bad parts

bad parts

- view encapsulation
- ngFor + list updates
- custom form components
- ngIf, ngFor, ngLet?
- state management

@michahell @ ZIVVER

view encapsulation

@michahell @ ZIVVER

view encapsulation

- overwriting 3rd party styles
- Angular Material specifically

@michahell @ ZIVVER

*ngFor list updates

- finished with MWE and... it just works ¯\(ツ)/¯
- tried to reproduce bug we encountered in OWA but.. no
- anyway
- when you use ngFor, irregardless of changeStrategy used
- you can run into **Angular does not render updates** - issues
- ⚡ **stackBlitz**

custom form components

@michahell @ ZIVVER

custom form components

the bad

- underdocumented
- uses multiple interfaces, classes and styles
- `formControlName !== formcontrolname`

read about

- **Reactive forms** (programmatic) *vs* **template-driven** (angularJs style) forms
- **FormControl**, **FormGroup**, **FormArray** and **Validator**s
- **ControlValueAccessor**
- **MatFormFieldControl**

@michahell @ ZIVVER

- documentation: not up to date.
- Form API exists of lots of stuff.
- lots of ids, refererences passed as strings.
- READ: especially the last two are important to read about.

ngIf ngFor ngLet? .....it be

ngIf, ngFor | asyc... ngLet ?

the good

- **| async** makes it easy to use observables
- **Store.select()** = not a problem: **ReplaySubject**
- ⚡ stackBlitz

the bad

- **DRY**: issue, another issue
- not **Store.select()**? multiple (unicast) subscriptions

possible solution:

- NGRX utils ngLet
- ⚡ stackBlitz
- (PROD-1196-owa-send-flow-improvements) ✅🚀🔥🙂

@michahell @ ZIVVER

state management [NGRX] + component lifecycle

@michahell @ ZIVVER

state management (NGRX)

- encapsulation vs. everything redux ?
- **do** s
- **don't** tsss...smh
- **caveats**

state management (NGRX) DOs

- immutability
- think out: Action / reducer / effect
  - for single items
  - or for list of items

state management (NGRX)

```
case authActions.LOGGED_IN : {
  const newState = state;
  newState.zivverRefreshToken = action.payload.response.refresh_token;
  return newState;
}
```

```
case draftActions.OWA_SUBJECT_FETCHED : {
  const draft = Draft.clone(state);
  draft.subject = action.payload as string;
  return draft;
}
```

- **{...}** or **[state].map(i => new Item(i))** (deep clone!)

state management (NGRX) DONTs

- don't dispatch multiple actions and assume synchrononicity
- **don't write ALL business logic in @Effects**
- don't dispatch Actions from @Effects in **tap()** operators.

Because, from least bad to horribad:

- CPU spike
- memory leaks
- loops
- deadlocks

1) Needs example: asynchronicity due to Effects
3) Try as much as you can to put business logic in services. Reusable, easier to test. use service methods in @Effects.
4) Unpredictability
- CPU: due to large amounts of Actions being dispatched
- memleaks: uncompleted Observables
- loops: chain loops
- deadlocks: ?

state management (NGRX) caveats

- write silent failing code.
- completion and **.catchError()** in @Effects
- RXJS error handling

Easy to do this. examples: 1) Action does not get dispatched anymore 2) .catchError in main @Effect pipeline 3) observable completes while it should not (first, take, until)

component lifecycle (NGRX)

- dumb components, smart components ✅
- dumb component: gets only **@Input**'s and **Output()**s
- (parent) smart component: knows about **store.select()**