Reactive programming with RxJS

You describe search, live status, and user input as streams with RxJS and test time-dependent code without real waiting time.

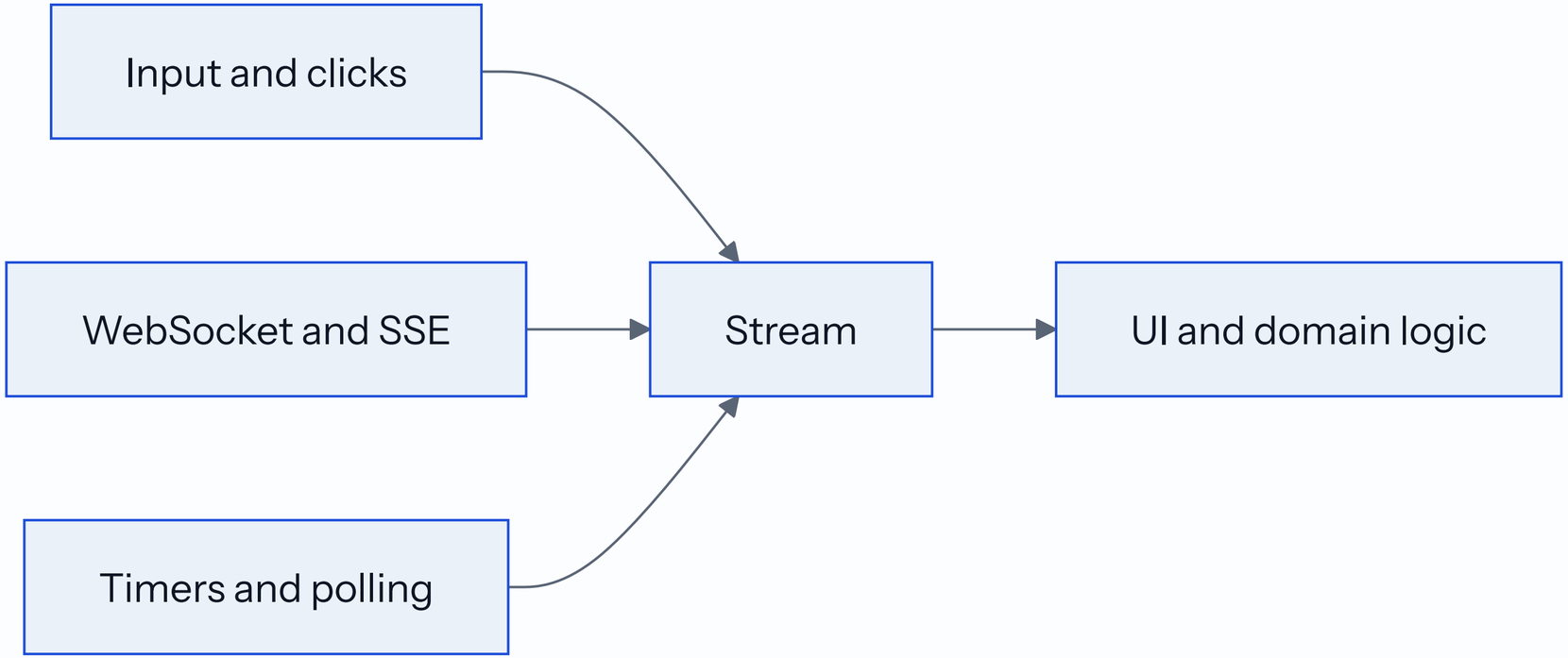
Mike Bild

Exercises: github.com/MikeBild/Reactive-Programming

Push, Observable, creation

When is reactive programming worth it? · Push instead of pull · Nothing happens without subscribe · Almost everything becomes an Observable

Values arrive over time



Callbacks and flags quickly become unreadable

- Each event handler manages its own state
- Timers, cancellation, and retry spread across the code
- Stale responses overwrite new results

```
let timer: ReturnType<typeof setTimeout> | undefined
let latestTerm = ''
input.addEventListener('input', () => {
  clearTimeout(timer)
  timer = setTimeout(async () => {
    latestTerm = input.value
    const response = await fetch(`/api/search?q=${latestTerm}`)
    if (input.value === latestTerm) render(await response.json())
  }, 300)
})
```

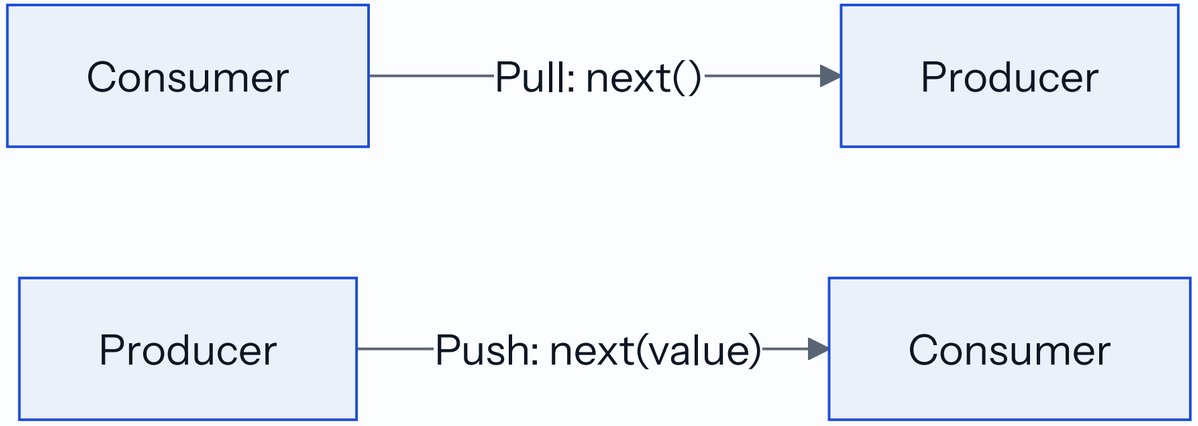
The same search as a stream

```
fromEvent(input, 'input')
  .pipe(
    map(() => input.value.trim()),
    debounceTime(300),
    distinctUntilChanged(),
    switchMap((term) =>
      fromFetch(`/api/search?q=${encodeURIComponent(term)}`, {
        selector: (response) => response.json(),
      }),
    ),
  )
  .subscribe(render)
```

When you do not need RxJS

- Exactly one result: async await is enough
- Simple local state: Signals or State Hooks are enough
- Team without Rx experience: plan for the learning curve

Who triggers the next value?



Four ways to deliver values

- Pull with one value: function
- Pull with many values: iterator and generator
- Push with many values: Observable

When pull is the better choice

- The consumer sets its own pace
- Data already exists or arrives page by page
- Backpressure comes without extra effort

Same operators, different model

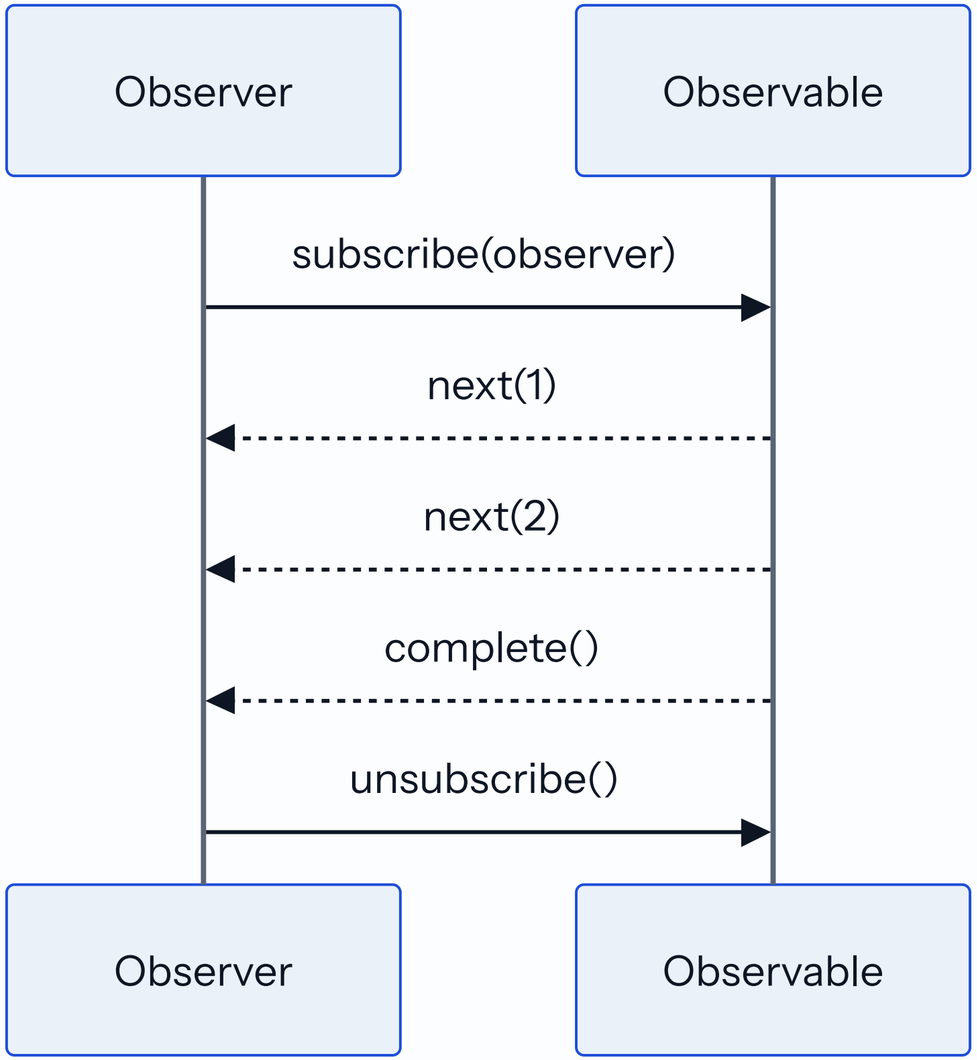
```
import { from, filter, map } from 'rxjs'

function* numbers() {
  for (let n = 1; n <= 6; n++) yield n
}

// Pull: the loop gets each value
for (const n of numbers()) if (n % 2 === 0) console.log(n)

// Push: from sends each value to the Subscriber
from(numbers())
  .pipe(filter((n) => n % 2 === 0), map((n) => n * 10))
  .subscribe((n) => console.log(n))
```

Observable, Observer, Subscription



The contract is strict

- Any number of next calls, then at most one error or complete
- After error or complete, nothing else arrives
- An error ends the stream like a throw ends a function

Lazy like a function

- Without subscribe, no timer and no request starts
- Each subscribe restarts the producer function
- Two Subscribers mean two independent executions

Write an Observable yourself

```
import { Observable } from 'rxjs'

const ticker = (ms: number) =>
  new Observable<number>((subscriber) => {
    let count = 0
    const id = setInterval(() => subscriber.next(count++), ms)
    return () => clearInterval(id)
  })

const subscription = ticker(1_000).subscribe({
  next: (value) => console.log(value),
  error: (error: unknown) => console.error(error),
  complete: () => console.log('fertig'),
})
setTimeout(() => subscription.unsubscribe(), 3_500)
```

Values, arrays, iterables

```
import { of, from, EMPTY, throwError } from 'rxjs'

of(1, 2, 3).subscribe(console.log)           // 1, 2, 3, complete
from(['a', 'b']).subscribe(console.log)      // a, b, complete
from(new Set([1, 1, 2])).subscribe(console.log) // 1, 2, complete

EMPTY.subscribe({ complete: () => console.log('leer') })
throwError(() => new Error('kaputt')).subscribe({ error: console.error })
```

Events and timers become Observables

- fromEvent registers a separate listener per Subscriber
- interval counts up from zero at a fixed interval
- timer waits once and can keep counting afterward

```
import { fromEvent, interval, timer } from 'rxjs'

const clicks$ = fromEvent<MouseEvent>(document, 'click')
const ticks$ = interval(1_000)       // 0, 1, 2, … every second
const once$ = timer(500)             // 0 after 500 ms, then complete
const later$ = timer(2_000, 1_000)  // after 2 s, then every second
```

Make Promises lazy with defer

```
import { from, defer } from 'rxjs'
import { fromFetch } from 'rxjs/fetch'

// Starts immediately, even without subscribe
const eager$ = from(fetch('/api/user'))

// Starts only on subscribe, anew for each subscribe
const lazy$ = defer(() => fetch('/api/user').then((r) => r.json()))

// Lazy and cancellable via AbortController
const cancellable$ = fromFetch('/api/user')
```

Connect callback APIs

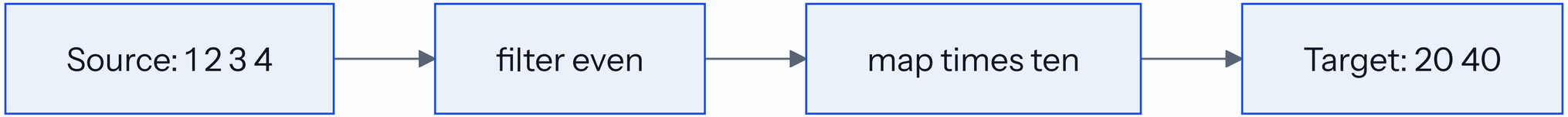
```
import { Observable } from 'rxjs'

const position$ = new Observable<GeolocationPosition>((subscriber) => {
  const id = navigator.geolocation.watchPosition(
    (position) => subscriber.next(position),
    (error) => subscriber.error(error),
  )
  return () => navigator.geolocation.clearWatch(id)
})
```

Operators, lifetime, combining

Operators are functions in a pipe · Every subscription needs an end · Which operator combines your streams?

An operator turns one stream into a new one



Read Marble diagrams

- One timeline per stream, time runs to the right
- Circles are values, a bar is complete, an X is error
- The source is on top, the operator result is below

```
Quelle:   --1--2--3--4--|
filter:   -----2-----4--|
map x10:  -----20----40-|
```

State over time with scan

```
import { fromEvent, map, scan, distinctUntilChanged } from 'rxjs'

const clicks$ = fromEvent(button, 'click')

clicks$
  .pipe(
    map(() => 1),
    scan((count, increment) => count + increment, 0),
    map((count) => (count >= 10 ? 'viel' : 'wenig')),
    distinctUntilChanged(),
  )
  .subscribe(render)
```

Your own operators with pipe

```
import { pipe, filter, map, type OperatorFunction } from 'rxjs'

type Message = { type: string; payload: unknown }

export const ofType = <T>(type: string): OperatorFunction<Message, T> =>
  pipe(
    filter((message) => message.type === type),
    map((message) => message.payload as T),
  )

messages$.pipe(ofType<Order>('order-placed')).subscribe(saveOrder)
```

Endless streams hold on to resources

- Event listeners stay attached to the DOM element
- Timers keep running even though no one is watching
- Callbacks keep components in memory

Tie resources to the stream

- Teardown closes what subscribe opened
- finalize logs and cleans up after every end
- Subscription.add bundles multiple subscriptions

Three ways to end a subscription

```
import { interval, fromEvent, take, first, takeUntil } from 'rxjs'

// 1. Explicit
const subscription = interval(1_000).subscribe(tick)
subscription.unsubscribe()

// 2. After a fixed number of values
interval(1_000).pipe(take(5)).subscribe(tick)
fromEvent(button, 'click').pipe(first()).subscribe(start)

// 3. Until another stream emits
interval(1_000).pipe(takeUntil(fromEvent(stopButton, 'click'))).subscribe(tick)
```

One signal for all subscriptions

```
import { Subject, interval, fromEvent, takeUntil, finalize } from 'rxjs'

const destroy$ = new Subject<void>()

interval(1_000)
  .pipe(finalize(() => console.log('Timer gestoppt')), takeUntil(destroy$))
  .subscribe(updateClock)

fromEvent(window, 'resize').pipe(takeUntil(destroy$)).subscribe(relayout)

const close = () => {
  destroy$.next()
  destroy$.complete()
}
```

Four ways to connect streams

```
a$:              --a-------b-------|
n$:              -----1-------2----|

merge:           --a--1----b--2----|
combineLatest:   -----a1---b1-b2---|
zip:             -----a1------b2---|
forkJoin:        ------------------(b2|)
```

Current state with combineLatest

```
import { combineLatest, debounceTime, switchMap } from 'rxjs'

const filter$ = combineLatest({
  term: term$,
  category: category$,
  sort: sort$,
})

const results$ = filter$.pipe(
  debounceTime(200),
  switchMap((filter) => searchProducts(filter)),
)
```

One trigger, lots of context

- withLatestFrom: only the main stream triggers
- The other streams quietly provide their latest value
- Typical case: save click with the current form state

```
const save$ = saveClick$.pipe(
  withLatestFrom(form$, user$),
  map(([, form, user]) => ({ ...form, authorId: user.id })),
)
```

forkJoin waits for complete

- Ideal for multiple requests with one response each
- Emits nothing if a source never ends
- Stops as soon as one source signals an error

Cold and hot, async, flattening

Cold or hot, who starts the producer? · Promises and Observables work together · switchMap, mergeMap, concatMap, or exhaustMap?

On-demand movie or live broadcast

- Cold: each subscriber starts the movie from the beginning
- Hot: everyone sees the same broadcast in progress
- Hot: anyone who joins late misses the beginning

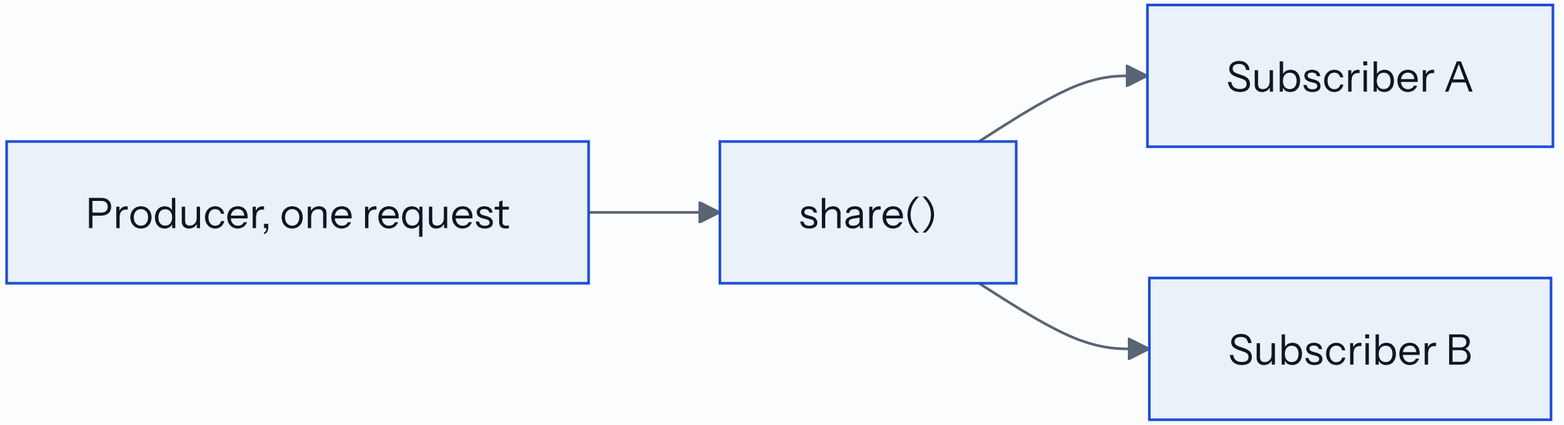
Two subscribers, two requests

```
import { tap } from 'rxjs'
import { fromFetch } from 'rxjs/fetch'

const user$ = fromFetch('/api/user').pipe(
  tap(() => console.log('Response da')),
)

user$.subscribe(renderHeader)
user$.subscribe(renderSidebar)
```

Share one producer



When hot becomes a problem

- Late subscribers miss earlier values
- Shared streams hide who starts the producer
- Errors end the stream for all subscribers

A Promise can only be canceled with a signal

- A Promise starts immediately when called
- It emits exactly one value or one error
- Cancellation only works through a passed AbortSignal

Where each world belongs

- Observables for events, real-time data, and UI flows
- Promises for one-off calls in services and tests
- Transitions at the edges, not in the middle of the chain

Back to await

```
import { firstValueFrom, lastValueFrom, take, toArray } from 'rxjs'

const user = await firstValueFrom(user$)

const total = await lastValueFrom(upload$)

const firstThree = await firstValueFrom(prices$.pipe(take(3), toArray()))

const maybe = await firstValueFrom(empty$, { defaultValue: null })
```

Connect async functions with cancellation

```ts
import { Observable } from 'rxjs'

export const fromAbortable = <T>(task: (signal: AbortSignal) => Promise<T>) =>
  new Observable<T>((subscriber) => {
    const controller = new AbortController()
    task(controller.signal).then(
      (value) => {
        subscriber.next(value)
        subscriber.complete()
      },
      (error: unknown) => subscriber.error(error),
    )
    return () => controller.abort()
  })
```

One value starts a new stream

```
import { map, mergeMap } from 'rxjs'

// Without flattening: a stream of streams
const nested$ = term$.pipe(map((term) => search(term)))

// With flattening: a stream of results
const results$ = term$.pipe(mergeMap((term) => search(term)))
```

Four strategies for the running stream

```
Source:       --a-----b--c--------|
inner:        a: --A|  b: ----B|  c: --C|

switchMap:    ----A---------C|      b canceled
mergeMap:     ----A-------BC|       all parallel
concatMap:    ----A-----B---C|      b, then c
exhaustMap:   ----A-----B|          c ignored
```

The Use Case decides

- switchMap for search, filters, and navigation
- concatMap for autosave and ordered writes
- exhaustMap for login, submit, and refresh buttons

Where flattening loses data or overloads

```
import { mergeMap, concatMap, catchError, EMPTY } from 'rxjs'

const uploads$ = files$.pipe(mergeMap((file) => upload(file), 4))

const saved$ = changes$.pipe(
  concatMap((change) =>
    save(change).pipe(catchError(() => EMPTY)),
  ),
)
```

Errors, Subjects, time, tests

An error ends the stream · Subjects share values, share shares work · What if more events arrive than you can process? · Testing time without waiting

An error is the end

```
Source:       --a--b--X
map:          --A--B--X
subscribe:    A, B, then error, then no more values
```

catchError belongs in the inner stream

```
import { switchMap, catchError, of } from 'rxjs'

const results$ = term$.pipe(
  switchMap((term) =>
    search(term).pipe(
      catchError((error: unknown) => of({ kind: 'error' as const, error })),
    ),
  ),
)
```

Errors are values for the UI

- Errors as values in the stream instead of error signals
- A type with kind distinguishes loading, result, and error
- The UI renders every state without try and catch

Retry with backoff

```
import { retry, timer, timeout } from 'rxjs'

const orders$ = loadOrders().pipe(
  timeout({ first: 2_000 }),
  retry({
    count: 3,
    delay: (_error, retryCount) => timer(2 ** retryCount * 250),
  }),
)
```

Observable and Observer at the same time

```
import { Subject } from 'rxjs'

const clicks = new Subject<string>()

clicks.subscribe((id) => console.log('A', id))
clicks.subscribe((id) => console.log('B', id))

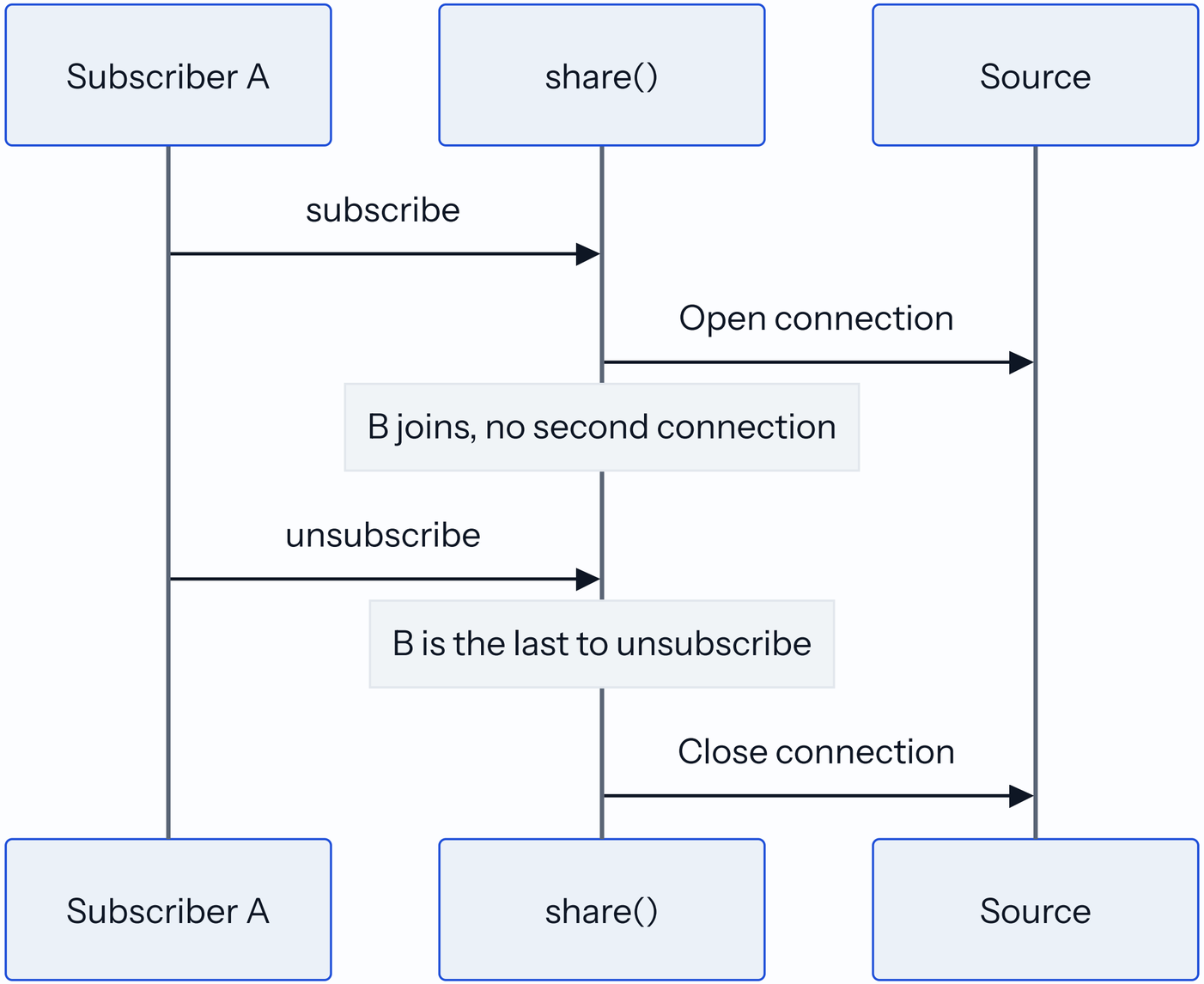
clicks.next('save')   // A save, B save
clicks.complete()
```

Three Subjects, three memories

- Subject: late subscribers get only new values
- BehaviorSubject: initial value and always the current value
- ReplaySubject: the last n values for every new subscriber

```
const user = new BehaviorSubject<User | null>(null)
user.next({ id: '42', name: 'Ada' })
user.subscribe(render) // bekommt Ada sofort
console.log(user.value)
```

The last one turns off the lights



Caching with shareReplay

```
import { defer, shareReplay } from 'rxjs'

const config$ = defer(() => fetch('/config').then((r) => r.json())).pipe(
  shareReplay({ bufferSize: 1, refCount: true }),
)

config$.subscribe(initHeader)
config$.subscribe(initFooter) // no second request
```

Push has no brakes

- The producer sends as fast as it can
- The consumer cannot ask for a pause
- You decide which values may be lost

Schedulers control when work runs

- asyncScheduler schedules work like setTimeout
- asapScheduler schedules work as a microtask
- animationFrameScheduler schedules work before the next frame

```
import { interval, animationFrameScheduler, observeOn } from 'rxjs'

interval(0, animationFrameScheduler).subscribe(drawFrame)
updates$.pipe(observeOn(animationFrameScheduler)).subscribe(render)
```

Four strategies against the flood

```
Source:         -a-b-c-------d-e-f-g-------|
debounceTime:   ----------c-----------g----|   after silence
throttleTime:   -a----------d-------g-----|   first, then pause
auditTime:      -----c-----------e-----g---|   latest per window
bufferTime:     -----[abc]-------[def]--[g]|   all collected
```

Why real time breaks tests

- Tests with setTimeout take as long as the timer
- Tight timings fail randomly in CI
- Fake timers do not affect every scheduler

Marble syntax

```
'-'       one frame, so one millisecond
'a'       a value, takes one frame
'|'       complete
'#'       error
'(ab)'    several events in the same frame
' 300ms ' time passes without an event
'^'       subscription point for hot
```

A test with virtual time

```
import { TestScheduler } from 'rxjs/testing'
import { debounceTime } from 'rxjs'
import { deepStrictEqual } from 'node:assert/strict'
import { test } from 'node:test'

const createScheduler = () =>
  new TestScheduler((actual, expected) => deepStrictEqual(actual, expected))

test('debounce wartet auf Ruhe', () => {
  createScheduler().run(({ cold, expectObservable }) => {
    const input$ = cold('ab 300ms c|')
    expectObservable(input$.pipe(debounceTime(300))).toBe('301ms b - (c|)')
  })
})
```

Testing without mocks

```
const searchResults = (
  term$: Observable<string>,
  search: (term: string) => Observable<string[]>,
) => term$.pipe(debounceTime(300), switchMap((term) => search(term)))

test('sucht nach Ruhe', () => {
  createScheduler().run(({ cold, hot, expectObservable }) => {
    const term$ = hot('a 400ms b')
    const search = (term: string) => cold('-r|', { r: [term] })
    expectObservable(searchResults(term$, search)).toBe('301ms x 400ms y', {
      x: ['a'],
      y: ['b'],
    })
  })
})
```

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen