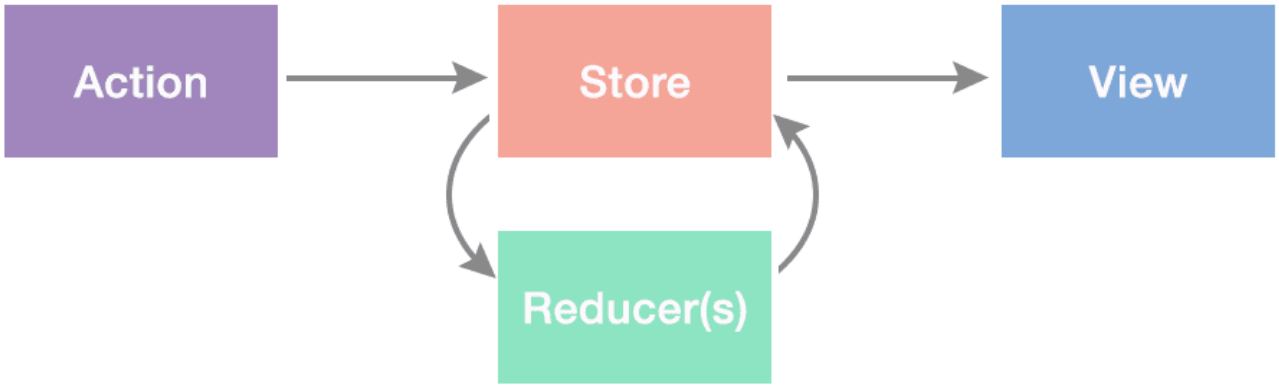
NgRx

Intérêt de NGRX

- Le 1er avantage est le modèle **unidirectionnel** avec lequel nous travaillerons, ce qui n'est pas le cas du standard MVC qui est **bidirectionnel**.
- Le 2ème avantage est **"l'historisation"**. Comme tous les changements transitent par le store, chaque update/modification est loggé. Grâce à cela, nous pouvons remonter dans l'historique et trouver quelle mutation a créer un bug : c'est une sorte de **state machine**.



Termes

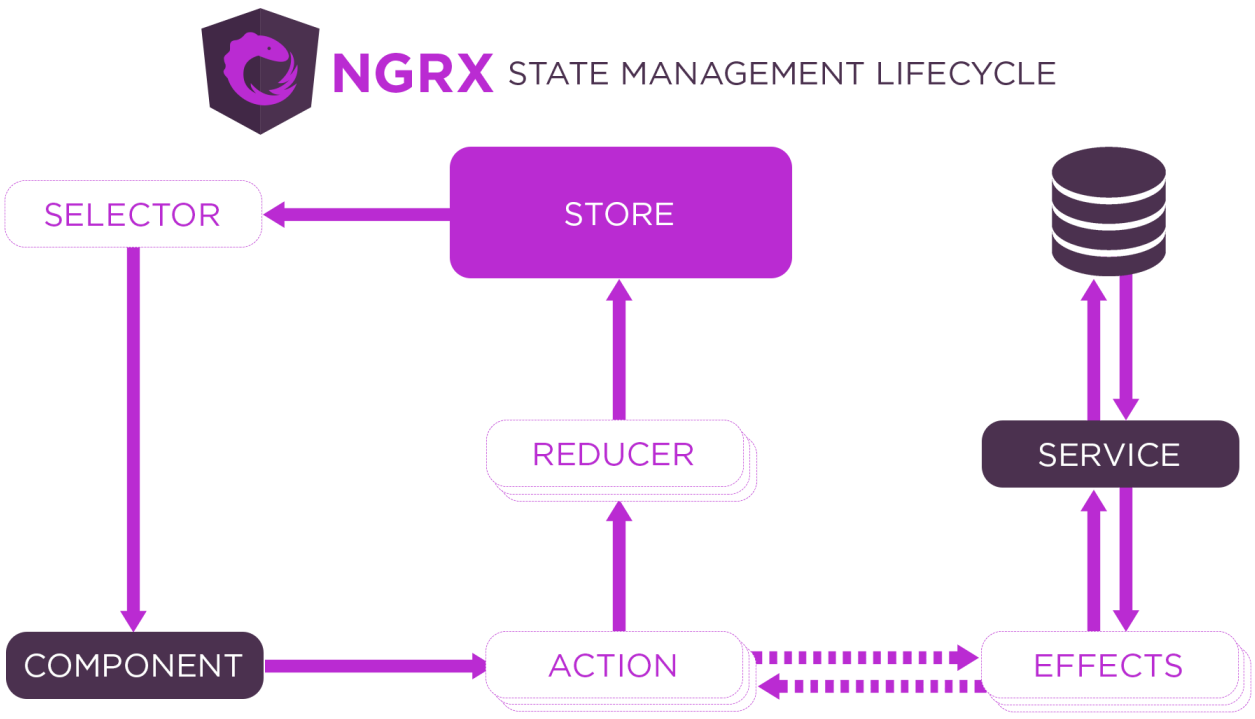
store : Le magasin est ce qui contient l'état de l'application.

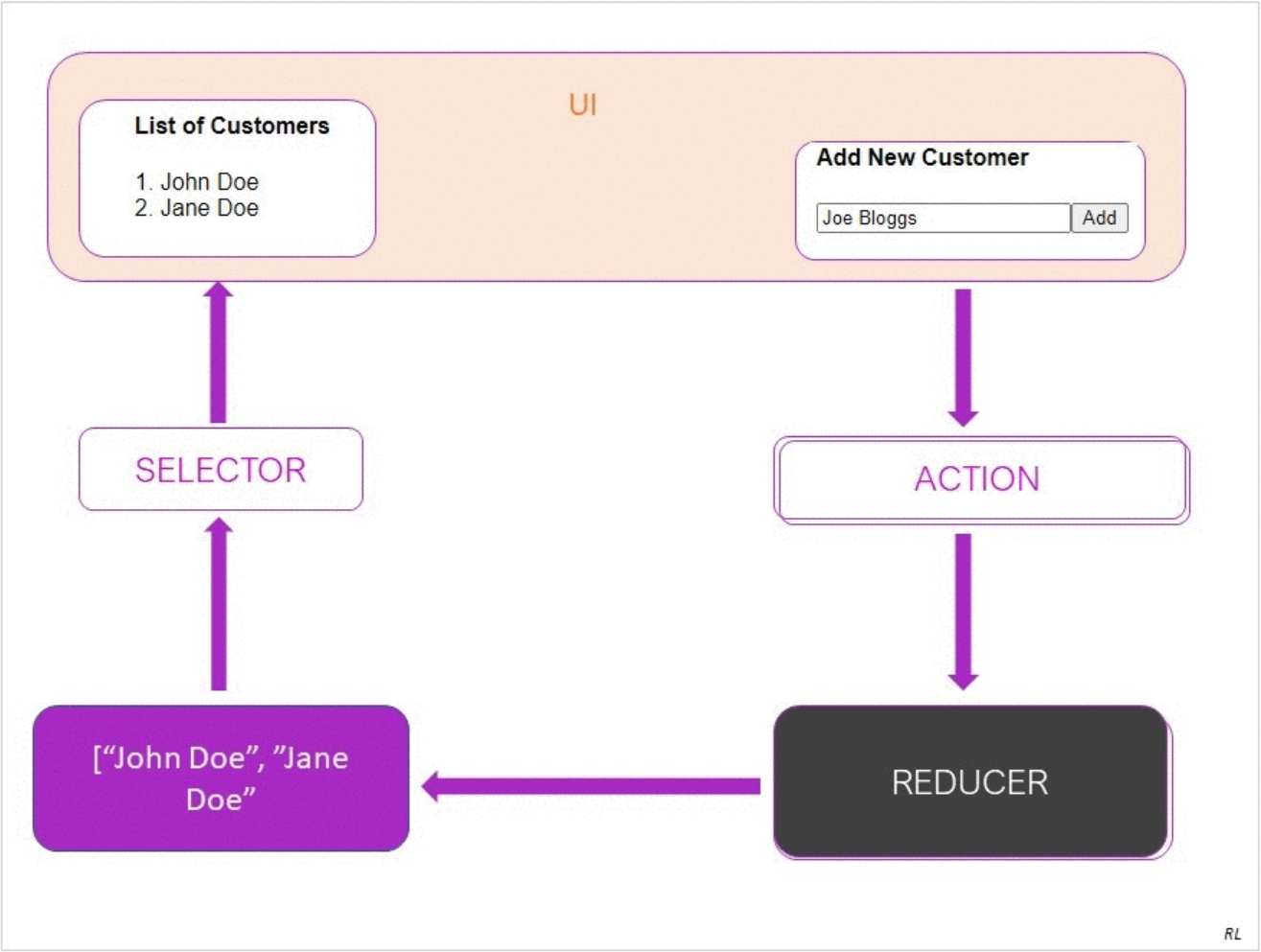
action : Un événement unique envoyé par les composants et les services qui décrivent comment l'état doit être modifié. Par exemple, "Ajouter un client" peut être une action qui modifiera l'état (c'est-à-dire ajouter un nouveau client à la liste).

reducer : Tous les changements d'état se produisent à l'intérieur du réducteur ; il répond à l'action et, en fonction de cette action, il créera un nouvel état immuable et le renverra au magasin.

selector : Le sélecteur est une fonction utilisée pour obtenir une partie de l'état à partir du magasin.

effect : Un mécanisme qui écoute les actions envoyées dans un flux observable, traite la réponse du serveur et renvoie les nouvelles actions soit immédiatement, soit de manière asynchrone au réducteur pour changer l'état.


NGRX STATE MANAGEMENT LIFECYCLE
SELECTOR
STORE
REDUCER
SERVICE
COMPONENT
ACTION
EFFECTS


UI
List of Customers
1. John Doe
2. Jane Doe
Add New Customer
Joe Bloggs
Add
SELECTOR
ACTION
["John Doe", "Jane Doe"
REDUCER
RL

Installation

```
npm install @ngrx/store
```

Store

> Notez que les codes si-dessous sont minimalistes. Il permet d'avoir des exemples pour chaque partie. Nous allons détailler la structure et les types durant le live coding de la formation

Pour simplifier, supposons que nous ayons une liste de clients dans l'application, et c'est l'état que nous essayons de gérer. Certains appels API et certaines entrées utilisateur pourraient modifier l'état (c'est-à-dire la liste) en ajoutant ou en supprimant des clients. Le changement d'état devrait se refléter dans l'interface utilisateur et les autres éléments dépendants. Dans ce cas particulier, nous pouvons certainement disposer d'une variable globale pour conserver la liste, puis ajouter/supprimer des clients, puis écrire le code pour mettre à jour l'interface utilisateur et les dépendances. Mais cette conception comporte de nombreux pièges. Le NgRx est une excellente conception pour la plupart des besoins de gestion de l'État.

Création de l'état

Dans le dossier `store`

```
export interface State {
    title: string
}

export const initialState: State = {
    title: ''
}
```

Création d'une action

C'est juste fonction retournant un objet sous le format { type: string } (au minimum)

```
export const appLoading = () => {
    return {
        type: '[App] Loading'
    }
}
```

Peut être simplifier par la fonction `createAction`:

```
import { createAction } from "@ngrx/store"

export const appLoading = createAction('[App] Loading')
```

Note

Si nous souhaitons faire une action avec des données:

```
import { createAction, props } from "@ngrx/store"

export const appLoading = createAction('[App] Loading', props<{
    info: string
}>())
```

Création d'un reducer

C'est juste une fonction

```
export const appReducer = (state: State, action) => {
    if (action.type == '[App] Loading') {
        return {
            title: 'Mon App'
        }
    }
}
```

Peut être simplifier par la fonction `createReducer`:

```
import { createReducer, on } from '@ngrx/store'

export const appReducer = createReducer(
    initialState,
    on(appLoading, (state, action) => {
        return {
            title: 'Mon App'
        }
    })
)
```

Mettre dans le module

```
import { myReducer } from './store/app/app.reducer';

StoreModule.forRoot({ 'myapp': myReducer })
```

Utilisation dans le composant

Utiliser dans le composant:

1. Créer une propriété qui va récupérer que le titre (c'est un Observable)
2. Déclencher l'action avec `dispatch`

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { Store } from '@ngrx/store';
import { appLoading } from './store/app/app.action';

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styles: ['appt.component.css']
})
export class AppComponent implements OnInit {
  private title$: Observable<string> = this.store.select(state => state.myapp.title)

  constructor(
    private store: Store<{ myapp: any }>
  ) { }

  ngOnInit() {
      this.store.dispatch(appLoading())
  }
}
```

Dans le template

```
<h1>{{ title | async }}</h1>
```

Note

Nous pouvons déclencher l'action avec des données

```
 this.store.dispatch(appLoading({info: 'test' ))
```

Les effets

Installation préalable

```
npm install @ngrx/effects
```

Remarquez qu'il faut

- Un service AppService pour faire la requête

- Une reducer à nouveau pour faire une nouvelle action (appLoaded)

```
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { mergeMap, map, catchError } from 'rxjs/operators';
import { AppService } from '../core/services/app.service';
import { appLoaded } from './app.action';
import { throwError } from 'rxjs';

@Injectable({
    providedIn: 'root'
})
export class AppEffect {

    loadTitle$ = createEffect(() => {
        return this.actions$.pipe(
            ofType('Action Name'),
            mergeMap(() => this.appService.get()
                .pipe(
                    map(title => appLoaded({ title })),
                    catchError(val => throwError('error'))
                ))
        )
    })

    constructor(
        private actions$: Actions,
        private appService: AppService
    ) {}
 }
```

Mettre l'effet dans un module

```
import { AppEffect } from './store/app/app.effect'

EffectsModule.forRoot([AppEffect])
```

Création un selector

```
import { createFeatureSelector, createSelector } from "@ngrx/store";
import { State } from './app.reducer';

export const getAppState = createFeatureSelector<State>('mylist');

export const getTitle = createSelector(
    getTodoListState,
    (state: State) => state.title
)
```

Utilisation dans le composant

La propriété `title` devient dans le composant

```
private title$: Observable<string> = this.store.select(getTitle)
```