Infos

- Change Detection in Angular - Pt.1 View Checking
- Change Detection in Angular Pt.2 - The Role of ZoneJS (2023)
- Change Detection in Angular Pt.3 - OnPush Change Detection Strategy
- 3.4 Зачем нужен метод changeDetectionRef.detach()?
- 3.3 В чем отличие markForCheck() и detectChanges()?
- Angular is about to get its most IMPORTANT change in a long time...
- I used Angular's signals to build an actual app
- Why didn't the Angular team just use RxJS instead of Signals?
- The end of Angular's "service with a subject" approach?
- Is this how we will handle errors reactively with signals in Angular?
- Angular zoneless change detection

https://medium.com/ngconf/simplified-angular-change-detection-e74809ff804d

https://medium.com/@bencabanes/angular-change-detection-strategy-an-introduction-819aaa7204e7

https://medium.com/@andre.schouten_ff/whats-the-difference-between-markforcheck-and-detectchanges-in-angular-fff4e5f54d34

https://dev.to/this-is-angular/angular-signals-everything-you-need-to-know-2b7g

https://netbasal.com/converting-observables-to-signals-in-angular-what-you-need-to-know-4f5474c765a0

https://betterprogramming.pub/rxjs-declarative-pattern-in-angular-cafba3983d21

https://netbasal.com/converting-signals-to-observables-in-angular-what-you-need-to-know-971eacd3af2

https://javascript.plainenglish.io/angular-zone-js-3b5e2347b7

https://betterprogramming.pub/zone-js-for-angular-devs-573d89bbb890

https://netbasal.com/understanding-angular-injection-context-18a0780ede2d

https://angular.io/guide/rxjs-interop

https://angular.io/guide/signals

https://angular.io/guide/rxjs-interop

What is Change Detection ?

Change Detection-ը դա մի մեխանիզմ է, որը հետևում է application-ի state-ի փոփոխություններին և սինխրոնացնում է այն application-ի view-ի հետ:

Յուրաքանչյուր front-end framework ունի իր Change Detection-ի մեխանիզմի ռեալիզացիան, մենք կխոսենք Angluar-ի մասին, թե ինչպես է այն աշխատում այստեղ:

Եկեք Change Detection-ի պրոցեսը բաժանենք երկու մասի

1. **View Checking**
   Այս պրոցեսը նշանակում է այն, երբ Angluar-ի view-ն սինխրոնացվում է համապատասխան **data model-ի** հետ:

2. **Re-run View Checking**
   Այս պրոցեսը ավտոմատացնում է **View Checking-ը** երբ application-ի state-ում ինչ որ փոփոխութուններ են տեղի ունենում:

Ավտոմատացման պրոցեսը մենք կարող ենք անջատել:

View Checking

Դիտարկենք մի օրինակ հասարակ component-ի վրա, որը ունի template և հայտարարված փոփոխականներ որոնք օգտագործվում են այդ template-ում:
Բոլոր այդ binding-ները մտնում են **Change Detection-ի** հետաքրքրության մեջ, որովհետև այդ binding-ները ուղիղ կապ ունեն թե ինչպես և ինչը պետք է re-render լինի այդ component-ում:

Պատկերացնենք որ որոշ ժամանակ հետո, մեր հայտարարած փոփոխականներից մեկը փոխվում է: Քանի որ մենք ավտոմատ **View Checking-ը** անջատել ենք, մեր view-ն re-render չի լինում և սինխրոնացված չէ մեր **data model-ի** հետ: Փոփոխությունները տեսնելու համար պետք է **View Checking Proccess-ը** աշխատացնենք ձեռքով: Այդ ամենը անելու համար մեզ կոգնի **ChangeDetectorRef-ի detectChanges** մեթոդը, որը նորից աշխատացնում է **View Checking-ը:**

Երբ մենք կանչում ենք **detectChanges** մեթոդը root component-ում (մեր դեպքում դա AppComponent-ն է) Angular-ը անցնում է բոլոր data binding-ների վրայով և թարմացնում է view-ն: Բայց ոչ միայն re-render է անում տվյալ component-ը այլ նաև իր child comopnent-ները: Այսինք եթե մենք **detectChanges-ը** աշխատացնում ենք AppComponent-ում, Angular-ը ամբողջ tree-ով re-render է անելու:

detectChanges-ը իր մեջ կանչում է **refreshView** ֆունկցիան որը իր մեջ ցիկլով պտտվում է բոլոր view-երի վրայով և re-render է անում:

Հարց կառաջանա, եթե մենք անջատել ենք ավտոմատ **View Checking-ը,** ինչպես է Angular-ը առաջին անգամ նկարում մեր component-ները: Angular-ը իր մեջ ունի **tick** մեթոդը, որը root component-ի bootstraping-ի ժամանակ կանչվում է, և իր մեջ ցիկլով պտտվում է բոլոր view-երի վրայով և կանչում է իրենց **detectChanges** մեթոդը:

Re-run View Checking (Zone.js)

Սկզբի համար հասկանանք, թե երբ է մեր state-ը փոխվում և երբ է Change Detection-ը աշխատում:

1. **MacroTasks and MicroTasks ինչպիսիք են setTimeout կամ setInterval, Promise.then**

2. **Handling Events ինչպիսիք են click, focuse և այլն**

3. **HTTP request completes (անսինխրոն գործողությունների ավարտից հետո)**

Այդ իրադարձությունների մասին Angular-ին տեղյակ պահելու համար Zone.js օգտագործում է **Monkey Patching** տեխնիկան:

Ինչ է իրենից ներկայացնում **Monkey Patching-ը,** երբ մենք copy-ենք անում օրիգինալ ֆունկցիան, սարքում ենք wrrapper այդ ֆունկցիայի համար, այդ wrrapper-ի մեջ կանչում ենք մեր օրիգինալ ֆունկցիան իրեն հասանելի արգումենտներով և այդ wrrapper ֆունկցիայի մեջ ավելացնում ենք մեր կողմից այլ գործողություններ, ինչնել անում է **Zone.js-ը** իր ֆունկցիոնալը ավելացնելու browser-ի կողմից տրամադրվող ֆունկցիաների մեջ:

Zone.js-ը ունի իր սեփական **execution context-ը:** Դրանք երկուսն են

1. **run** գործողությունները կատարվում են **Zone.js-ի execution context-ում** և աշխատացնում է **Change Detection-ը**

2. **runOutsideAngular** գործողությունները կատարվում են **Zone.js-ի execution context-ից** դուրս և **Change Detection-ը** չի աշխատում

Այս ամենը մենք ինքներս կարող ենք օգտագործել:

Հիմա հասկանանք թե ինչպես է **Zone.js-ը** Angular-ին տեղյակ պահում փոփոխությունների մասին: **Zone.js-ը** ուղղակի հետևում է callStack-ի փոփոխություններին և մինչև թասքի callStack-ից դուրս գալը տեղի են ունենում որոշակի ստգուգումներ, որոնցից մեկն է արդյոք կան թասքեր, որոնք սպասում են իրենց ավարտին, եթե չկան **Zone.js-ը** ուղղակի **onMicrotaskEmpty EventEmitter-ին** էմիթ է անում null:

Zone.js-ը **NgZoneChangeDetectionSchdeluer service-ում,** initialize-ից անմիջապես հետո սկսում է հետևել **onMicrotaskEmpty EventEmitter-ին,** և next-ի ժամանակ երբ **taskQue-ն** դատարկ է, կանչում է **tick** ֆունկցիան որը նայել ենք վերևում:

Event-ները աշխատացնում են **Change Detectior-ը** միայն այն ժամանակ, երբ մենք binde-ենք արել այդ event-ը և ունենք այդ event-ի համար listener: Մնացած դեպքում **Change Detectior-ը** չի աշխատի:

Zone.js-ի հիմնական խնդիրը կայանում է նրանում, որ նա չի կարող ասել Angular-ին թե որտեղ ու ինչ component է փոխվել, կամ ընդանրապես այդ component-ում ինչ որ բան փոխվել է թե ոչ: Նա ուղղակի ասում է որ միգուցե ինչ որ state ինչ որ component փոխվել է, դրա համար Angular-ը աշխատացնում է **View Checking-ը** ամբող **view tree-ի** համար և կատարում է անիմաստ, ավելորդ, չպետքական գործողություն:

Այս խնդիրը ինչ որ չափով լուծվել է **onPush strategy-ի** միջոցով, բայց դա այդքան էլ այդպես չէ: Իրական լուծումը մեզ տալիս են **Signal-ները:**

Change Detection Strategy onPush

Երբ մեր component-ի **ChangeDetectionStrategy-ին** փոխում ենք **Default-ից onPush-ի** Change Detection-ը աշխատում է հետևյալ փոփոխությունների ժամանակ:

1. **Input-ի ռեֆեռանսը փոխվելուց**
2. **Event աշխատելուց (component-ում կամ child component-ներում) (click, scroll, @Output() + EventEmitter)**
3. **Երբ ձեռքով կանչում ենք markForCheck()**
4. **Երբ template-ում օգտագործում ենք async pipe**

Այս բոլոր դեպքերում մենք կարիք չունենք Angular-ին տեղյակ պահելու որ կան փոփոխություններ: Angular-ը նշելու է **darty** այն բոլոր component-ներին և իրենց ծնողներին որոնք կապ ունեն այդ փոփոխված արժեքների հետ և աշխատացնելով **Change Detection-ը** (բացի **markForCheck-ի** ժամանակ) re-render է անելու այդ փոփոխված component-ները:

onPush-ով մենք Angular-ին ասում ենք որ պետք չի գուշակություններ անել թե որտեղ ինչ է փոխվել, Angular-ը կախված վերոնշյալ փոփոխությունների տվյալ component-ի subtree-ն update է անում:

onPush ստռատեգիայով մենք կարող ենք խուսափել ավելորդ re-rendering-ից Input-ի ռեֆեռանսը փոխվելու ժամանակ, քանի որ **Change Detection-ը** համեմատում է հին և նոր արժեքները: Angular-ը կատարում է re-rendering միայն այն child component-ներում որտեղ փոխվել է այդ Input-ի ռեֆերանսը, կամ եթե փոփոխականը փոխվել է event handler-ի մեջ: Իսկ մնացած փոփոխությունների ժամանակ չի կատարվելու subtree-ի re-rendering (եթե իհարկե չենք օգտագործում **async pipe**):

Նախքան Signal-ներին անցնելը, խոսենք **ChangeDetectorRef-ի** մեթոդների մասին:

1. **detectChanges** աշխատացնում է **Change Detector-ը** տվյալ component-ի և իր child component-ների համար: Աշխատում է նաև **detach** եղած component-ների համար:

2. **markForChack** տվյալ component-ից սկսած, իր բոլոր parent component-ներին մինչև root component տալիս է darty flag, որպեսի հաջորդ ցիկլի ժամանակ այդ view tree-ն նույնպես ստուգվի:

3. **detach** տվյալ component-ը իր child component-ների հետ միասին հանում է change detection tree-ից

4. **reattach** տվյալ component-ը իր child component-ների հետ միասին հետ է մտցնում change detection tree

5. **chackNoChanges** ստուգում է արդյոք տվյալ component-ը կամ իր child component-ները ունեն փոփոխություններ. եթե այո նետում է Error

Signals

Signal հասկացողությունը եկել է SolidJS-ից: Signal-ները իրենցից ներկայացնում են reactiv primitive-ներ, որոնց փոփոխության ժամանակ update են լինում այն ամենը ինչը օգտագործում են դրանք:

Signal-ները շատ նման են Behavior Subject-ներին, ուղղակի Signal–ները կարիք չունեն subscribe/unsubscribe լինելու: Բայց սա միակ տարբերությունը չէ, Signal-ները աշխատում են **սինխրոն արժեքների հետ (Synchronous Reactivity),** իսկ Observable-ները կարող են աշխատել նաև **անսինխրոն արժեքների հետ (Asynchronous Reactivity):**

Signal-ները չեն եկել փոխարինելու RxJs-ը, նրանք եկել են ավելի հեշտացնելու և պարզեցնելու մեր աշխատանքը: Մենք խնդիրները կարող ենք լուծել թե Signal-ներով և թե RxJs-ի օգնությամբ, ուղղակի Signal-ներով ավելի պարզ է և հասկանալի:

Angular-ը մեզ հնարավորություն է տալիս, համատեղել այս երկու տեխնոլոգիաները հետևյալ մեթոդների շնորհիվ

1. **toSignal**

2. **toObservable**

Signal-ների արժեքը ստանալու համար իրենց պետք է կանչել ֆունկցիայի նման:

Signal-ների API-ին մեզ տրամադրում է հետևյալ ֆունկցիաները

1. **set** վերագրում է արժեք

2. **update** փոխում է արժեքը (array-ների և object-ների դեպքում նոր reference) , առգումենտ ստանում է call-back որն էլ իր հերթին ստանում է signal-ի հին արժեքը և վերադարձնում է նոր արժեք

3. **mutate** փոխում է արժեքը (նախատեսված է array-ների և object-ների համար) , առգումենտ ստանում է call-back որն էլ իր հերթին ստանում է signal-ի հին արժեքը և վերադարձնում է նոր արժեք: Այս ֆունկցիան կատարում է փոփոխություն օրիգինալ array-ի կամ object-ի վրա

4. **computed** վերադարձնում է նոր signal, և թարմացնում է այն կախված իր callback-ի մեջ օգտագործվող signal-ների փոփոխությունների: Վերադարձրած signal-ը non setable է, որին չենք կարող արժեք **set** անել

5. **effect** աշխատում է այն ժամանակ, երբ իր callback-ի մեջ օգտագործվող signal-ներից ցանկացած մեկը փոխվում է: **effect-ի** մեջ մենք նաև կարող ենք օգտագործել **untracked** մեթոդը, որի մեջ կարող ենք գրել signal և այդ signal-ի փոփոխությունները չեն աշխատեցնի **effect-ը**: **effect-ի** callback-ը առաջին առգումենտով ստանում է **onCleanup** ֆունկցիա, նա որպես առգումենտ ստանում է callback, որը աշխատում է նախքան հաջորդ փոփոխությունը կամ **effect-ի destroy-ի** ժամանակ: **effect-ը** վերադարձնում է **EffectRef** որը ունի **destroy** մեթոդը: Դրա շնորհիվ մենք մանուալ կարող ենք ավարտել **effect-ի** tracking-ը

toSignal-ը, toObservable-ն (երբ **requireSync** option-ը true է) **և effect-ը** unsubscribe լինելու համար օգտագործում են **inject** ֆունկցիան և **DestroyRef-ի onDestroy hook-ը:** Դրա համար հարկավոր է նրանց կանչել **injection context-ում** կամ որպես երկրորդ առգումենտ փոխանցել **injector:**