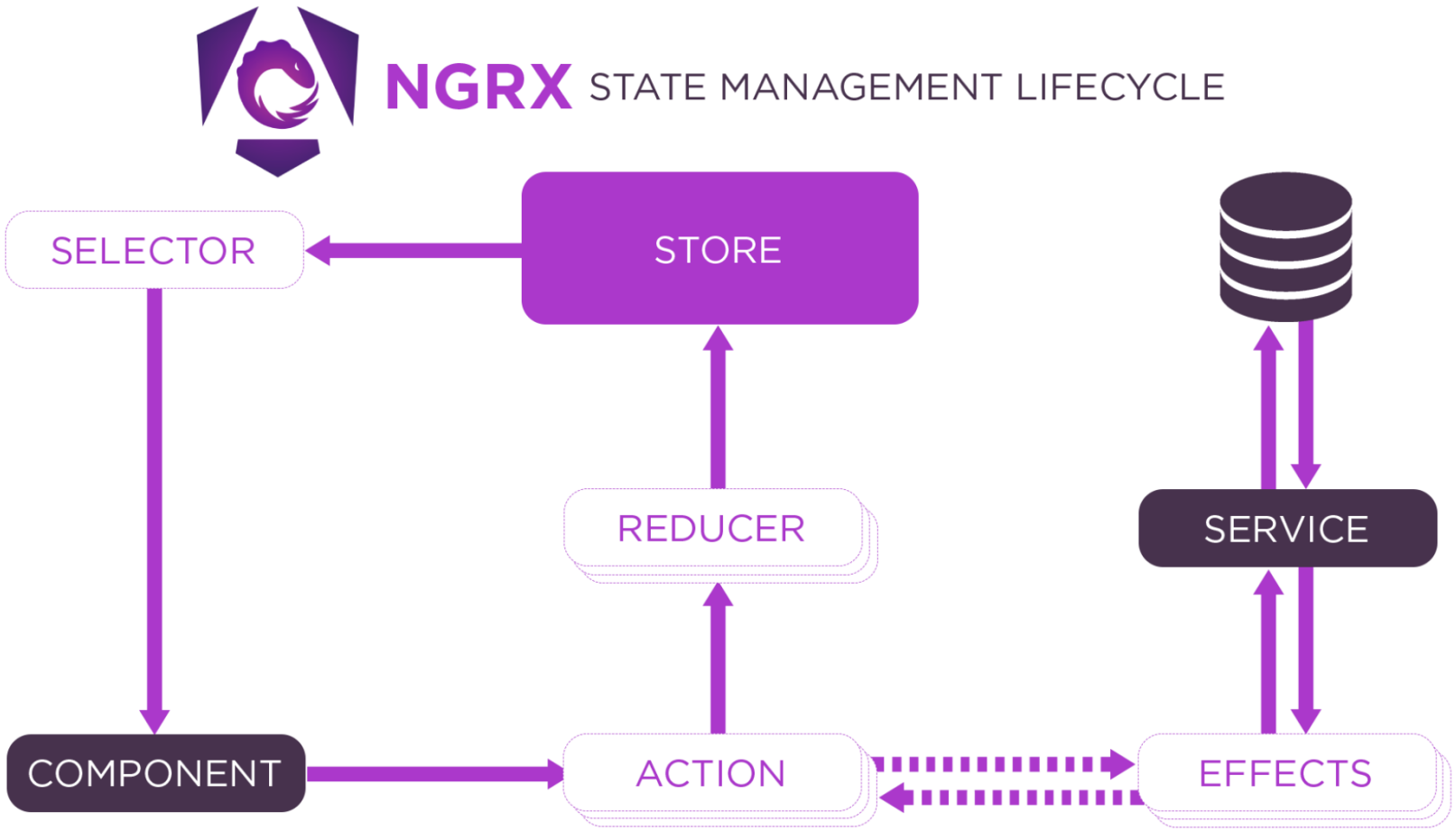
NGRX

- 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
SELECTOR
STORE
REDUCER
SERVICE
COMPONENT
ACTION
EFFECTS

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/lmhkpmbekcpmknklioeibfkpmmfibljd?pli=1

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

    Edge: https://microsoftedge.microsoft.com/addons/detail/reduxdevtools/nnkgneoiohoecpdiaponcejilbhhikei

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 shéma 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 évenements.
- **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 débugger.

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 été 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}))
```

NGRX: Selectors

- Les sélecteurs sont de fonctions "pures" utilisées pour obtenir des morceaux d’état du store. Les sélecteurs offrent de nombreuses fonctionnalités lors de la sélection de morceaux de notre store.
  - Portabilité
  - Memoization
  - Testabilité
  - Composition
  - Type safety

Quand vous utilisez createSelector. Le dernier resultat est mémorisé lors de son appel ce qui permet de ne pas declancher la métode subscribe de tous les observables attaché a notre store. Cela permet un gain de performance. C'est ce que l'on appelle: memoization.

NGRX: Selectors

Selectors sont généralement défini dans un fichier a part:
- yourstorename.selectors.ts

```
const userState = createFeatureSelector('user')
export const selectUser = createSelector(
  userState,
 (state: AppState) => state.description;
)
```

Appel du selector dans le store:

```
this.counter = this.store.select(selectAllBooks)
```

NGRX: Effects - installation

```
ng add @ngrx/effects
```

Cela vas mettre à jour app.module.ts. On y déclarera nos effets

```
EffectsModule.forRoot([MovieEffects]),
```

Effects

Dans l'approche traditionnelle, un composant utilisant un service pour effectuer des appels http.
par exemple, si nous avons à faire à une collection de films:

- Utilisation du service pour effectuer un *side effect*, appel d'une API externe pour récupérer les films
- Changer l'état de nos films dans notre component:

```
@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);
  }
}
```

Effects

Les effets lorsqu'ils sont utilisés avec les Store diminuent le couplage du composant.
Écrivons notre premier effet, généralement dans son propre fichier :

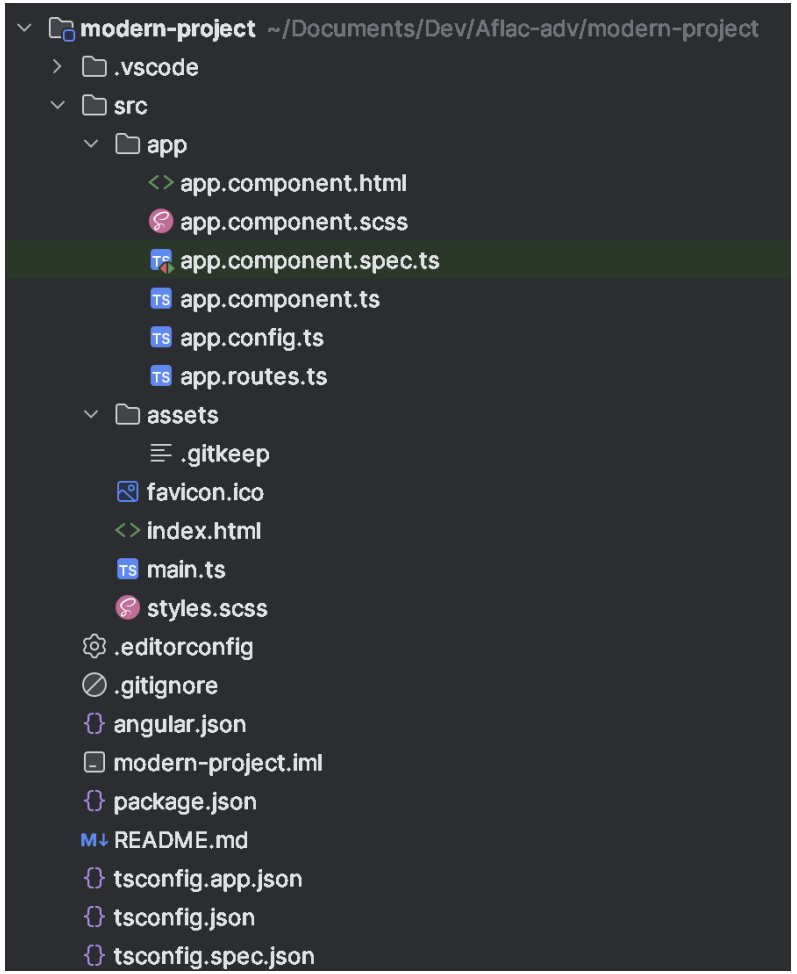
```
@Injectable()
export class MoviesEffects {

  loadMovies$ = createEffect(() => this.actions$.pipe(
      ofType('[Movies Page] Load Movies'),   // <-- Action que l'effet "écoute", appelée par dispatch
      exhaustMap(() => this.moviesService.getAll()
        .pipe(
        // Une fois terminé, nous passons à une autre action à l'aide d'une fonction RxJs
          map(movies => ({ type: '[Movies API] Movies Loaded Success', payload: movies })),
          catchError(() => EMPTY)
        ))
    )
  );
  constructor(
    private actions$: Actions,
    private moviesService: MoviesService
  ) {}
}
```

Angular 17

- Nouveau logo !
- Nouvelle documentation : https://angular.dev/
- Nouveau flux de contrôle intégré
- Vues "deffered"
- Composant standalone
- Composant basé sur l'api signal
- Applications de rendu hybride : des builds jusqu’à 87 % plus rapides.
- Applications de rendu côté client : des builds jusqu’à 67 % plus rapides.

Structure du projet

- Il n’y a plus de app.module.ts
- Les composants sont standalone par default
- Configuration déplacé de app.module.ts vers app.config.ts
- app.routes.ts est généré par default..



@if

Support de *control flow blocks* qui vous permettent d'afficher, de masquer et de répéter des éléments de manière conditionnelle.

@if

```
@if (a > b) {
   {{a}} is greater than {{b}}
} @else if (b > a) {
   {{a}} is less than {{b}}
} @else {
   {{a}} is equal to {{b}}
}
```

@for

@for:

- nécessite de définir "track"

```
@for (item of items; track item.id) {
    {{ item.name }}
}
```

- De nombreuses variables peuvent être utilisées

Variable	Signification
$count	Nombre d'éléments d'une collection
$index	Index de la ligne actuelle
$first	Si la ligne actuelle est la première ligne
$last	Si la ligne actuelle est la dernière ligne
$even	Si l'index de ligne actuel est pair
$odd	Si l'index de ligne actuel est impair

@for

@for:

- Exemple de boucle for plus avancé

```
@for (item of items; track item.id; let idx = $index, e = $even) {
  Item #{{ idx }}: {{ item.name }}
}
```

@Empty keyword

```
@for (item of items; track item.name) {
    <li> {{ item.name }}</li>
} @empty {
    <li> There are no items.</li>
}
```

@switch

```
@switch (condition) {
  @case (caseA) {
    Case A.
  }
  @case (caseB) {
    Case B.
  }
  @default {
    Default case.
  }
}
```

Nouveau composants "standalone"

- Méthode simplifiée pour créer des nos applications
- Réduire le besoin de NgModules
- Tous les composants générés sont désormais standalone par défaut

Comment ça marche

- Déclarer en tant que "standalone: true" dans les metadatas
  - Cela signifie que le composant n’a plus besoin d’être déclaré dans un app.module.ts.
- Besoin d’importer tout ce que le composant utilise

```
@Component({
  standalone: true,
  selector: 'photo-gallery',
  imports: [ImageGridComponent],
  template: `
    ... <image-grid [images]="imageList"></image-grid>
  `,
})
export class PhotoGalleryComponent {
  // component logic
}
```

Qu’est-ce que c’est

- Suivi d'un état permettant à Angular d’optimiser les mises à jour de rendu.
- Un **signal** est un wrapper autour d’une valeur. Les signaux peuvent contenir n’importe quelle valeur, des primitives simples aux structures de données complexes. La lecture d'une valeur se fait toujours via un getter ce qui permet à Angular de suivre où le signal est utilisé. Signal peut être soit *writable* ou *read-only*.

Définir et lire

Définition d’un signal

```
count = signal(0);
albums = signal<Album[]>([])
```

Signals are getter functions.

```
console.log('The count is: ' + count());
```

```
<div> Count: {{count()}}</div>
```

Changer une valeur

- Modification de la valeur
- Utilisation de 'set' pour modifier directement la valeur
- Utiliser 'update' pour avoir la valeur actuelle

```
count.set(10)
count.update(c => c+1)
```

Valeurs "Computed"

- Jusqu'à présent, les signaux var que nous avons définis sont de type `WritableSignal`.
- Nous pouvons définir certains signaux spécifiques en fonction d'autres variables du même type.
- On utilise la fonction `computed`
- Type: `Signal`

```
count: WritableSignal<number> = signal(0);
doubleCount: Signal<number> = computed(() => this.count() * 2);
```

Effets secondaires.

- Un **effet** est une opération qui s'exécute chaque fois qu'une ou plusieurs valeurs de signal changent.
- Vous pouvez créer un effet avec la fonction effect():
- Les effets fonctionnent toujours **au moins une fois**
- Doivent être définis dans le constructeur !

```
constructor(){
 effect( () => {
          console.log(`The current count is: ${count()}`);
     });
}
```

Signaux avec OnPush

- Lorsqu’un composant OnPush utilise la valeur d’un signal dans son modèle, Angular suit le signal en tant que dépendance de ce composant. Lorsque ce signal est mis à jour, Angular marque automatiquement le composant pour s’assurer qu’il est mis à jour lors de la prochaine exécution de Change Detection.
- La voie aux applications sans zonejs est ouverte! C'est là l’avenir d’Angular

Signal Inputs

- Signal Inputs
  - Valeurs exposées à l'aide d'un signal peuvent changer au cours du cycle de vie de votre composant.
  - Type safe
- Les valeurs peuvent être facilement dérivées chaque fois qu'une entrée change à l'aide de computed()
- Utilisation de Effect
- Read Only

```
// Dans le composant
firstName = input<string>();  //InputSignal<string|undefined>
age = input(0); // InputSignal<number>

lastName = input.required<string>(); // InputSignal<string>

// Dans le template, lastName est obligatoire
<app-user [lastName]="MacGowan"></app-user>
```

Signal model inputs

Model inputs are a special type of input allowing to do communication between two component (two-way binding)

```
//app.component
person: Person = {
  name: 'MacGowan',
  age: 65
}
```

```
<app-person [(person)]="person"></app-person>
```

```
// person.component.ts
person = model.required<Person>();

increasePersonAge() {
  this.person.update(
       p => ({ ...p, age: p.age + 1})
  )}
}
```

Signal queries

approche classique	approche Signal	Utilisation

@viewChildren	viewChildren	`<div #el></div>`
		`@if (show) {`
		`  <div #el></div>`
		`}`
		
		`divEls = viewChildren<ElementRef>('el');// Signal<ReadonlyArray<ElementRef>>`

@viewChild	viewChild	`<div #el></div> <my-component />`
		
		`divEl = viewChild<ElementRef>('el'); // Signal<ElementRef|undefined>`
		`cmp = viewChild(MyComponent); // Signal<MyComponent|undefined>`

@contentChildren	contentChildren	`divEls = contentChildren<ElementRef>('h');` `//Signal<ReadonlyArray<ElementRef>>`

@contentChild	contentChild	`headerEl = contentChild<ElementRef>('h'); // Signal<ElementRef|undefined>`
		`header = contentChild(MyHeader); // Signal<MyHeader|undefined>`

Principes clés

- **Simple et intuitif** : API simple et intuitive permettant aux développeurs de travailler efficacement avec Angular Signals.
- **Léger et performant.**
- **Programmation déclarative .**
- **Modulaire, extensible et évolutif :** des blocs de code indépendants combinés pour des implémentations flexibles et évolutives.
- **Type-safe :** Prévention des erreurs au moment de la compilation.

Creating the store

- signalStore function
  - Defining store features (cart, products, etc)
  - Defining store methods
  - Defining computed signals
  - Returns an injectable (can be injected wherever we want)

Créer le store

```
const initialState: CounterStore = {
  count: 10
}

export const CounterStore = signalStore(
  withState(initialState)
)
```

Et c'est tout ! Comme le store est injectable, il est prêt à être injecté dans n'importe quel composant.
Comme il s'agit d'un service et que nos composants sont autonomes, nous devons soit fournir l'injectable dans les providers de métadonnées du composant, soit le fournir dans l'injecteur racine.

```
export const CounterStore = signalStore(

 { providedIn: 'root' },// <-----
  withState(initialState)
)
```

Obtenir des valeurs du store

Comme le signal ne fonctionne pas de manière asynchrone, nous pouvons initier notre valeur dans les paramètres de notre composant.

```
//counter.component.ts

store= inject(CounterStore)
count = this.store.count
```

```
//counter.component.html

<span>value: {{count()}}</span>
```

Modification du store

Ajoutons quelques méthodes dans notre store en utilisant la function withMethods.

```
export const CounterStore = signalStore(

...

withMethods((store) => ({
  increment() {
    patchState(store, ({count: store.count() + 1}))
  },

...
```

Ensuite, dans notre composant :

```
this.store.increment()
```

Computed values

En tant que magasin basé sur le signal, nous pouvons définir des computed values:

```
export const CounterStore = signalStore(

...

withComputed(({count}) => ({
  doubleCount: computed(() => count() * 2)
})

...
```

Then in our component:

```
this.store.withComputed()
```

Async http calls

Nous pouvons récupérer des données directement dans notre store à l'aide d'API basées sur des Promises :
- Nous devons d'abord injecter notre service dans notre store
- Nous devons convertir notre observable en Promise en utilisant l'operateur `firstValue`

```
...

withMethods((store, albumservice = inject(AlbumService)) => ({

...
    async countAlbums() {
      const albums = await firstValueFrom(albumservice.findAll())
      patchState(store, {count: albums.length})
    }

...
```

Ensuite, dans notre composant :

```
this.store.countAlbum()
```

Approche plus complexe utilisant des opérateurs RxJs

Pour les opérations plus complexes, nous pouvons utiliser les opérateurs RxJs :

```
...

withMethods((store, albumservice = inject(AlbumService)) => ({

...
    countAlbumsRxjs: rxMethod<void>(
      pipe(
        switchMap(() => {
          return albumservice.findAll().pipe(
            tap(
              (albums) => patchState(store, { count: albums.length}),
            )
          );
        })
      )
    ),
    )

...
```

Guards

Permet de contrôler l'accès à des routes spécifiques dans votre application

- Exemple: Lorsqu'un utilisateur est enregistré, ce dernier pourra avoir accès a une route, sinon

on le redirige vers une page de login

- 2 types de guards principaux que l'on retrouve sous forme d'interface ayant chacun leur méthode à implémenter:
  - canActivate
  - canDeactivate

Guards: CanActivate

CanActivate décidera de si un utilisateur a accès à une route ou non.

- Un guard est un service (donc, injectable)
- Dois implémenter l'interface CanActivate et sa méthode
  - `canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {`
- Doit retourner une valeur de type **boolean** ou **Promise<boolean>** ou **Observable<boolean>**
  - Possibilité d'attendre un résultat asynchrone

```
// Guard utilisant un boolean en valeur de retour
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private router: Router) { }

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot){
    const isAuthenticated = //... votre logique pour
vérifier si l'utilisateur est authentifié

    if (isAuthenticated) {
      return true;
    } else {
      this.router.navigate(['/login']);
      return false;
    }
  }

}
```

Guards: CanDeactivate

CanDeactivate décidera de si un utilisateur peux quitte une route ou non...
Si la méthode canDeactivate du composant renvoie true, la navigation continue. Si elle renvoie false, la navigation est annulée.

- Un guard est un service (donc, injectable)
- Dois implémenter l'interface CanDeactivate et sa méthode
  - `canDeactivate(component: MyCantEscapeComponent) {`
- Doit retourner une valeur de type **boolean** ou **Promise<boolean>** ou **Observable<boolean>**
  - Possibilité d'attendre un résultat asynchrone

- Il est important que la **méthode canDeactivate soit générique**, on se doit de spécifier sûr quel composant on agit.

```
@Injectable({
  providedIn: 'root'
})
export class CanDeactivateGuard implements
CanDeactivate<MyCantEscapeComponent> {
  canDeactivate(component: MyCantEscapeComponent) {
    return component.canDeactivate ?
component.canDeactivate() : true;
  }
}

}
```

Guards: Enregistrer un guard

```
{path: 'home', loadComponent: () =>
import('./components/home/home.component').then(c =>
c.HomeComponent), canActivate: [AuthGuard]
},
```

```
{path: 'home', loadComponent: () =>
import('./components/home/home.component').then(c =>
c.HomeComponent), canDeactivate: [StayHereGuard]
},
```

Route Resolvers

- Middleware qui joue un rôle entre une route et un composant.
- Permet par exemple de charger des données avant d'activer une route

- Meilleure expérience utilisateur : les utilisateurs bénéficient de transitions transparentes entre les routes, car les données sont pré-chargés

- Logique de composant simplifiée : Séparation de la récupération de données et de la logique de composant. Code plus propre et plus facile à maintenir.

Route Resolvers

- Avant angular 16:
  - Service, doit implementer l'interface Resolve
- Après angular 16:
  - Basculement d'un resolver orienté classe en faveur d'une fonction
  - Dois retourner un type: ResolveFn<ReturnType>
  - ResolveFn peut retourner n'importe quelle valeur, y compris des valeurs de type Observable ou Promise

```
// Dans son propre fichier
export const loadUsersResolver:
ResolveFn<Observable<User[]>> = (route, state) => {
   let userService = inject(UserService)
   return userService.findAll()
};
```

```
// dans le composant
private route  = inject(ActivatedRoute)
ngOnInit() {
  console.log( this.route.snapshot.data['users'])
}
```

```
// dans une route
{path: users', resolve: {users: loadUsersResolver}},
```

Lazy-loading de nos routes

Tout comme un module, Il est possible de lazy loader nos composants standalones.

- Téléchargera un "chunk" de notre composant lors de l'activation de la route

```
export const ROUTES: Route[] = [
  {path: 'admin', loadComponent: () => import('./admin/panel.component').then(mod =>
mod.AdminPanelComponent)},
  // ...
];
```

Lazy-loading de nos routes

Le préchargement améliore l'expérience utilisateur en chargeant des parties de votre application en arrière-plan. Vous pouvez précharger des modules et des composants standalone.

```
//Dans une application orienté module
RouterModule.forRoot(
appRoutes,
    {
    preloadingStrategy:
    PreloadAllModules
    }
)
```

```
//Dans une application standalaone
export const appConfig: ApplicationConfig =
{
    providers: [
                provideRouter(
                        routes,

        withPreloading(PreloadAllModules)
                ),
    ],
};
```

Internationalisation

- La team Angular propose un package pour gérer l'internationalisation de nos pages.
  - Installation:

```
ng add @angular/localize
```

  - Modification de angular.json

```
"album-wholesale-v17": {
  "i18n": {
    "sourceLocale": "en",
    "locales": {
      "fr": "src/locale/messages.fr.xlf"
    }
  },
…
     "architect": {
       "build": {
         "builder": "@angular-devkit/build-angular:application",
         "options": {
                      "localize": ["fr"],
```

Internationalisation

- Maintenant que nous avons configurer notre app. Il nous faut dire à Angular ce qui doit être traduit.

```
<h1 i18n>Bonjour le monde</h1>
```

- Nous pouvons maintenant générer le fichier de traduction:

```
ng extract-i18n --output-path src/locale
```

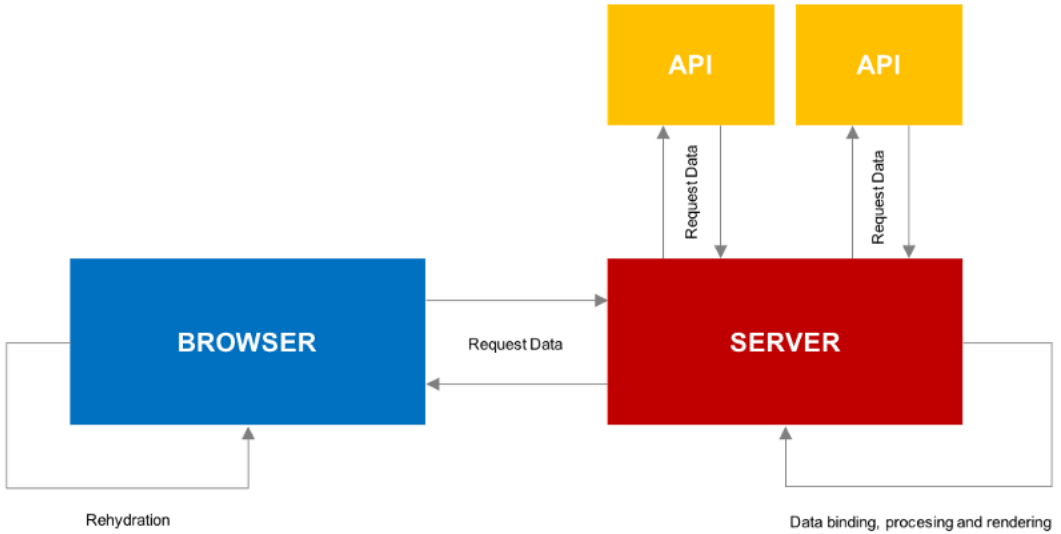
- Il ne nous reste plus qu'a copier-coller le fichier pour chaque langue:
  ex messages.en.xls

```
<trans-unit id="7886570921510760899" datatype="html">
  <source>Bonjour</source>
  <target>Hello</target>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/components/albums/album-
add/album-add.component.html</context>
    <context context-type="linenumber">34,35</context>
  </context-group>
</trans-unit>
```

SSR

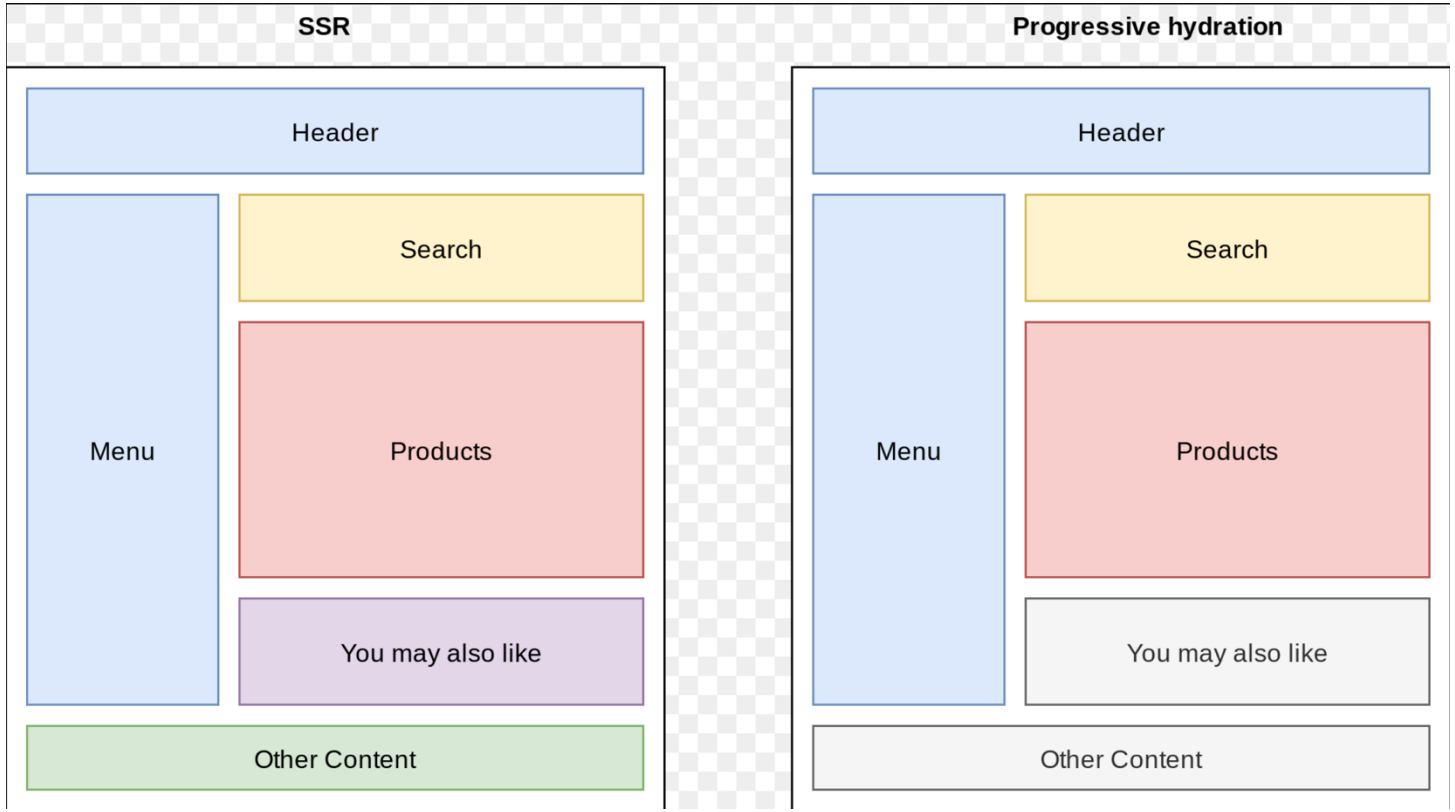
SSR (Server-Side Rendering) avec Angular offre plusieurs avantages :

- Amélioration des performances de chargement initial: le serveur envoie une page entièrement rendue au navigateur, ce qui peut réduire le temps de chargement initial.
  - Optimisation pour les moteurs de recherche (SEO).
  - Prévisualisation des liens sur les réseaux sociaux** : Les plateformes de médias sociaux génèrent des aperçus de liens en récupérant le HTML d'une page.



SSR: Hydration

L’hydratation est le processus qui restaure l’application rendue côté serveur sur le client. Cela inclut des éléments tels que la réutilisation des structures DOM rendues par le serveur, la persistance de l’état de l’application, le transfert des données d’application déjà récupérées par le serveur et d’autres processus.



SSR: Utilisation

- Utilisation de ng add pour l'instalation

```
ng add @angular/ssr
```

- Puis lancement classique

```
ng serve
```

Jest

- Jest, développé par Facebook, est un framework de test complet et facile à configurer qui est livré avec de nombreuses fonctionnalités prêtes à l’emploi, telles qu’une bibliothèque d’assertions intégrée, la prise en charge de mock et des tests instantanés. Jest exécute également des tests en parallèle, ce qui peut le rendre plus rapide que Karma dans les grandes bases de code.

Jest: Avantages

- Faible Configuration
- Snapshots
- Isolé
- Api riche
- Rapide
- Propose du code coverage
- Mocking simple

Jest : installation

```
npm install --save-dev jest jest-preset-angular @types/jest
```

- Ensuite, nous devons créer deux fichiers a la racine de notre projet:

```
import 'jest-preset-angular/setup-jest';
```

- jest.config.js

```
// jest.config.js
module.exports = {
  preset: 'jest-preset-angular',
  setupFilesAfterEnv: ['./setup-jest.js'],
};
```

Nous devons ensuite ajouter le type "jest" dans notre tsconfig.spec.json

Jest : installation

- Nous n'avons plus besoin de Karma, nous pouvons donc le supprimer

```
npm uninstall karma karma-chrome-launcher karma-coverage-
istanbul-reporter karma-jasmine karma-jasmine-html-reporter
karma-coverage jasmine-core
```

- Nous pouvons supprimer la configuration de test dans angular.json puisque Jest n'a pas besoin de configuration si explicite

Jest et test avec Angular : Méthodes

Fonction	Description
describe()	Regroupe un ensemble de tests relatifs à une fonctionnalité spécifique ou à un module.
it() ou test()	Définit un test individuel. it() est un alias de test().
beforeEach()	Exécutée avant chaque test dans le bloc describe(). Utilisée pour initialiser les variables d'état avant chaque test.
afterEach()	Exécutée après chaque test dans le bloc describe(). Utilisée pour nettoyer l'état après chaque test.
beforeAll()	Exécutée une fois avant tous les tests dans le bloc describe(). Utilisée pour des configurations coûteuses en temps qui peuvent être réutilisées pour tous les tests.
afterAll()	Exécutée une fois après tous les tests dans le bloc describe(). Utilisée pour nettoyer les ressources coûteuses en temps après tous les tests.
expect()	Définit une assertion. Généralement utilisée avec un "matcher" pour vérifier que le code se comporte comme prévu.
jest.fn()	Crée une fonction mock. Peut être utilisée pour espionner les appels à la fonction et simuler des comportements.
jest.spyOn()	Espionne les appels à une fonction existante et peut être utilisée pour simuler des comportements.
mockReturnValue()	Utilisée avec jest.fn() ou jest.spyOn() pour simuler une valeur de retour pour la fonction mock.
mockResolvedValue()	Utilisée avec jest.fn() ou jest.spyOn() pour simuler une valeur de résolution pour une fonction mock qui retourne une promesse.
mockRejectedValue()	Utilisée avec jest.fn() ou jest.spyOn() pour simuler une valeur de rejet pour une fonction mock qui retourne une promesse.

Example de test avec Jest

```
describe('HelloComponent', () => {
  let component: HelloComponent;
  let fixture: ComponentFixture<HelloComponent>;
  let compiled: any;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [HelloComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(HelloComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
    compiled = fixture.debugElement.nativeElement;
  });

  it('should create the component', () => {
    expect(component).toBeTruthy();
  });

  it('should render the message in a h1 tag', () => {
    expect(compiled.querySelector('h1').textContent).toContain('Bonjour le monde!');
  });
});
```

Cypress : installation

```
npm install cypress --save-dev
```

- Lancer cypress avec la commande et laissez vous guider

```
npx cypress open
```

- Modification du package.json

```
"scripts": {
...
      "cypress:open": "cypress open",
      "cypress:run": "cypress run"
}
```