Consistent UIs
with RxJS and NgRx

Michele Stieven

Angular GDE, Founder of accademia.dev

@MicheleStieven

www.accademia.dev

RxJS

Masterclass

Functional

JavaScript

Traits of inconsistent UIs

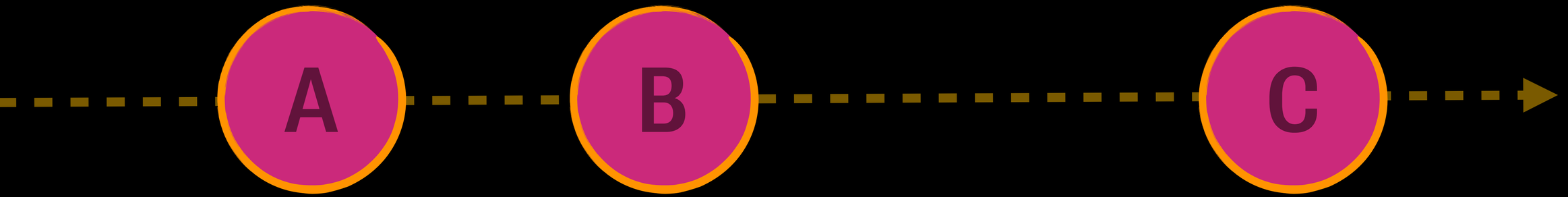
- Derived states not recalculated → template out of sync
- Multiple sources of truth → components out of sync
- HTTP concurrency (queries / mutations)
  - Duplicated calls → flashing loading states
  - Out of order calls → wrong result
  - Aborted calls → no feedback

Traits of inconsistent UIs

- Derived states not recalculated → 😡 **FRUSTRATION**
- Multiple sources of truth → 😡 **FRUSTRATION**
- HTTP concurrency (queries / mutations)
  - Duplicated calls → 😡 **FRUSTRATION**
  - Out of order calls → 😡 **FRUSTRATION**
  - Aborted calls → 😡 **FRUSTRATION**

🤔 Observable?

🚀 A stream of values over time, that can be observed



Derived States

Reactivity → Declarative Dependencies

Invoice

Consultancy ✏️

Angular	1h	100€
RxJS	2h	200€
TypeScript	1h	100€
	tot.	400€

Invoice

Consultancy ✏️

Angular	1h	100€
RxJS	2h	200€
TypeScript	**2h**	**200€**
	tot.	400€

?

```
export class InvoiceComponent {

  total = 0;

  onItemQuantityChange(i: number, qty: number) {
    ...
    this.total = this.items.reduce((tot, item) => tot + item.qty, 0);
  }
```

```
export class InvoiceComponent {

  total = 0;

  onItemQuantityChange(i: number, qty: number) {
    ...
    this.total = this.items.reduce((tot, item) => tot + item.qty, 0);
  }

  onInvoiceLoad() {
    ...
    this.total = this.items.reduce((tot, item) => tot + item.qty, 0);
  }

  onItemAdd() {
    ...
    this.total = this.items.reduce((tot, item) => tot + item.qty, 0);
  }

  onItemRemove() {
    ...
    this.total = this.items.reduce((tot, item) => tot + item.qty, 0);
  }
}
```

```
nvoiceComponent {


tyChange(i: number, qty: number) {
 = this.items.reduce((tot, item) ⇒ tot + item.qty, 0);

d() {
 = this.items.reduce((tot, item) ⇒ tot + item.qty, 0);

{
 = this.items.reduce((tot, item) ⇒ tot + item.qty, 0);

() {
 = this.items.reduce((tot, item) ⇒ tot + item.qty, 0);
```

- Scattered dependencies
- Error prone
- Not future-proof
- Repeated code
- Ad hoc functions (`recalculateTotal()`) don’t solve the problem

```
export class InvoiceComponent {

  items$: Observable<Item[]>;

  total$ = this.items$.pipe(
    map(items ⇒ items.reduce((tot, item) ⇒ tot + item.qty, 0))
  )
}
```

- Explicit dependencies (**items** → **total**)
- Always in sync, no manual updates

```
export class InvoiceComponent {

  items$: Observable<Item[]>;
  invoiceInfo$: Observable<Info>;

  total$ = combineLatest([this.items$, this.invoiceInfo$]).pipe(
    map(([items, info]) => {
      const itemsTotal = items.reduce((tot, item) => tot + item.qty, 0);
      return itemsTotal + info.stampDuty;
    })
  )
}
```

```
export class InvoiceComponent {

  form: FormGroup = ...;

  total$ = valueChanges(this.form).pipe(
    map(({ items, stampDuty }) => {
      const itemsTotal = items.reduce((tot, item) => tot + item.qty, 0);
      return itemsTotal + stampDuty;
    })
  )
}
```

- Suggested: custom operator instead of **form.valueChanges** (for now)

  https://michelestieven.medium.com/angular-derived-values-from-forms-with-rxjs-48760807ed1e

Single Source of Truth

Avoid duplicate states as much as possible

Invoices

RxJS Course x 1 ✏️

TypeScript Course ✏️

Consultancy (2h) ✏️

Invoice

Consultancy (2h) ✏️

Angular	1h
RxJS	1h

Invoices

RxJS Course x 1 ✏️

TypeScript Course ✏️

❓ Consultancy (2h) ✏️

Invoice

Consultancy (4h) ✏️

Angular	1h
RxJS	2h
TypeScript	1h

👍 Cambiamenti salvati con successo

```
export class InvoiceListComponent {

  invoices$ = this.store.select(selectInvoices);

  ngOnInit() {
    this.store.dispatch(getInvoices());
  }
}
```

```
export class Inv

  onInvoiceSave
    const invoi
    this.store.
  }
}
```

```
voices);
```

```
export class InvoiceComponent {

  onInvoiceSave() {
    const invoice = this.form.value;
    this.store.dispatch(saveInvoice({ invoice }));
  }
}
```

```
export const invoicesReducer = createReducer(
  initialState,
  on(saveInvoiceSuccess, (state, action) => {
    return {
      ...state,
      [action.invoice.id]: action.invoice
    }
  })
)
```

```
reloadOnSave$ = createEffect(() => this.actions$.pipe(
  ofType(saveInvoiceSuccess),
  map(getInvoices)
))
```

Concurrency

From Event Handlers to Declarative Effects

Invoice

Consultancy ✏️

Angular	1h
RxJS	2h
TypeScript	1h

Delete

```
export class InvoiceComponent {

  id$ = this.route.paramMap.pipe(
    map(params ⇒ params.get('id'))
  )

  onInvoiceDelete() {
    this.id$.pipe(
      switchMap(id ⇒ this.invoicesService.delete(id))
    ).subscribe()
  }
}
```

Problems

- Duplicated calls (missing `take(1)`)
- Concurrency

```
<button [disabled]="deleting">Delete</button>
```

```
onInvoiceDelete() {
  this.deleting = true;
  this.id$.pipe(
    take(1),
    switchMap(id => this.invoicesService.delete(id)),
    finalize(() => this.deleting = false)
  ).subscribe()
}
```

Problems

- Components bloated with variables, error prone
- Business logic tied to the UI, it’s likely to break

```
<button [disabled]="deleting">Delete</button>
```

```
onInvoiceDelete() {
  this.deleting = true;
  this.id$.pipe(
    take(1),
    switchMap(id => this.invoicesService.delete(id)),
    finalize(() => this.deleting = false)
  ).subscribe()
}
```

Suggested dev.to article

No, disabling a button is not app logic.

- David K. 🎹

```ts
export class InvoiceComponent {

  id$ = this.route.paramMap.pipe(...);

  delete$ = new Subject<void>();
  sub = Subscription | null = null;

  ngOnInit() {
    this.sub = delete$.pipe(
      withLatestFrom(this.id$),
      exhaustMap(([, id]) => this.invoiceService.delete(id))
    ).subscribe();
  }

  ngOnDestroy() {
    this.sub?.unsubscribe();
  }
}
```

```html
<button (click)="delete$.next()">Delete</button>
```

```ts
export class InvoiceComponent {

  id$ = this.route.paramMap.pipe(...);

  delete$ = new Subject<void>();
  sub = Subscription | null = null;

  ngOnInit() {
    this.sub = delete$.pipe(
      withLatestFrom(this.id$),
      exhaustMap(([, id]) => this.invoiceService.delete(id))
    ).subscribe();
  }

  ngOnDestroy() {
    this.sub?.unsubscribe();
  }
}
```

```html
<button (click)="delete$.next()">Delete</button>
```

```
export class InvoiceComponent {

  id$ = this.route.paramMap.pipe(...);

  delete$ = new Subject<void>();
  sub = Subscription | null = null;

  ngOnInit() {
    this.sub = delete$.pipe(
      withLatestFrom(this.id$),
      exhaustMap(([, id]) => this.invoiceService.delete(id))
    ).subscribe();
  }

  ngOnDestroy() {
    this.sub?.unsubscribe();
  }
}
```

Problem

- Verbosity

```
export class InvoiceStore extends ComponentStore<InvoiceState> {

  id$ = this.route.paramMap.pipe(...);

  deleteInvoice = this.effect((delete$: Observable<void>) => {
    return delete$.pipe(
      withLatestFrom(this.id$),
      exhaustMap(([, id]) => this.invoiceService.delete(id).pipe(
        tapResponse(

          // Do stuff

        )
      ))
    )
  });
}
```

NgRx Component Store

```ts
export class InvoiceStore extends ComponentStore<InvoiceState> {

  id$ = this.route.paramMap.pipe(...);

  deleteInvoice = this.effect((delete$: Observable<void>) => {
    return delete$.pipe(
      withLatestFrom(this.id$),
      exhaustMap(([, id]) => this.invoiceService.delete(id).pipe(
        tapResponse(


        )
      ))
    )
  });
}
```

```ts
export class InvoiceComponent {

  constructor(private store: InvoiceStore) {}

  onDelete() {
    this.store.deleteInvoice();
  }
}
```

Effects

UI-based effects

Invoice

Consultancy ✏️

Angular	1h
RxJS	2h
TypeScript	1h

Save

```
saveInvoice$ = createEffect(() => this.actions$.pipe(
  ofType(saveInvoice),
  switchMap(invoice => this.invoiceService.save(invoice.id, invoice).pipe(
    map(invoice => saveInvoiceSuccess({ invoice })),
    catchError(error => of(saveInvoiceError({ error })))
  ))
))
```

NgRx

Problem

- `switchMap` is the right choice… but only if we're working on 1 invoice at a time!

Invoices

RxJS Course x 1 ✏️

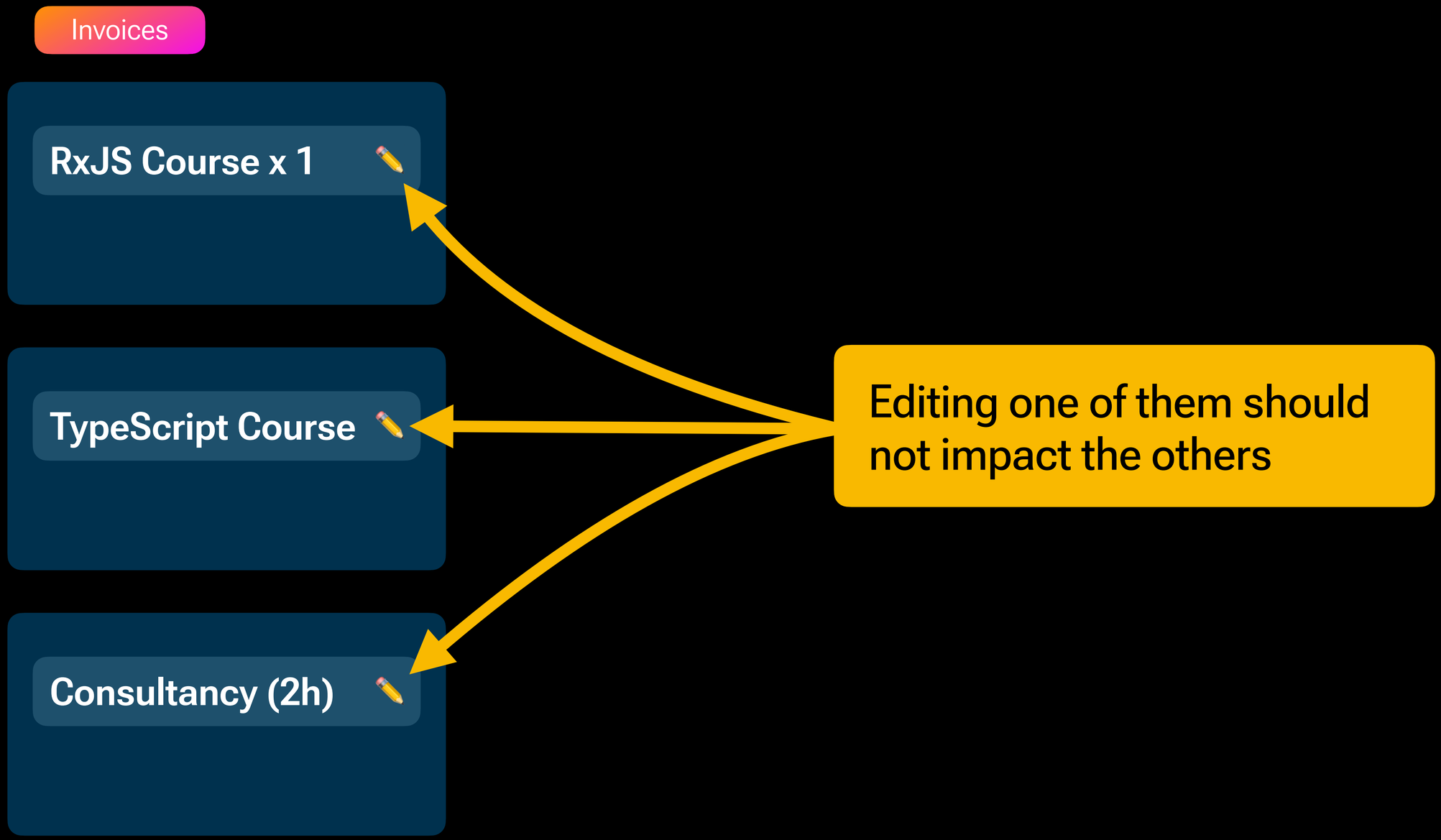
TypeScript Course ✏️

Consultancy (2h) ✏️

Invoice

Consultancy ✏️

Angular	1h
RxJS	2h
TypeScript	1h


Invoices
RxJS Course x 1
TypeScript Course
Consultancy (2h)
Editing one of them should not impact the others

```
saveInvoice$ = createEffect(() => this.actions$.pipe(
  ofType(saveInvoice),
  groupBy(invoice => invoice.id),
  mergeMap(group$ => group$.pipe(
    switchMap(invoice => this.invoiceService.save(invoice.id, invoice).pipe(
      map(invoice => saveInvoiceSuccess({ invoice })),
      catchError(error => of(saveInvoiceError({ error })))
    ))
  ))
))
```

Suggested YouTube video

What GroupsBy in Vegas, stays in Vegas

- Mike Ryan & Sam Julien

```
saveInvoice = this.effect((save$: Observable<Invoice>) ⇒ save$.pipe(
  groupBy(invoice ⇒ invoice.id),
  mergeMap(group$ ⇒ group$.pipe(
    switchMap(invoice ⇒ this.invoiceService.save(invoice.id, invoice).pipe(
      tapResponse(...)
    ))
  ))
))
```

NgRx Component Store

Suggested YouTube video

What GroupsBy in Vegas, stays in Vegas

- Mike Ryan & Sam Julien

Recap

- Derive states by mapping Observables
- Propagate changes with a global store
- Avoid async logic inside event handlers, use Effects instead
  - Try to avoid `subscribe()`, lean on the async pipe
- Use the appropriate Flattening Operator
- Group actions to avoid concurrency collisions

Slides: *https://github.com/UserGalileo/talks*

👋

Michele Stieven

Angular GDE, Founder of accademia.dev

@MicheleStieven