NgRx

Reactive State for Angular

TABLE OF CONTENTS

01

What is NgRx?

NgRx (Angular Reactive Extensions)

NgRx is a framework for building reactive applications in Angular. NgRx is a state management system that is based on the Redux pattern. NgRx provides libraries for:

- Managing global and local state
- Isolation of side effects to promote a cleaner component architecture
- Entity collection management
- Integration with the Angular Router
- Developer tooling that enhance developers experience when building many different types of applications

02

State

Packages of State

Store

Effects

Router Store

Entity

ComponentStore

Why use NgRx Store for State Management?

NgRx Store provides state management for creating maintainable explicit applications, by single state and the use of actions in order to express state changes.

When Should I Use NgRx Store for State Management

A good substance that might answer the question "Do I need NgRx", is the SHARI principle:

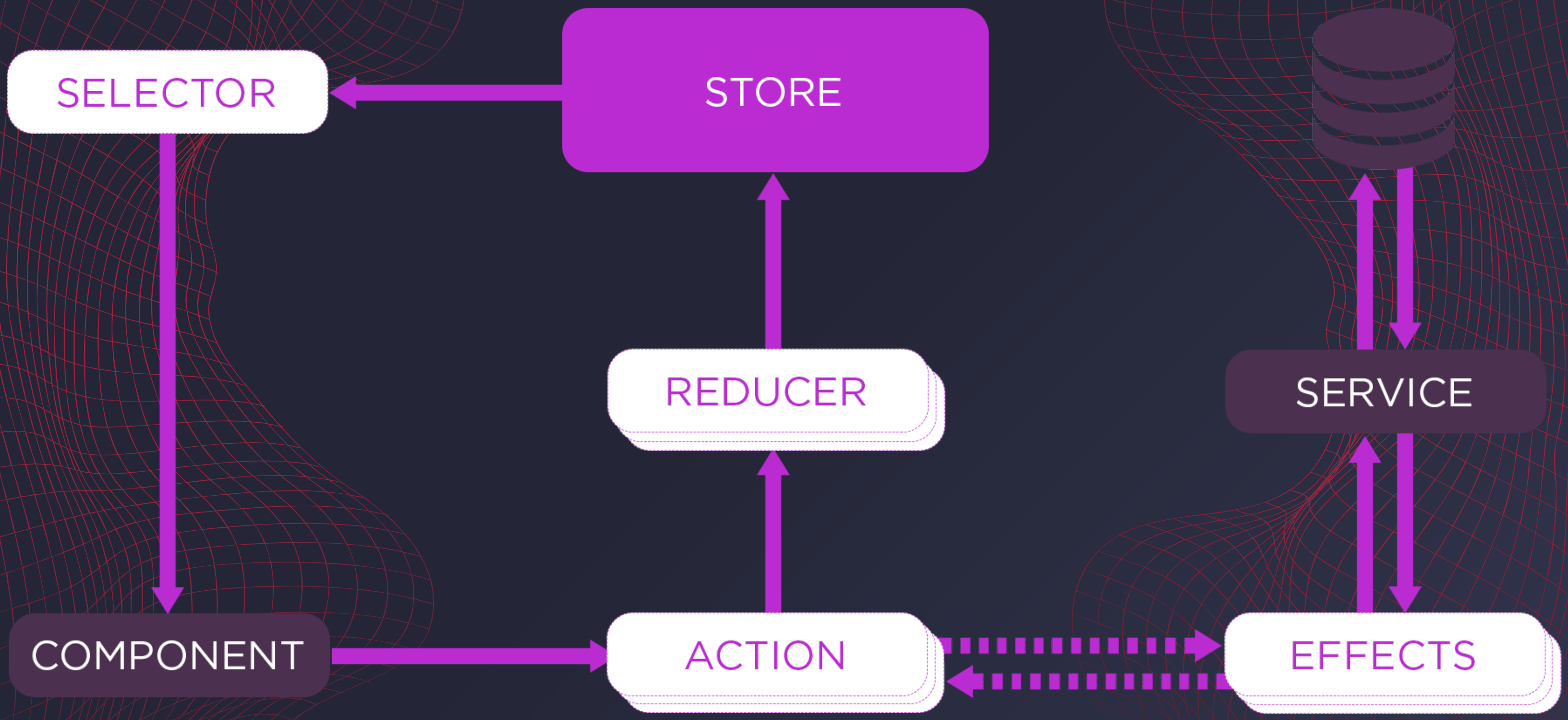
- Shared: state that is accessed by many components and services.
- Hydrated: state that is persisted and rehydrated from external storage.
- Available: state that needs to be available when re-entering routes.
- Retrieved: state that must be retrieved with a side-effect.
- Impacted: state that is impacted by actions from other sources.

@ngrx/store

Store is RxJS powered global state management for Angular applications, inspired by Redux. Store is a controlled state container designed to help write performant, consistent applications on top of Angular.

- Actions describe unique events that are dispatched from components and services.
- State changes are handled by pure functions called reducers that take the current state and the latest action to compute a new state.
- Selectors are pure functions used to select, derive and compose pieces of state.
- State is accessed with the Store, an observable of state and an observer of actions.

NGRX STATE MANAGEMENT LIFECYCLE

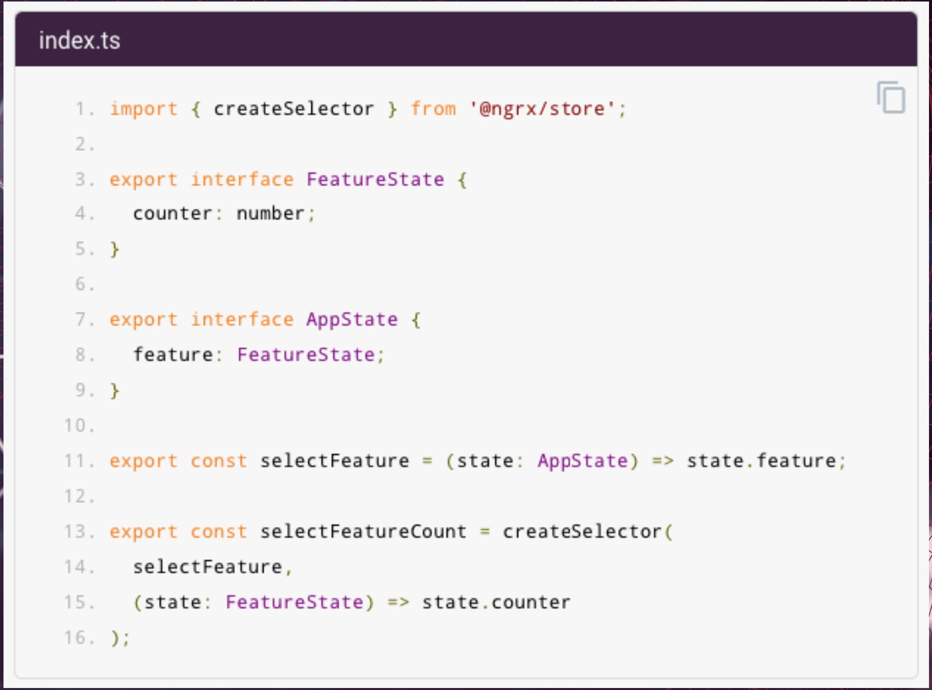


Action

- Upfront - write actions before developing features to understand and gain a shared knowledge of the feature being implemented.
- Divide - categorize actions based on the event source.
- Many - actions are inexpensive to write, so the more actions you write, the better you express flows in your application.
- Event-Driven - capture *events* **not** *commands* as you are separating the description of an event and the handling of that event.
- Descriptive - provide context that are targeted to a unique event with more detailed information you can use to aid in debugging with the developer tools.

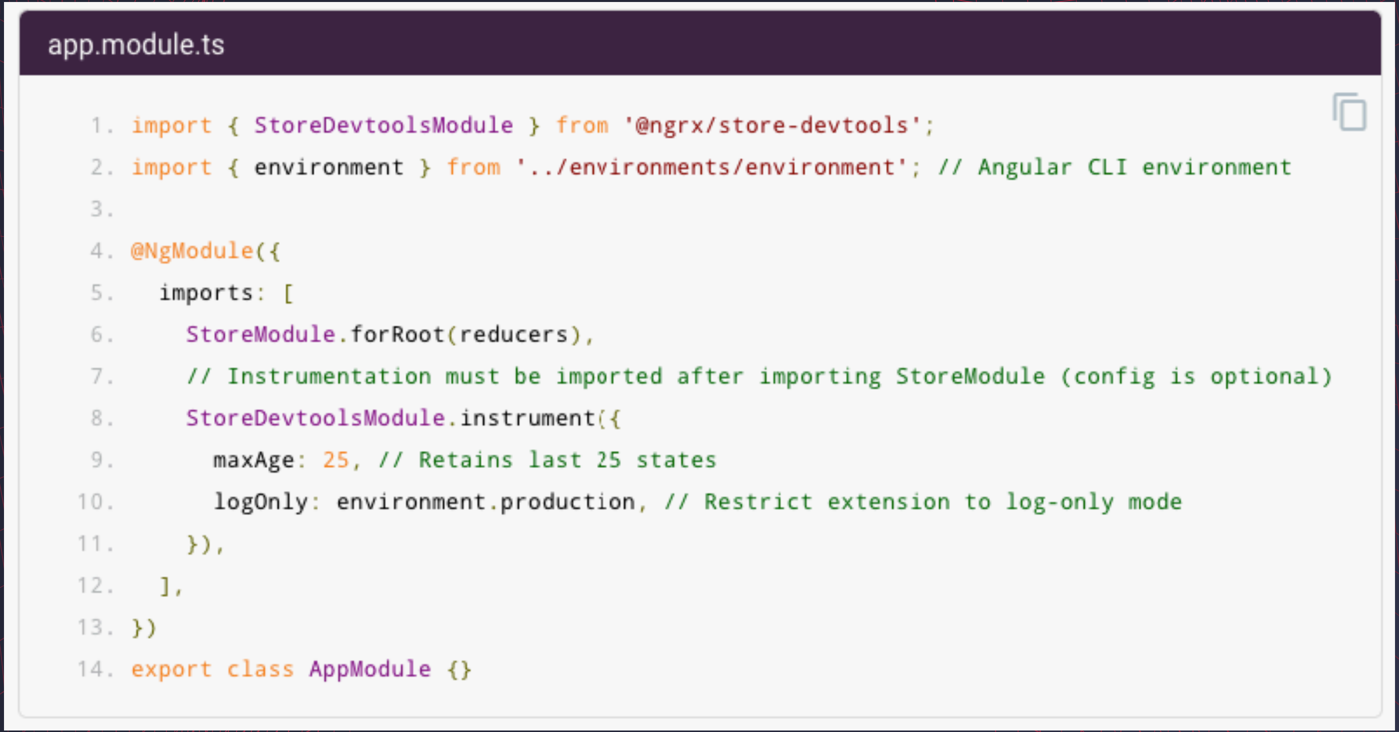
Selectors

- Portability
- Memoization
- Composition
- Testability
- Type Safety


index.ts

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

export interface FeatureState {
  counter: number;
}

export interface AppState {
  feature: FeatureState;
}

export const selectFeature = (state: AppState) => state.feature;

export const selectFeatureCount = createSelector(
  selectFeature,
  (state: FeatureState) => state.counter
);
```

@ngrx/store-devtools



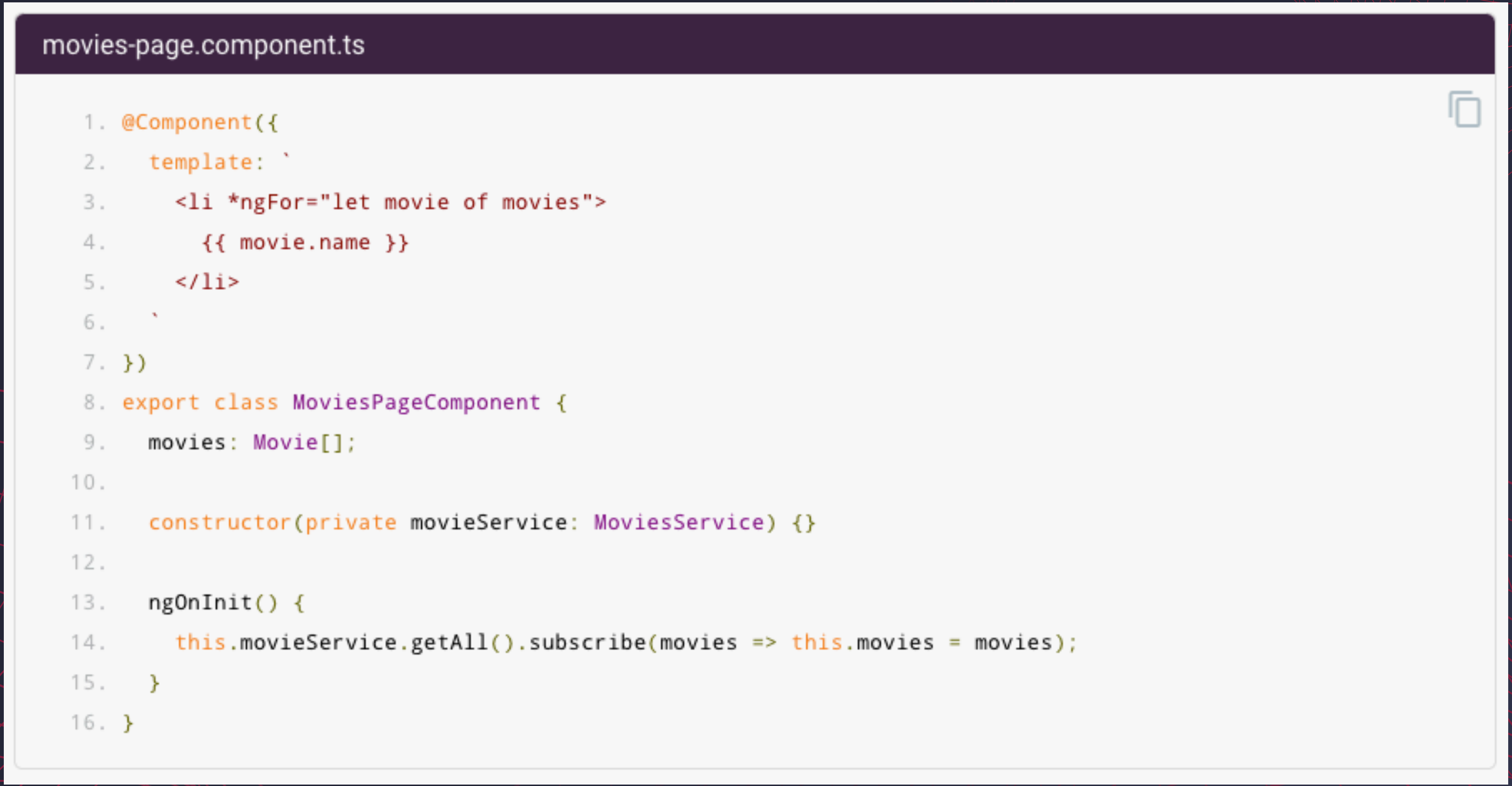
app.module.ts

```
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import { environment } from '../environments/environment'; // Angular CLI environment

@NgModule({
  imports: [
    StoreModule.forRoot(reducers),
    // Instrumentation must be imported after importing StoreModule (config is optional)
    StoreDevtoolsModule.instrument({
      maxAge: 25, // Retains last 25 states
      logOnly: environment.production, // Restrict extension to log-only mode
    }),
  ],
})
export class AppModule {}
```

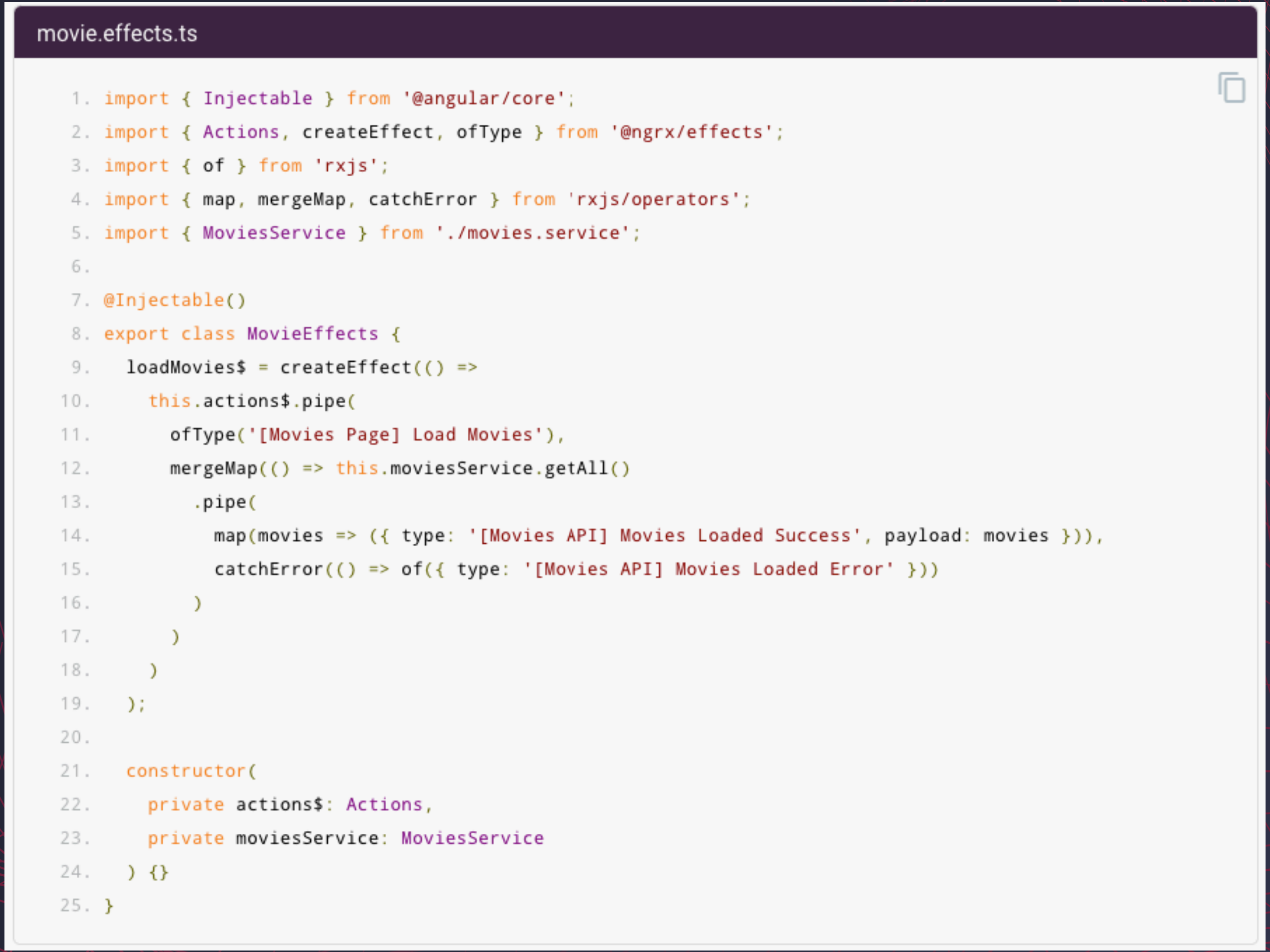
@ngrx/effects

Effects are an RxJS powered side effect model for Store. Effects use streams to provide new sources of actions to reduce state based on external interactions such as network requests, web socket messages and time-based events.

- Effects isolate side effects from components, allowing for more *pure* components that select state and dispatch actions.
- Effects are long-running services that listen to an observable of *every* action dispatched from the Store.
- Effects filter those actions based on the type of action they are interested in. This is done by using an operator.
- Effects perform tasks, which are synchronous or asynchronous and return a new action.

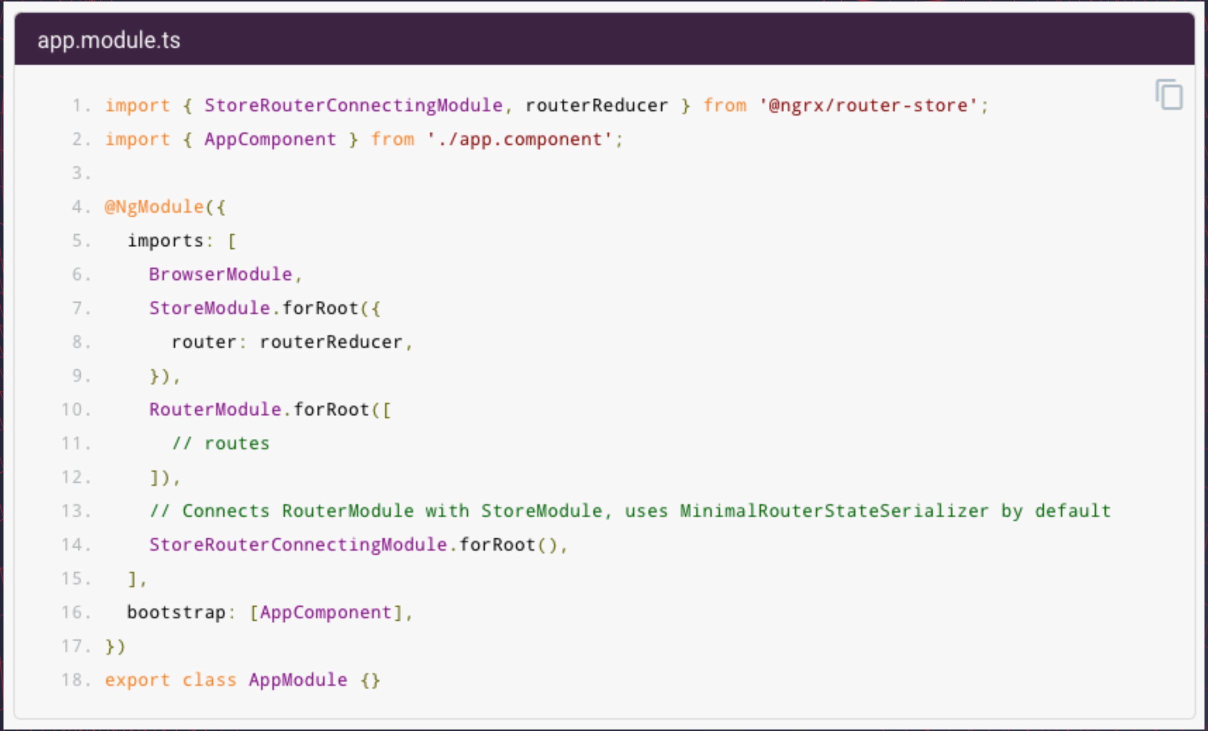

movies-page.component.ts

```typescript
@Component({
  template: `
    <li *ngFor="let movie of movies">
      {{ movie.name }}
    </li>
  `
})
export class MoviesPageComponent {
  movies: Movie[];

  constructor(private movieService: MoviesService) {}

  ngOnInit() {
    this.movieService.getAll().subscribe(movies => this.movies = movies);
  }
}
```


movie.effects.ts

```typescript
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { of } from 'rxjs';
import { map, mergeMap, catchError } from 'rxjs/operators';
import { MoviesService } from './movies.service';

@Injectable()
export class MovieEffects {
  loadMovies$ = createEffect(() =>
    this.actions$.pipe(
      ofType('[Movies Page] Load Movies'),
      mergeMap(() => this.moviesService.getAll()
        .pipe(
          map(movies => ({ type: '[Movies API] Movies Loaded Success', payload: movies })),
          catchError(() => of({ type: '[Movies API] Movies Loaded Error' }))
        )
      )
    )
  );

  constructor(
    private actions$: Actions,
    private moviesService: MoviesService
  ) {}
}
```

@ngrx/router-store



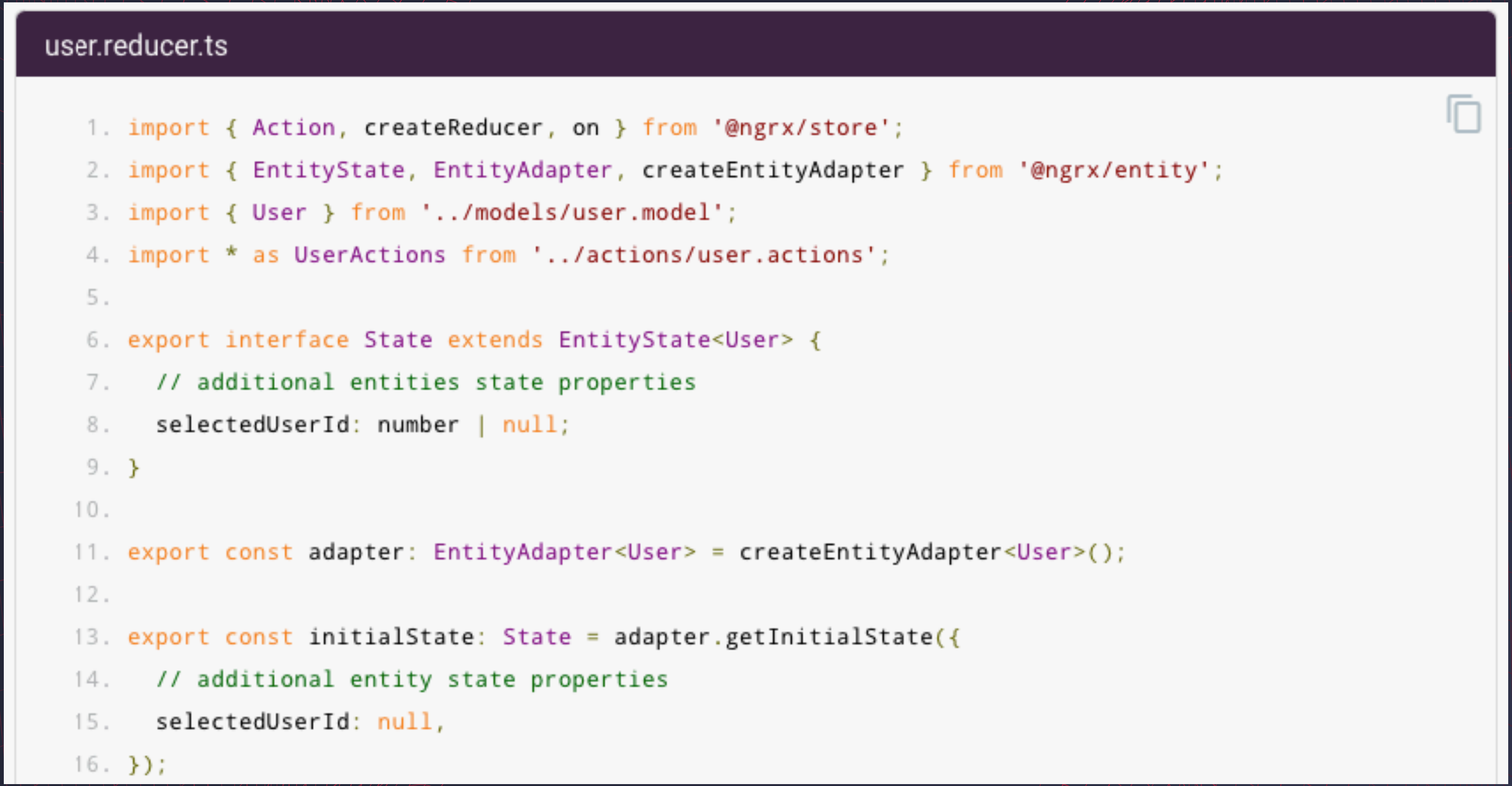
app.module.ts

```
import { StoreRouterConnectingModule, routerReducer } from '@ngrx/router-store';
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    StoreModule.forRoot({
      router: routerReducer,
    }),
    RouterModule.forRoot([
      // routes
    ]),
    // Connects RouterModule with StoreModule, uses MinimalRouterStateSerializer by default
    StoreRouterConnectingModule.forRoot(),
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

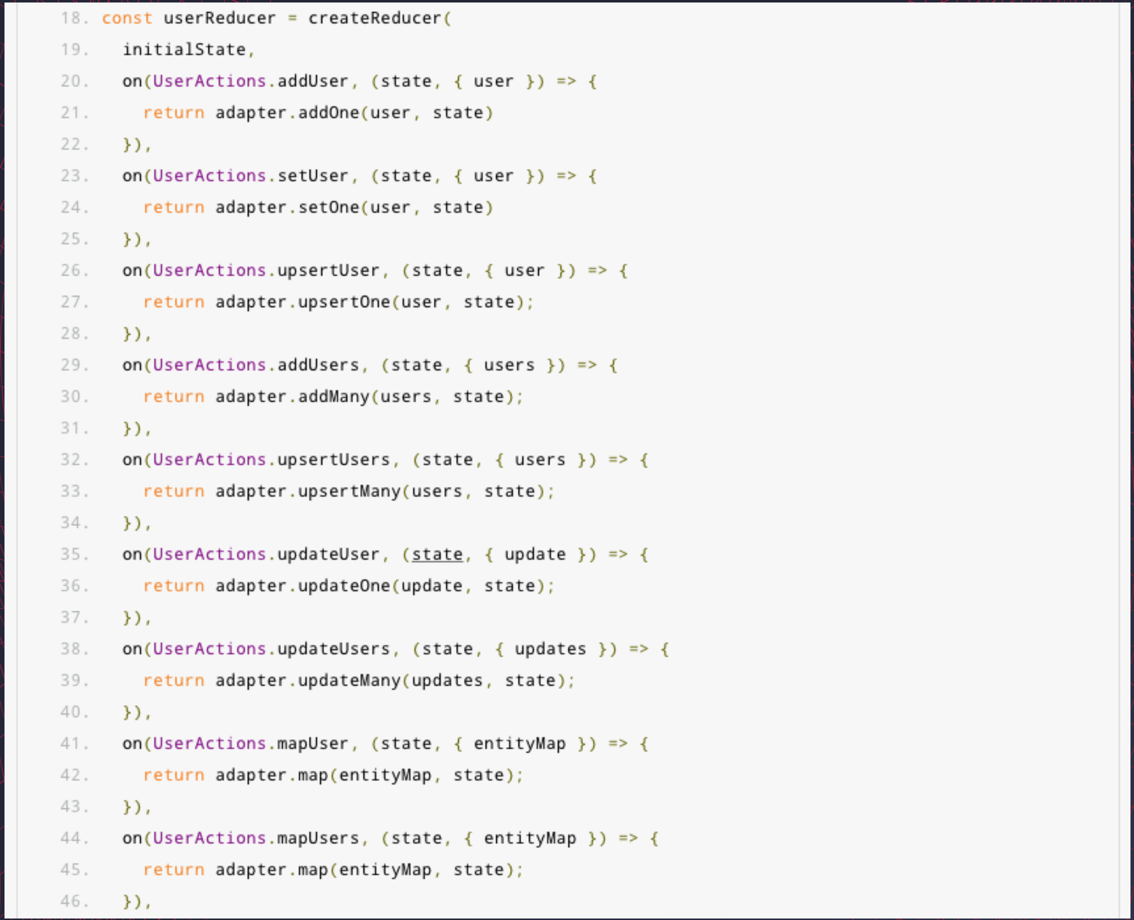
@ngrx/entity

Entity promotes the use of plain JavaScript objects when managing collections. *ES6 class instances will be transformed into plain JavaScript objects when entities are managed in a collection*. This provides you with some assurances when managing these entities:

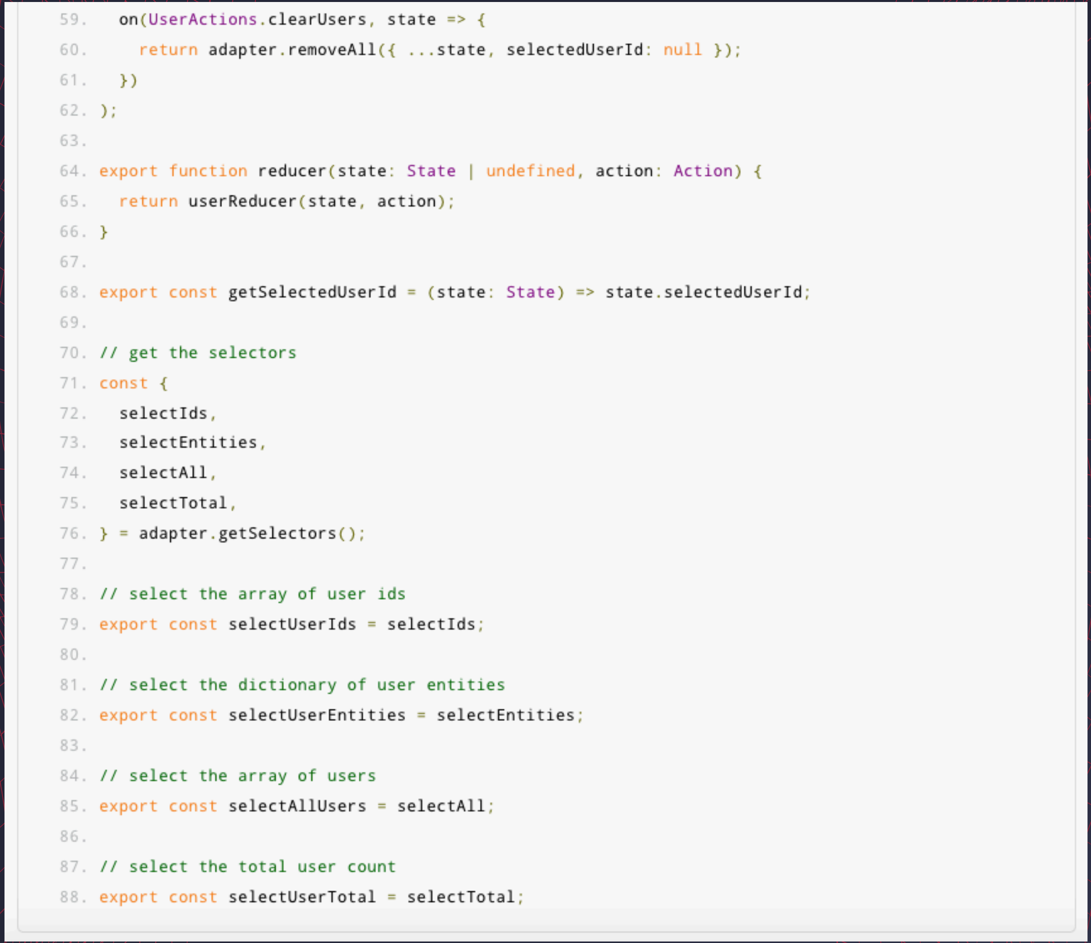
- Guarantee that the data structures contained in state don't themselves contain logic, reducing the chance that they'll mutate themselves.
- State will always be serializable allowing you to store and rehydrate from browser storage mechanisms like local storage.
- State can be inspected via the Redux Devtools.


user.reducer.ts

```typescript
import { Action, createReducer, on } from '@ngrx/store';
import { EntityState, EntityAdapter, createEntityAdapter } from '@ngrx/entity';
import { User } from '../models/user.model';
import * as UserActions from '../actions/user.actions';

export interface State extends EntityState<User> {
  // additional entities state properties
  selectedUserId: number | null;
}

export const adapter: EntityAdapter<User> = createEntityAdapter<User>();

export const initialState: State = adapter.getInitialState({
  // additional entity state properties
  selectedUserId: null,
});
```



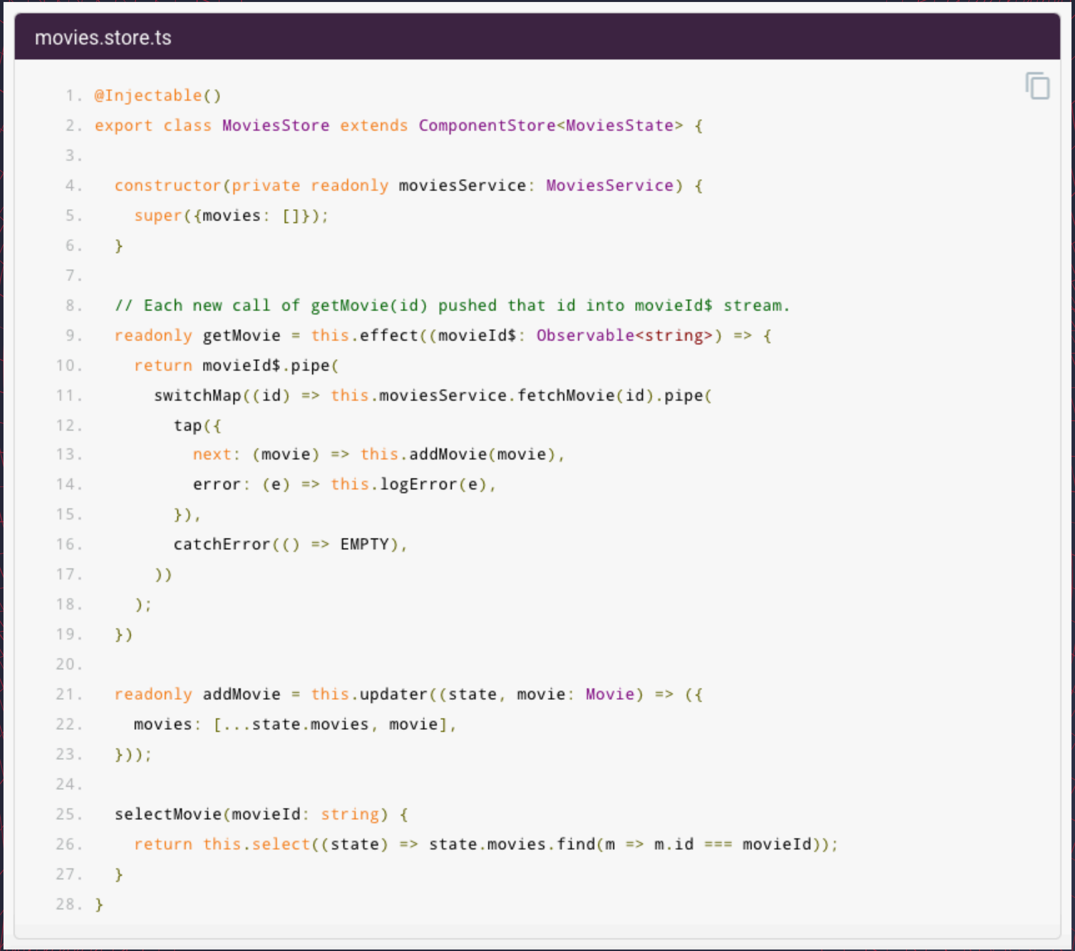
```
const userReducer = createReducer(
  initialState,
  on(UserActions.addUser, (state, { user }) => {
    return adapter.addOne(user, state)
  }),
  on(UserActions.setUser, (state, { user }) => {
    return adapter.setOne(user, state)
  }),
  on(UserActions.upsertUser, (state, { user }) => {
    return adapter.upsertOne(user, state);
  }),
  on(UserActions.addUsers, (state, { users }) => {
    return adapter.addMany(users, state);
  }),
  on(UserActions.upsertUsers, (state, { users }) => {
    return adapter.upsertMany(users, state);
  }),
  on(UserActions.updateUser, (state, { update }) => {
    return adapter.updateOne(update, state);
  }),
  on(UserActions.updateUsers, (state, { updates }) => {
    return adapter.updateMany(updates, state);
  }),
  on(UserActions.mapUser, (state, { entityMap }) => {
    return adapter.map(entityMap, state);
  }),
  on(UserActions.mapUsers, (state, { entityMap }) => {
    return adapter.map(entityMap, state);
  }),
```



```
  on(UserActions.clearUsers, state => {
    return adapter.removeAll({ ...state, selectedUserId: null });
  })
);

export function reducer(state: State | undefined, action: Action) {
  return userReducer(state, action);
}

export const getSelectedUserId = (state: State) => state.selectedUserId;

// get the selectors
const {
  selectIds,
  selectEntities,
  selectAll,
  selectTotal,
} = adapter.getSelectors();

// select the array of user ids
export const selectUserIds = selectIds;

// select the dictionary of user entities
export const selectUserEntities = selectEntities;

// select the array of users
export const selectAllUsers = selectAll;

// select the total user count
export const selectUserTotal = selectTotal;
```

@ngrx/component-store

ComponentStore is a stand-alone library that helps to manage local / component state. It's an alternative to reactive push-based "Service with a Subject" approach.


movies.store.ts

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

  constructor(private readonly moviesService: MoviesService) {
    super({movies: []});
  }

  // Each new call of getMovie(id) pushed that id into movieId$ stream.
  readonly getMovie = this.effect((movieId$: Observable<string>) => {
    return movieId$.pipe(
      switchMap((id) => this.moviesService.fetchMovie(id).pipe(
        tap({
          next: (movie) => this.addMovie(movie),
          error: (e) => this.logError(e),
        }),
        catchError(() => EMPTY),
      ))
    );
  })

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

  selectMovie(movieId: string) {
    return this.select((state) => state.movies.find(m => m.id === movieId));
  }
}
```

03
Data

@ngrx/data

- automates the creation of actions, reducers, effects, dispatchers, and selectors for each entity type
- provides default HTTP GET, PUT, POST, and DELETE methods for each entity type
- holds entity data as collections within a cache which is a slice of NgRx store state
- supports optimistic and pessimistic save strategies
- enables transactional save of multiple entities of multiple types in the same request
- makes reasonable default implementation choices
- offers numerous extension points for changing or augmenting those default behaviors

04

View

@ngrx/component

Component is a set of primitive reactive helpers to enable fully reactive, Zoneless applications. They give more control over rendering, and provide further reactivity for Angular applications.

THANKS!

Does anyone have any questions?

blog.hg-world.com