ANGULAR
ARCHITECTS

Best Choices

Alexander Thalhammer | @LX_T

Best Choices

- Templete-driven vs reactive forms
- Constructor-based vs functional inject()
- Class-based vs functional guards
- Using inputs vs. content projection
- Structural directives vs new control flow
- Reactivity with RxJS subjects vs signals
- Global vs. local state management
- Structuring your application(s)

Agenda

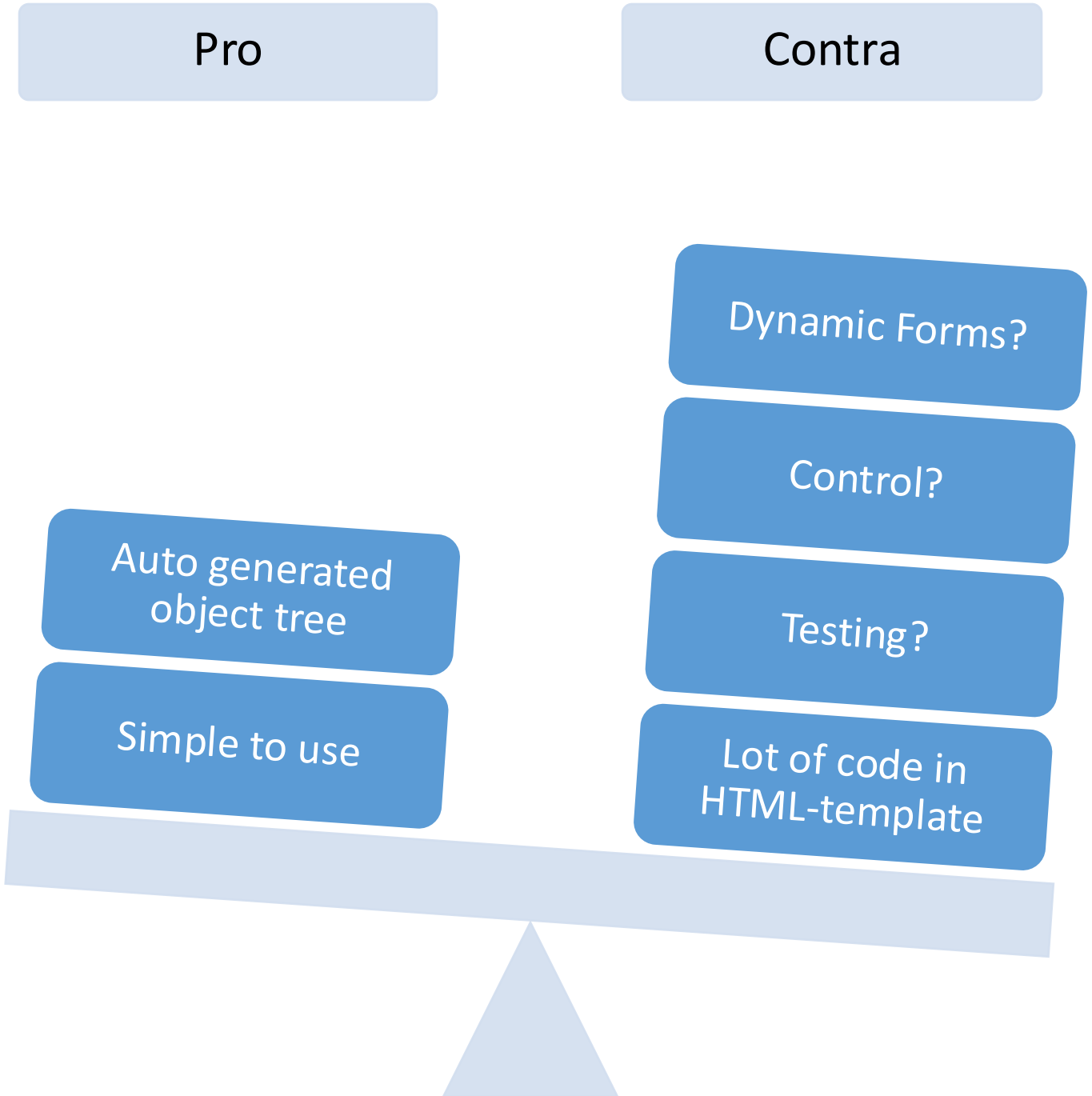
Which one is better?

Template-driven

- Add ngModel within the HTML-template
- Angular creates object tree for form
- FormsModule

Reactive

- We create the object tree in our component (.ts)
- More control, more power
- ReactiveFormsModule


Pro
Contra
Dynamic Forms?
Control?
Testing?
Lot of code in HTML-template
Auto generated object tree
Simple to use

Demo

Template-driven vs reactive forms

Dependency injection and route guards

Discuss this post on Github:
/angular/angular/issues/50234

ANGULAR
ARCHITECTS

Thought experiment

- What if <app-flight-card> would handle use case logic?
  - e.g. communicate with API
- Number of requests ==> Performance?
- Traceability?
- Reusability?

Smart vs. Dumb Components

Smart / Controller	Dumb / Presentational
• 1 use case / route • Business logic • Container	• Independent of Use Case • Reusable • Often Leafs

View vs. Content

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

View vs. Content

```
@Component({
selector: 'app-tab',
template: `
    @if (visible) {
        <h1>{{title}}</h1>
        <div>
            <ng-content>No content</ng-content>
        </div>
    }
`
})
export class TabComponent {
    @Input() title = '';
    visible = true;
}
```

View

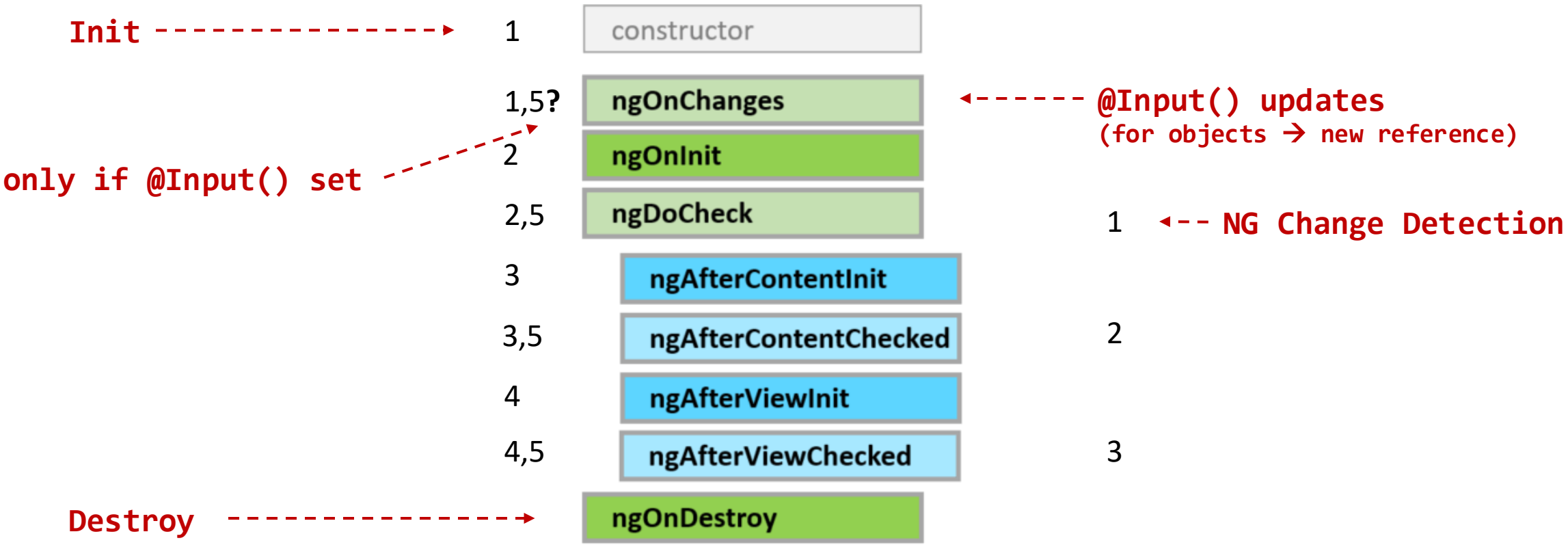
```
<app-tab title="Booked">
    Sample Text ...
</app-tab>
```

Content

Demo

Content projection

Lifecycle Hooks (in order of execution)



Where do we subscribe?

- 1 **Field initializer or constructor**
- 2 **If** @Input(s) needed → **ngOnInit** hook (needs destroyRef)
- 3 Elsewhere (needs injected destroyRef)

Why asynchronicity?

Asynchronous operations (API requests)

Interactive behavior (user input)

Websockets

Server Send Events (Push)

Managing your RxJS subscriptions

- Problem: Components create subscriptions without closing them
- Identify: .subscribe() without .unsubscribe() or other methods
- Solution: Unsubscribe from all Observables in your App
  - Except Angular Router Params

Why do we (always!) need to close sub?

Avoid side effects

Avoid memory leaks

Also for HttpClient‘s get / post …

How are subscriptions cancelled?

Observables	Observer
complete() error()	unsubscribe()

RxJS Subscription Management

- Explicitly with reference
  - readonly subscription = observable$.subscribe(…); // field initializer
    // subscription?.add(otherObservable$.subscribe(…)); // also possible since V6
    subscription?**.unsubscribe()**; // ngOnDestroy hook
- Implicitly with take until
  - ~~observable$.pipe(takeUntil(otherObservable)).subscribe(…);~~
  - observable$.pipe(**takeUntilDestroyed()**).subscribe(…);

last operator!

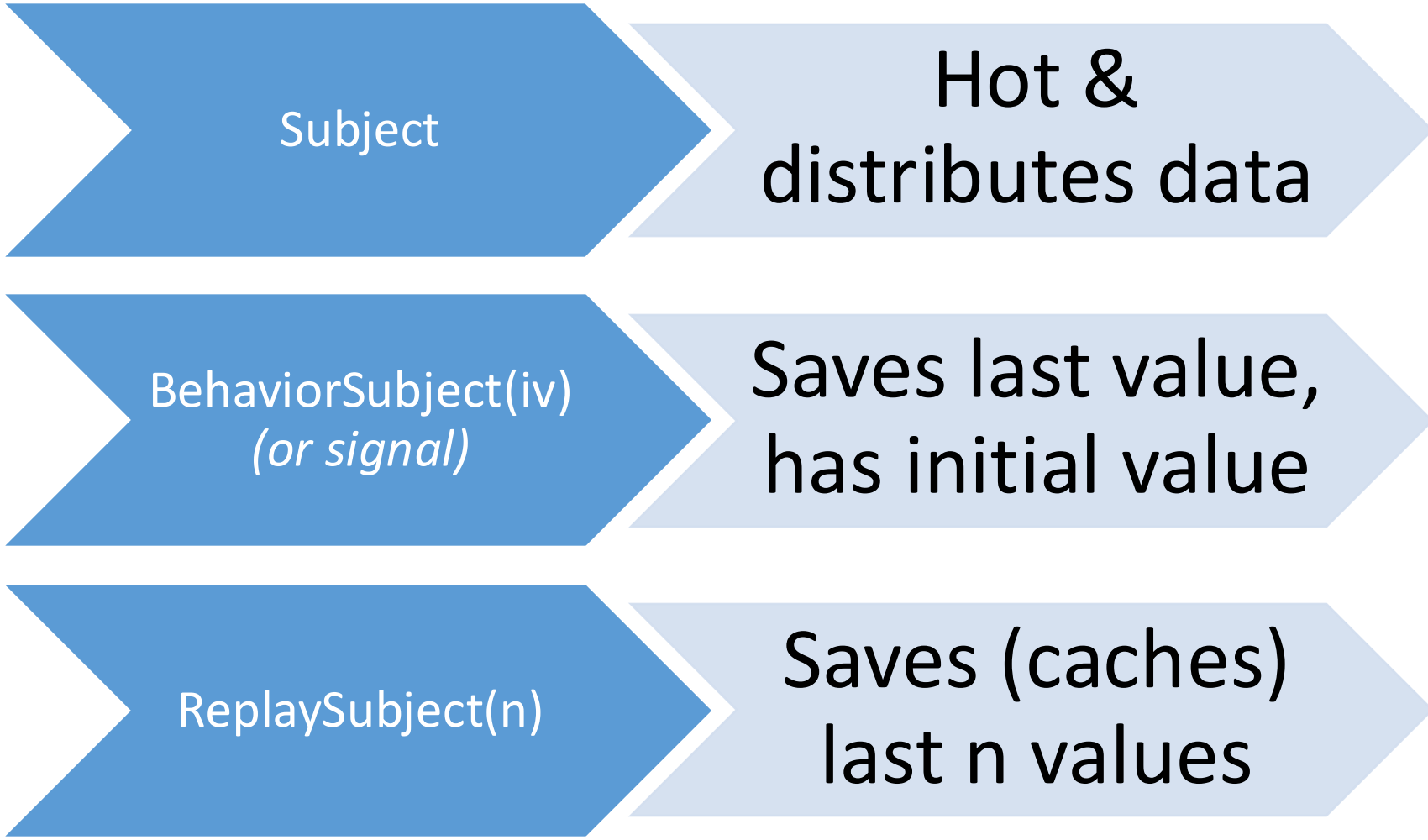
- Implicitly with async Pipe managed by Angular or using a Signal
  - {{ observable$ | **async** }} → **also triggers a cdr.markForCheck for OnPush ☺**
- Automatically managed by Angular
  - Router Params / ParamMap (only 1 I know where unsubscribing is not needed)

Demo

RxJS subscription management

ANGULAR
ARCHITECTS

Subjects


Subject
Hot &
distributes data
BehaviorSubject(iv)
(or signal)
Saves last value,
has initial value
ReplaySubject(n)
Saves (caches)
last n values

Subjects vs Signals – Details

RxJS Subjects (Eventing, State, Comparing)	Angular Signals (State)
Complex usage	Lightweight usage (especially getting current value)
Subscription management necessary	No subscription needed (done internally)
More features	Less powerful
Choose between • Eventing/Messaging (Subject) • State (BehaviorSubject) or • Comparing (ReplaySubject)	No choices, clearly opinionated
RxJS provides a **ton of functionality** to operate on observables like the map, filter, debounceTime & distinctUntilChanged, delay and retry operators	**Angular** provides two operators: • effect() *like tap() and subscribe* and • computed() *for all others* ☺
Using multiple subjects may lead to glitches	Diamond problem solved using multiple signals
RxJS currently by HTTP Client, Forms & Router	Optional for component inputs, outputs and queries

Demo

BehaviorSubject vs Signals

ANGULAR
ARCHITECTS

@angular/core/rxjs-interop

- toObservable(signal)
- toSignal(observable$)
- takeUntilDestroyed()
- outputFromObservable()

Demo

Signals computed & effected

Signal Components

(Starting with Angular 17.1)

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Demo

Signal inputs & queries

ANGULAR
ARCHITECTS

Conclusion

Fine-grained CD

Zone-less Future

Convertible to Observables and vice versa!

No need to unsubscribe!

No need to update code!

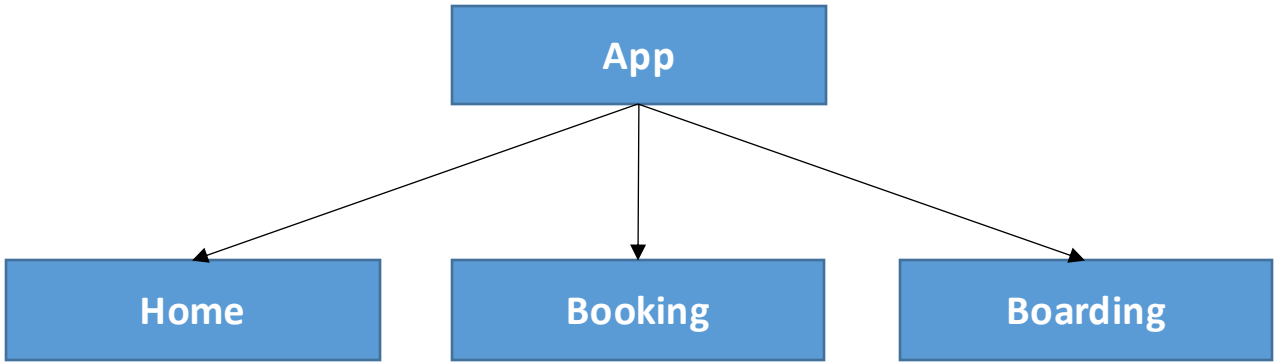
Angular Roadmaps

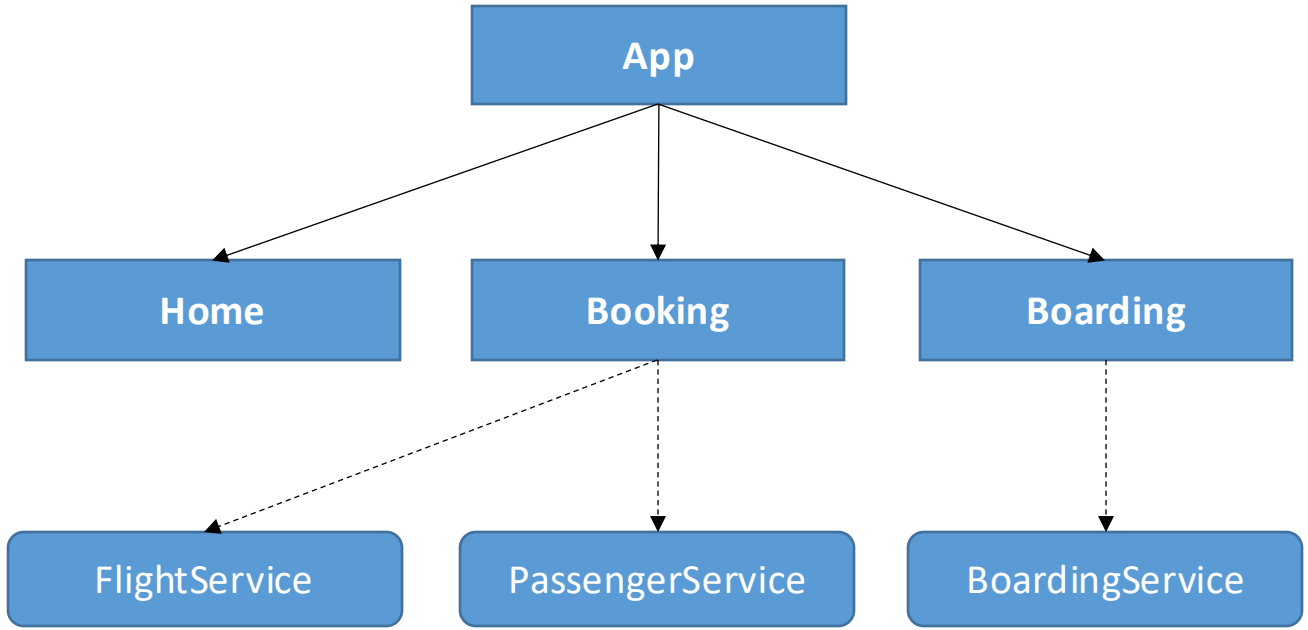
- Angular Feature Roadmap - caniuse?
  https://www.angular.courses/caniuse

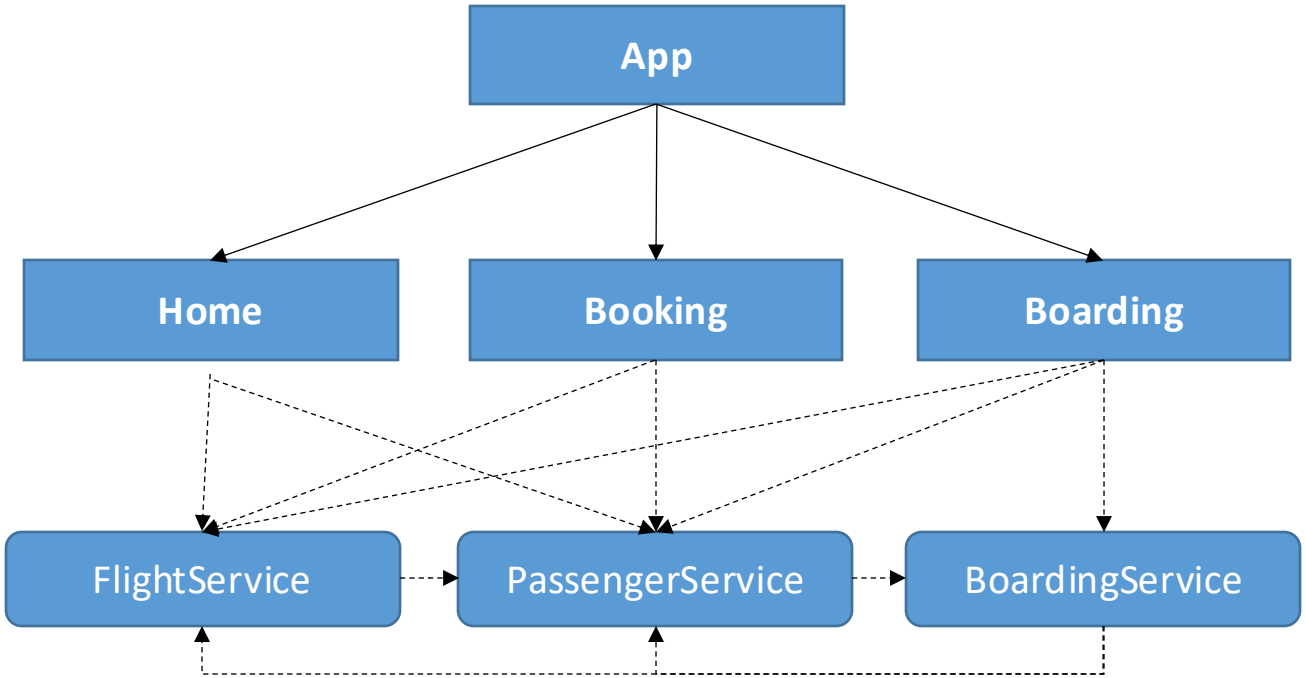
- Angular Reactivity Roadmap
  https://github.com/orgs/angular/projects/31/views/2

State Management

Motivation


App
Home
Booking
Boarding


App
Home
Booking
Boarding
FlightService
PassengerService
BoardingService


App
Home
Booking
Boarding
FlightService
PassengerService
BoardingService

Why State Management? I

- Single source of truth ☺
- Good predictability
- Good performance
- Clear & consistant architecture
  - no discussions btw. Devs or Teams
- Better maintainability
- Very smooth w. Angular
  - ChangeDetectionStrategy.OnPush &
  - Signals

Why State Management? II

- Easy to debug (with Redux DevTools)
- Easy to persist (e.g. localStorage)
- Easy to onboard new Devs
- Easy undo/redo
- Easy to test

State Management cons

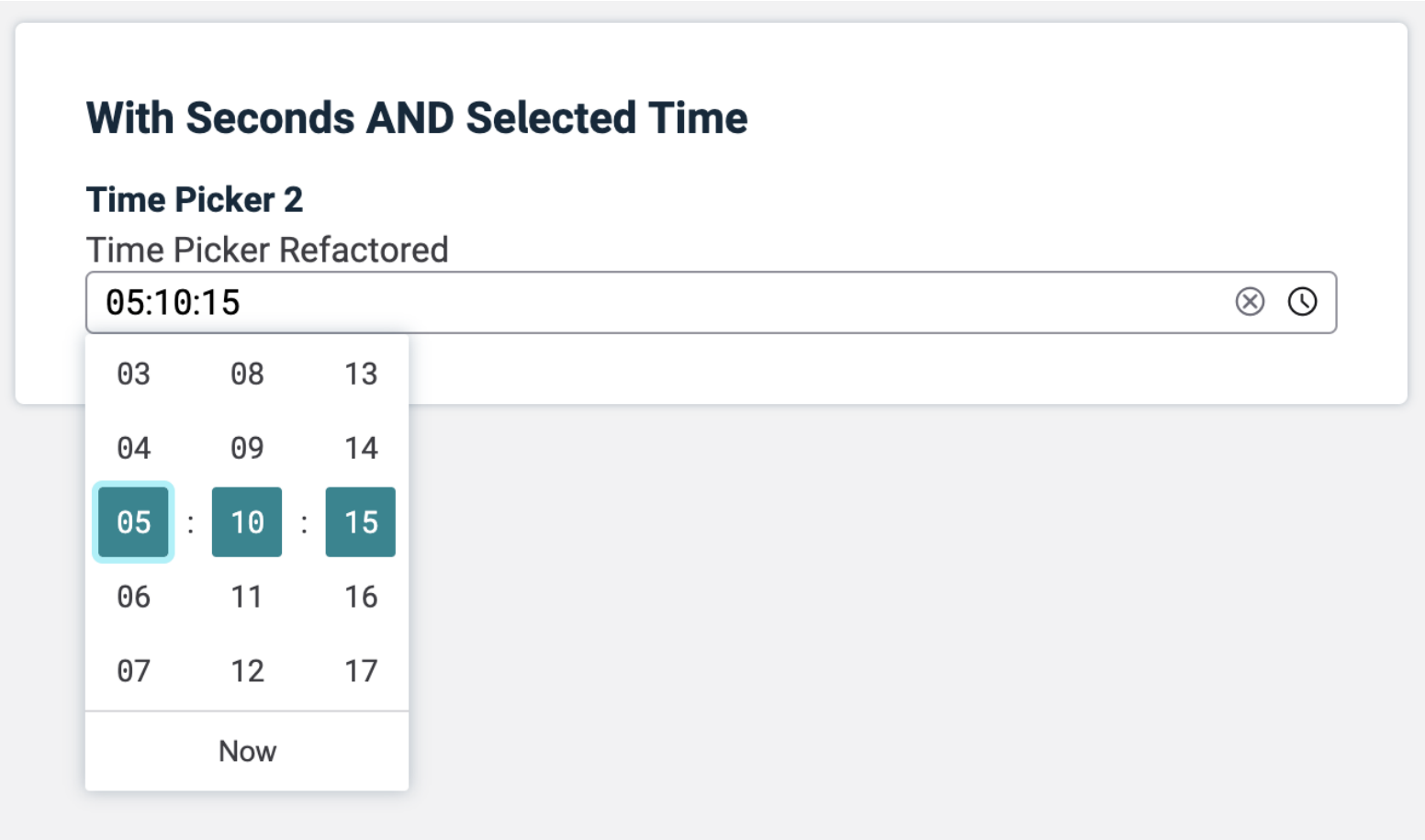
- Needs to be learned (steep curve like Angular)
- Strict architecture
- Less freedom
- Boilerplate

When do I need State Management?

- Complex applications
  - Checkout process
  - Draft / Edit process (e.g. multiple comp. or serv. accessing the same thing)
  - Filters / pagination
  - State used by multiple routes
  - State retrieved (e.g. from API)

- Complex components
  - Container/Controller (for features)
  - real world example: TimePicker (internally in presentational component)

Example: TimePickerComponent



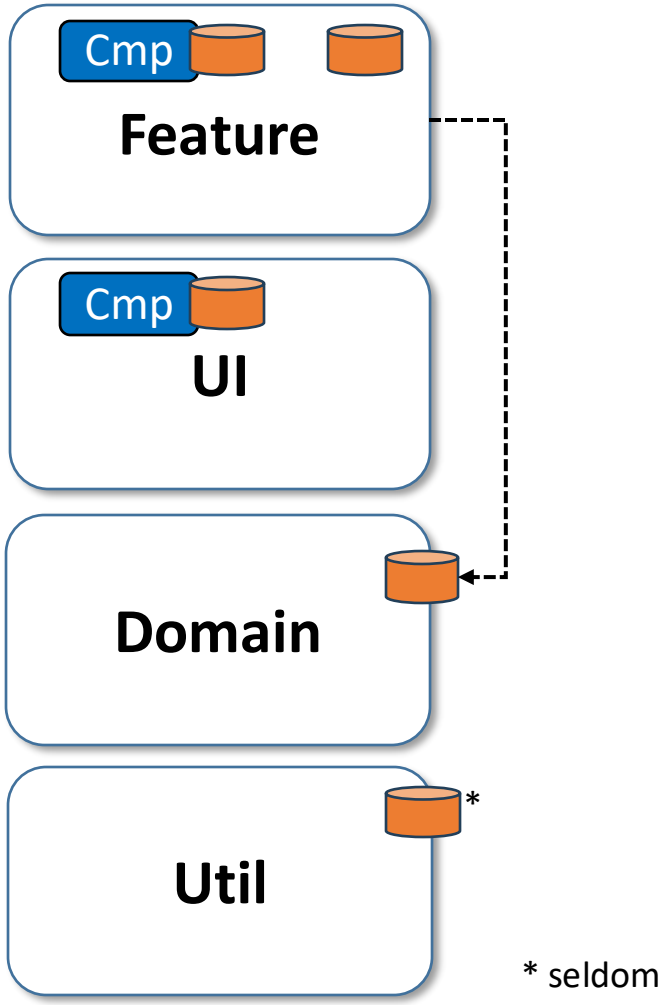
Global & features vs component (aka local)

- Global store
  - classic approach
  - 1 app store, can be split into features (like modules)
  - maybe even sub features (like sub modules)

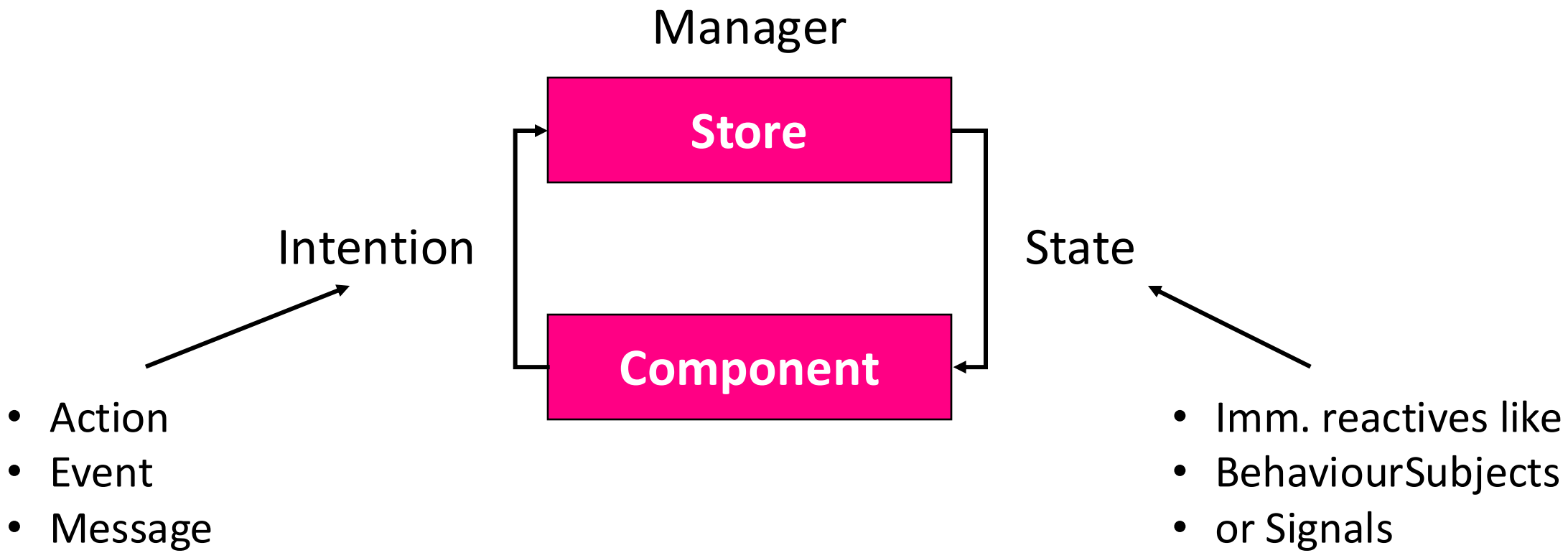
  - feature stores (mirroring your architecture)
    - for Enterprise apps
    - 1 store per feature (lib/folder)
    - Components that belong together share store

- Local store / component store
  - 1 store for 1 component

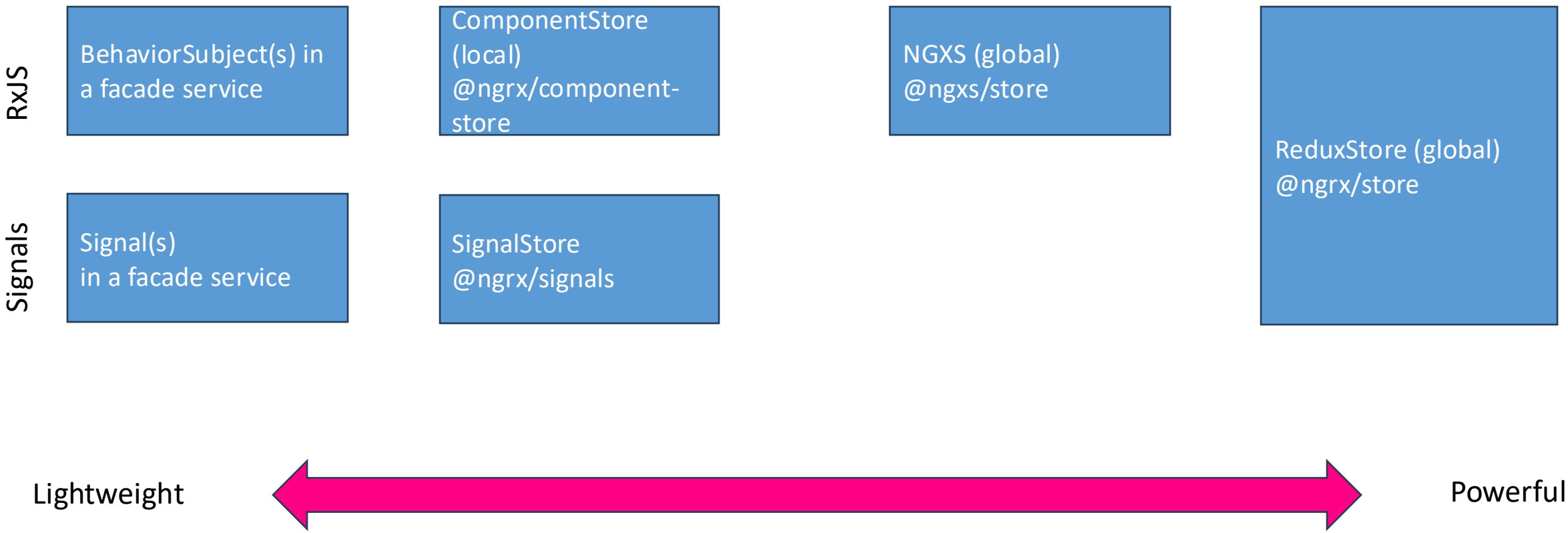
State Management and DDD



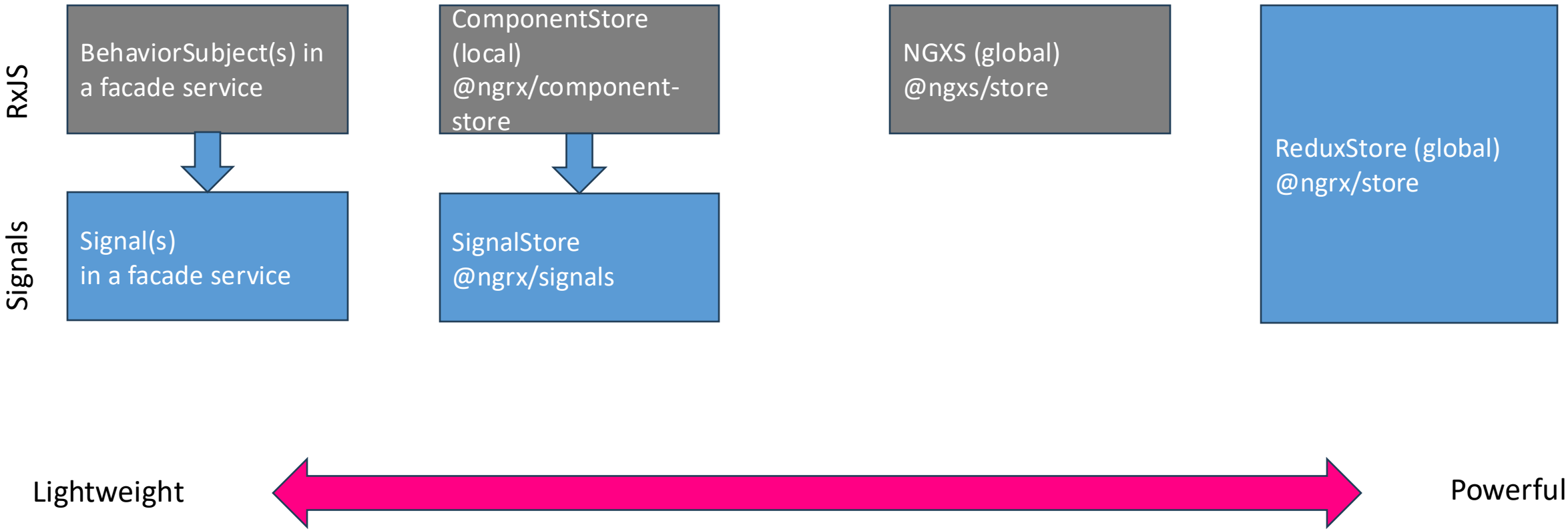
The store and the flow



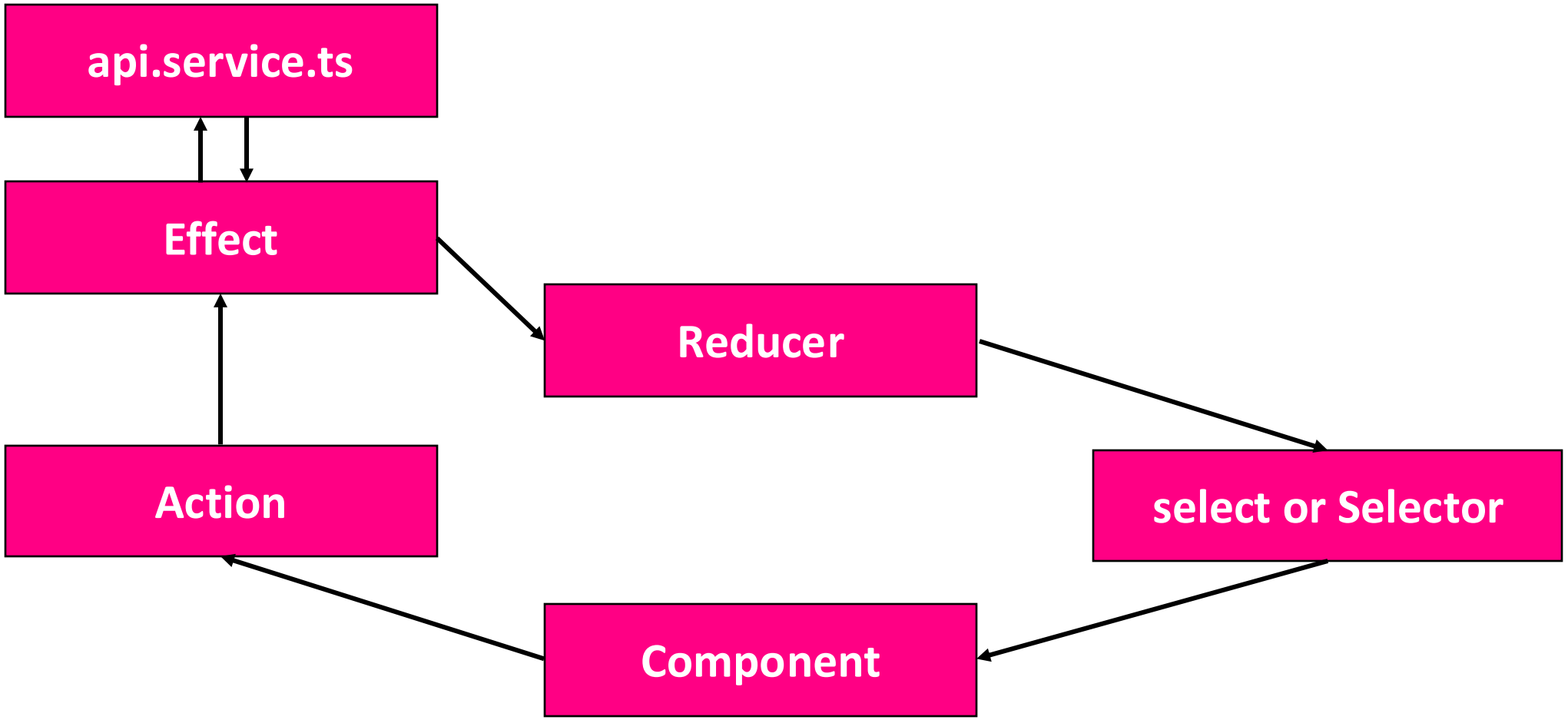
Which State Management solution?



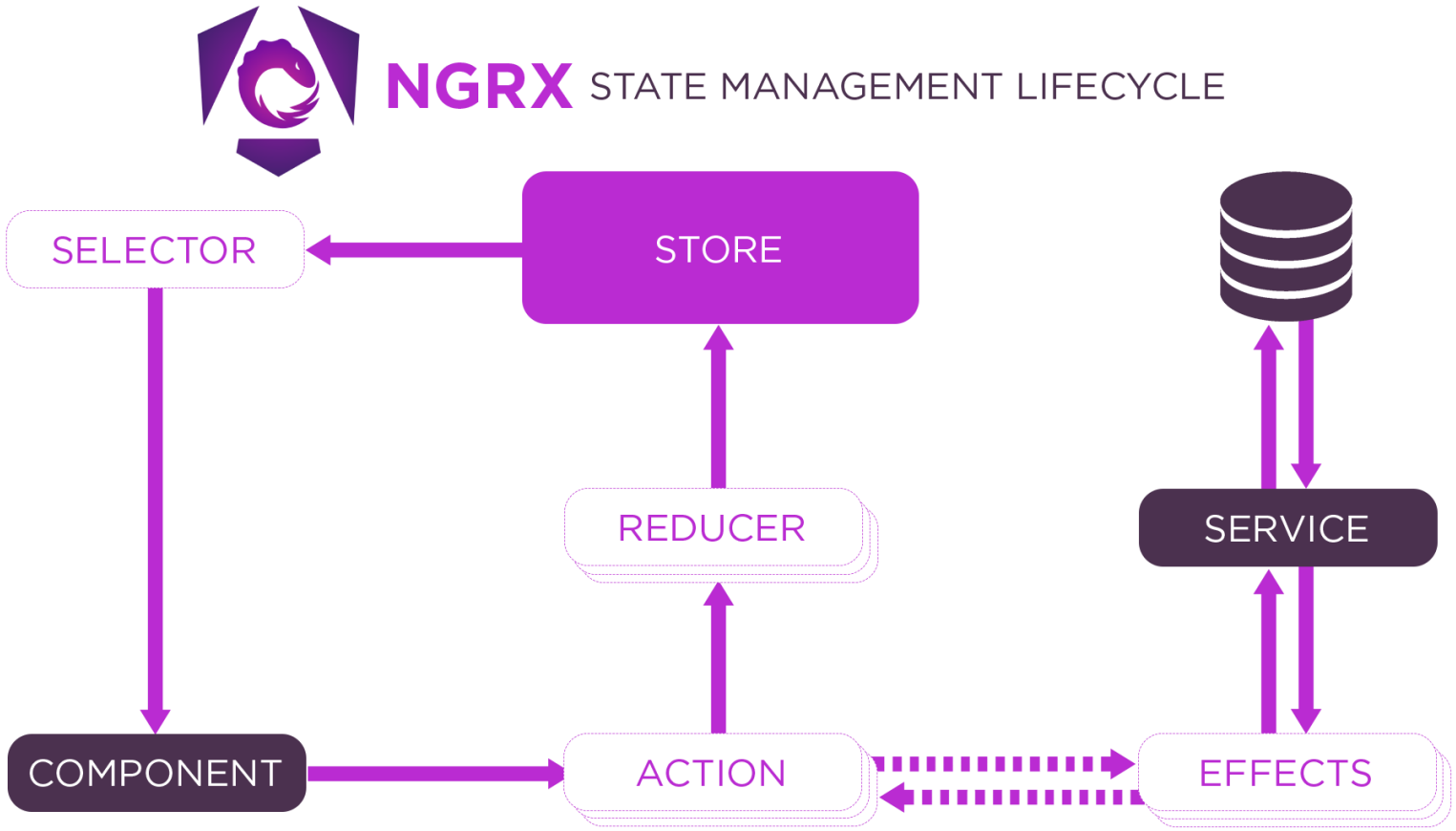
Which State Management solution?



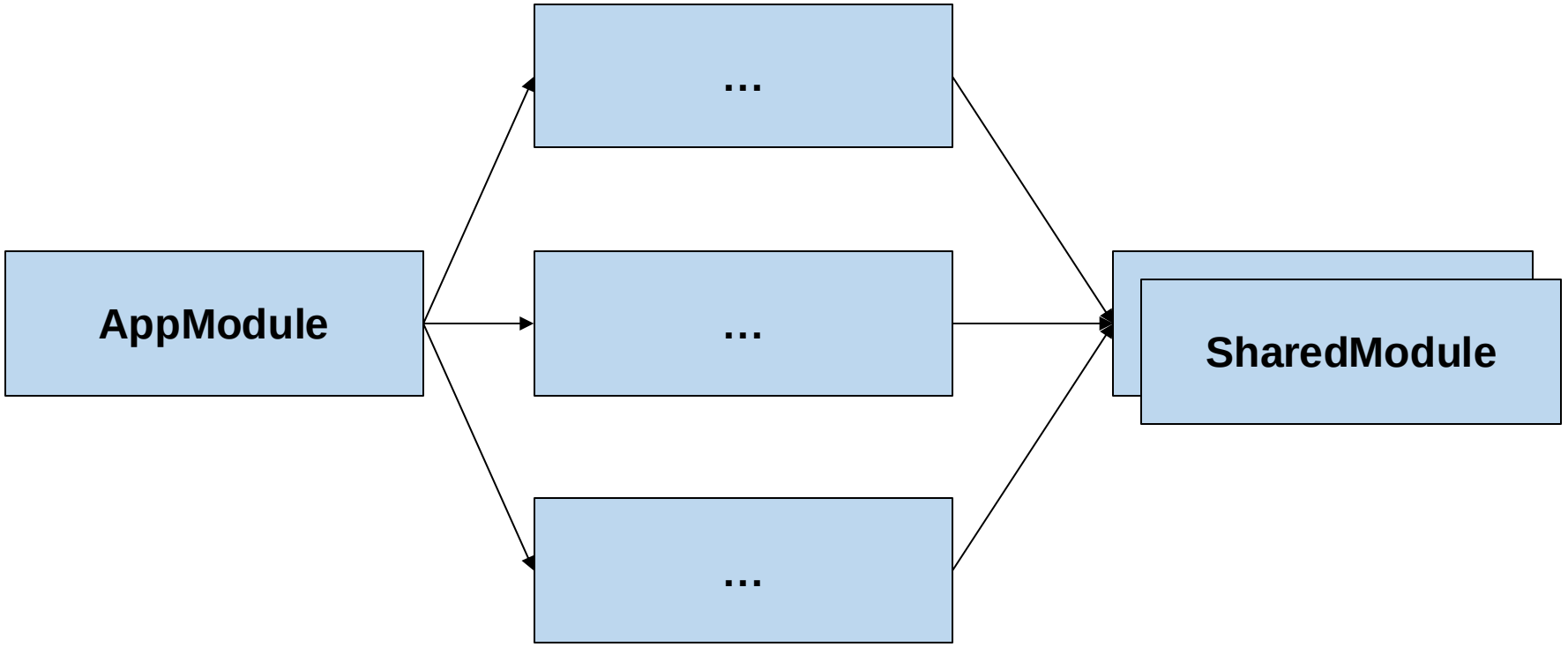
@ngrx/store and the redux flow



@ngrx/store and the redux flow



Typical Module Structure (deprecated)



Root Module **Feature Modules** **Shared Modules**


This is what Strategic DDD prevents

Outline

- Nx Monorepos or feature folders

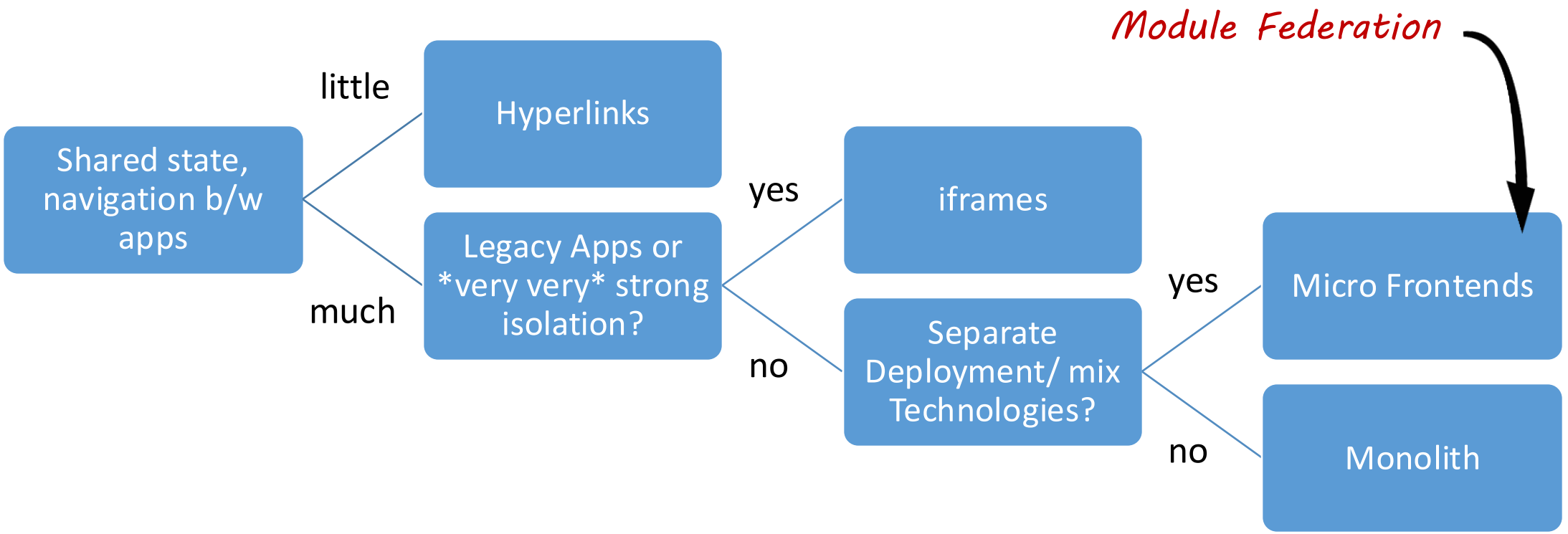
- Strategic Design and DDD

Demo

Nx Monorepo & DDD

ANGULAR
ARCHITECTS

Some General Advice



Best Choices

- Use templete-driven and reactive forms
- Use constructor-based or functional inject()
- Use class-based or functional guards
- Use smart vs dumb comp & content project.
- Use new control flow, migrate old directives
- Use RxJS subjects if needed, else signals
- Use state management (global or local)
- Use Nx monorepo or else folders with DDD

Conclusions