good parts

good parts

- TypeScript
- app component tree
- modules (root, features)
- JIT / AOT
- change strategies
- animations
- ng-container, if #else
- transclusion

- decorators
- marble testing

@michahell @ ZIVVER

good parts

- reactive forms
- view encapsulation
- guards, interceptors
- unit testing
- observables (RXJS)
- state management (NGRX)

@michahell @ ZIVVER

reactive forms

@michahell @ ZIVVER

reactive forms

- opposed to *template-driven forms*
- still requires template bindings
- but forms can be tested **without** needing the template
- create forms dynamically (easier), think specials (Angular example)
- does **Angular**, or do **you** create and manage **form** and **form controls** ?

reactive forms

reactive:

```
<input type="text" [formControl]="name">
```

template-driven:

```
<input type="text" [(ngModel)]="model.name">
```

@michahell @ ZIVVER

reactive forms

```
new FormControl(
  { value: 'defaultValue', disabled: false },
  [
    Validators.required,
    Validators.requiredTrue,
    Validators.min(2),
    Validators.max(10),
    Validators.minLength(2),
    Validators.maxLength(10),
    Validators.email,
    Validators.pattern(/[a-z]/),
    complexCustomValidator
  ]
)

// setting validators at a later time, sync:
formControl.setValidators(syncValidator)

// async:
formControl.setAsyncValidators(asyncValidator)
```

@michahell @ ZIVVER

reactive forms

- angular documentation
- reactive forms (custom) validators

@michahell @ ZIVVER

view encapsulation

@michahell @ ZIVVER

view encapsulation

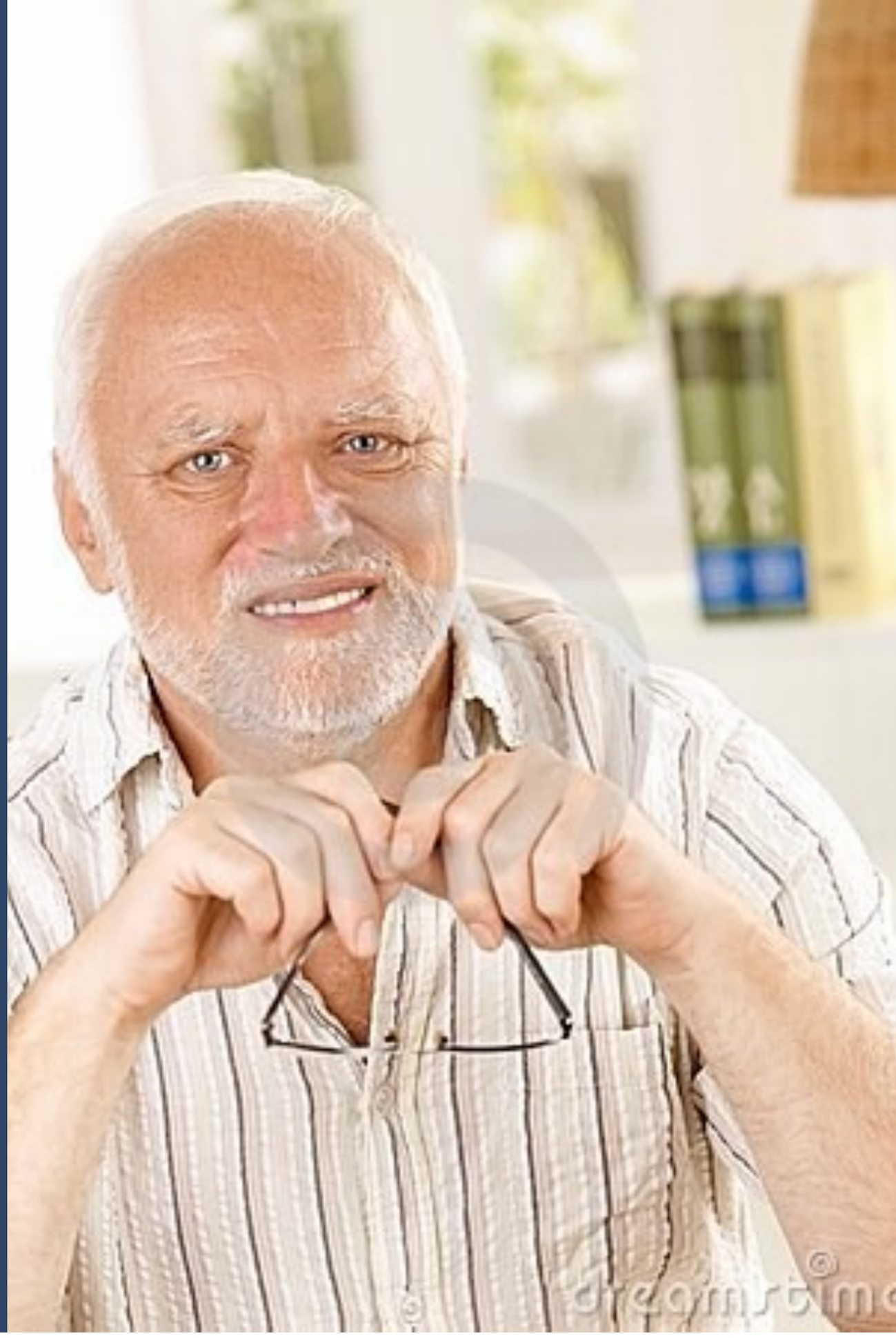
- VE in Angular (pascal precht)
- ViewEncapsulation.**None**
- ViewEncapsulation.**Emulated** (default)
- ViewEncapsulation.**Native**
- **/deep/, >>>, and ::ng-deep** (deprecated)
- Angular Material about styling other components

view encapsulation

my code works. I don't know why

- ⚡ **stackBlitz** <- don't click. everything just works

@michahell @ ZIVVER



guards, interceptors

@michahell @ ZIVVER

guards

- comparable to the AngularJs **resolve** route guard
- authentication
- guaranteeing resources exist
- check for supported browser features
- angular docs route guards
- protecting routes using guards

```ts
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
    console.log('AuthGuard#canActivate called');
    return true;
  }
}
```

```
{
    path: 'compose-overview',
    canActivate: [
      AuthenticatedGuard,
      CanComposeGuard
    ],
    component: RightpaneComposeOverviewComponent
}
```

interceptors

- used to exist in AngularJs
- came back in Angular 4.3
- intercept any or all Angular HTTP requests*
- authentication
- retrying
- [httpclient interceptors]

*** only Angular requests**

so if you use a third party library, like OfficeJs, and it does it's own requests, those are NOT intercepted.

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { AuthService } from './auth/auth.service';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class TokenInterceptor implements HttpInterceptor {

  constructor(public auth: AuthService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    request = request.clone({
      setHeaders: {
        Authorization: `Bearer ${this.auth.getToken()}`
      }
    });
    return next.handle(request);
  }
}
```

unit testing

@michahell @ ZIVVER

unit testing in general

- clear way of setting up tests
- documentation is good
- easy to mock dependencies (w/ useClass + useValue)
- shallow testing with **NO-ERROR-SCHEMA**
- easy to test: services, reducers, guards, interceptors
- ok: components
- harder: components, directives (hosts), @effects

services, reducers easy to test and contain lots of business logic. then, Effects, then shallow components.

```ts
import {async, ComponentFixture, TestBed} from '@angular/core/testing';
import {NO_ERRORS_SCHEMA} from '@angular/core';

import {NoopComponent} from './noop.component';

describe('NoopComponent', () => {
  let component: NoopComponent;
  let fixture: ComponentFixture<NoopComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      providers: [],
      declarations: [NoopComponent],
      schemas: [NO_ERRORS_SCHEMA]
    })
      .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(NoopComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

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

```ts
import {async, ComponentFixture, TestBed} from '@angular/core/testing';
import {NO_ERRORS_SCHEMA} from '@angular/core';

import {NoopComponent} from './noop.component';

describe('NoopComponent', () => {
  let component: NoopComponent;
  let fixture: ComponentFixture<NoopComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      providers: [],
      declarations: [NoopComponent],
      schemas: [NO_ERRORS_SCHEMA]
    })
      .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(NoopComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

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

more elaborate component
(not even that much deps...)

```
import {MockUtilService} from '@@lib/testing/mocks';
import {async, ComponentFixture, TestBed} from '@angular/core/testing';
import {CUSTOM_ELEMENTS_SCHEMA} from '@angular/core';
import {RouterTestingModule} from '@angular/router/testing';
import {Router} from '@angular/router';
import {Store} from '@ngrx/store';
import {of} from 'rxjs';
import {OwaContextService, UtilService, ValidationService} from '@@lib/services';
import {AppState, selectConversation, selectDraft, selectValidation} from '@@lib/state';
import {Conversation, Draft, DraftValidation, Violation, ZivverDraftType} from '@@lib/models';
import {RightpaneComposeOverviewComponent} from './rightpane-compose-overview.component';
import {SectionBackButtonComponent} from '../../components/section-back-button/section-back-
button.component';
import {SectionHeaderComponent} from '../../components/section-header/section-header.component';
import {ExtLinkDirective} from '../../directives/extLink.directive';

const mockDraft: Draft = new Draft();
mockDraft.zivverType = ZivverDraftType.plain;

const mockValidation: DraftValidation = new DraftValidation({
  validDraft: true
});

const mockConversation: Conversation = new Conversation();

describe('RightpaneComposeOverviewComponent', () => {
  let component: RightpaneComposeOverviewComponent;
  let fixture: ComponentFixture<RightpaneComposeOverviewComponent>;
  let router: Router;
```

```
beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [
        RouterTestingModule.withRoutes([
          { path: 'compose-overview', component: RightpaneComposeOverviewComponent }
        ])
      ],
      providers: [
        {
          provide: OwaContextService,
          useValue: {
            getComposeItemId: () => Promise.resolve(),
            getLanguage: () => 'NL'
          },
        }
      },
      {
        provide: Store,
        useValue: {
          select: (selectFn: (s: AppState) => any) => {
            switch (selectFn) {
              case selectDraft: return of(mockDraft);
              case selectValidation: return of(mockValidation);
              case selectConversation: return of(mockConversation);
            }
          },
          dispatch: () => {}
        }
      },
```

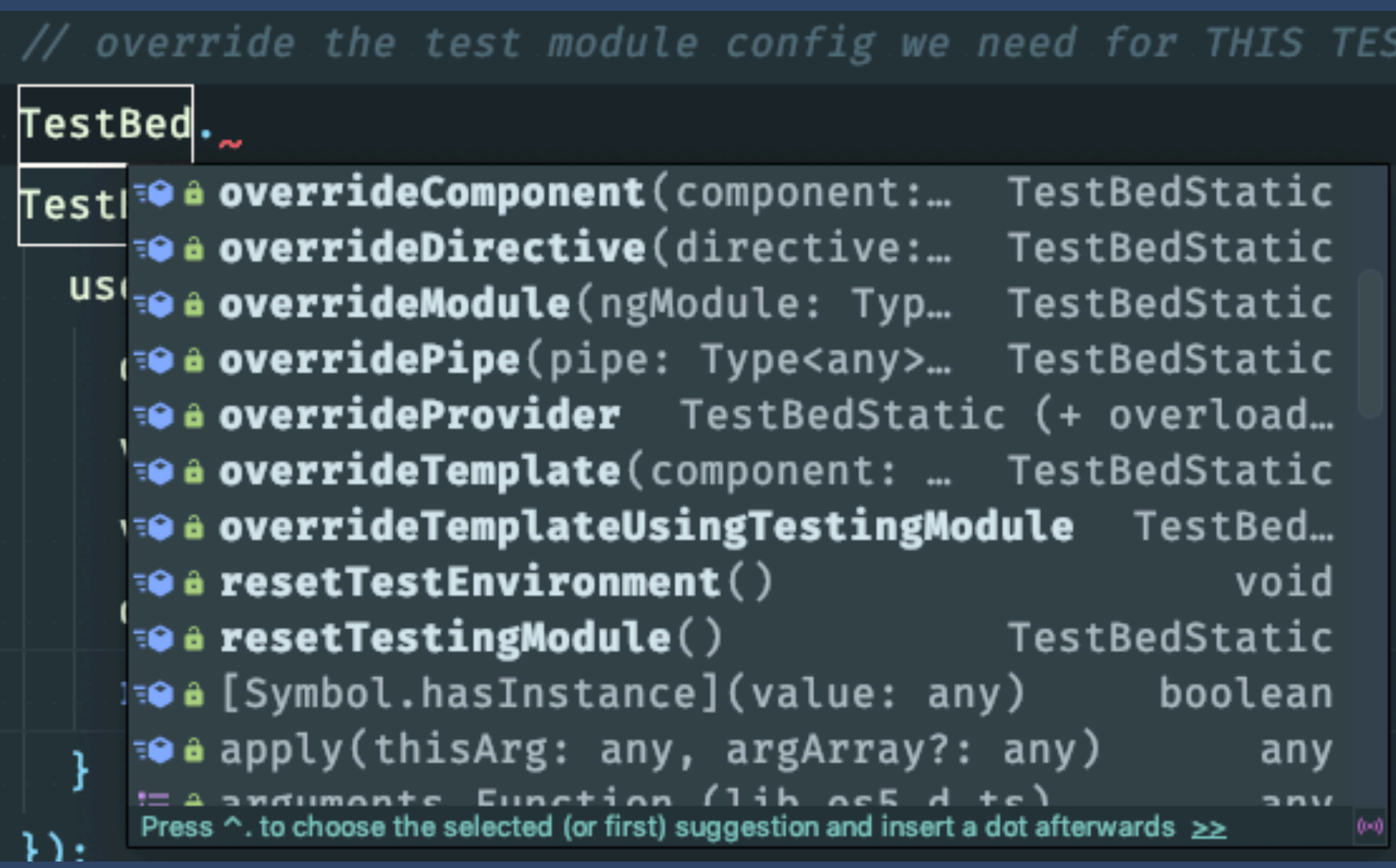
```
      {
        provide: ValidationService,
        useValue: {
          validationResult$: of(new DraftValidation()),
          violations$: of([new Violation()]),
        }
      },
      {
        provide: UtilService,
        useClass: MockUtilService
      }
    ],
    declarations: [
      RightpaneComposeOverviewComponent,
      SectionHeaderComponent,
      SectionBackButtonComponent,
      ExtLinkDirective
    ],
    schemas: [CUSTOM_ELEMENTS_SCHEMA]
  })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(RightpaneComposeOverviewComponent);
    router = TestBed.get(Router);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

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

abstract boilerplate away

- have a default **TestBed**, everything dep mocked
- **.override()** deps you need

```
beforeEach(() ⇒ {
  // start with base testbed
  TestBed.configureTestingModule(
    createDefaultTestBed(ValidationDetailsComponent, configureStore(mockAppState))
  );
  // override the test module config we need for THIS TEST (f.e. providers)
  TestBed.overrideProvider(ValidationService, {
    useValue: {
      draft$: of(mockDraft),
      validationResult$: of(mockDraftValidation),
      violations$: of(mockViolations),
      dismissibleViolations$: of(''),
      resolveViolation: () ⇒ {}
    }
  });
});
```

```
export function createDefaultTestBed (forClass: any, initialStoreState: any) : TestModuleMetadata {
  return {
    imports: [
      MatSnackBarModule,
      StoreModule.forRoot(APP_REDUCERS, {
        'initialState': initialStoreState
      }),
    ],
    providers: [
      Store,
      ActionsSubject,
      // ...
      {
        provide: ValidationService,
        useValue: {
          draft$: of(''),
          validationResult$: of(''),
          violations$: of(''),
          dismissibleViolations$: of(''),
          resolveViolation: () => {}
        }
      }
      // ...
    ],
    declarations: [
      forClass
    ],
    schemas: [
      NO_ERRORS_SCHEMA
    ]
  };
}
```

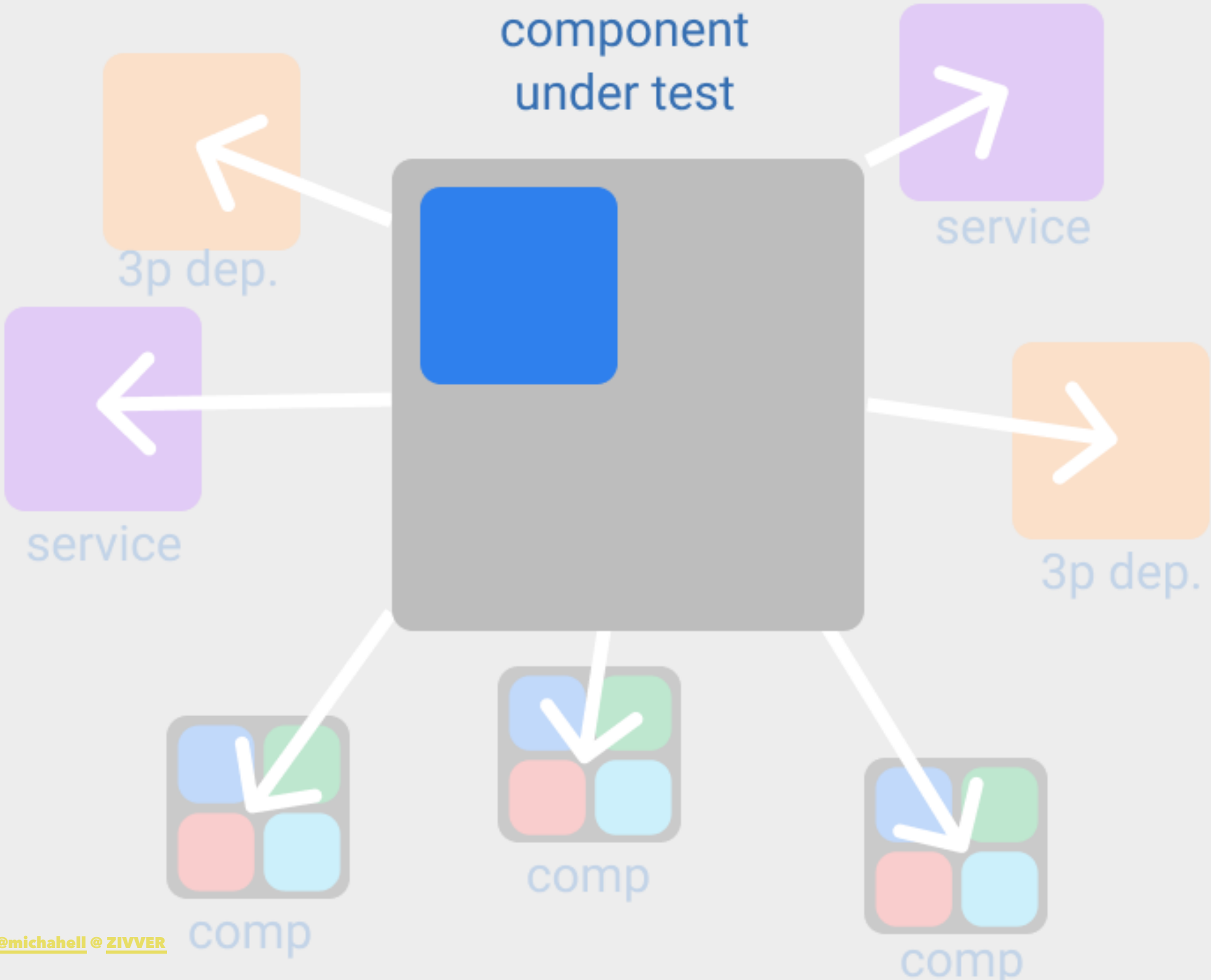

// override the test module config we need for THIS TEST (f.e. providers)
TestBed.
overrideComponent(component:… TestBedStatic
overrideDirective(directive:… TestBedStatic
overrideModule(ngModule: Typ… TestBedStatic
overridePipe(pipe: Type<any>… TestBedStatic
overrideProvider TestBedStatic (+ overload…
overrideTemplate(component: … TestBedStatic
overrideTemplateUsingTestingModule TestBed…
resetTestEnvironment() void
resetTestingModule() TestBedStatic
[Symbol.hasInstance](value: any) boolean
apply(thisArg: any, argArray?: any) any
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards >>

testing components

- component class (w/o tmpl)
- shallow testing (w/ tmpl + shallow)
- integration testing (w/ tmpl)

three ways of testing

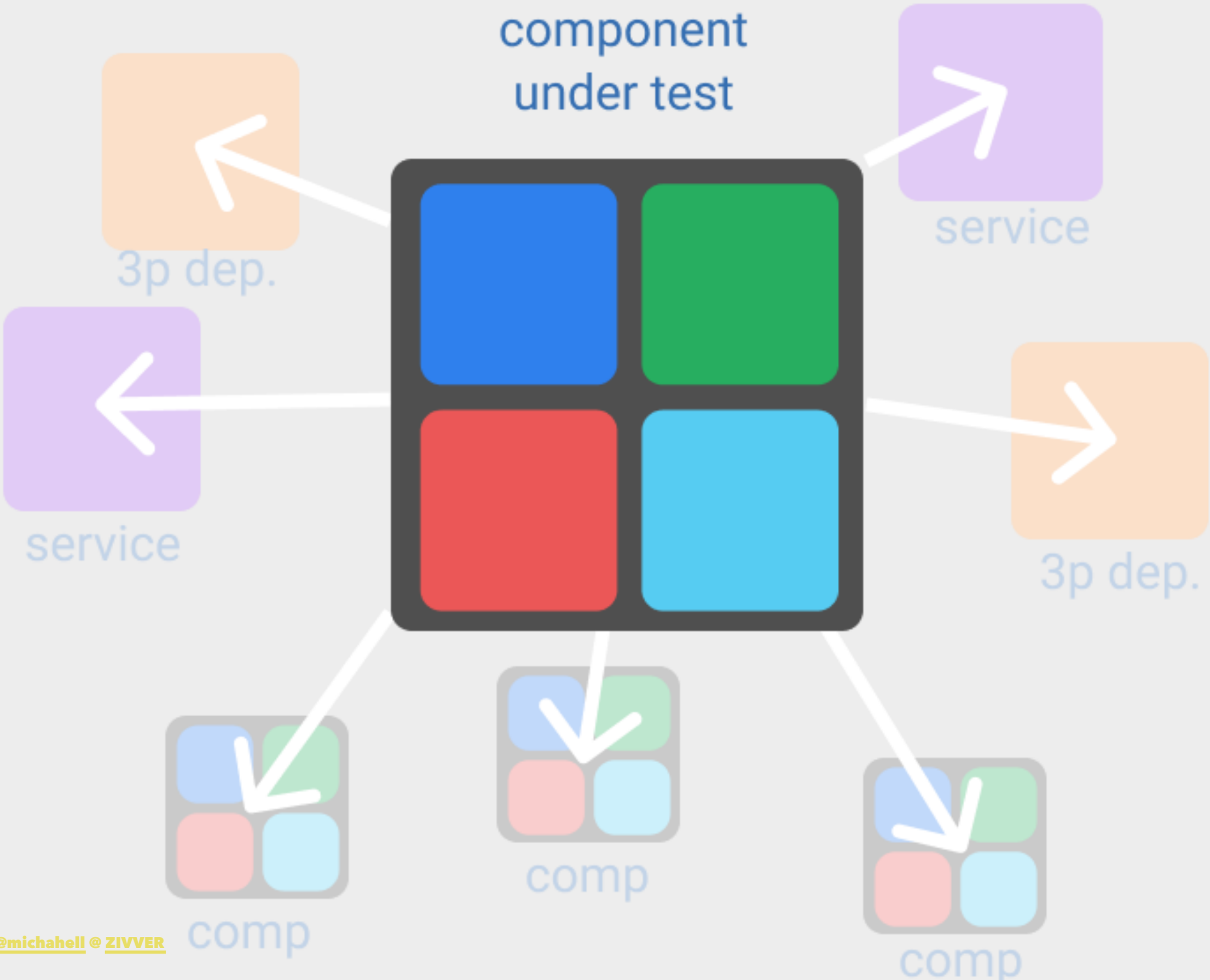
want to go over this quick because it helps determining what you actually want to test.



component class

- when?
- only instantiate the component class itself
- to test local logic (smart / no redux-like state mgmt)

```
const c = new ComposeComponent(<any>route, time, actions);

// performing an action
c.form.setValue({
  title: 'Categorical Imperative vs Utilitarianism',
  body: 'What is more practical in day-to-day life?'
});
c.onSubmit();

// reading the emitted value from the subject
// to make sure it matches our expectations
expect(actions.value.conversationId).toEqual(11);
```

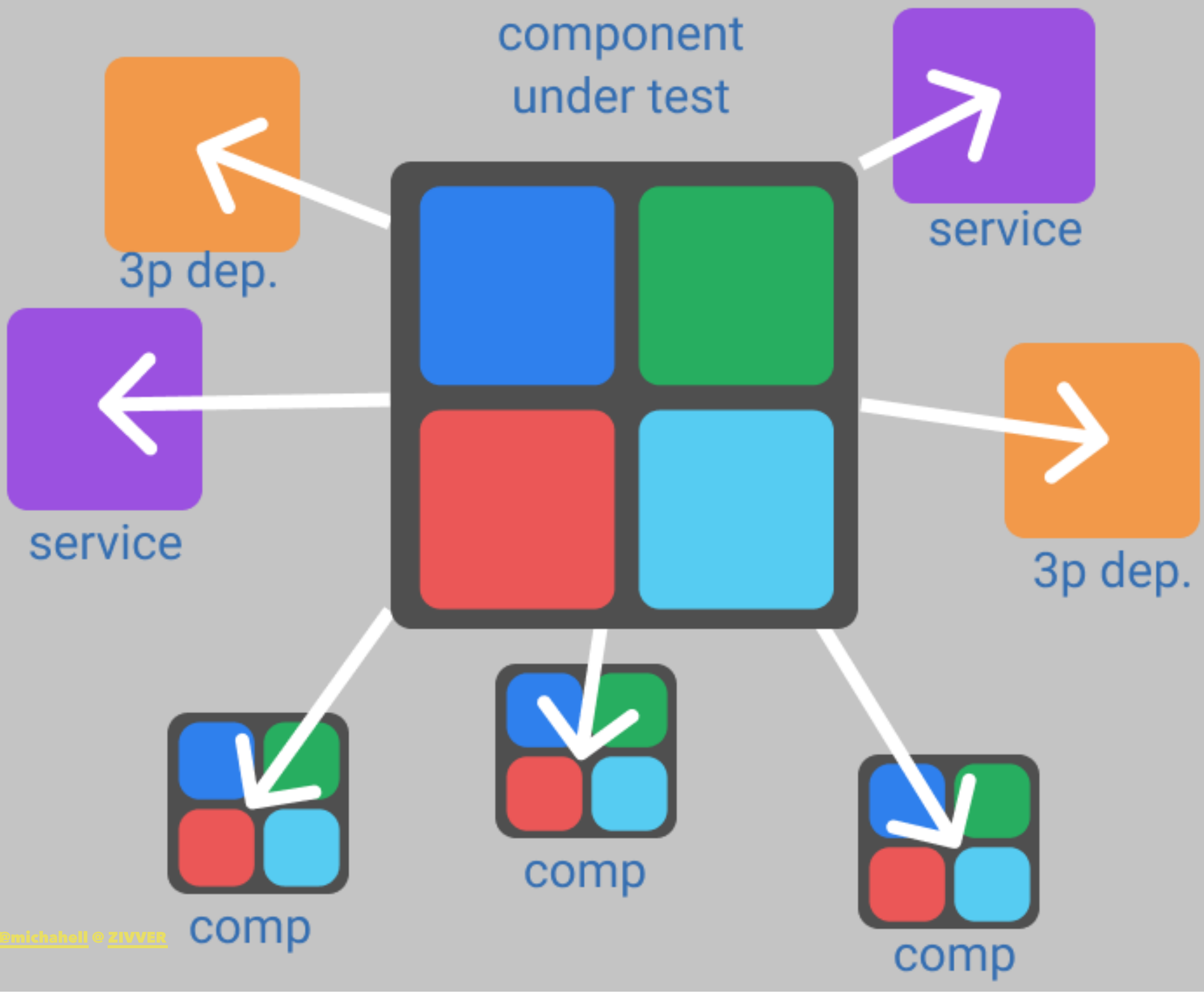


shallow testing

- schemas: [**NO_ERRORS_SCHEMA**]
- when?
- make sure UI elements are visible
- have the correct state (enabled / disabled)
- basic behaviour

Did you ever think about why the test is called .spec?

All unit tests in OWA are **Shallow tests**.



integration testing

- default when using Angular CLI because no schema
- when?
- test component ánd integration with 3p. deps / services
- + all of shallow testing

mocking dependencies

3 ways of mocking dependencies:

- **useValue**:

```
provide: RealService,
useValue: {
    realMethodName : () => 'fakeResult';
}
```

- **useClass**:

```
provide: RealService, useClass: MockRealService
```

- use realService**.spyOn()** without **.callthrough()**

unit testing caveats

- correct return types in mocked methods!
- don't use global methods/ functions -> Services
- it used to be easier to not mock the **Store** and just populate it with mock data
- but now, it should be mocked... when *shallow testing*

2) global methods harder to test / mock.

General advice is to put them in services. in OWA -> UtilService

worth looking at

- auto mocking libraries
- model mock data generators
- **Spectator**
- mostly just a wrapper library
- reduces boilerplate, like CodeceptJs for e2e tests: cleaner syntax

auto mocking libraries

Alwin -> library.

observables

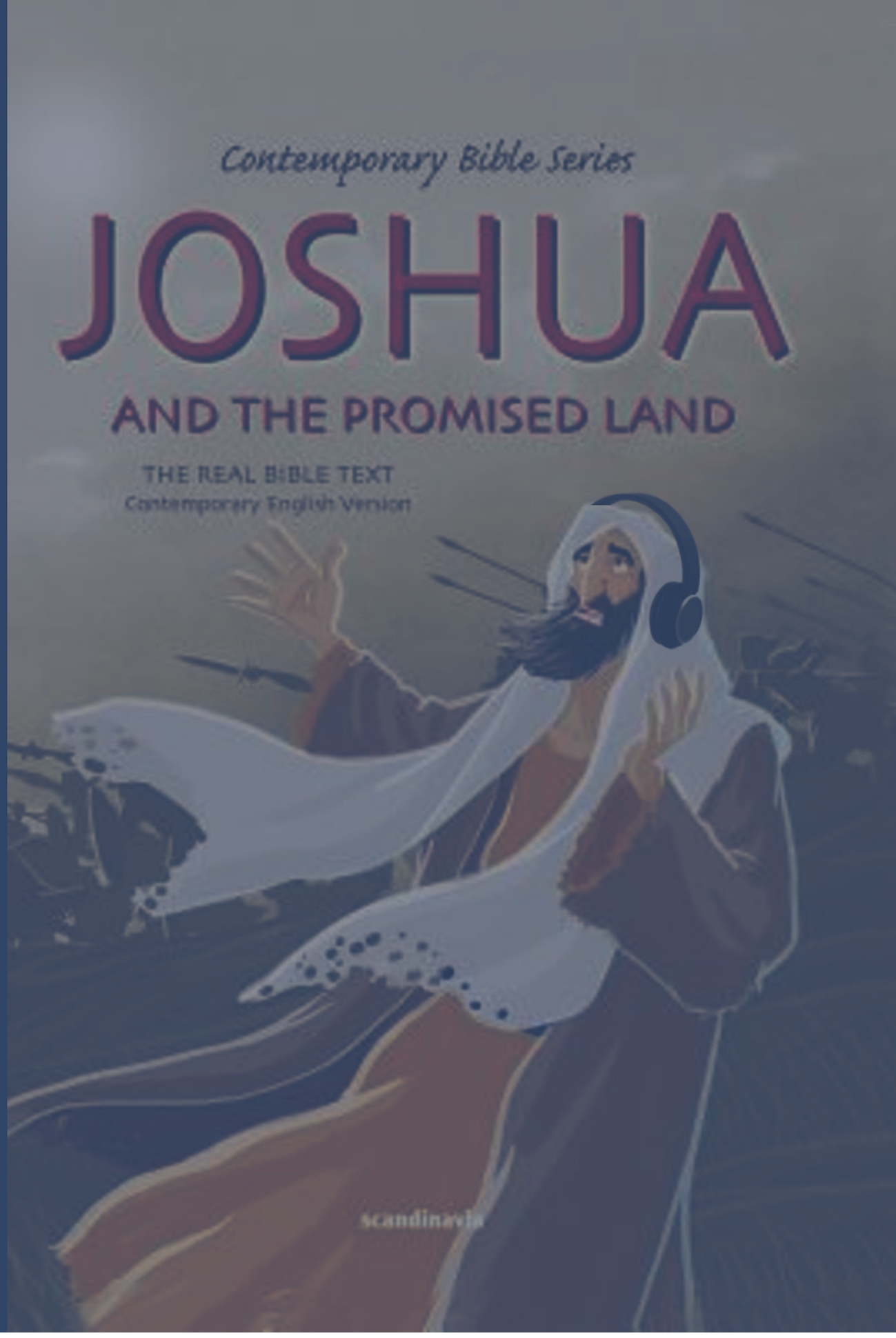
- observable or promise?
- .pipe() or subscribe()?
- pipe() + tap() or new obs$?
- observable lifetime
- hot vs cold, unicast vs multicast
- pro's and cons

@michahell @ ZIVVER

observable or promise?

- **promise**: one-off, eager, not cancelable
- **observable**: multiple emits, lazy, **cancelable**
- handler functions: promise -> async, Obs -> sync.
- promise -> observable : **from(promise);**
- observable -> promise : **observable.toPromise();**
- some operators accept **Promise**s directly

@michahell @ ZIVVER



.pipe(code) or subscribe(code)?

- .subscribe(**onEmit**, onError, onComplete)
- 99% can be done with **operators** so prefer **.pipe()**
- pipe reuse (lazy evaluated), pure (no side effects)
- don't know what operator you need? Use **.subscribe()**, figure it out later (teammate / google)
- ⚡ stackBlitz

pipe reuse: because
Observable, not
Subscription
if you don't know up front,
you can always code in
the **.subscribe()** first,
but treat it as a red flag!

.pipe() + tap() or new obs$?

- observable 'chain' that does something
- you realise you still need the 'original'
- you throw in a **tap()**. Done!
- **but it feels wrong**... why?
- I struggled with this 👶
- ⚡ stackBlitz

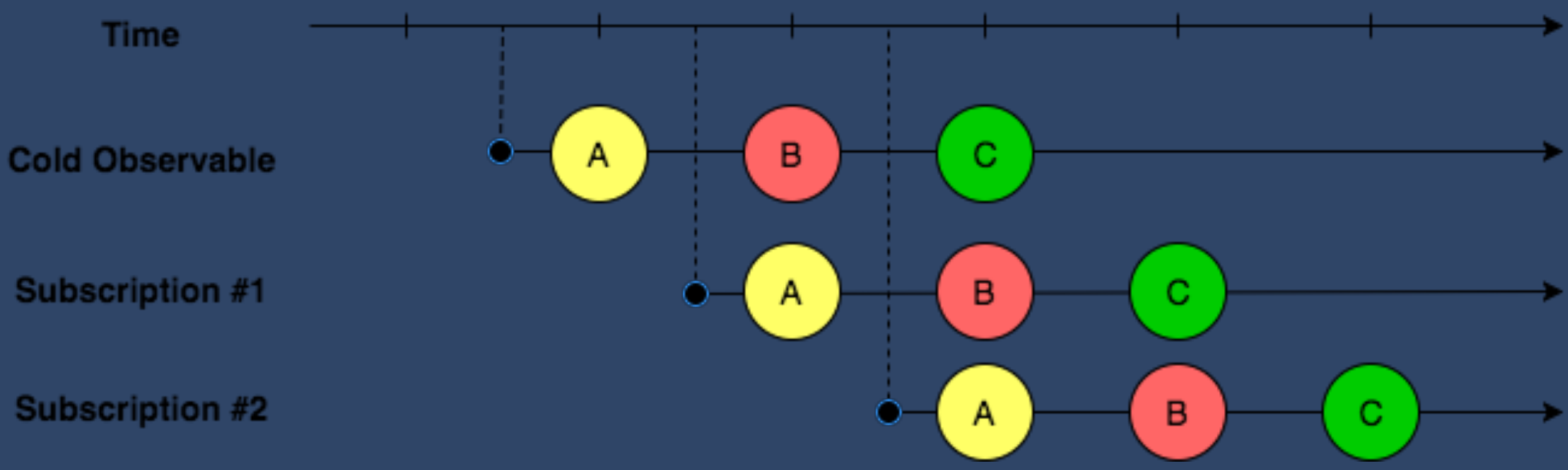
observable lifetime

- subscription**.unsubscribe()**
- template **| async** vs **subscribe()**
- simple completion operators: **take(), first(), last()**, etc.
- complex completion operators: **takUntil(), takeWhile()**
- don't .unsubscribe()
- ⚡ stackBlitz
- **switchMap()**: ⚡ stackBlitz

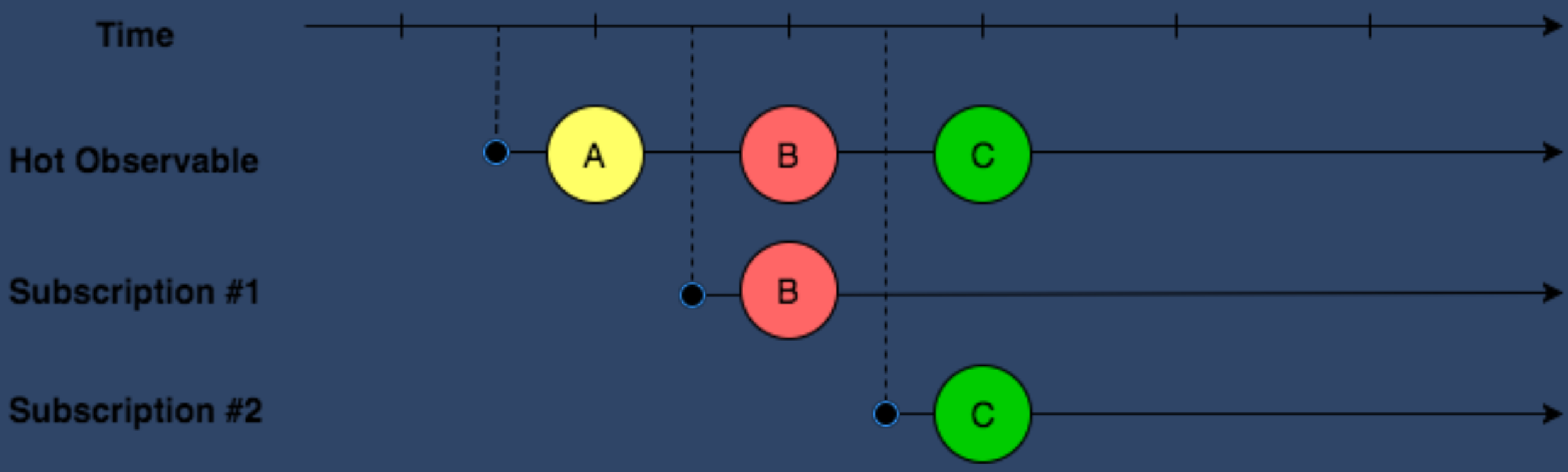
🔥 hot vs. cold 🌧️ observables

- **cold**: data source does not exist. is instantiated upon **.subscribe()**
- **hot**: data source **exists** and is instantiated only *once*.
- relates to **unicast / multicast (multicast operators)**
- ⚡ stackBlitz

unicast


Time
Cold Observable
A
B
C
Subscription #1
A
B
C
Subscription #2
A
B
C

multicast



pro's

- really great to work with once you wrap your head around thinking **declarative + streams**
- makes for very clear lists of declarative steps your code takes
- splitting code paths (if, switch-case etc.) can be tackled by separating into separate 'chains'

cons

- harder to write error handling code. debug. test.
- infinite waiting mistakes
- new way of causing memory leaks
- operators. read and **test what they do!** (⚡)
- some **higher-order observables** are harder to reason about: **concatMap(), mergeMap()**
- some are easier: **forkJoin(), withLatestFrom(), combineLatest()**

operators

- every operator = different.
- Confusing. operator syntax is simple, results can be very different.

caveats

- **always** check what kind of observable you are chaining at any point, is it a source observable? or a 3-level deep **inner observable**?

helpful tooling

- RxViz (visualising RXJS operators)
- RXJS documentation 🔥 firebaseapp 🔥
  (accurate / up to date)

RXJS documentation:

🔥 firebaseapp 🔥

further reading

- Angular RXJS documentation
- learning observable by building observable
- 6 operators you must know
- multicasted observables in RXJS
- ngx-take-until-destroy

state management (NGRX)

@michahell @ ZIVVER

state management (NGRX)

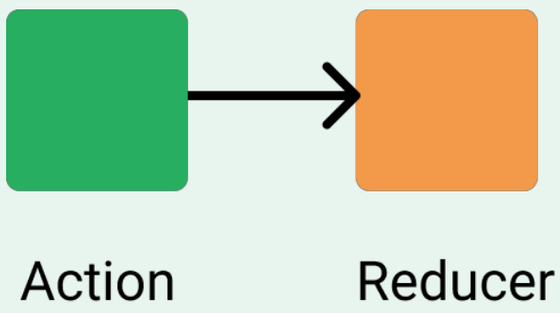
- ~~basics: **Store, Action, Reducer, Effect**~~
- ~~dealing with collections of things~~
- ~~reusing Actions~~
- **@Effect**s and most common patterns
- store-devtools

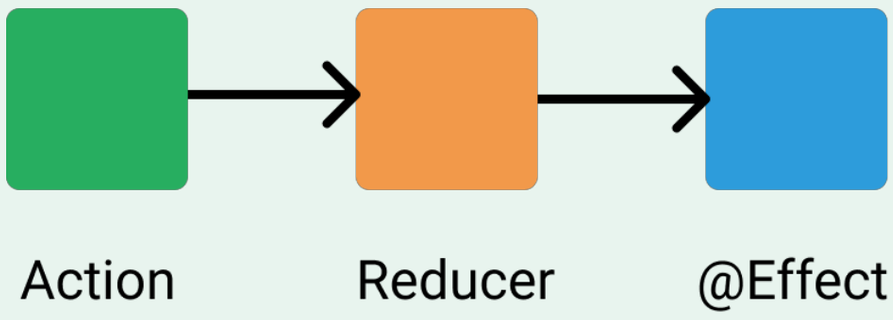
@Effects

- why? manage side effects, manage actions
- splitter: **Action** > [n **Action**s]
- aggregator: [N **Action**s] > **@Effect**
- chains: **Action** > **Action** > **Action**
- stop using effects for that

@Effects

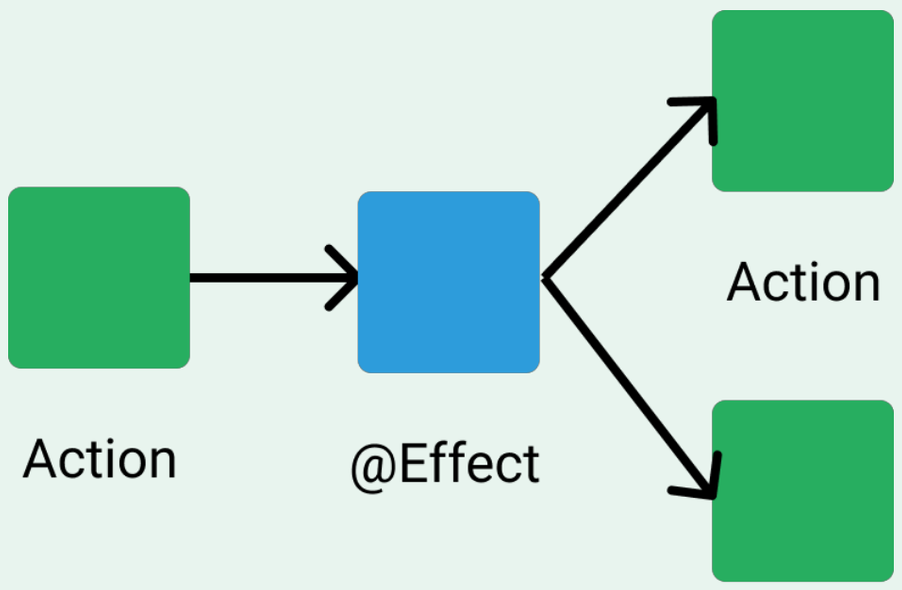
```
constructor(
  private actions$: Actions,
  private store: Store<AppState>,
  private localStorage: LocalStorageService,
) {}


/**
 * resets all draft localStorage update flags in the Draft reducer
 */
@Effect()
public resetDraftLsUpdates$ = this.actions$
  .pipe(
    ofType(LocalStorageActions.FETCH_ITEM_FROM_LOCAL_STORAGE),
    tap(() => console.log('@@ItemLocalStorageEffects.resetDraftLsUpdates$')),
    map(() => new DraftActions.ResetLsDraftUpdateFlagsAction())
  );
```





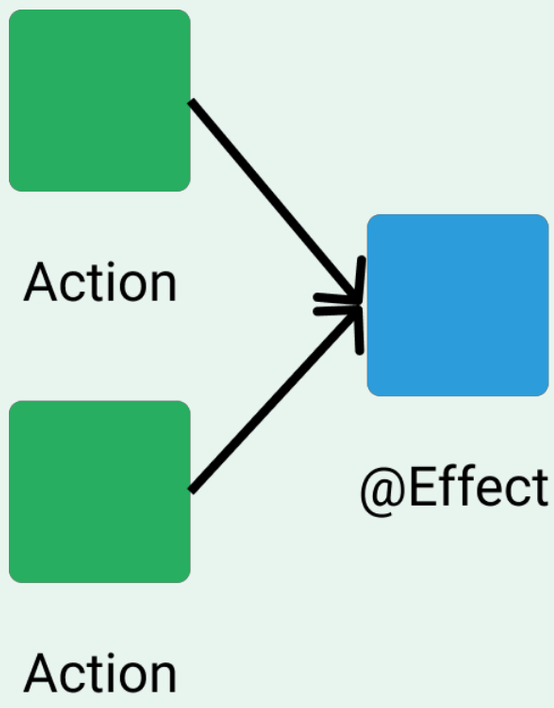
basics

in owa: most of all action -> effects sequences



splitter

in owa: auto update (body, to/cc recipients, subject)



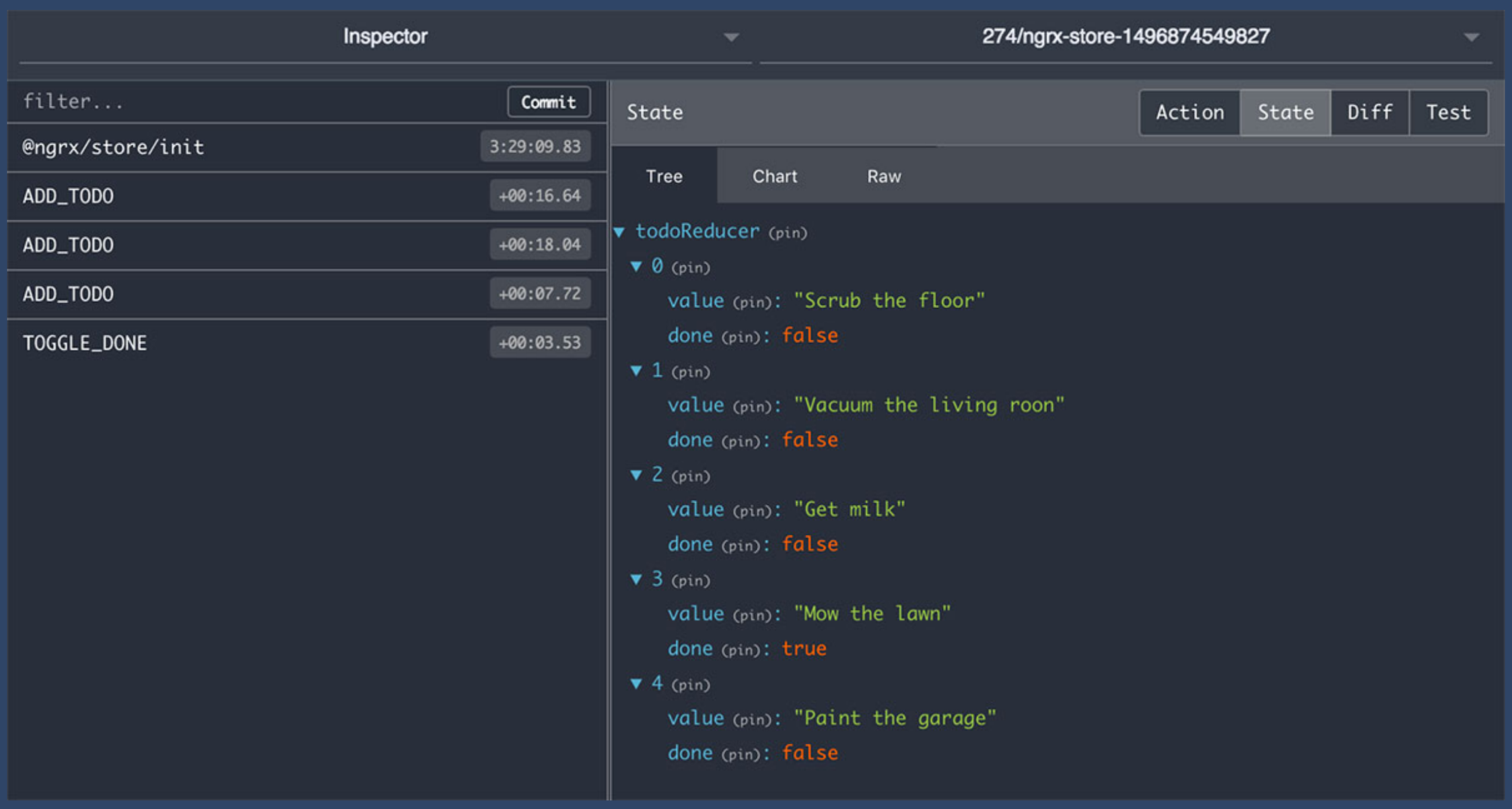
aggregator

in owa: validation, fetch ZIVVER contact details

chain

in owa: conversation, grant token, messages

Redux devtools



state management (NGRX)

- one source of truth is really nice
- in practice... OWA -> localStorage (proxy @Andrej)

NGRX further reading

- don't fear the boilerplate (interesting read)
- NGRX testing documentation
- splitter and aggregator pattern
- NGRX patterns

dont fear the boilerplate, argues in favour of Andrej's point regarding state management.