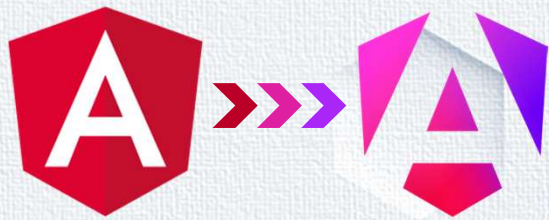


C:\ng new_



Angulars Renaissance



Hi, I'm...



Kobi Hari

- Freelancer
- Developer, Instructor and Consultant
- Angular, Async development, .NET core



My email:

hari@applicolors.com



Courses on Udemy: <https://www.udemy.com/user/kobi-hari/>



My Angular Channel: <https://www.youtube.com/@kobihari>

We have a GitHub Repository!

[kobi-hari-courses/2508-glassbox-ngnew](https://github.com/kobi-hari-courses/2508-glassbox-ngnew)



Session Summary

Commits		
Application Setup		
#	Link	Description
1	Initial application	Created an empty application using the angular CLI: <code>ng new redux-pop-quiz</code>
2	Added Material Design	Added angular material using the cli: <code>ng add @angular/material</code>
3	Custom Theme	Defined the theme palettes in partial <code>_common.scss</code> and generated the material styles in <code>styles.scss</code>
4	Tested theme	Used some angular material components like icon and button to test the new theme
5	Grid, Flex layout	Created <code>about.component.html</code> to display grid or flex for easier layout management
6	CSS Variables	Added <code>about.component.css</code>
Building the UI		
7	Toolbar	Added mat

Project Code with
Commits Summary

 [presentations](#)

 [projects](#)

 [LICENSE](#)

 [README.md](#)

This presentation and other
Useful Links

Our Agenda



inject() – The new Dependency Injection



Routing - Redirected



Signals – The new Reactivity



Resources – async Signals

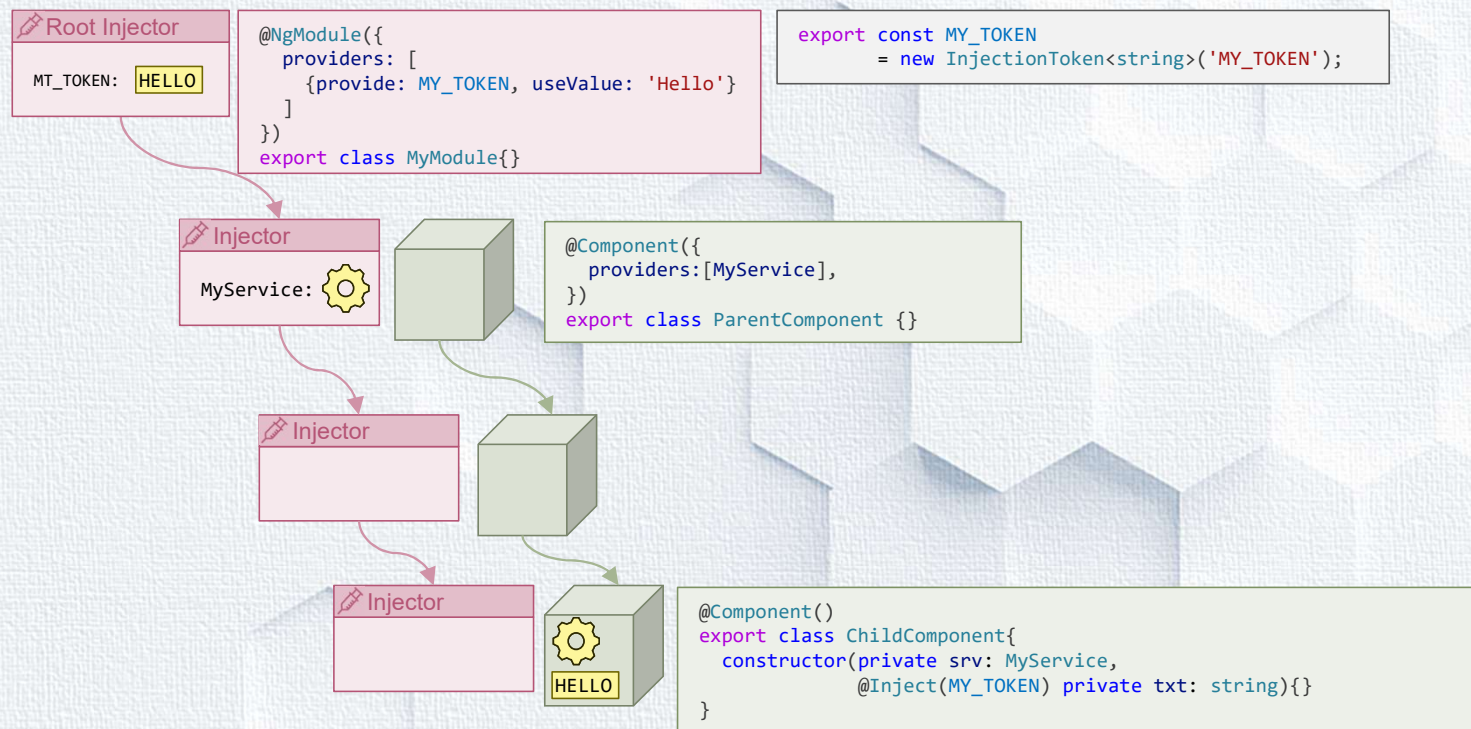


inject()

The new Dependency Injection



Dependency Injection - Recap





The Injector

```
@Component({
  providers: [{provide: MY_NUMBER, useValue: 42}]
})
export class AppComponent {

  constructor(private injector1: Injector) {
    const injector2 = Injector.create({
      providers: [{provide: MY_TOKEN, useValue: 'WORLD'}],
      parent: injector1
    });

    const val = injector2.get(MY_TOKEN); // val === 'WORLD'
    const num = injector2.get(MY_NUMBER); // num === 42
  }
}
```




13



inject()

ng old (<14)

```
@Component()  
export class ChildComponent{  
  constructor(private srv: MyService,  
               @Inject(MY_TOKEN) private txt: string){}  
}
```

ng new (14+)

```
@Component()  
export class ChildComponent{  
  private srv = inject(MyService);  
  private txt = inject(MY_TOKEN);  
}
```




Injection Context

- The `inject()` function only works in injection context:
 - Inside the constructor
 - Field initializers
 - Factory function of provider

```
@Component()
export class ChildComponent{
  private str = inject(MY_TOKEN); // will work

  constructor() {
    const txt = inject(MY_TOKEN); // will work
  }
}
```


Can you use `inject` in these cases? Angular

```
// in property setters
set myProp(val: string) {
  inject(Service).refresh();
}

constructor() {
  // in subscribe body
  of(true).subscribe(val => {
    const txt2 = inject(MY_TOKEN); // error
  });

  // in promise continuation
  Promise.resolve(true).then(val => {
    const txt3 = inject(MY_TOKEN); // error
  })
}

// in lifecycle hooks
ngOnInit() {
  const txt = inject(MY_TOKEN); // error
}
```



Don't



run-In-Injector-Context()

- You can run functions that use inject() in other places by:
 - Storing the **Injector**
 - Using the **runInInjectorContext** function

```
@Component()  
export class ChildComponent {  
  private injector = inject(Injector);  
  
  ngOnInit() {  
    runInInjectorContext(this.injector, () => {  
      const num = inject(MY_TOKEN);  
    })  
  }  
}
```




Destroy Ref

- You can now inject “**DestroyRef**” to components, services, directives, pipes...
 - This replaces the need for “**OnDestroy**”
- If you want to auto-complete observables – you can use the **takeUntilDestroyed** operator
 - Note that it uses **inject()** to inject the **DestroyRef**
 - So you may only use this operator in injection context
 - Alternatively, you may use it anywhere and pass the **DestroyRef** as parameter

```
export class SomeComponent {  
  constructor(private destroyRef: DestroyRef) {  
    destroyRef.onDestroy(() =>  
      console.log('Destroyed'));  
  
    interval().pipe(  
      takeUntilDestroyed()  
    ).subscribe(val => console.log(val));  
  }  
  
  notInjectionContext() {  
    interval().pipe(  
      takeUntilDestroyed(this.destroyRef)  
    ).subscribe(val => console.log(val))  
  }  
}
```




Routing & Standalone

Redirected



 Angular

14



Importing providers

ng old (<15)

```
// app.module.ts
@NgModule({
  imports: [
    HttpClientModule,
    BrowserAnimations,
    RouterModule.forRoot(routes),
    ModuleWithProviders
  ],
})
```

ng new (15+)

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(),
    provideRouter(routes),
    provideAnimations(),
    importProvidersFrom(ModuleWithProviders)
  ]
};
```




App Initialization

ng old (<19)

```
// app.module.ts
@NgModule({
  imports: [{
    provide: APP_INITIALIZER,
    useFactory: (service: InitService) => () => service.init(),
    deps: [InitService],
    multi: true
  }],
})
```

ng new (19+)

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideAppInitializer(() => inject(InitService).init()),
  ]
};
```




No routing module

ng old (<15)

```
// app-routing.module.ts
const routes: Routes = [
  {path: 'home', component: HomeComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

ng new (15+)

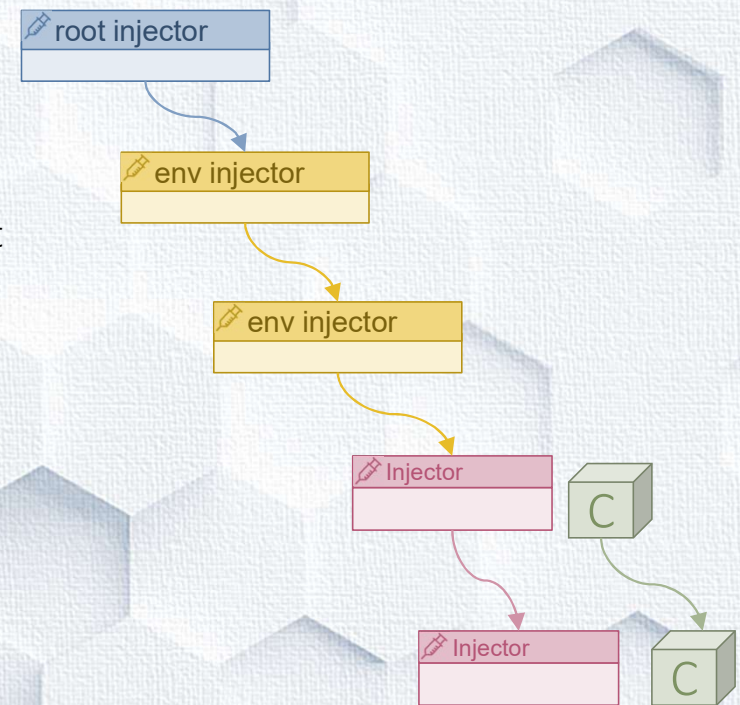
```
// app.routes.ts
export const routes: Routes = [
  { path: 'home', component: HomeComponent }
];
```




Environment Injectors

- You can define **providers** that will be applicable to subset of routes
- This creates an additional Hierarchy of **injectors**
- Once the component Injectors are exhausted, The environment injector Hierarchy is used

```
// app.routes.ts
export const routes: Routes = [
  {
    path: 'home', component: HomeComponent,
    providers: [{
      provide: MY_TOKEN, useValue: 100
    }],
    children: [
      {path: 'page-a', component: PageAComponent},
      {path: 'page-b', component: PageBComponent}
    ]
  }
];
```





Lazy Loading

- You can load one standalone component
- Or you can load a set of routes

```
// app.routes.ts
export const routes: Routes = [
  {
    path: 'courses',
    loadChildren: () => import('./courses/courses.component')
      .then(m => m.CoursesComponent)
  }
];
```

```
// app.routes.ts
export const routes: Routes = [
  {
    path: 'courses',
    loadChildren: () => import('./courses/courses.routes')
      .then(m => m.coursesRoutes)
  }
];

// courses.routes
export const coursesRoutes: Route[] = [
  { path: 'list', component: CourseListComponent },
  { path: 'top', component: TopCourseComponent }
];
```



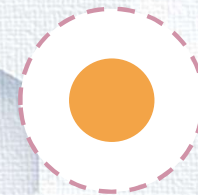

Demo – Standalone Routing



- ✓ Define Router
- ✓ Lazy load components
- ✓ Lazy load routes set
- ✓ Use Environment Injectors
- ✓ Initialize Environment Injector

Other Goodies

- Directive Composition
- Router Input Bindings
- Functional Guards
- Functional Resolver





Directive Composition

- Directives and components may have **hostDirectives**
- These are applied to the component or directive itself
- You can bind inputs and outputs of the hosted directives to the inputs and outputs of the host directive

```
@Directive({  
  selector: '[appMy]',  
  standalone: true,  
  hostDirectives: [HostedDirective]  
})  
export class MyDirective {}
```

```
@Directive({  
  selector: '[appMy]',  
  standalone: true,  
  hostDirectives: [{  
    directive: HostedDirective,  
    inputs: ['hostedInput: myInput'],  
    outputs: ['onHostedEvent: myEvent']  
  }]  
})  
export class MyDirective {}
```




Functional Guards and Resolvers

- Guards and Resolvers are classes that wrap a single function
- The only reason we used a class was... for Dependency Injection
- But now, with the `inject()` function, we can inject into functions.
- Use the `ResolveFn<T>` and `CanActivateFn` Typescript types to help you create such functions

```
// app.routes.ts
export const routes: Routes = [
  {
    path: 'admin',
    canActivate: () => inject(AuthService).isAdmin()
  }
];
```

```
// app.routes.ts
export const routes: Routes = [
  {
    path: 'admin',
    resolve: {user: () => inject(DateService).getUser()}
  }
];
```




Router Input Bindings

- Components may receive inputs from the router
 - Router parameters
 - Query parameters
 - Route data from the data property
 - Route data from the resolvers
- You can respond to value changes by using:
 - Property Setters
 - **OnChanges** hook

```
// app.routes.ts
export const routes: Routes = [
  {
    path: 'course/:id',
    data: { courseType: 'online' },
    resolve: { topic: () => inject(DateService).getTopic() },
    component: CourseComponent
  }
];

// course.component.ts
export class CourseComponent {
  @Input() id?: string;
  @Input() courseType?: string;
  @Input() topic?: string;
}
```




Dynamic Redirect

- Your “redirectTo” property can now be a function
- The function can either return “string” or “UrlTree”
- You can return an async value (Promise, or Observable)
- You can use “inject” in the function
- Your function can receive a parameter containing
 - The Data
 - The Url
 - The Title
 - Params
 - And more...

```
export const routes: Routes = [  
  {path: '', pathMatch: 'full',  
    redirectTo: () => getRandomPage() },  
  {path: 'page-1', component: Page1},  
  {path: 'page-2', component: Page2},  
  {path: 'page-3', component: Page3},  
];  
  
function getRandomPage(): Observable<string> {  
  const pages = ['/page-1', '/page-2', '/page-3'];  
  const randomIndex = Math.floor(Math.random() * 3);  
  const randomPage = pages[randomIndex];  
  return of(randomPage).pipe(  
    delay(3000) // Simulate a delay of 1 second  
  );  
}
```




Demo – Standalone Routing



- ✓ Define functional guard
- ✓ Define functional resolver
- ✓ Feed component inputs
- ✓ Use Directive Composition



Signals

The new reactivity



 Angular

15



Signals

- Normal properties do not maintain their relationships with each other
- With signals, atomic values “signal” when they change, so computed values remain correct
- Yes... It’s kinda like **BehaviorSubject**

```
let width = 10;  
let height = 20;  
let area = width * height; // 200
```

```
width = 15;  
// area is still 200
```

```
let width = signal(10);  
let height = signal(20);  
let area = computed(() => width() * height()); // 200
```

```
width.set(15);  
console.log(area()); // area is 300
```




The signal primitives

- You create a signal with the `signal()` function
- You create a computed signal with the `computed()` function
 - Do not cause side effects inside it
 - Do not create new signals inside it
 - This should be a pure function that depends on other signals – that's it
- You can read from a signal by simply calling it like a function
 - It is synchronous
 - It will return a value instantly
 - No need to subscribe – key difference from observables

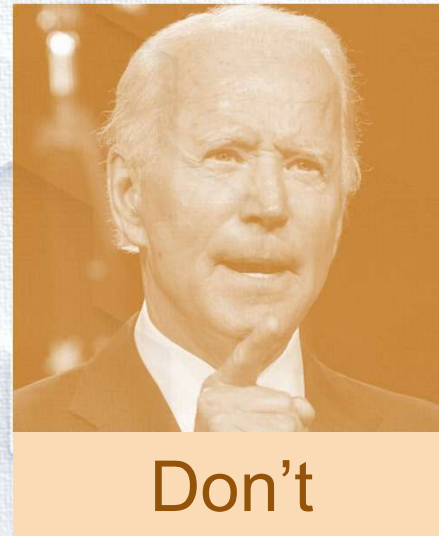
```
let firstName = signal('Kobi');
let lastName = signal('Hari');

let fullName = computed(() =>
  firstName() + ' ' + lastName());

console.log(fullName());
```


Inside computed, can you: ?

```
// Create a new signal?  
let fullName = computed(() =>  
  signal('Mr')  
    + firstName() + ' '  
    + lastName());  
  
// Cause side effect?  
let fullName = computed(() =>  
  console.log('HEY!!!'));  
  
// do asynchronous stuff  
let fullName = computed(() =>  
  await userService.getName());
```





Updating signals

- You can only update atomic (writeable) signals
- Use `set()` when you want to set a new value that does not depend on the previous
- Use `update()` when you want to set a new value that **does** depend on previous value
- ~~Use `mutate()` when you want to modify the original value (mutate array or object inside signal).~~
 - Don't...

Do

```
firstName.set('Yakov');  
lastName.update(val => val.toLowerCase());
```

Don't

```
firstName.set('Mr ' + firstName());
```

Instead, do

```
let titledName = computed(() => 'Mr ' + firstName());  
  
// or  
  
firstName.update(val => 'Mr ' + val);
```




effect

- Use effect to respond to changes in signals
 - The effect will run once when defined
 - It will run again every time one the signals it depends on change
- You may not change other signals inside effect
 - Ish...
 - This started off as forbidden (in Angular <=18) and then became “not recommended” in Angular 19
 - But you **Can** modify signals inside effect after **await**

```
effect(() => {  
  console.log('full name: ', fullName());  
});
```

Don't

```
let salutation = signal('');  
effect(() => {  
  salutation.set('Hello' + fullName());  
});
```

Instead, do

```
let salutation = computed(() => 'Hello' + fullName());
```

This is, actually, allowed

```
effect(async () => {  
  if (fullName() === 'Kobi') {  
    let res = await (someService.someMethod());  
    resultSignal.set(res);  
  }  
});
```




Limitations

1. Signals are synchronous – so they must have initial value when created
2. `effect()` must be called in Injection Context
 - Because they rely on `DestroyRef` to complete and unsubscribe



Is RxJs dead?

- Signals are great for
 - State
 - Component binding
 - They are the future of angular
- But...
 - You cannot replace higher order observables
 - ~~• You cannot do anything asynchronous with them~~
 - You cannot filter them...
 - They rely on injection context so you cannot use them anywhere else
 - ~~• They are still for review...~~



Signals - Verdict

- Simple – but the simplicity is deceptive
- Lots of Magic
 - Relies too heavily on **InjectionContext**
 - Has a complex algorithm for change flow, that may cause bugs
 - Effects can't really call any service, create new signals, or even change them.
- But they are the future – so get on board
- Recommended: Use it with **NgRx Signal Store**



Modern Angular 20 with Signals

The missing guide

by Kobi Hari



NgRx Signal Store 19

The missing guide

by Kobi Hari



Theming Angular 20 & Material MD3

The missing guide

by Kobi Hari

Demo – Signals



- ✓ Create Writeable signals
- ✓ Create computed signals
- ✓ Present Signals in components
- ✓ Modify Signals
- ✓ Create side effects
- ✓ See the limitations of signals



 Angular

16



 Angular

17



Signal Components

Zoneless we go!



The new way to write components

- The new way is called “**Signal Components**”
 - Simpler
 - More Powerful
- Alternative for **@Decorators** using functions
- No more need for lifecycle hooks
- A step on the way to zoneless apps
- Actually – not just for components.
 - Directives
 - Pipes





Signal Inputs

ng old (<18)

```
export class SelectorComponent {  
  @Input() options: string[] = [];  
  
  @Input({required: true}) selectedOption!: string;  
}
```

ng new (18+)

```
export class SelectorComponent {  
  readonly options = input<string[]>([]);  
  
  readonly selectedOption = input.required<string>();  
}
```




Inputs are now signals

- You do not need lifecycle hooks to respond to changes
- You can derive computed values from them
- You can respond to their changes using effects
- You can derive observables from them using **toObservable**

```
readonly selectedIndex = computed(() =>  
  this.options().indexOf(this.selectedOption()));
```

```
constructor() {  
  effect(() => {  
    localStorage.setItem('selectedOption',  
      this.selectedOption());  
  })  
}
```




Input value transformations

- Sometimes we want to apply transformation of values received from parent components
- You can define a custom transformation function on the input
- You can use a set of predefined transformations

```
readonly num = input(  
  { default: 0, transform: toNumber }  
);
```

```
function toNumber(val: string | number): number {  
  return typeof val === 'string' ? parseInt(val, 10) : val;  
}
```




Output functions

ng old (<18)

```
export class SelectorComponent {  
  @Output() selectedOptionChange = new EventEmitter<string>();  
}
```

ng new (18+)

```
export class SelectorComponent {  
  selectedOptionChange = output<string>();  
}
```




The new `output()`

- It's not a signal – but an event emitter
- Interoperability with RxJS
 - Create output from observable using `outputFromObservable`
 - Create observable from output using `outputToObservable`

```
import { outputFromObservable } from
  '@angular/core/rxjs-interop';

timesUp = outputFromObservable(timer(3000).pipe(
  map(_ => 'Time is up!'))
);
```




models

- Model is a new approach to create two-way binding in angular
- A model input is essentially a writeable signal (while normal inputs are read-only)
- Child component may pass data to parent
 - By calling an event
 - Or, by setting a writeable signal which is passed by reference



Defining and Using model inputs

- Define a model using the model function
 - You can use `model.required` mandatory inputs
- In the parent component, you can pass value like any other inputs
 - Using expression
 - Using signal values
- In the parent component, you can receive changes
 - Using an event handler
 - Using a writeable signal

```
// counter.component.ts
export class CounterComponent {
  readonly value = model.required<number>();

  increment() {
    this.value.update(v => v + 1);
  }
}
```

```
// app.component.html
<app-counter [value]="2"/>

<app-counter [value]="mySignal()"/>
```

```
// app.component.html
<app-counter [value]="2"
  (valueChange)="counterChanged($event)"/>

<app-counter [(value)]="mySignal"/>
```




[(ngModel)] is back!

- The “banana-in-a-box” syntax is back
- Two-way-binding
- You can (once again) use ngModel to bind inputs to signals
- Works with any form control.
- Handle with care...





Signal Queries

ng old (<18)

```
// counter.component.html
<div #counterValue>{{value()}}</div>

// counter.component.ts
@ViewChild('counterValue')
title: ElementRef | undefined = undefined;
```

ng new (18+)

```
// counter.component.ts
readonly counterValue = viewChild<ElementRef>('counterValue');
```




What about ``afterViewInit``

- The query returns a `Signal<ElementRef | undefined>`
- We can therefore just respond to the signal
 - Using `computed`
 - Or using `effect`
- The signal query refreshes, so if the view element is destroyed or replaced, we will get a new value
- Just like in the `@ViewChild` decorator, you can set the `"read"` property to query components, directives, templates, view container refs, or elements.



Making signal query required

- Just like with **input** and **model**, you can set the view child to be **required** and then it will never return undefined
- Angular will return error if the query does not match anything in the view



Signal Queries

ng old (<18)

```
// app.component.ts
@ViewChild(CounterComponent)
counters!: QueryList<CounterComponent>;

// counters is a QueryList<CounterComponent> and
// you can subscribe to it
```

ng new (18+)

```
// app.component.ts
readonly counters = viewChildren(CounterComponent);

// counters is Signal<CounterComponent[]>
```




`contentChild` and `contentChildren`

- You can use `contentChild` to query from the component content into a signal
- Use `contentChildren` to query a collection of items from the component content into a signal of array
- You can use “`read`” properties here too.
- You can make them required:
`contentChild.required(“feature”)`



18



Linked Signal

A combination of computed and writeable signal

- Like **computed** – it has a computed expression
- But you can override the value
- using **set** and **update**

TS linked-signal.ts

```
readonly products =  
  signal(['Apple', 'Banana', 'Cherry']);  
  
readonly selectedProduct =  
  linkedSignal<string[], string>({  
    source: this.products,  
    computation: (products, prev) => {  
      if (!prev) return products[0];  
      if (products.includes(prev.value))  
        return prev.value;  
      return products[0];  
    }  
  });
```




Signal Resources

Asynchronous Signals



Resource signals

Bridges between signals and **async** primitives

- **resource** – for promises
- **rxResource** – for observables
- **httpResource** – Specifically for http

TS resource.ts

```
readonly keyword = signal('');

readonly results = resource({
  params: () => this.keyword(),
  loader: options =>
    this.service.search(options.request)
})
```




Resource signals

- Contains several signals:
 - **value** – the latest value
 - **error** – the latest error, or undefined
 - **status** – the current async status
 - **isLoading** – guess...
- Methods:
 - **set, update** – modify to a local value
 - **reload** – reloads with the latest params
 - **asReadonly** – readonly, non writeable version

```
TS resource.ts

constructor() {
  effect(() => {
    console.log('results status',
      this.results.status());

    console.log('results data',
      this.results.value());

    console.log('results error',
      this.results.error());

    console.log('results loading',
      this.results.isLoading());
  })
}
```


Resource Signals - Statuses



Idle

Not loaded yet



Error

Latest load failed



Loading

Triggered by new request



Reloading

Triggered by reload method



Resolved

Value loaded



Local

Value overridden locally



19



HttpResource – wraps HttpClient

- Accepts a URL that is wrapped with “**effect**”
- Can also accept params as object
- Fetches data from the url whenever the url **changes**
- Supports “**reload**”, and **local** value
- All Http Client **configurations** are supported
 - Yes, including **Interceptors**

TS http-resource.ts

```
#searchResult = httpResource<Book[]>(() => ({  
  url: `${this.apiBase}/search`,  
  params: { q: this.#keyword() },  
  
}), {  
  defaultValue: [],  
})
```




resource – with streaming

- Resource supports 2 modes
 - Loader
 - Stream
- The “stream” callback is called once per request for setup
- It is required to return a signal
- It is supposed to set a “background” process that updates the signal
- Usually, creates an observable and subscribes to it
- Relatively complex syntax... 😞

TS stream-resource.ts

```
#selectedStock = resource({
  params: () => ({id: this.#selectedBookId()}),
  stream: async (options) => {
    const res = signal<ResourceStreamItem<number>>({value: 0
  });

    const o$ = interval(1000);
    o$.subscribe(val => {
      res.set({value: val})
    });
    // need to also take care of cancellation
  }

  return res;
})
```




rxjsResource – best for streaming

- Works with Observables
- Has **only** stream option
- The stream function needs to calculate a new observable based on the params (and perhaps previous value)
- It will subscribe to the observable and copy its value to the resource
- Automatically supports cancellation by unsubscribing from the observable when needed

```
TS rx-resource.ts

#selectedStock = rxResource({
  params: () => ({id: this.#selectedBookId()}),
  stream: (options) => {
    if(!options.params.id) return of(0);
    return interval(1000)
  }
})
```




Going Zone-less

- Zone JS has many Cons
 - Performance Hits
 - Bundle overhead
 - Startup time
 - Debugging Experience and Stack trace
- Today You can already disable ZoneJS
 - `provideExperimentalZonelessChangeDetection()`
 - `provideZonelessChangeDetection()`
 - Remove the import of `zone.js` in the `polyfills` file
- Get ready to go zoneless
 - Use `ChangeDetectionStrategy.OnPush` as much as possible
 - Remove reliance on `NgZone`
 - Avoid `onMicrotaskEmpty`, `isStable`, `onStable`, `onUnstable`
 - You can still use `NgZone.run` and `NgZone.runOutsideAngular` – these are compatible with zoneless but are not required anymore



Demo – Resource Signals



- ✓ Using `httpResource`
- ✓ Using resource with promises
- ✓ Streaming with `rxResource`
- ✓ Streaming with web socket