Advanced State Management using NgRx

Agenda

Advanced State Management Overview

Introduction to the Redux Pattern

Understanding & Using NGRX

Using Store, Actions and Reducers

Using Effects

Using NgRx Entity

State Management Overview

State Managing Options

State Managing Options for Angular / Ionic Applications:

- No Centralized State – Fetch Data on every request

Manage State using Services / Providers

- Stateful Services

Use a State Management Library

- ngrx

Observable-Store ...

What is State?

Data from REST Apis

User Information

- Profile, Token, ...

User Input

- SecectedItems, Filters, ...

UI State (is some Component, Tag visible | not)

Router / Location State

Stateful Services

Might reduce Round Trips to Server

Steps to implement:

- Get Remote Data
- Store in local Array
- Expose as BehaviorSubject
  - Keeps last State for late subscribers
  - Call next() on change to local array

```
private arrMarkers: Marker[] = [];
private markers: BehaviorSubject<Marker[]> = new BehaviorSubject(
  this.arrMarkers
);

private loadMarkersRemote() {
  this.httpClient
    .get<Marker[]>("http://localhost:5000/api/markers")
    .subscribe(data => {
      this.arrMarkers = data;
      this.markers.next(this.arrMarkers);
    });
}
```

Introduction to Redux

Redux

Redux is a predictable state container for JavaScript apps.

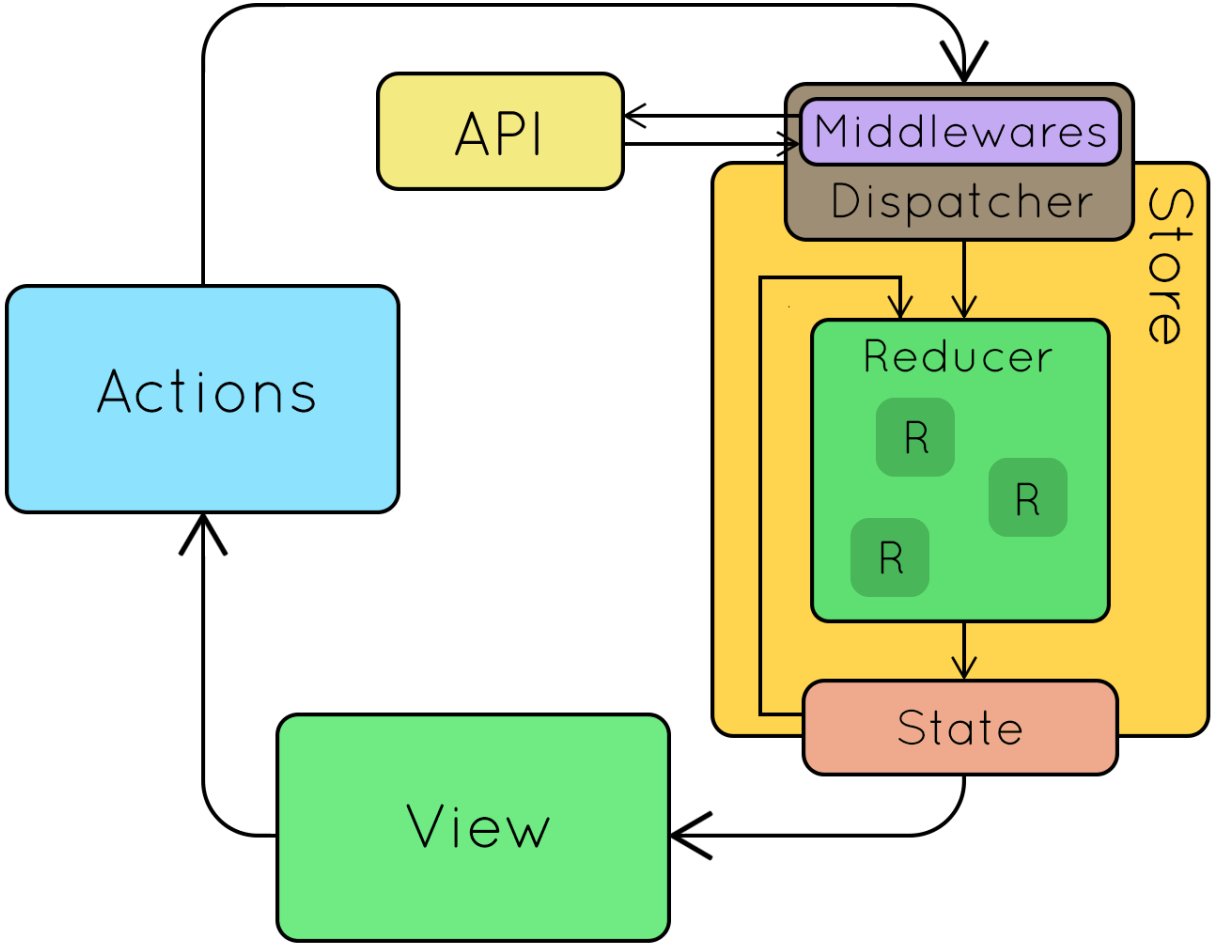
Defines an app as initial state followed by a series of actions

Each action reduces state to a new predictable state

To enable state changes to be predictable, the following constraints applied to state changes

- Single Source of Truth
- State is Immutable (replace with new state)
- Changes are made with Pure Functions

Redux Flow


API
Middlewares
Dispatcher
Store
Actions
Reducer
R
R
R
State
View

Single Source of Truth

Following the pattern of Flux, all data flows through a Redux system in a unidirectional matter

All changes to the state comes from actions applied to the state, and all actions are funneled into Redux

No part of the system can ever receive data from two sources

Additionally, the state managed by Redux is the state of the whole application (with minor exceptions, such as form control entry)

State is Read-Only

State can never be mutated

New states are produced by applying an action to the current state (known as reduction) from which a new state object is produced

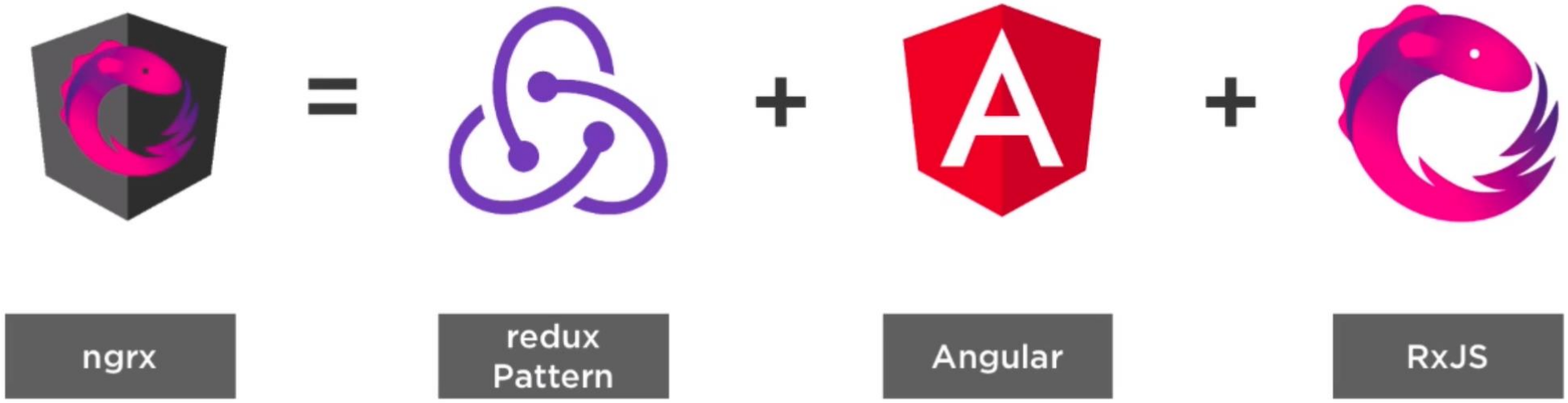
Immutable programming techniques need to be utilized

ngrx

What is ngrx

NgRx is a group of Angular libraries for Reactive Extensions for JavaScript

Not recommended to use ngrx for small projects because of overhead



Base Elements

Store -> Holds State

Action -> Trigger Actions applied by Reducers

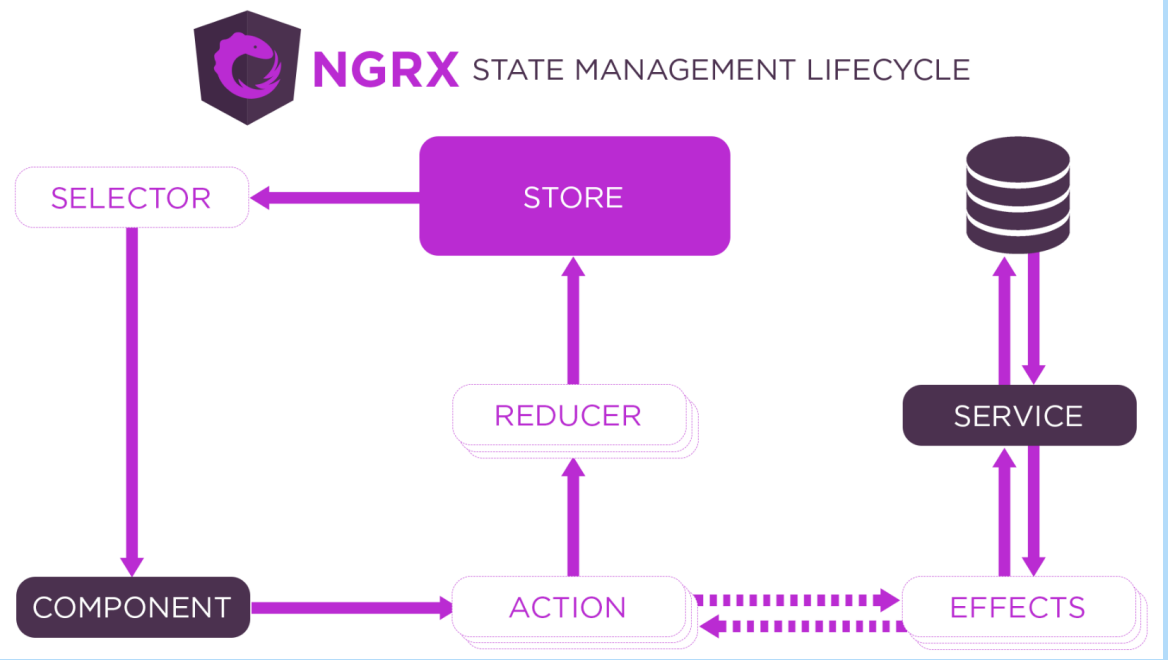
Reducers -> Apply Modifications

Effects -> Communicate with Api

Selectors -> Slice State

Facades -> Abstract Logic, Dispatch Actions

Schematics -> Lift burden of writing Boilerplate



ngrx - Libraries

Typically the following packages will be installed as a minimal configuration:

- @ngrx/core
- @ngrx/store
- @ngrx/effects

Additional Packages:

- @ngrx/entity
- @ngrx/data

Dev Tooling

- @ngrx/store-devtols
- @ngrx/schematics

@ngrx/schematics

Schematics allow us to scaffold ngrx-elements just like the Angular CLI

- ng generate store State
- ng generate action ActionName
- ng generate reducer ReducerName
- ...
- Installation:
  - ng add @ngrx/schematics
  - Switches default Schematics collection in angular.json

@ngrx/store

Benefits of @ngrs/store:

- Single source of truth
- Testability
- Performance benefits
  - Change DetactionStrategy.OnPust
  - Immutable @Inputs
- Root and Feature Module support

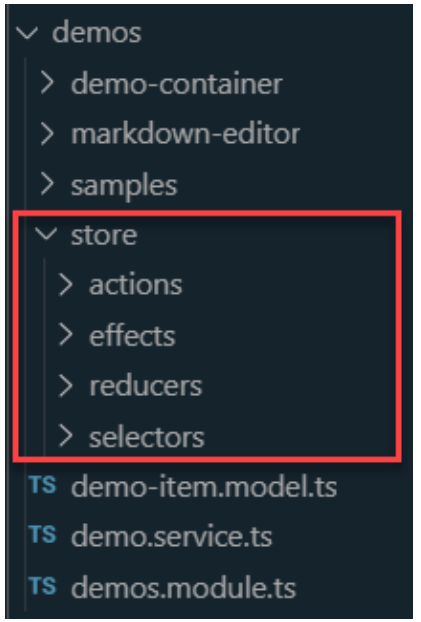
Create Store

Store is composed by one ore more State Slices

- RootState
- FeatueState: DemoState, SkillState, ...

Must be registered in

- Root Module
  - StoreModule.forRoot({}),
- Feature Modules
  - Might use a predefined slice of the state
  - StoreModule.forFeature(demo_slice, reducer),
- Scaffolding:
  - ng g store State --root true | false




```ts
export interface State {
  app: AppState;
  routerReducer: RouterReducerState<RouterStateUrl>;
  auth: AuthState;
  // demos: DemoState  -> Lazy Loaded
}
```

Actions

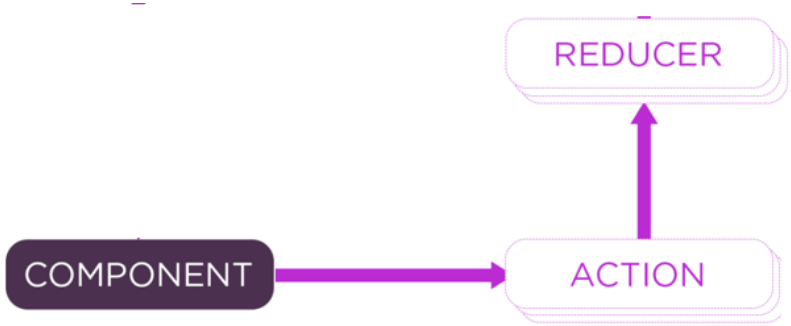
Actions represent the events with optional payloads that are sent from the Component to the Store to execute Reducers. Example: Add Skill

Scaffolding:

- ng g action store/skill --group

Consist of

- ActionTypes: Used to identify the Action
- ActionCreators: Create the Action with op
- Action Union Types



Actions - Elements

- ActionTypes:
  - Used to identify the Action
- ActionCreators:
  - Create the Action
  - Payload is assigned using constructor
- Action Union Types
  - Used in xx

```
export enum SkillActionTypes {
  LoadSkills = "[Skill] Load Skills",
  AddSkill = "[Skill] Add Skills"
}

export class LoadSkills implements Action {
  readonly type = SkillActionTypes.LoadSkills;
}

export class AddSkill implements Action {
  readonly type = SkillActionTypes.AddSkill;
  constructor(public payload: Skill) {}
}

export type SkillActions = LoadSkills | AddSkill;
```

Reducers

Follows the pattern of the reduce function available on the Array prototype in JavaScript

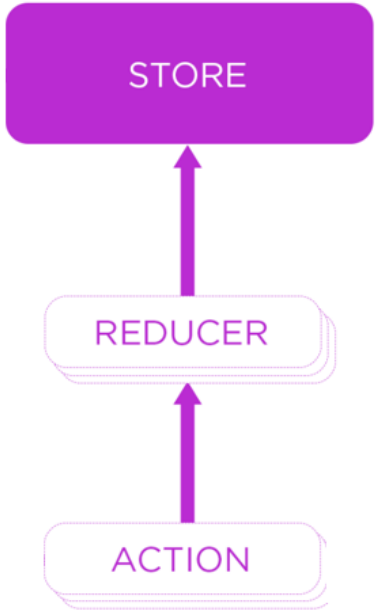
Receives the current state and an action, the function produces a new state based upon the type of action, and its associated data

Pure function – output results from inputs only, no side-effects

Should be configured to create an initial state during the first run

Scaffolding:

- ng g reducer store/skill --group



State

Holds the State of the Application

Implemented in the "reducers" folder

Consists of

- Feature Key
- State
- Initial State
- Feature Reducer

```
import { SkillActions, SkillActionTypes } from "../actions/skill.actions";

export const skillFeatureKey = "skill";

export const initialState = [{ id: "123", name: "ngrx" }];

export function SkillReducer(state = initialState, action: SkillActions) {
  switch (action.type) {
    case SkillActionTypes.AddSkill:
      return [...state, action.payload];
    default:
      return state;
  }
}
```

Selectors

Selectors are pure functions used for obtaining slices of store state

Two major implementations:

- createFeatureSelector
  - The createFeatureSelector is a convenience method for returning a top level feature state
- createSelector
  - Can be used to select some data from the state based on several slices of the same state

Selectors – Code Chart

```ts
export interface SkillsState {
  skills: Skill[];
  loading: boolean;
  loaded: boolean;
}
```

```ts
//Selectors
export const getSkillsLoaded = (state: SkillsState) => state.loaded;
export const getSkills = (state: SkillsState) => state.skills;
```

```ts
export const getSkillsState = createFeatureSelector<fromSkills.SkillsState>(
  stateFeatureKey
);

export const getSkillData = createSelector(
  getSkillsState,
  (state: fromSkills.SkillsState) => state.skills
);
```

```html
<div
  *ngFor="let s of skills$ | async"
  class="item"
  fxLayout="row"
  fxLayoutAlign="space-between center"
  fxFill
>
  <div fxFlex="3 1 auto" style="padding-left: 2rem">{{ s.name }}</div>
  <div fxFlex="1 1 140px">…
  </div>
  <div fxFlex="1 1 80px">…
  </div>
```

```ts
export class SkillsListComponent implements OnInit {
  constructor(private store: Store<SkillsState>) {}

  skills$: Observable<Array<Skill>> = this.store
    .select(getSkillData)
    .pipe(tap(data => console.log("data received from store", data)));
```

Register State

State has to be registers in Root Module (app.module.ts) or Feature Module

Root Modules

- StoreModule.forRoot({})

Feature Modules

- StoreModule.forFeature({})

```
@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    BrowserModule,
    FormsModule,
    BrowserAnimationsModule,
    HttpClientModule,
    MaterialModule,
    StoreModule.forRoot({ skills: SkillReducer }),
    !environment.production ? StoreDevtoolsModule.instrument() : []
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

ActionReducerMap

Combines Reduces from Root and Feature Modules

Reducers of Lazy Loaded Modules is combined when loading the Modules

Root Reducer Map is typically implemented in 'src/app/store/reducers/index.ts'

```
export const reducers: ActionReducerMap<State> = {
  app: AppReducer,
  auth: AuthReducer,
  routerReducer: routerReducer
};
```

Effects

Represent Async Actions Middleware implemented as Service

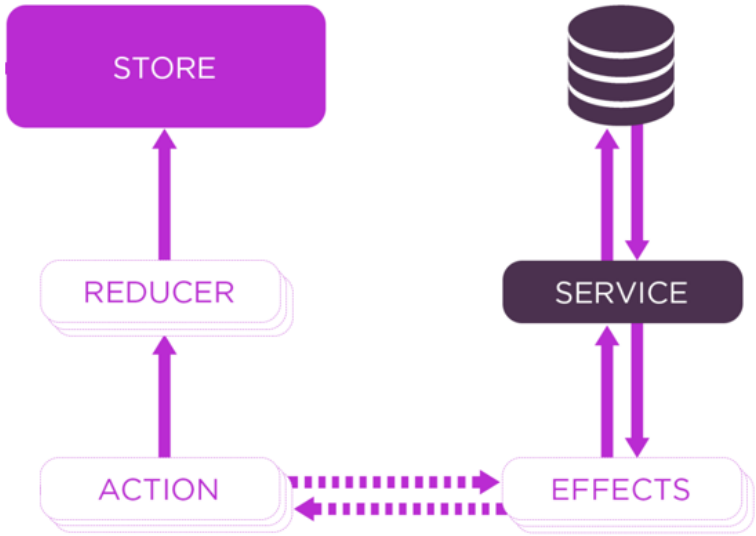
Effect typically use Service to communicate with the Database / API

Effects can

- Succeed
- Throw Errors

Effect might trigger other Actions

- … which might use Reducers to update State
- … Subscribers of this State the will be updated



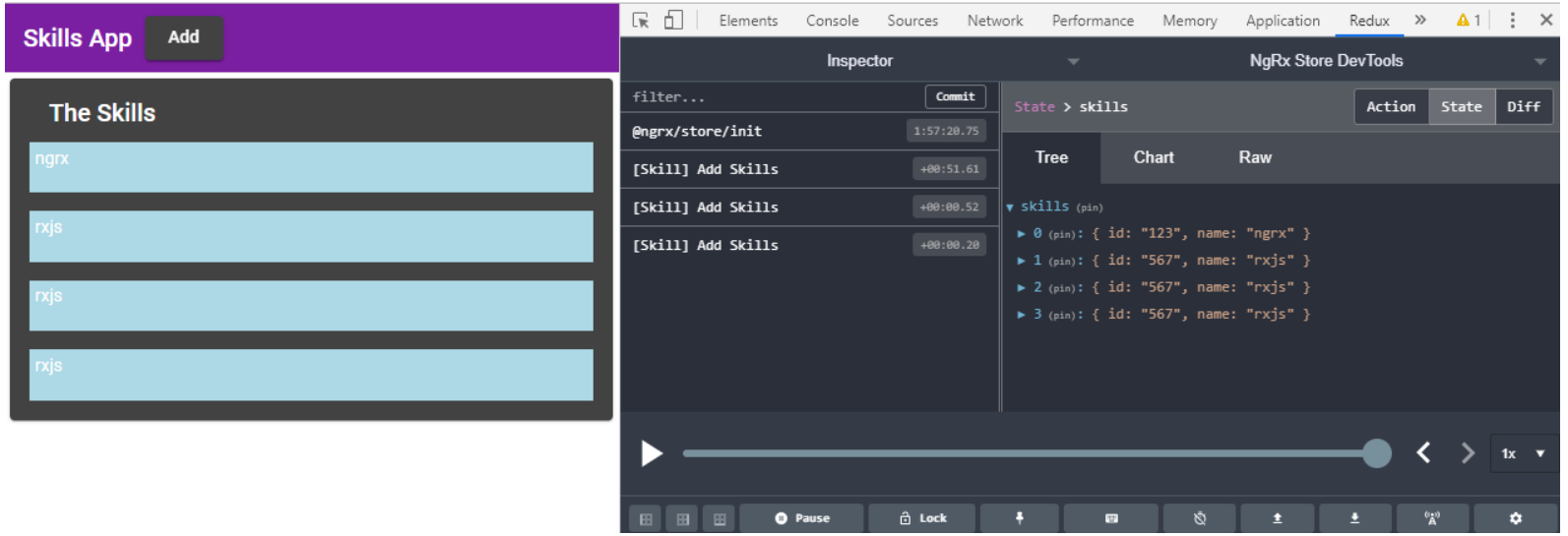
Redux Dev Tools

Redux Dev Tools

Enable Easy NgRx Debugging

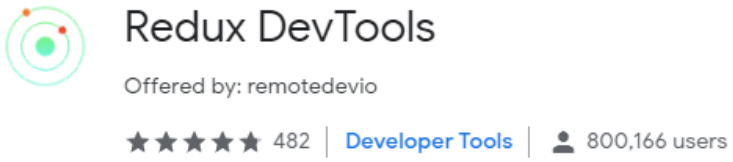
Features:

- Live Debugging
- Time Travel
- State Investigation
- Save / Restore State
- Dispatch Actions



Dev Tools - Tooling

- @ngrx/store-devtools
  - Typically register for dev build only
- Chrome Redux Dev Tools Extension

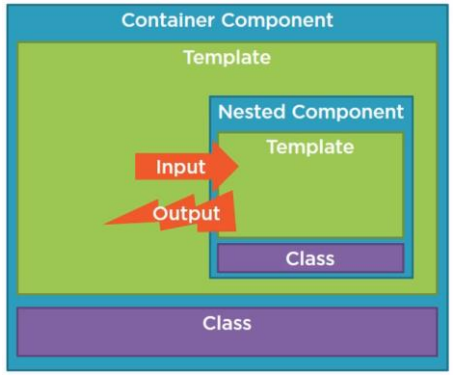



```
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule, HttpClientModule, StoreModule.forRoot(reducers, {
      metaReducers,
      runtimeChecks: {
        strictStateImmutability: true,
        strictActionImmutability: true,
      }
    }),
    !environment.production ? StoreDevtoolsModule.instrument() : []],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Container / Presentational

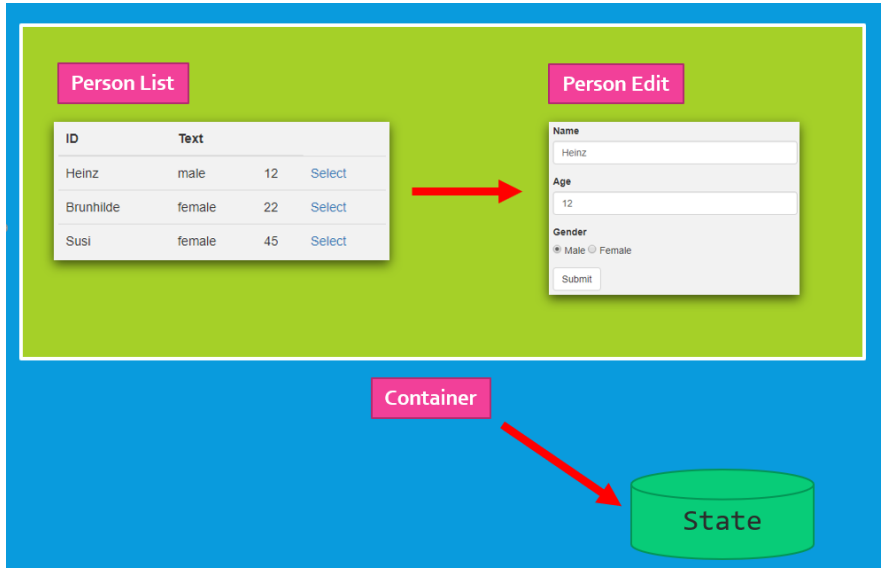
Container Component:

- Aware of Store
- Dispatches Actions
- Reads data from Store



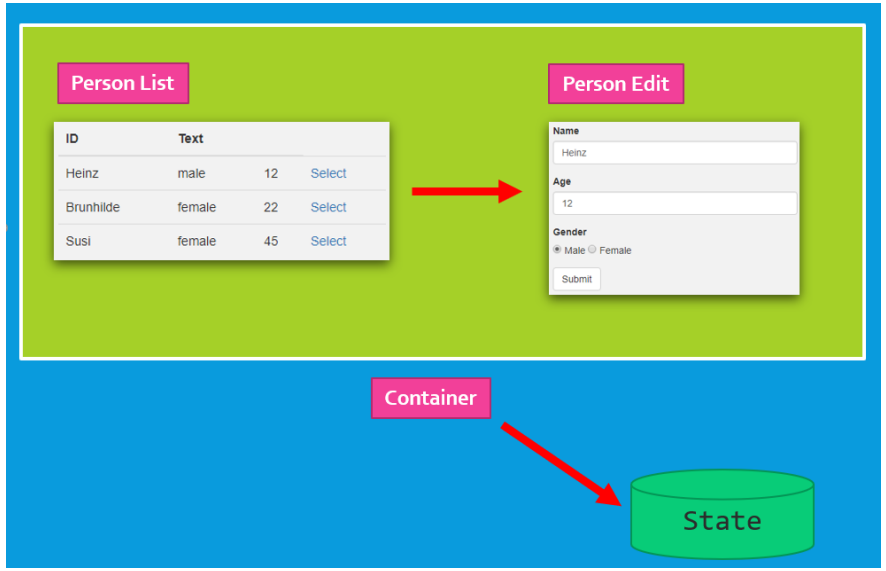
Presentational Component:

- Not aware of Store
- Simple Rendering
- Exchanges Data with Container using
  - @Input
  - @Output

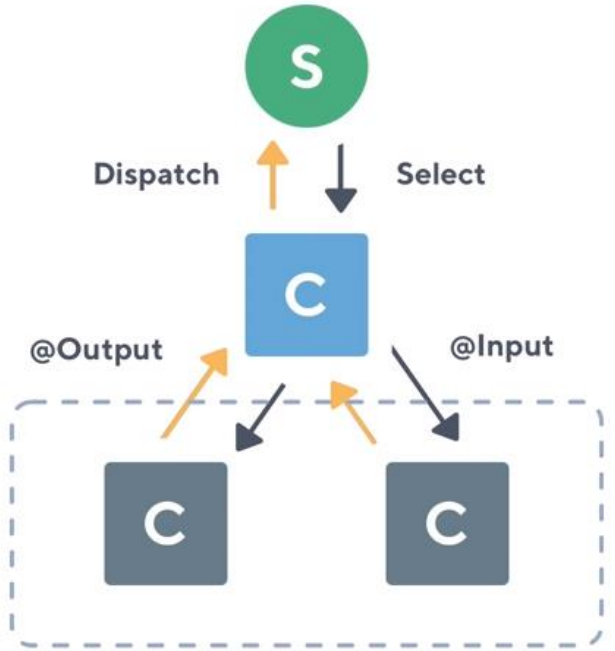


Container / Presentational using NgRx

Component View



NgRx View



@ngrx/entity

ngrx Entity

Entity State adapter for managing record collections

- Reduces boilerplate for creating reducers that manage a collection of models.
- Provides performant CRUD operations for managing entity collections.
- Provides bascic Selectors (idSelector) & configurable sortComparer

Consists of:

- EntityState<T>
- createEntityAdapter<T>

```
//internal entity structure
interface EntityState<T> {
  ids: string[];
  entities: { [id: string]: T };
}
```

Arrays vs Entity

When working with many items looping over the array is slow compard to using maps

Need to transform Arrays (we get from Apis) to Entity Structure

When binding we need to transform back

```
const entities = skills.reduce(
  (entities: { [id: number]: Skill }, skill: Skill) => {
    return {
      ...entities,
      [skill.id]: skill
    };
  },
  {
    ...state.entities
  }
);
```

```
skills: [
  { id: "123", name: "rxjs", completed: true },
  { id: "456", name: "ngrx", completed: false }
],
```

```
let target = {
  "123": {
    "id": "123",
    "name": "rxjs from api",
    "completed": true
  },
  "456": {
    "id": "456",
    "name": "ngrx from api",
    "completed": false
  }
}
```

Changes:

Change Reducer to return and deal with Entities

Change Selectors to Entities

- Change Structure of Result back to Array

```
export const getSkillsEntities = createSelector(
  getSkillsState,
  (state: fromSkills.SkillsState) => state.entities
);

export const getSkills = createSelector(
  getSkillsEntities,
  entities => {
    return Object.keys(entities).map(id => entities[parseInt(id, 10)]);
  }
);
```

Entity Adapter

Entity State adapter for managing record collections.

@ngrx/entity provides an API to manipulate and query entity collections.

- Reduces boilerplate for creating reducers that manage a collection of models.
- Provides performant CRUD operations for managing entity collections.
- Extensible type-safe adapters for selecting entity information.

Entity Adapter Methods

The adapter methods behave in the following way:

- addOne: add one entity to the collection
- addMany: add several entities
- addAll: replaces the whole collection with a new one
- removeOne: remove one entity
- removeMany: removes several entities
- removeAll: clear the whole collection
- updateOne: Update one existing entity
- updateMany: Update multiple existing entities
- upsertOne: Update or Insert one entity
- upsertMany: Update or Insert multiple entities

Entity Selectors

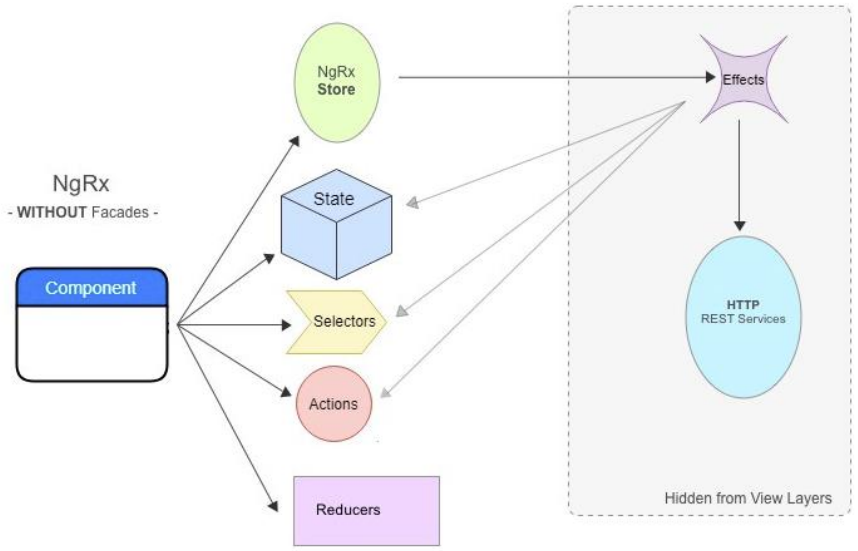
NgRx entity provides commonly needed selectors:

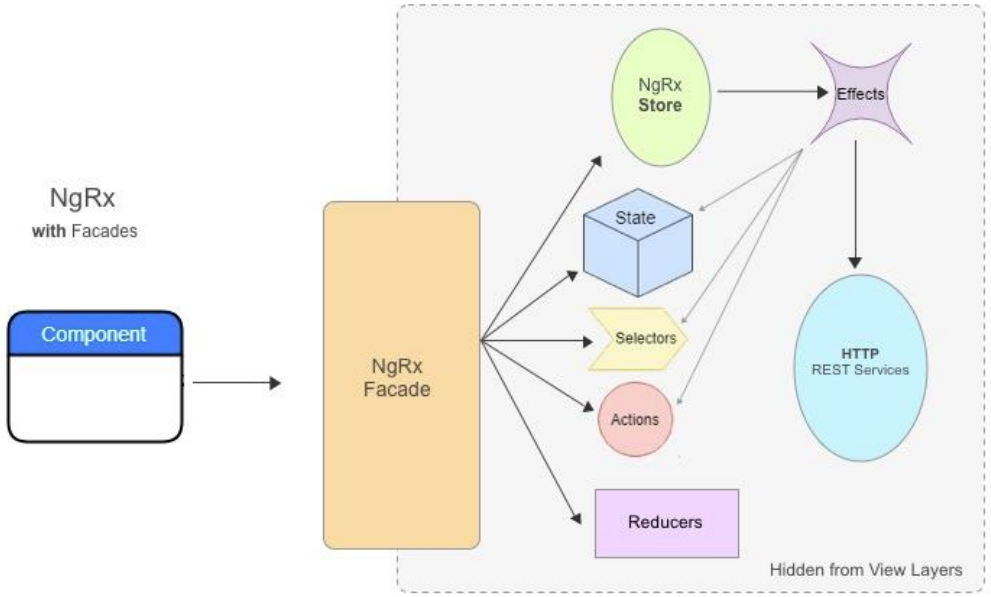
- selectAll,
- selectEntities,
- selectIds,
- selectTotal

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

NgRX Facades

A programming pattern in which a simpler public interface is provided to mask a composition of internal, more-complex, component usages





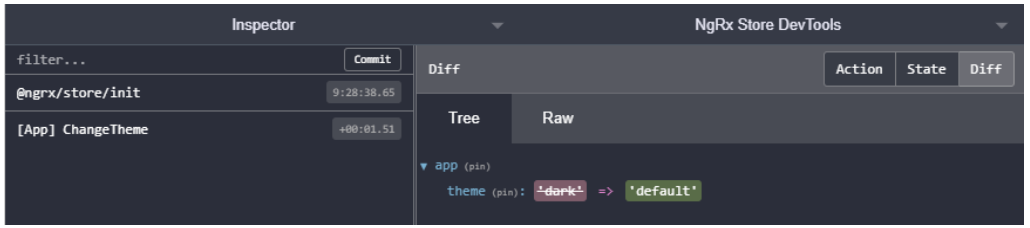
Using Facades

Refactoring code from Statefult Services to NgRx Facades is a good exercise to get Started

- Similar Code Patterns, easy to migrate

```
@Injectable({
  providedIn: "root"
})
export class ThemeService {
  constructor() {}

  private theme: string = "default";
  private currTheme: BehaviorSubject<string> = new BehaviorSubject(this.theme);

  toggleTheme() {
    this.theme = this.theme == "default" ? "dark" : "default";
    console.log("curr theme:", this.theme);
    this.currTheme.next(this.theme);
  }

  getTheme(): Observable<string> {
    return this.currTheme;
  }
}
```

```
@Injectable({
  providedIn: 'root'
})
export class ThemeFacade {
  constructor(private store: Store<AppState>) {}

  theme = this.store.select(getCurrentTheme);

  toggleTheme() {
    this.store.dispatch(new ChangeThemeAction(''));
  }
}
```



Action Creators

Creates a configured Creator function that, when called, returns an object in the shape of the Action interface

Reduce Boilerplate that has to be written for:

- ActionType Enum
- Action
- ActionUnion Type

```
export enum SkillActionTypes {
  LoadSkills = '[Skill] Load Skills',
}

export class LoadSkillsAction implements Action {
  readonly type = SkillActionTypes.LoadSkills;
}

export type SkillActionsUnion =  LoadSkillsAction
```

Implement Action Creators

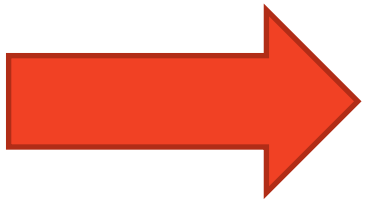
Action Creators can be implemented by:

- Specifying a Type
- Using Props
- With a function

```
export const loadSkills = createAction('[Skill] Load Skills');

export const loadSkillsSuccess = createAction(
  '[Skill] Load Skills Success',
  props<{ skills: Skill[] }>()
);
```

Simplified Reducers

```
export function SkillReducer(
  state: SkillsState = initialState,
  action: SkillActionsUnion
) {
  switch (action.type) {
    case SkillActionTypes.LoadSkills: {
      return { ...state, loading: true };
    }
    case SkillActionTypes.LoadSkills_Success: {
      return skillAdapter.setAll(action.payload, {
        ...state,
        loading: false,
        loaded: true,
      });
    }
    case SkillActionTypes.LoadSkills_Error: {
      return { ...state, loaded: false, loading: false };
    }
    case SkillActionTypes.AddSkill:
      return skillAdapter.addOne(action.payload, { ...state });
    case SkillActionTypes.DeleteSkill:
      return skillAdapter.removeOne(action.payload.id, { ...state });
    case SkillActionTypes.ToggleComplete:
      const updateSkill: Update<Skill> = {
        id: action.payload.id,
        changes: { completed: !action.payload.completed },
      };
      return skillAdapter.updateOne(updateSkill, { ...state });
    default:
      return state;
  }
}
```



```
export const skillReducer = createReducer(
  initialState,
  on(loadSkills, (state, action) => {
    return { ...state, loading: true };
  }),
  on(loadSkillsSuccess, (state, action) => {
    return skillAdapter.setAll(action.skills, {
      ...state,
      loading: false,
      loaded: true,
    });
  }),
  on(loadSkillsError, (state, action) => {
    return { ...state, loaded: false, loading: false };
  }),
  on(addSkill, (state, action) => {
    return skillAdapter.addOne(action.skill, { ...state });
  }),
  on(deleteSkill, (state, action) => {
    return skillAdapter.removeOne(action.skill.id, { ...state });
  }),
  on(toggleComplete, (state, action) => {
    const updateSkill: Update<Skill> = {
      id: action.skill.id,
      changes: { completed: !action.skill.completed },
    };
    return skillAdapter.updateOne(updateSkill, { ...state });
  })
);
```

Going Further

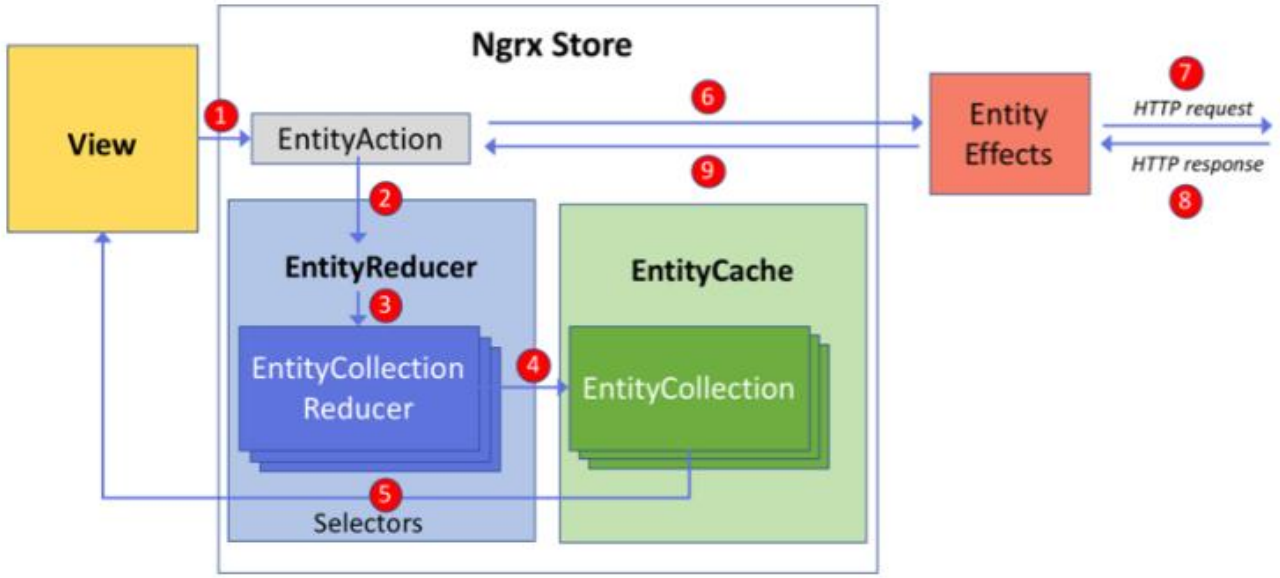
@ngrx/data

NgRx Data is an extension that offers a gentle introduction to NgRx by simplifying the management of entity data while reducing the amount of explicitness

Optimized for entities with 50.000+ items

Provides:

- CRUD on Entity Collection
- Filter, Sort
- Optimistic & Pessimistic Save



NgRx Auto-Entity

An add-on library for @ngrx that aims to greatly simplify use of @ngrx and reduce the implementation load of adding @ngrx to your application

Provides a set of ready-made, generic actions that cover most of the common use cases such as:

- loading entities and sets of entities
- creating,
- updating or
- replacing, and
- deleting entities