





# NGRX

- Bibliothèque de gestion de "state"(état) pour vos applications Angular
- Inspiré par Redux,
- Aide à gérer l'état de vos applications Angular de manière prévisible et centralisée

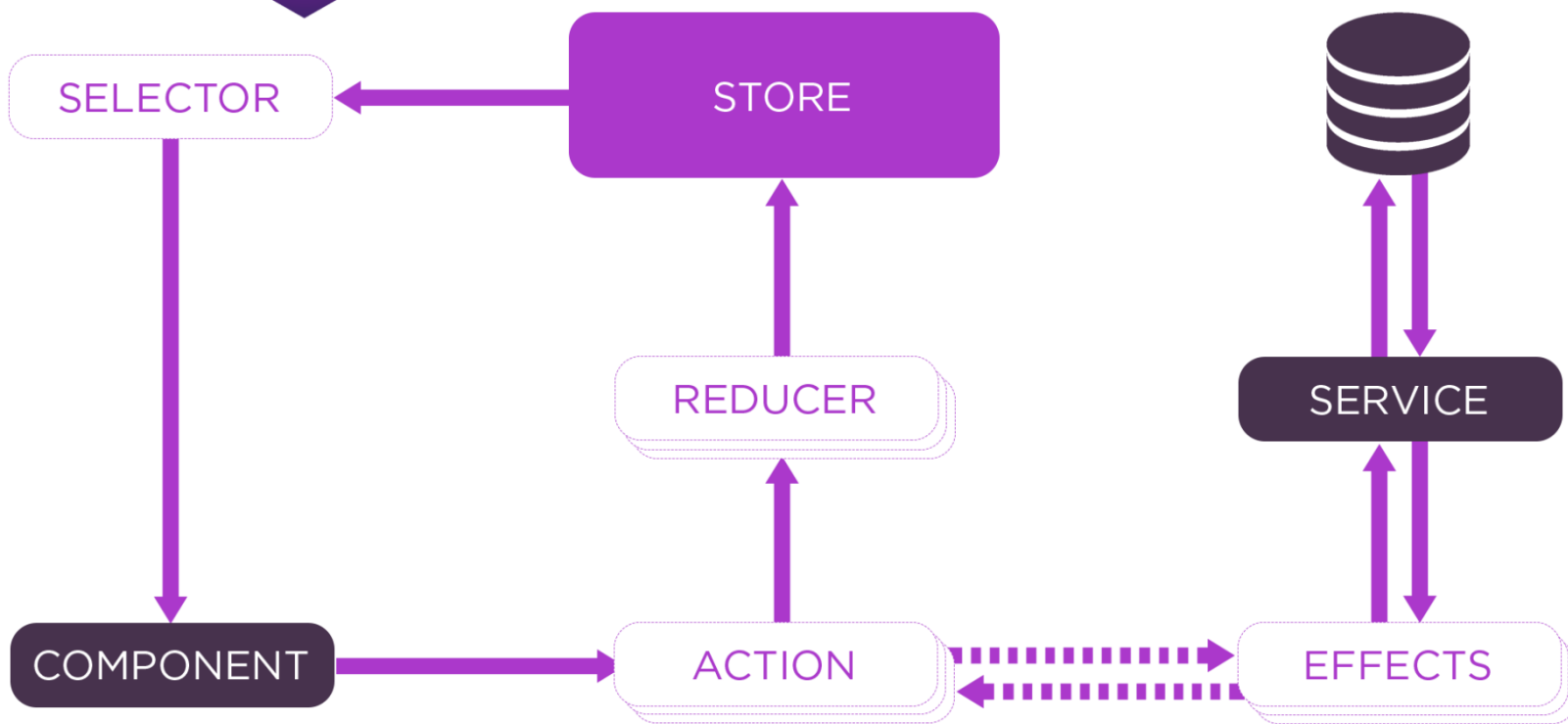
# NGRX

- Les composants clés de NgRx comprennent :
  - **Actions** : Représente des événements uniques ou des interactions utilisateur qui déclenchent des modifications de l'état de l'application.
  - **Reducers** : Des fonctions dite "pures" qui renvoient un nouvel état basé sur une action Les réducteurs sont responsables de la mise à jour de l'état en réponse aux actions.
    - Ont pour paramètre:
      - L'état actuel du store
      - Une action
  - **Selectors** : Des fonctions qui encapsulent la logique pour récupérer des morceaux spécifiques de votre store
  - **Effects** : Middleware de gestion d'effets secondaires tels que la récupération de données asynchrones. Les effets écoutent les actions dispatchées, effectuent des effets secondaires et puis dispatchent de nouvelles actions pour mettre à jour l'état.

# NGRX



## NGRX STATE MANAGEMENT LIFECYCLE



# NGRX

- NgRx

- D'abord, nous installons NgRx en utilisant la commande ng add, cela mettra à jour notre app.module.ts

```
ng add @ngrx/store
```

- Store-devtools

- Nous pouvons maintenant installer le package store-devtools, cela nous permettra de déboguer notre store dans notre navigateur. app.module.ts est à nouveau mis à jour.

```
ng add @ngrx/store-devtools
```

- Nous devons maintenant installer un plugin sur notre navigateur afin que nous puissions voir notre store dans l'onglet développeur

Chrome: <https://chromewebstore.google.com/detail/redux-devtools/lmhkpmbekcpmknklieibfkpmmfbljd?pli=1>

Firefox: <https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/>

Edge:

<https://microsoftedge.microsoft.com/addons/detail/reduxdevtools/nkkgneoiohoecpdiaponcejilbhhi-kei>

# NGRX: Store

Store: Gestion d'état globale orienté RxJS.

Par convention, le store est défini dans le fichier dit 'reducer' et enregistré dans le app.module.ts

```
//store/counter/counter.reducer.ts
export const initialState = 0;

export const counterReducer =
  createReducer(
    initialState,
  );
```

```
//app.module.ts
StoreModule.forRoot({countstore:
  counterReducer}, {}),
```

# NGRX: Lire une valeur

NgRx fournit un service appelé "Store" à injecter dans notre composant. La méthode 'select' retournera un observable qui émettra une valeur à chaque fois que le store change.


```
// counter.component.ts
```

```
count$: Observable<number>
```

```
constructor(private store: Store<number>) {  
    this.count$ = store.select('count');  
}
```

```
// Mise à jour de la vue:
```

```
Current Count: {{ count$ | async }}
```



# NGRX: Mise à jour du store

- Nous devons maintenant mettre à jour la valeur de notre store.
  - Le schéma classique des étapes est le suivant:
    - Le composant appelle la méthode "dispatch" avec une action en paramètre.
    - Le "reducer" déclenche le changement en fonction de l'action qui a été appelée.
    - Deux choses sont maintenant nécessaires :
      - Créer des actions que nous appellerons dans nos composants:
        - `counter.actions.ts`
      - Déclenche les changements lorsque ces actions sont appelées
        - `counter.reducer.ts`



# NGRX: Créer nos actions.

Action est composé d'une interface simple

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

- **En amont** : Ecriture des actions en premiere, cela permet d'avoir une vision global des besoins
- **Diviser** : catégoriser les actions en fonction des événements.
- **Grandes quantités d'actions** : Elles sont peu coûteuses à écrire, plus vous écrivez d'actions, mieux vous exprimez les évènements de votre application.
- **Descriptif** : fournissez un contexte ciblé sur un événement unique avec des informations: Plus simple a débbugger.

# NGRX: Créer nos actions.

Créons trois actions qui seront déclenchées par des clics sur nos boutons.

```
//counter.actions.ts

import { createAction } from '@ngrx/store';

export const increment = createAction('[Counter] Increment');
export const decrement = createAction('[Counter] Decrement');
export const reset = createAction('[Counter] Reset');
```

# NGRX: Mise à jour du reducer:

Définissons une fonction dans notre reducer afin de gérer les changements de valeur du compteur en fonction des actions.

Le reducer gère le nouvel état de manière immutable(!): retour d'une nouvelle valeur au lieu d'une modification.

```
export const counterReducer =
  createReducer(
    initialState,
    on(increment, (state) => state + 1),
    on(decrement, (state) => state - 1),
    on(reset, (state) => 0)
  );
```

# NGRX Dispatch des actions.

Nous appelons la méthode "dispatch" de notre "store" et nous passons une "action" pour déclencher notre mise à jour.

```
//app.component.ts
increment() {
  this.store.dispatch(increment())
}
reset() {
  this.store.dispatch(reset())
}
decrement() {
  this.store.dispatch(decrement())
}
```

# Passing values to our actions.

Pour l'instant, notre reducer exécute une action basée sur la valeur précédente de notre store. Et si maintenant, nous voulons mettre à jour le compteur en fonction d'une valeur définie à partir d'un formulaire par exemple?

Création d'une action avec une props comme deuxième paramètre.

```
export const updateValue = createAction('[Counter] Update Value',  
  props<{newValue: number}>());
```

Mise à jour du reducer:

```
...  
on(updateValue, (state, {newValue}) => newValue)  
...
```

Dispatch de l'action:

```
this.store.dispatch(updateValue({newValue: 50}))
```