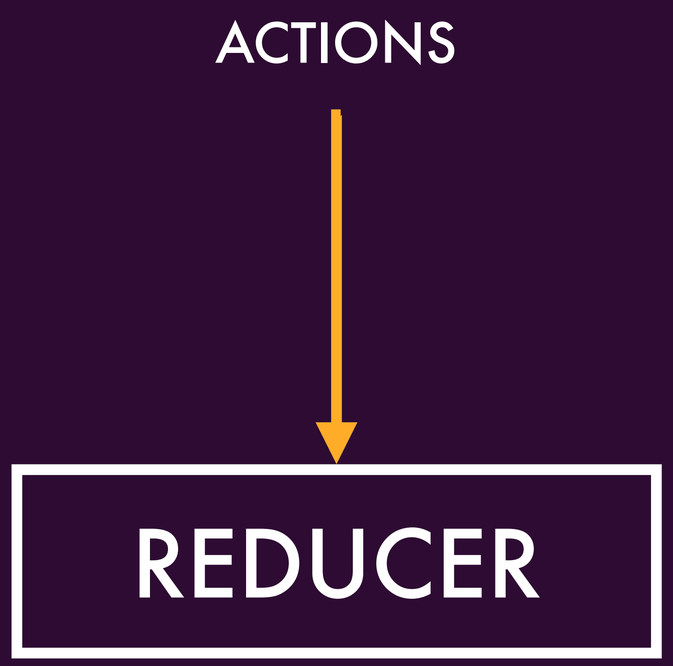
1. Open **books-api.actions.ts**
2. Create **strongly typed actions** that adhere to **good action hygiene** for getting all of the books, updating a book, creating a book, and deleting a book. Don’t worry about failure actions for this exercise.
3. Update **books-page.component.ts** to dispatch the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

REDUCERS

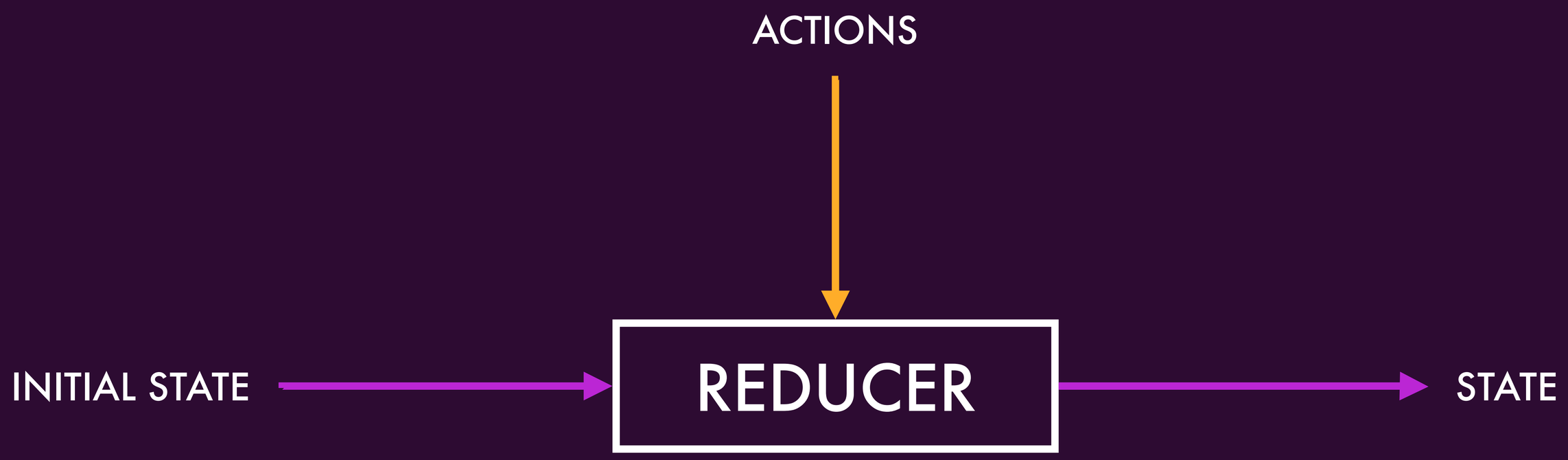
REDUCERS

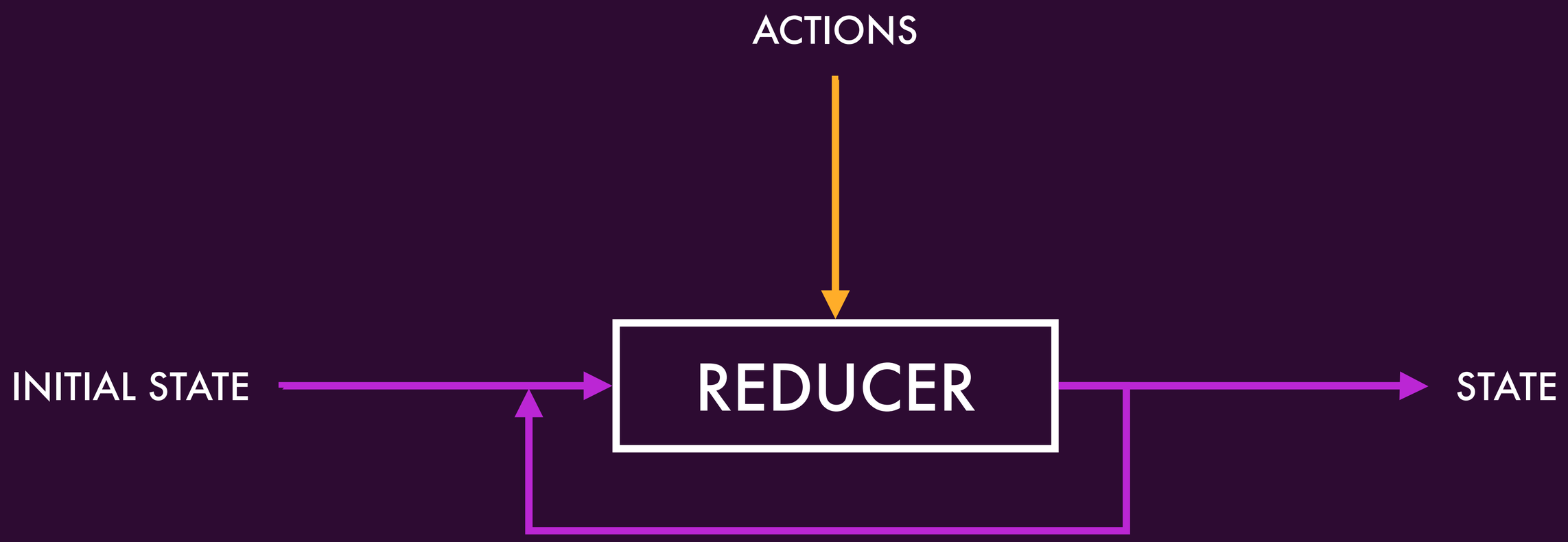
- ☑ Produce new states
- ☑ Receive the last state and next action
- ☑ Listen to specific actions
- ☑ Use pure, immutable operations

REDUCER


ACTIONS
REDUCER

ACTIONS
INITIAL STATE
REDUCER


ACTIONS
INITIAL STATE
REDUCER
STATE


ACTIONS
INITIAL STATE
REDUCER
STATE

```typescript
export interface State {
  collection: BookModel[];
  activeBookId: string | null;
}
```

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

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeBookId: null
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeBookId: null
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeBookId: null
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeBookId: null
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeBookId: null
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.enter, (state, action) => {
    return {
      collection: state.collection,
      activeBookId: null
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.enter, (state, action) => {
    return {
      ...state,
      activeBookId: null
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.enter, (state, action) => {
    return {
      ...state,
      activeBookId: null
    };
  }),
  on(
    BooksPageActions.clearSelectedBook,
    (state, action) => {
      return {
        ...state,
        activeBookId: null
      };
    }
  )
);
```

```
export const reducer = createReducer(
  initialState,
  on(
    BooksPageActions.enter,
    BooksPageActions.clearSelectedBook,
    (state, action) => {
      return {
        ...state,
        activeBookId: null
      };
    }
  )
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.selectBook, (state, action) => {
    return {
      ...state,
      activeBookId: action.bookId
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.selectBook, (state, action) => {
    return {
      ...state,
      activeBookId: action.bookId
    };
  })
);
```

```
export const reducer = createReducer(
  initialState,
  on(BooksPageActions.selectBook, (state, action) => {
    return {
      ...state,
      activeBookId: action.bookId
    };
  })
);
```

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 **reducer** using **createReducer**
4. Use the **on** function in the reducer to handle the **enter**, **clearSelectedBook**, and **selectBook** actions from **BooksPageActions**

SETTING UP THE STORE

STORE

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

```typescript
export interface State {
  collection: BookModel[];
  activeBookId: string | null;
}
```

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

```
import * as fromBooks from "./books/books.reducer";

export interface State {
  books: fromBooks.State;
}

export const reducers: ActionReducerMap<State> = {
  books: fromBooks.reducer
};
```

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

Demo

Challenge

1. Add the books **State** interface to the state feature's **State** interface
2. Add the book **reducer** to the feature's **reducers map**
3. Install the **Redux Devtools** extension in your browser
4. Using the **Redux Devtools**, verify the **actions** are being dispatched and the **active book state** is changing

BACK TO REDUCERS

IMMUTABILITY

```
const createBook = (
  books: BookModel[],
  book: BookModel
) => [...books, book];
const updateBook = (
  books: BookModel[],
  changes: BookModel
) =>
  books.map(book => {
    return book.id === changes.id
      ? Object.assign({}, book, changes)
      : book;
  });
const deleteBook = (
  books: BookModel[],
  bookId: string
) => books.filter(book => bookId !== book.id);
```

```
const createBook = (
  books: BookModel[],
  book: BookModel
) => [...books, book];
```

```
const updateBook = (
  books: BookModel[],
  changes: BookModel
) =>
  books.map(book => {
    return book.id === changes.id
      ? Object.assign({}, book, changes)
      : book;
  });
```

```
const deleteBook = (
  books: BookModel[],
  bookId: string
) => books.filter(book => bookId !== book.id);
```

Demo

Challenge

1. Create a **reducer handler** for the **booksLoaded** action that adds the books to the state
2. Create helpers to immutability **create a book**, **update a book**, and **delete a book**
3. Create a **reducer handler** for the **bookCreated** action that adds the book to the state
4. Create a **reducer handler** for the **bookUpdated** action that updates the book in the state
5. Create a **reducer handler** for the **bookDeleted** action that removes the deleted book from the state

SELECTORS

SELECTORS

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

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

```
export const selectActiveBookId = (state: State) => {
  return state.activeBookId;
};
```

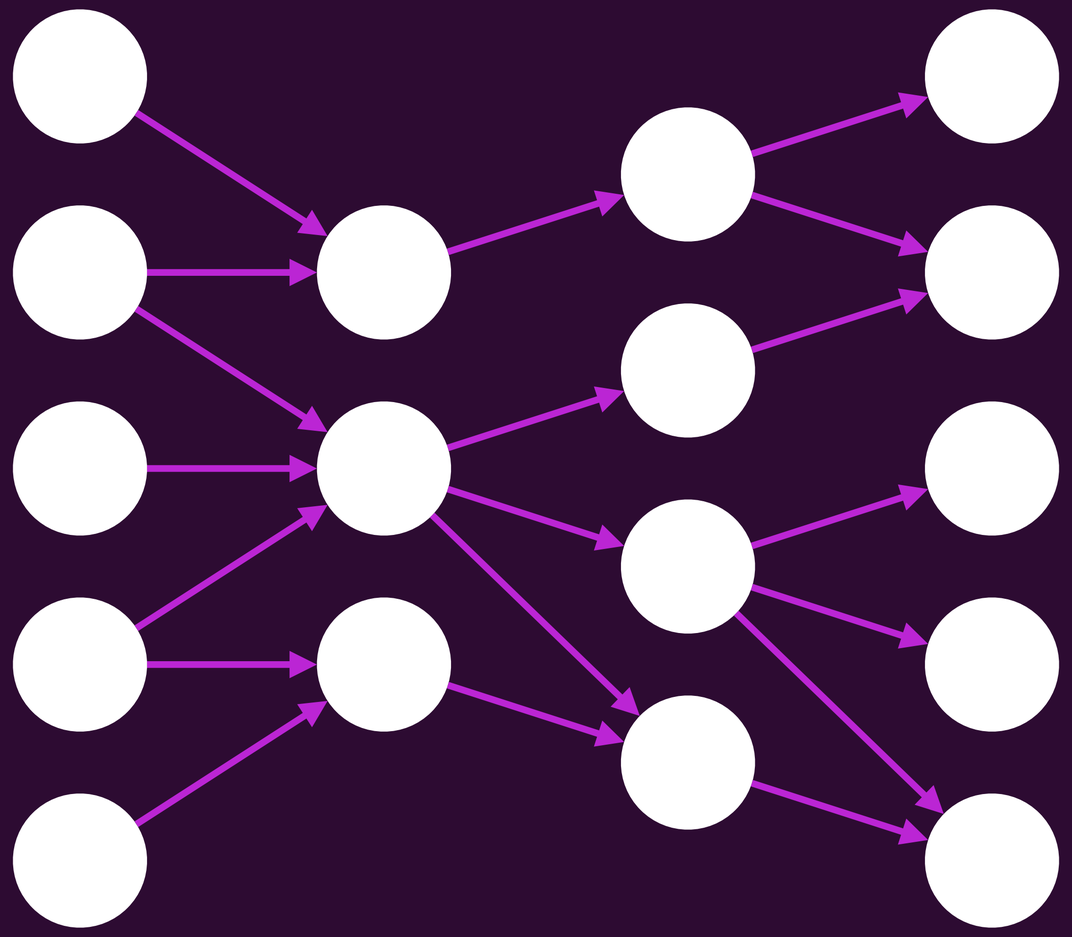
```
export const selectActiveBook = (state: State) => {
  const books = selectBooks(state);
  const activeBookId = selectActiveBookId(state);

  return (
    books.find(book => book.id === activeBookId) || null
  );
};
```

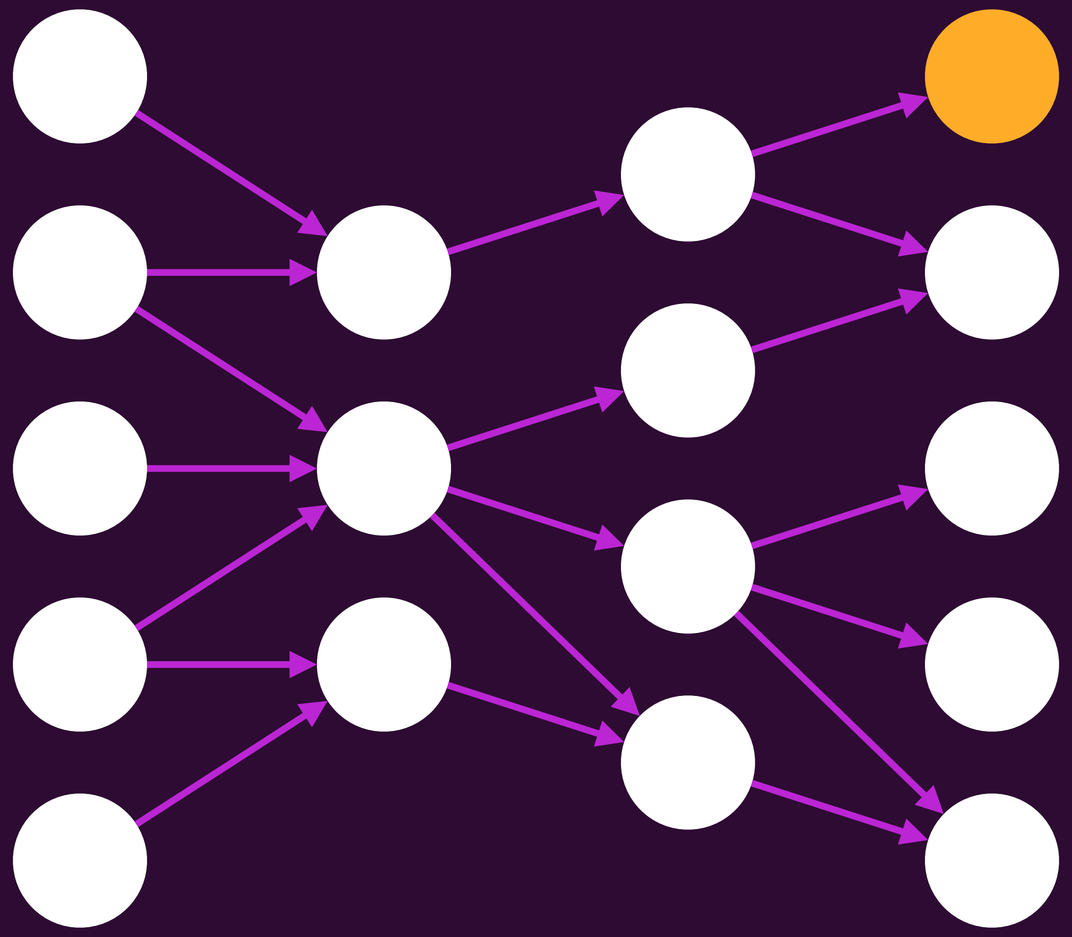
```
export const selectActiveBook = createSelector(
  selectBooks,
  selectActiveBookId,
  (books, activeBookId) => {
    return (
      books.find(book => book.id === activeBookId) || null
    );
  }
);
```

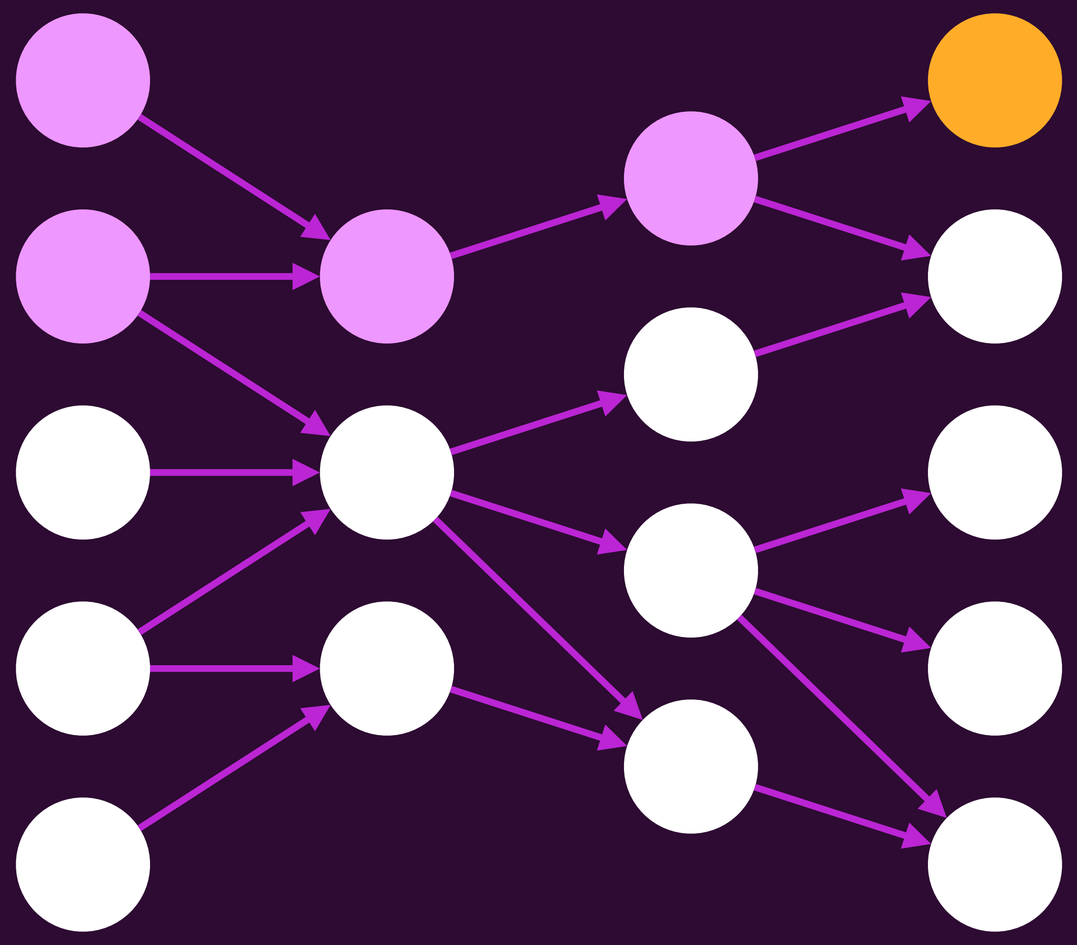
```
export const selectEarningsTotals = createSelector(
  selectBooks,
  books => {
    return calculateGrossBooksEarnings(books);
  }
);
```

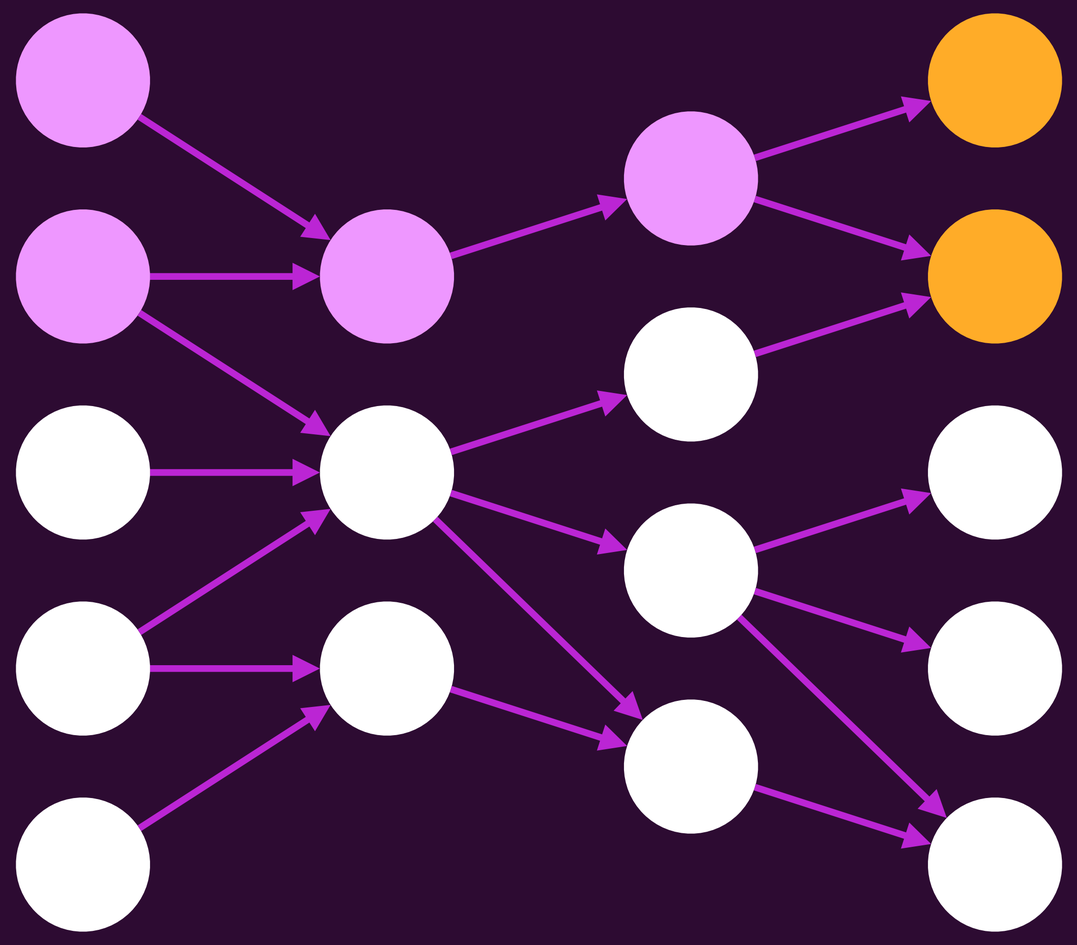
```
export const selectEarningsTotals = createSelector(
  selectBooks,
  calculateGrossBooksEarnings
);
```

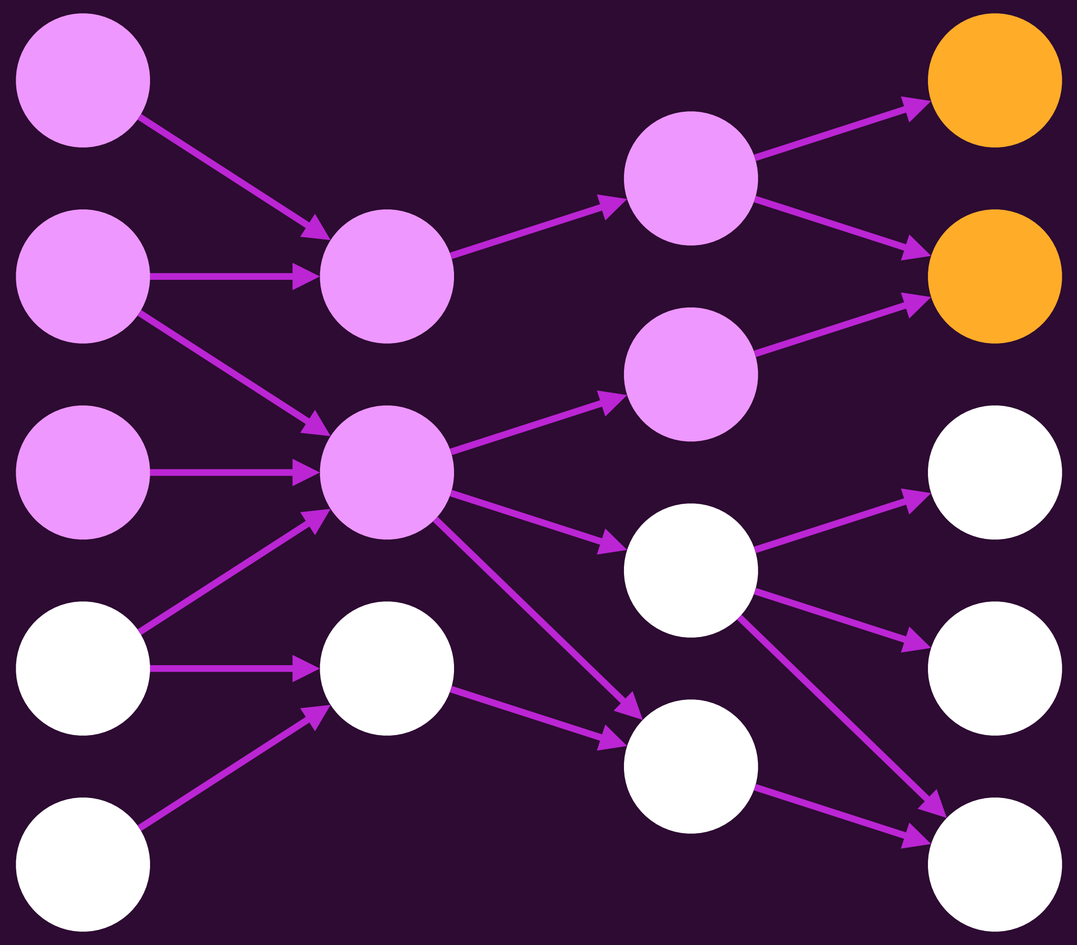
COMPOSITION











Write many selectors with few inputs

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**

USING SELECTORS

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

```
export const selectBooksState = (state: State) => {
  return state.books;
};

export const selectActiveBook = (state: State) => {
  const booksState = selectBooksState(state);

  return fromBooks.selectActiveBook(booksState);
};
```

```
export const selectBooksState = (state: State) => {
  return state.books;
};

export const selectActiveBook = createSelector(
  selectBooksState,
  fromBooks.selectActiveBook
);
```

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 BooksPageComponent {
  books: BookModel[] = [];
}
```

```
class BooksPageComponent {
  books$: Observable<BookModel[]>;

  constructor(store: Store) {
    this.books$ = store.select(selectAllBooks);
  }
}
```

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

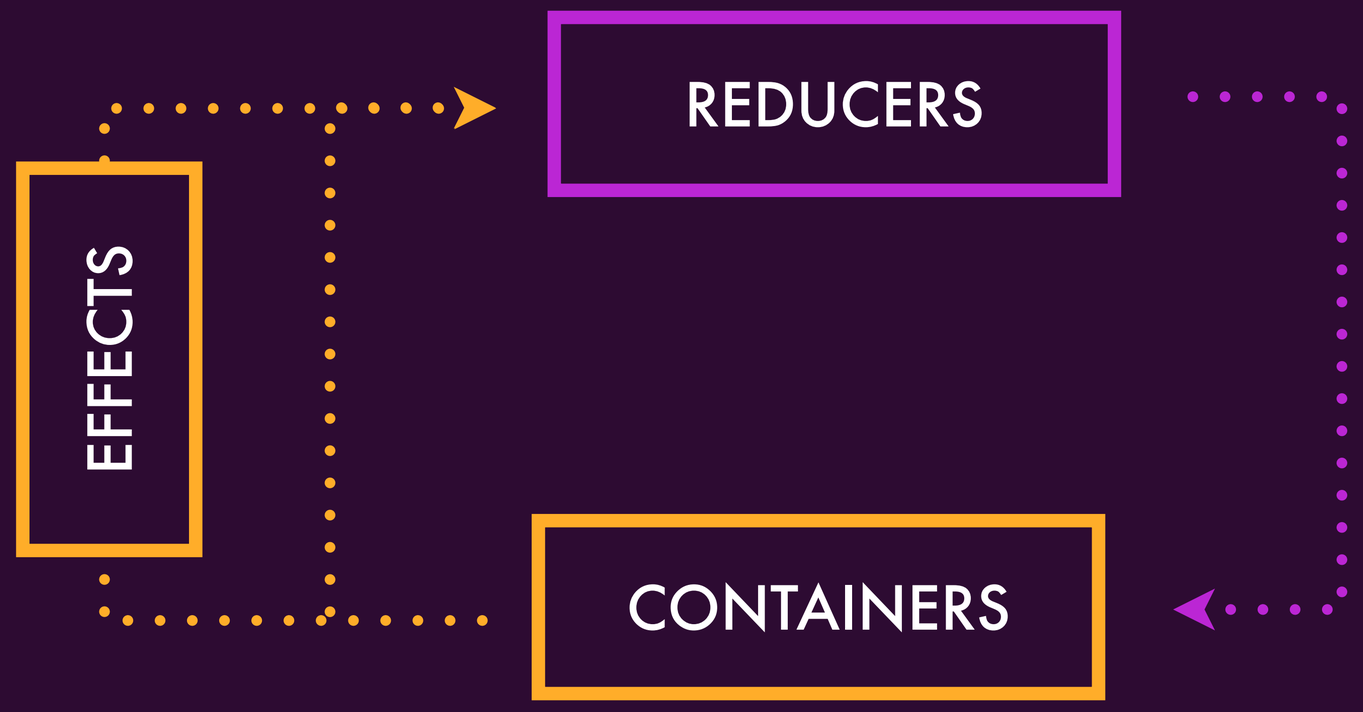
```
<app-books-list
  [books]="books$ | async"
  (select)="onSelect($event)"
  (delete)="onDelete($event)"
></app-books-list>
```

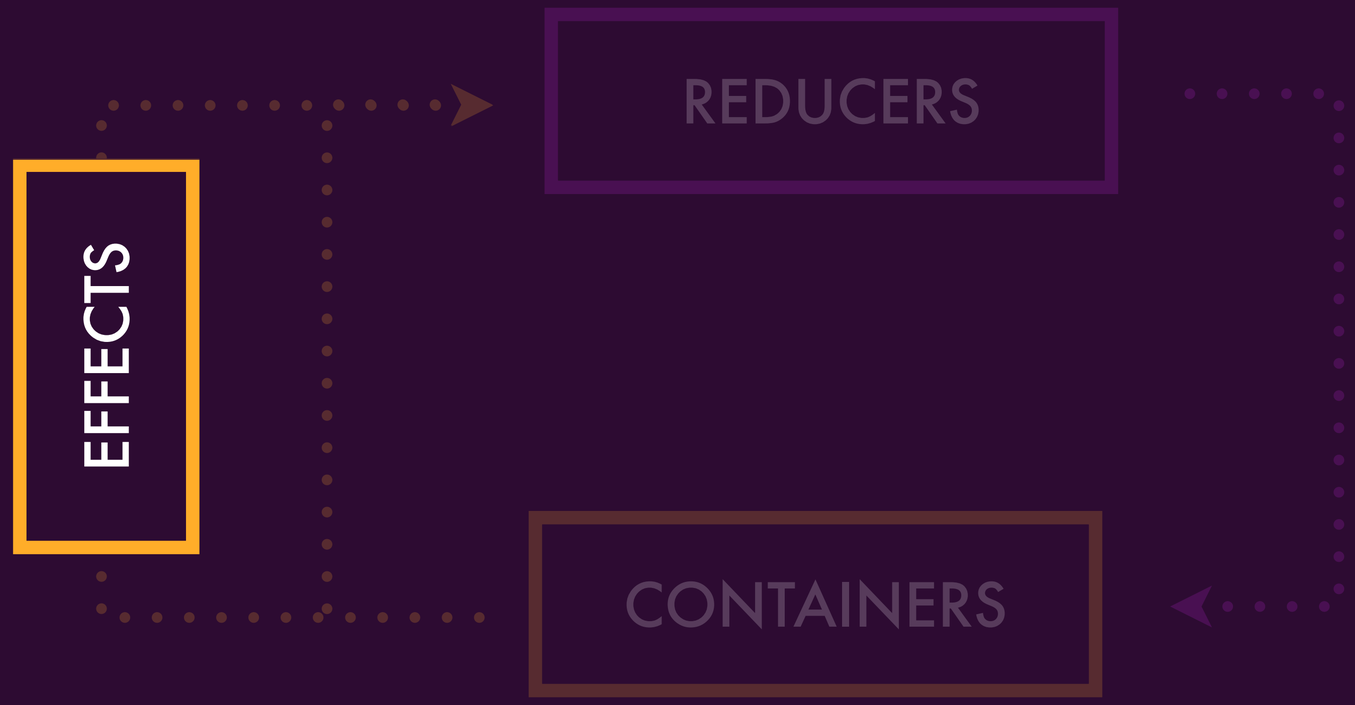
Demo

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.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


REDUCERS
EFFECTS
CONTAINERS

EFFECTS

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

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

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

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

```
@Injectable()
export class BooksEffects {
  constructor(
    private actions$: Actions,
    private booksService: BooksService
  ) {}
}
```

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

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

Demo

Challenge

1. Open **books-api.effects.ts**
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 **getBooks() method** in **books-page.component.ts** that gets all of the books

ADVANCED EFFECTS

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

```
@Injectable()
export class BooksApiEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BooksPageActions.enter),
      mergeMap(() =>
        this.booksService
          .all()
          .pipe(map((books) => BooksApiActions.booksLoaded({ books })))
      )
    )
  );
}
```

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 BooksEffects {
  @Effect() loadBooks$ = this.actions$.pipe(
    ofType(BooksPageActions.enter),
    mergeMap(() =>
      this.booksService.all().pipe(
        map(books =>
          BooksApiActions.booksLoaded({ books })
        )
      )
    )
  );
}
```

```
class BooksEffects {
  @Effect() loadBooks$ = this.actions$.pipe(
    ofType(BooksPageActions.enter),
    exhaustMap(() =>
      this.booksService.all().pipe(
        map(books =>
          BooksApiActions.booksLoaded({ books })
        )
      )
    )
  );
}
```

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

```
tick$ = createEffect(() => {
  return interval(/* Every minute */ 60_000).pipe(
    map(() => Clock.tickAction(new Date()))
  );
});
```

```
ws$ = createEffect(() => {
  return webSocket("/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);
      }
    }
  }));
});
```

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

```
promptToRetry$ = createEffect(() => {
  return 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)))
    )
  );
});
```

```
promptToRetry$ = createEffect(() => {
  return 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)))
    )
  );
});
```

Failed to save book. TRY AGAIN

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

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

```
uploadCover$ = createEffect(() => {
  return 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

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

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

```typescript
export interface BooksState extends EntityState<BookModel> {
  activeBookId: string | null;
}
export const adapter = createEntityAdapter<BookModel>();
export const initialState: BooksState = adapter.getInitialState(
  {
    activeBookId: null
  }
);
```

```typescript
export interface BooksState extends EntityState<BookModel> {
  activeBookId: string | null;
}
export const adapter = createEntityAdapter<BookModel>();
export const initialState: BooksState = adapter.getInitialState(
  {
    activeBookId: null
  }
);
```

```
export interface BooksState extends EntityState<BookModel> {
  activeBookId: string | null;
}
export const adapter = createEntityAdapter<BookModel>();
export const initialState: BooksState = adapter.getInitialState(
  {
    activeBookId: null
  }
);
```

```
export interface BooksState extends EntityState<BookModel> {
  activeBookId: string | null;
}
export const adapter = createEntityAdapter<BookModel>();
export const initialState: BooksState = adapter.getInitialState(
  {
    activeBookId: 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.setAll(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,
  (booksEntities, activeBookId) => {
    if (activeBookId) return booksEntities[activeBookId] ?? null;

    return null;
  }
);
```

Demo

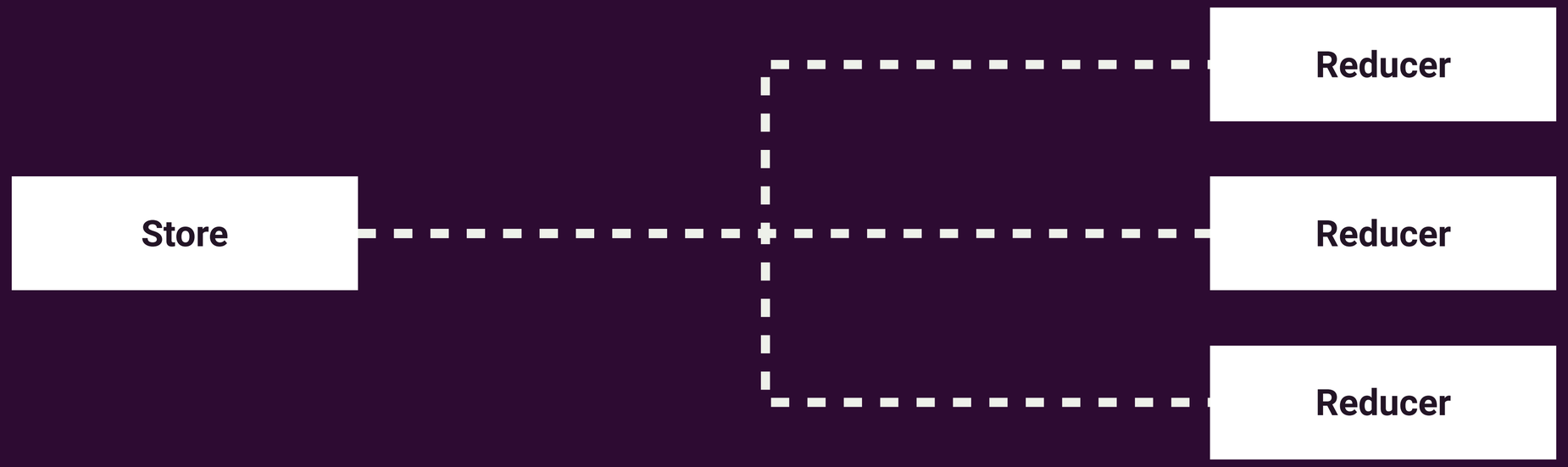
Challenge

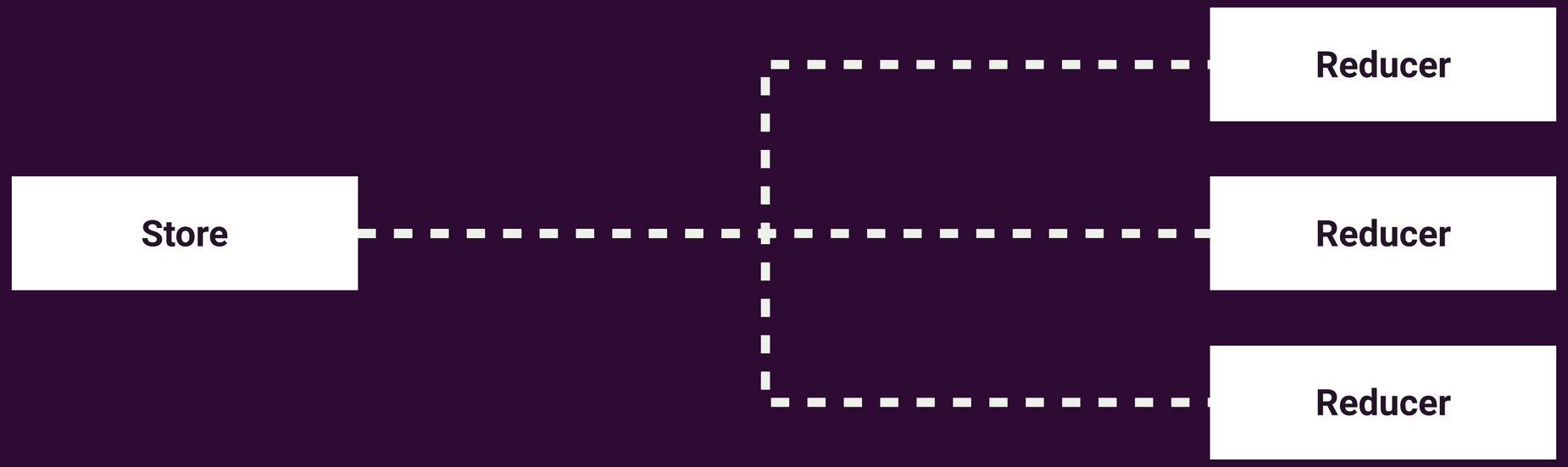
1. Update **books.reducer.ts** to use **EntityState** to define **State**
2. Create an **unsorted entity adapter** for **State** and use it to initialize **initialState**
3. Update the **reducer** to use the **adapter** methods
4. Use the adapter to replace the **selectAll** selector and to create a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

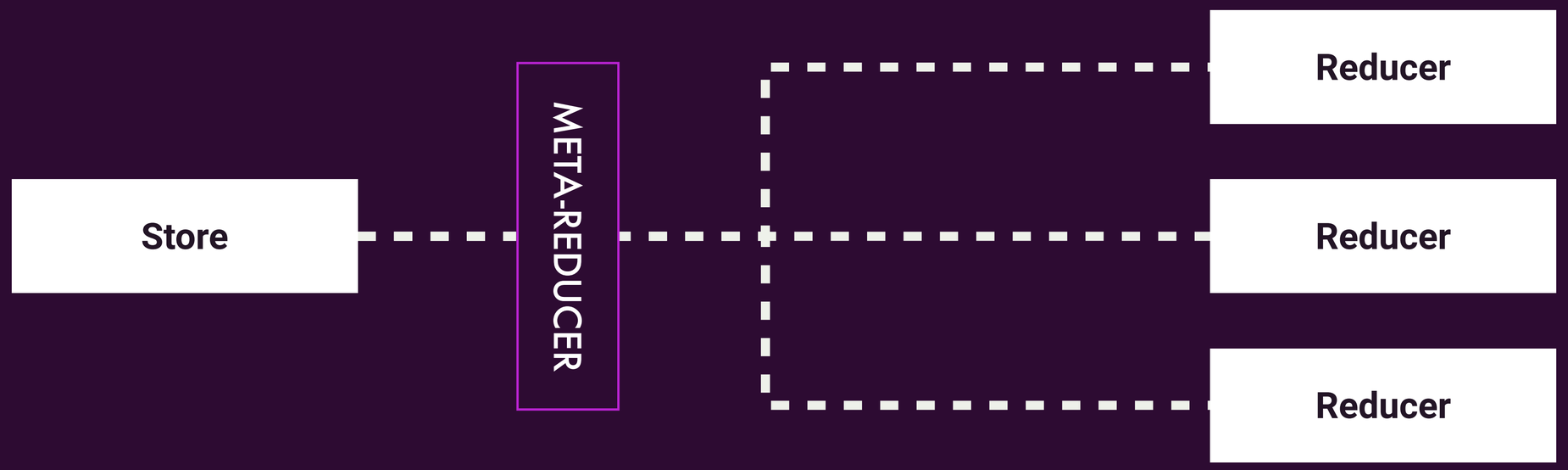
META-REDUCERS

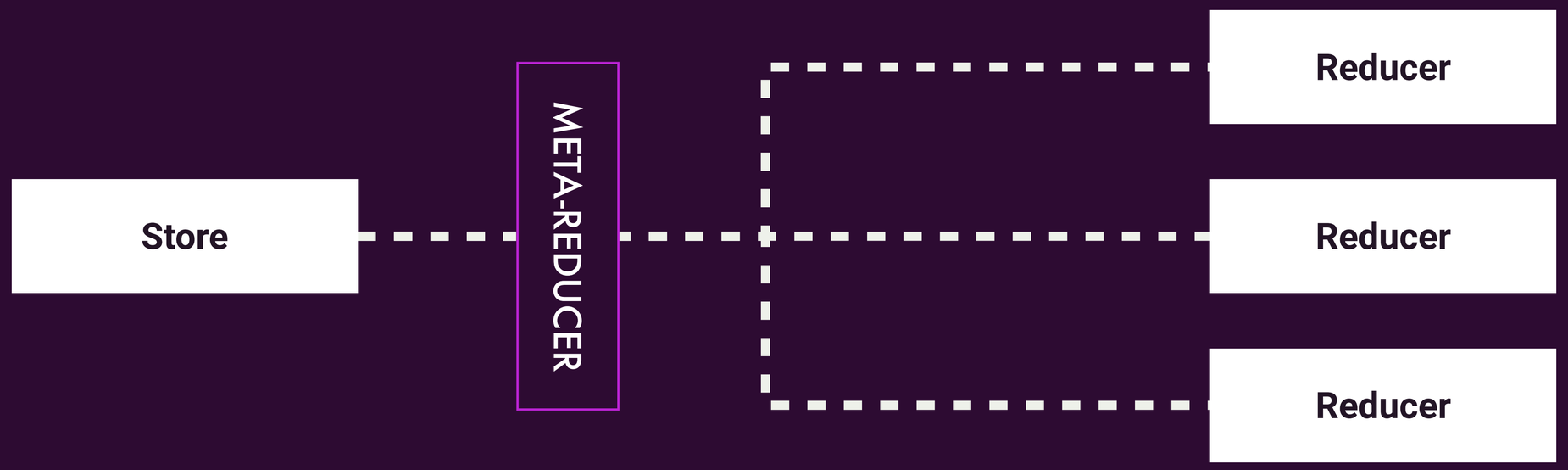
META-REDUCERS

- ✅ Intercept actions before they are reduced
- ✅ Intercept state before it is emitted
- ✅ Can change control flow of the Store


Store
Reducer
Reducer
Reducer


Store
Reducer
Reducer
Reducer


Store
META-REDUCER
Reducer
Reducer
Reducer


Store
META-REDUCER
Reducer
Reducer
Reducer

```
function logger(reducer: ActionReducer<any>) {
  return function(state: any, action: Action) {
    console.log("Previous State", state);
    console.log("Next Action", action);

    const nextState = reducer(state, action);

    console.log("Next State", nextState);

    return nextState;
  };
}
```

```
function logger(reducer: ActionReducer<any>) {
  return function(state: any, action: Action) {
    console.log("Previous State", state);
    console.log("Next Action", action);

    const nextState = reducer(state, action);

    console.log("Next State", nextState);

    return nextState;
  };
}
```

```
function logger(reducer: ActionReducer<any>) {
  return function(state: any, action: Action) {
    console.log("Previous State", state);
    console.log("Next Action", action);

    const nextState = reducer(state, action);

    console.log("Next State", nextState);

    return nextState;
  };
}
```

```
function logger(reducer: ActionReducer<any>) {
  return function(state: any, action: Action) {
    console.log("Previous State", state);
    console.log("Next Action", action);

    const nextState = reducer(state, action);

    console.log("Next State", nextState);

    return nextState;
  };
}
```

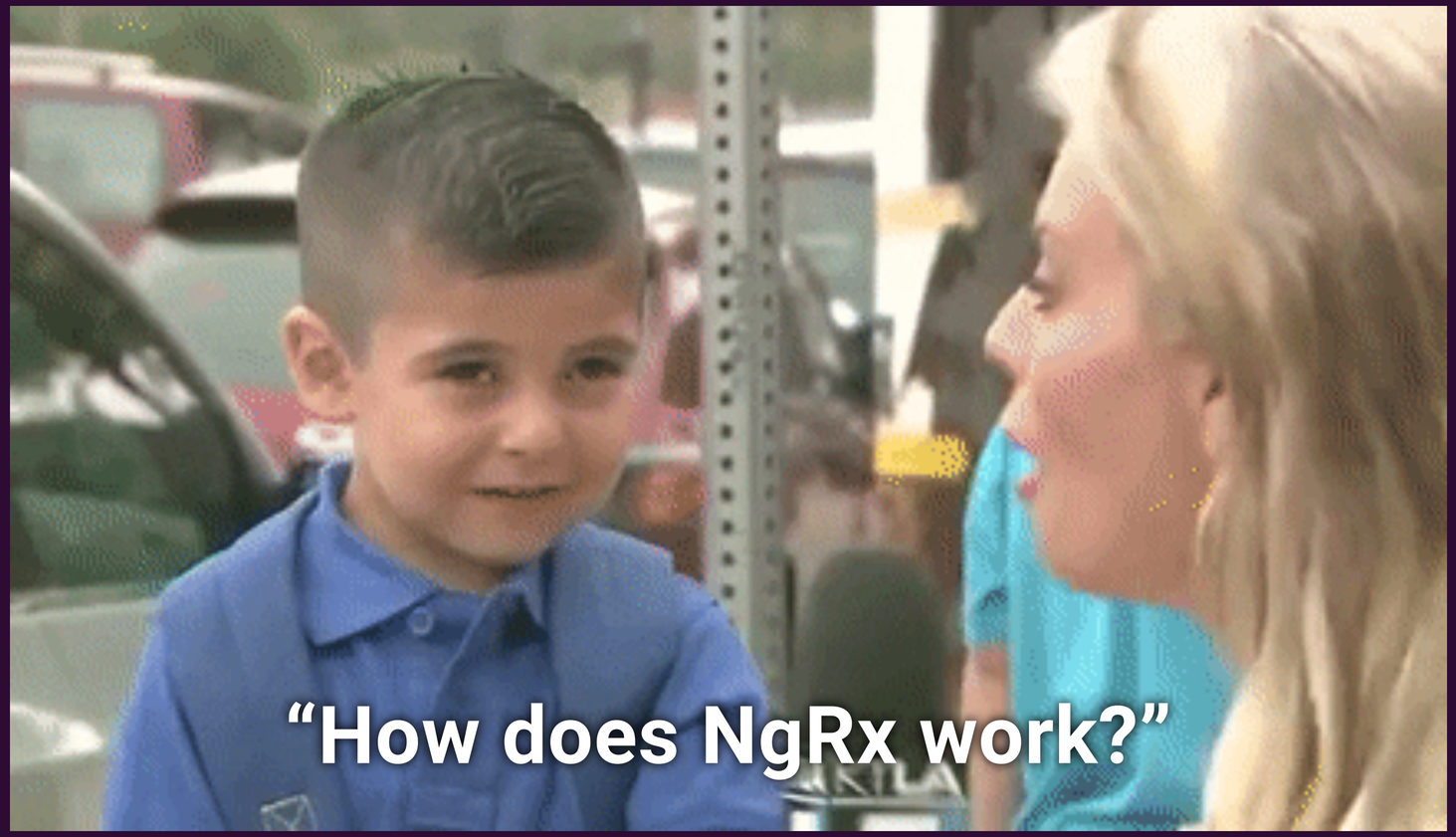
```
function logger(reducer: ActionReducer<any>) {
  return function(state: any, action: Action) {
    console.log("Previous State", state);
    console.log("Next Action", action);

    const nextState = reducer(state, action);

    console.log("Next State", nextState);

    return nextState;
  };
}
```

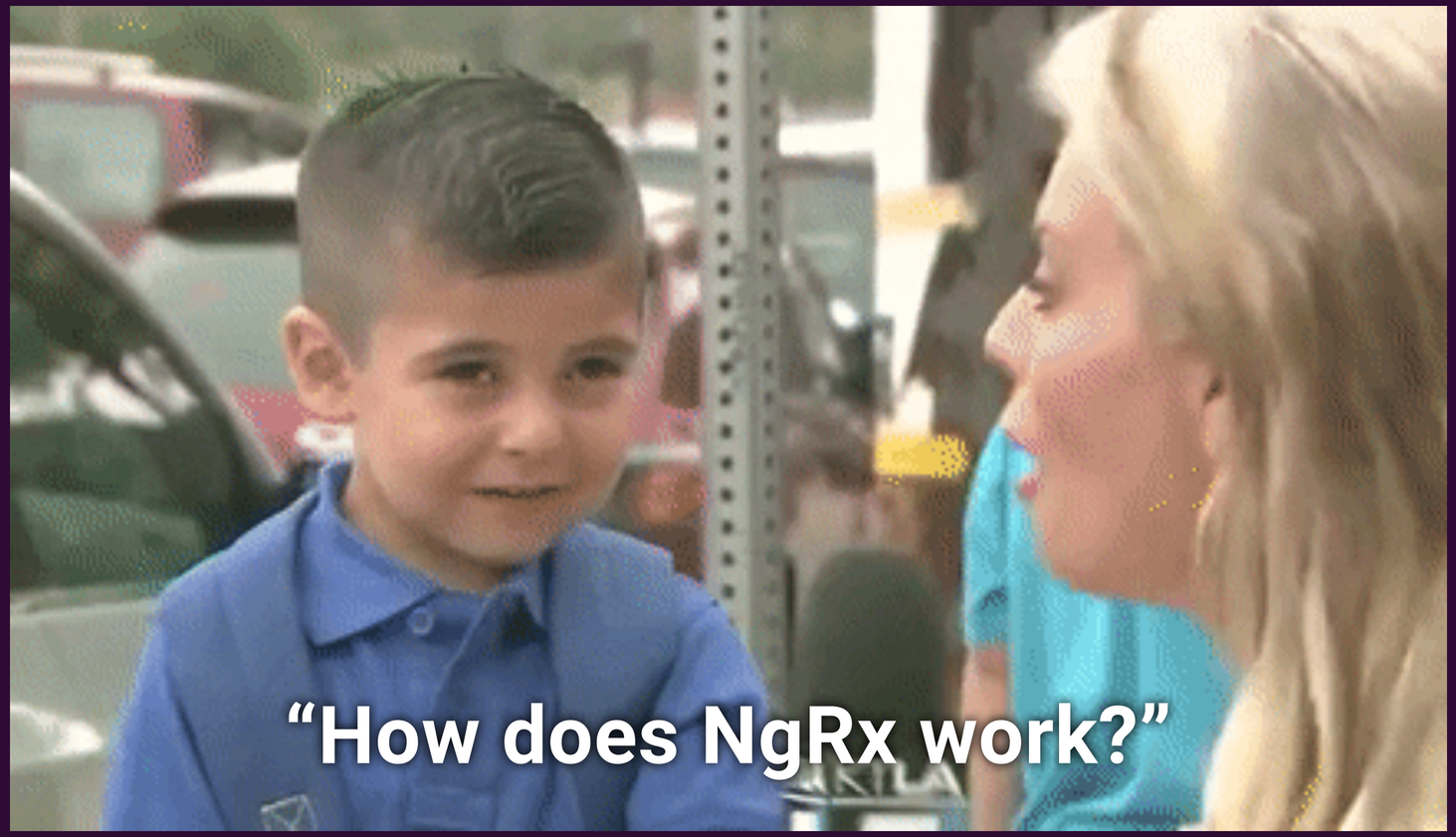
```
function logger(reducer: ActionReducer<any>) {
  return function(state: any, action: Action) {
    console.log("Previous State", state);
    console.log("Next Action", action);

    const nextState = reducer(state, action);

    console.log("Next State", nextState);

    return nextState;
  };
}
```

```
function logger(reducer: ActionReducer<any>) {
  return function(state: any, action: Action) {
    console.log("Previous State", state);
    console.log("Next Action", action);

    const nextState = reducer(state, action);

    console.log("Next State", nextState);

    return nextState;
  };
}
```

```
function logger(reducer: ActionReducer<any>) {
  return function(state: any, action: Action) {
    console.log("Previous State", state);
    console.log("Next Action", action);

    const nextState = reducer(state, action);

    console.log("Next State", nextState);

    return nextState;
  };
}
```

Demo


“How does NgRx work?”


“How does NgRx work?”

- [x] State flows down, changes flow up

- ☑ State flows down, changes flow up
- ☑ Indirection between state & consumer

- ✓ State flows down, changes flow up
- ✓ Indirection between state & consumer
- ✓ Select & Dispatch => Input & Output

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

ACTIONS

- ☑ Unified interface to describe events
- ☑ Just data, no functionality
- ☑ Has at a minimum a type property

REDUCERS

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

STORE

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

SELECTORS

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

EFFECTS

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

META-REDUCERS

- ✅ Intercept actions before they are reduced
- ✅ Intercept state before it is emitted
- ✅ Can change control flow of the Store

“How does NgRx work?”

“How does NgRx work?”

HELP US IMPROVE

bit.ly/3erckGH

NGRX CONF

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/schematics

@ngrx/router-store

@ngrx/schematics

@ngrx/router-store

@ngrx/data

@ngrx/schematics

@ngrx/router-store

@ngrx/data

ngrx.io

@ngrx/component-store

@ngrx/schematics

@ngrx/router-store

@ngrx/data

ngrx.io

THANK YOU!

@MikeRyanDev

@brian_love

THANK YOU