RxJS

Tomasz Ducin

13-15 February 2019

Wrocław

Tomasz Ducin

agenda - content

Fundamentals of Asynchronous JavaScript

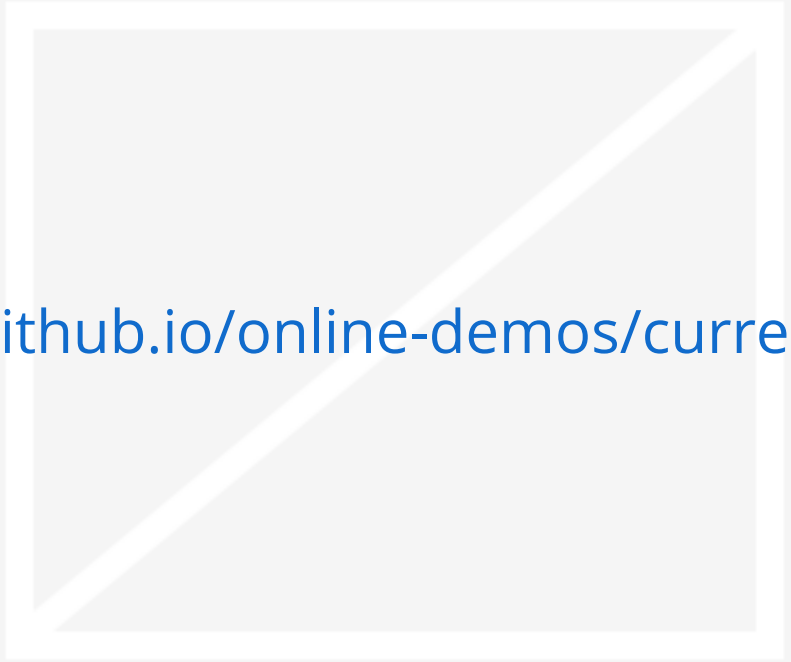
- Reactive Functional Programming
- Observable Pattern
- Observable Streams, as opposed to Promises & Async Await
- Managing Subscriptions
- Operators: Manipulating Data
- Operators: Manipulating Time
- Operators: Backpressure
- Higher Order Observables
- Error Handling
- Subjects
- Hot and Cold Observables
- Best Practices and Antipatterns

agenda - roadmap

1. Understand FRP
2. Understand streams & operators
3. Differentiate operators and their usecases
   (learn operators!)
4. Understand given problems,
   to pick up the right tool
5. Design & Architecture based on RxJS

we'll do...

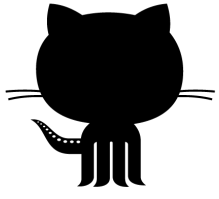
https://ducin.github.io/online-demos/waves/

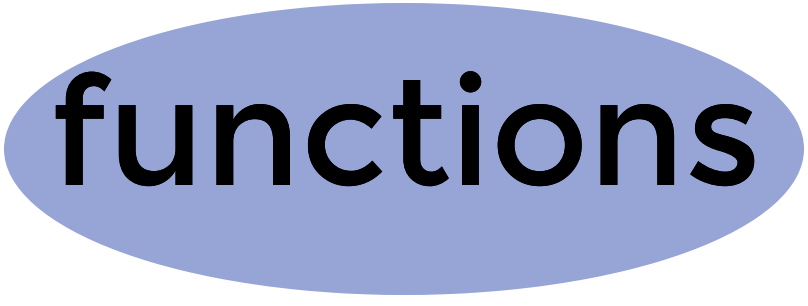


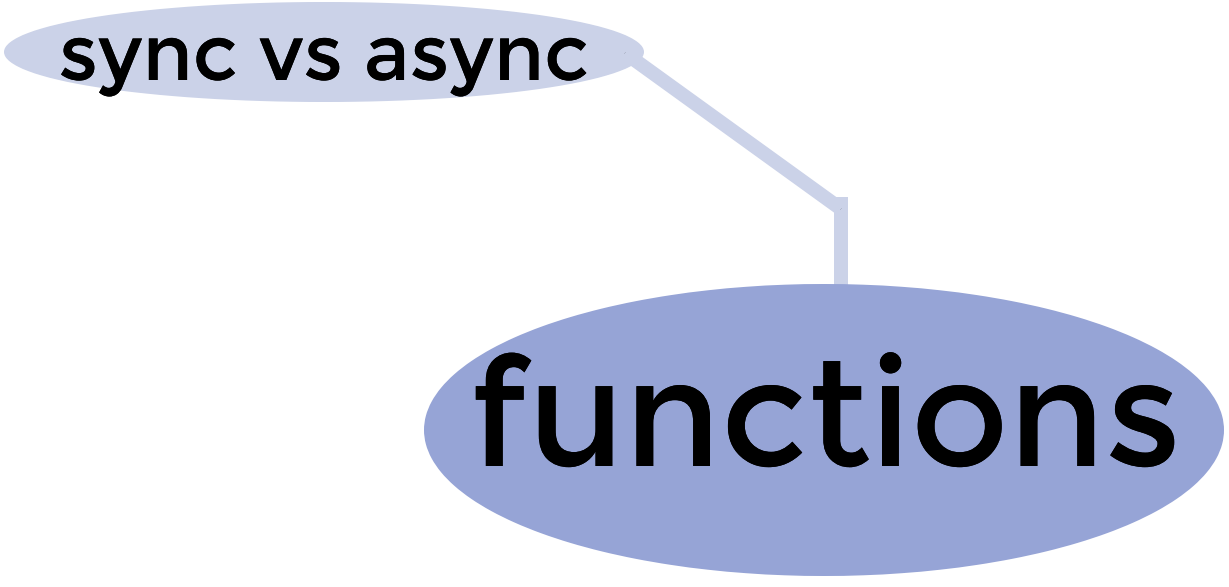
https://ducin.github.io/online-demos/currency-exchange/

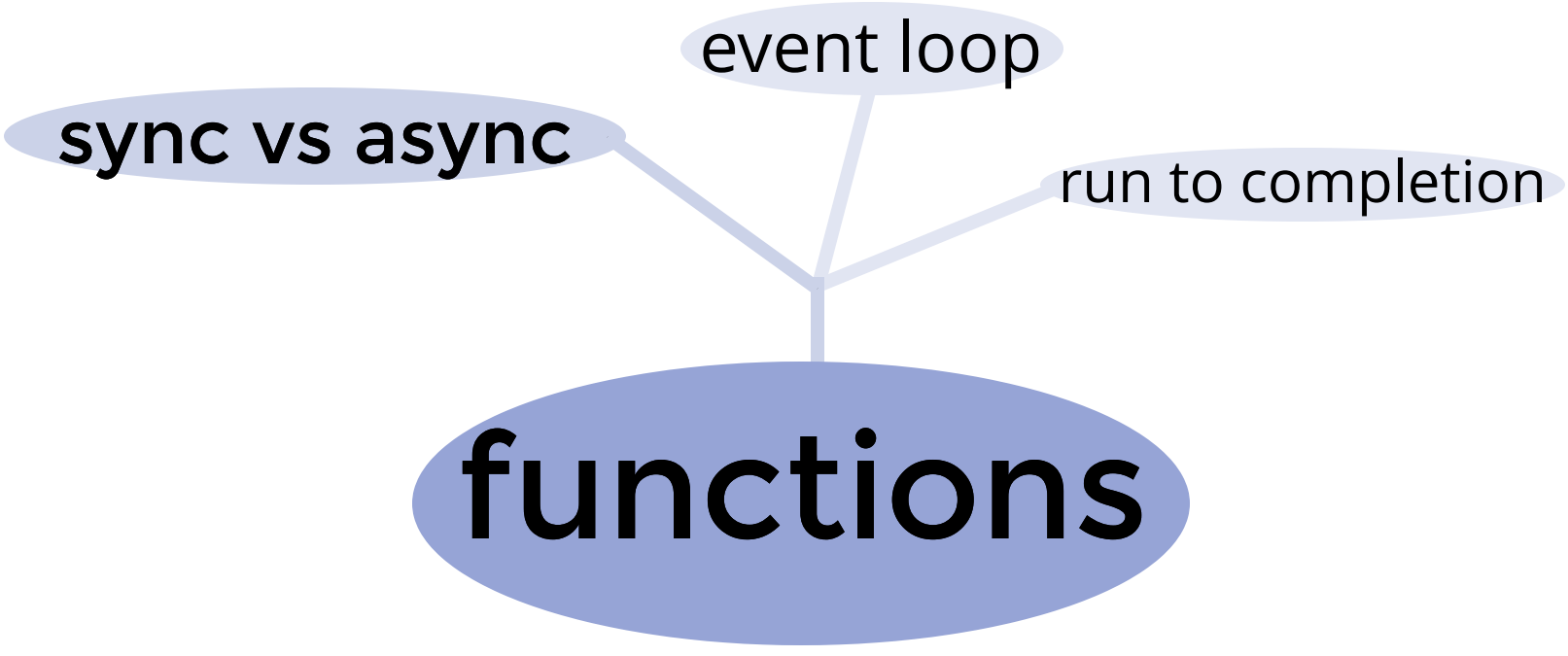
currency exchange

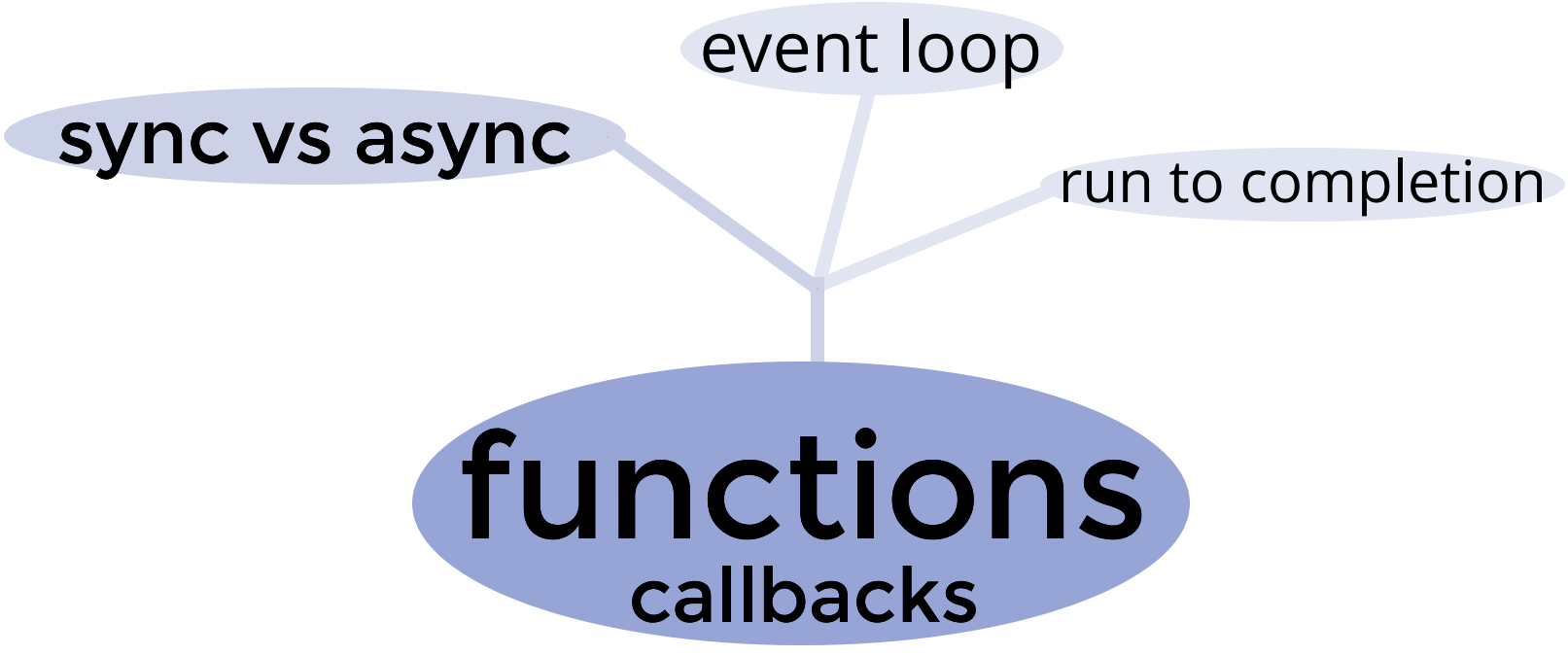
https://github.com/ducin-public/rxjs-training

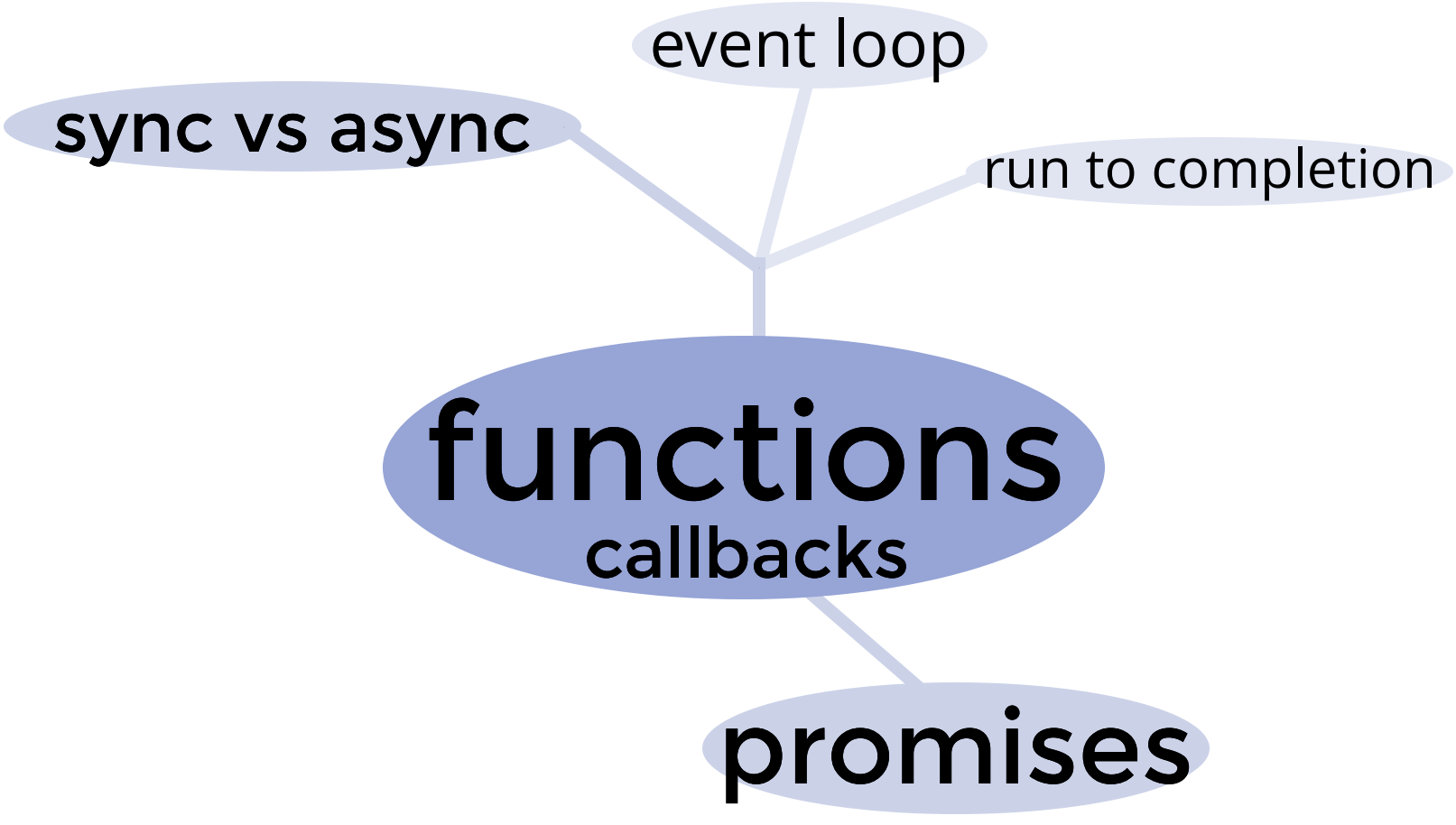


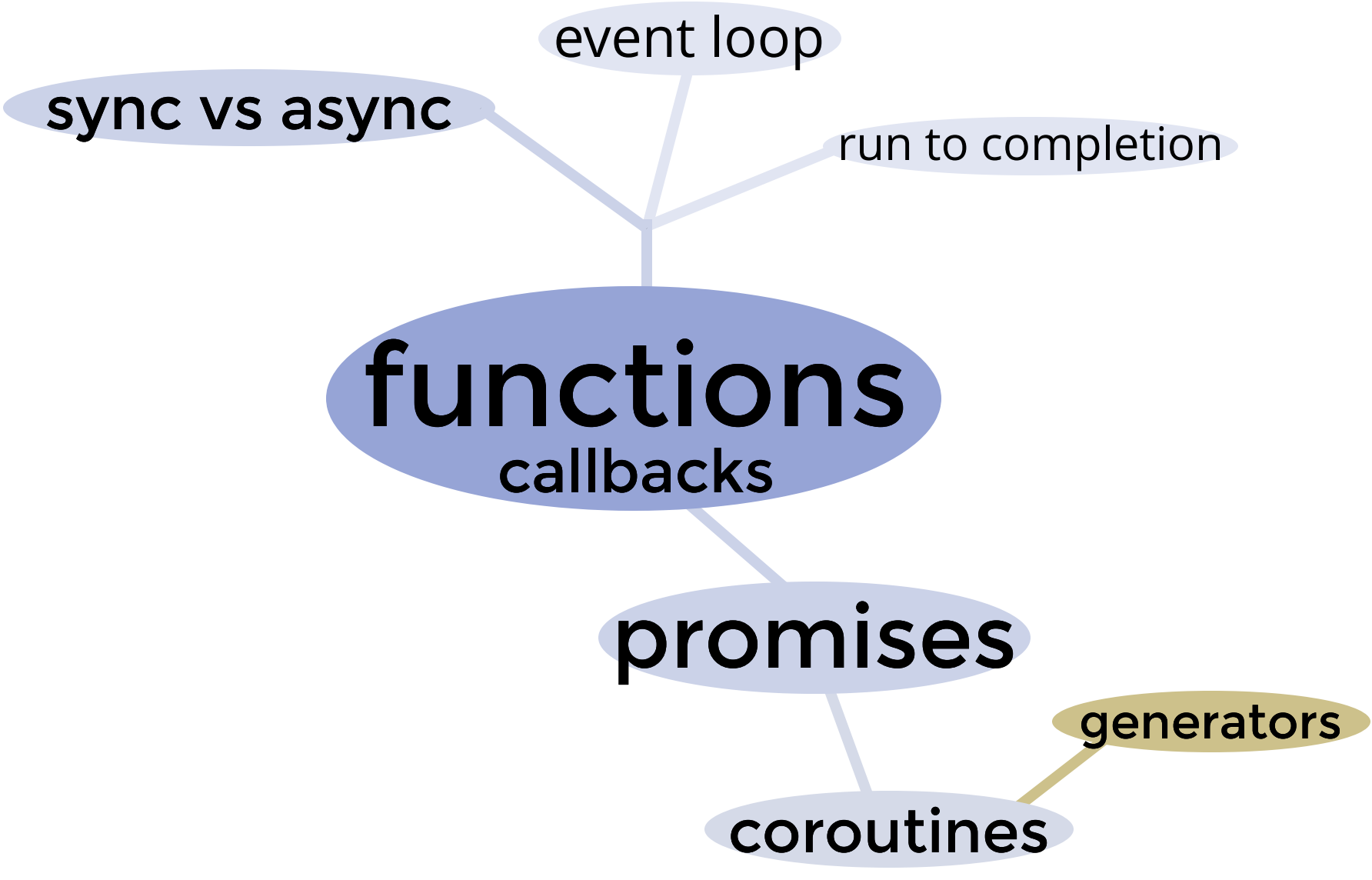

functions

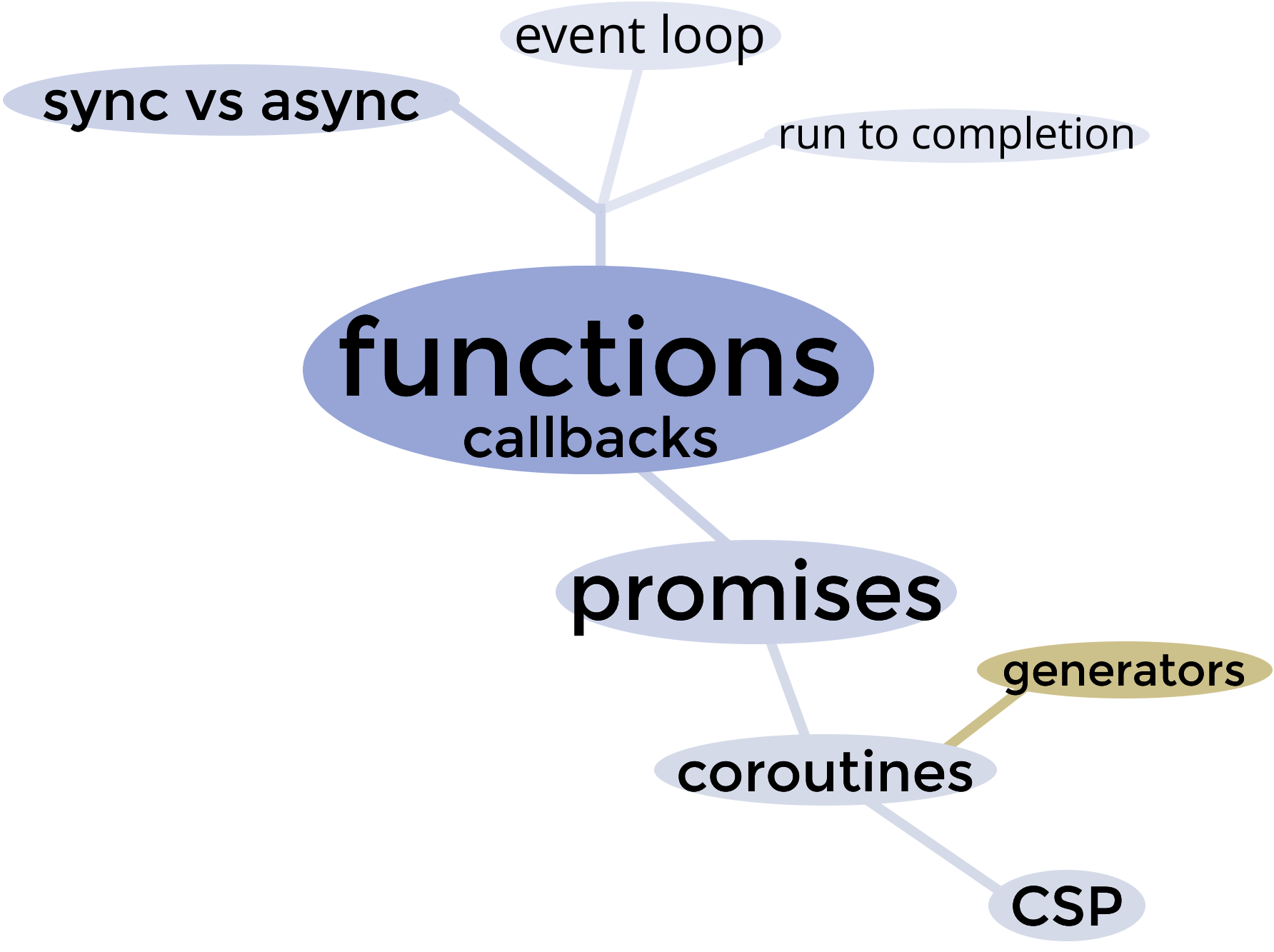

sync vs async
functions

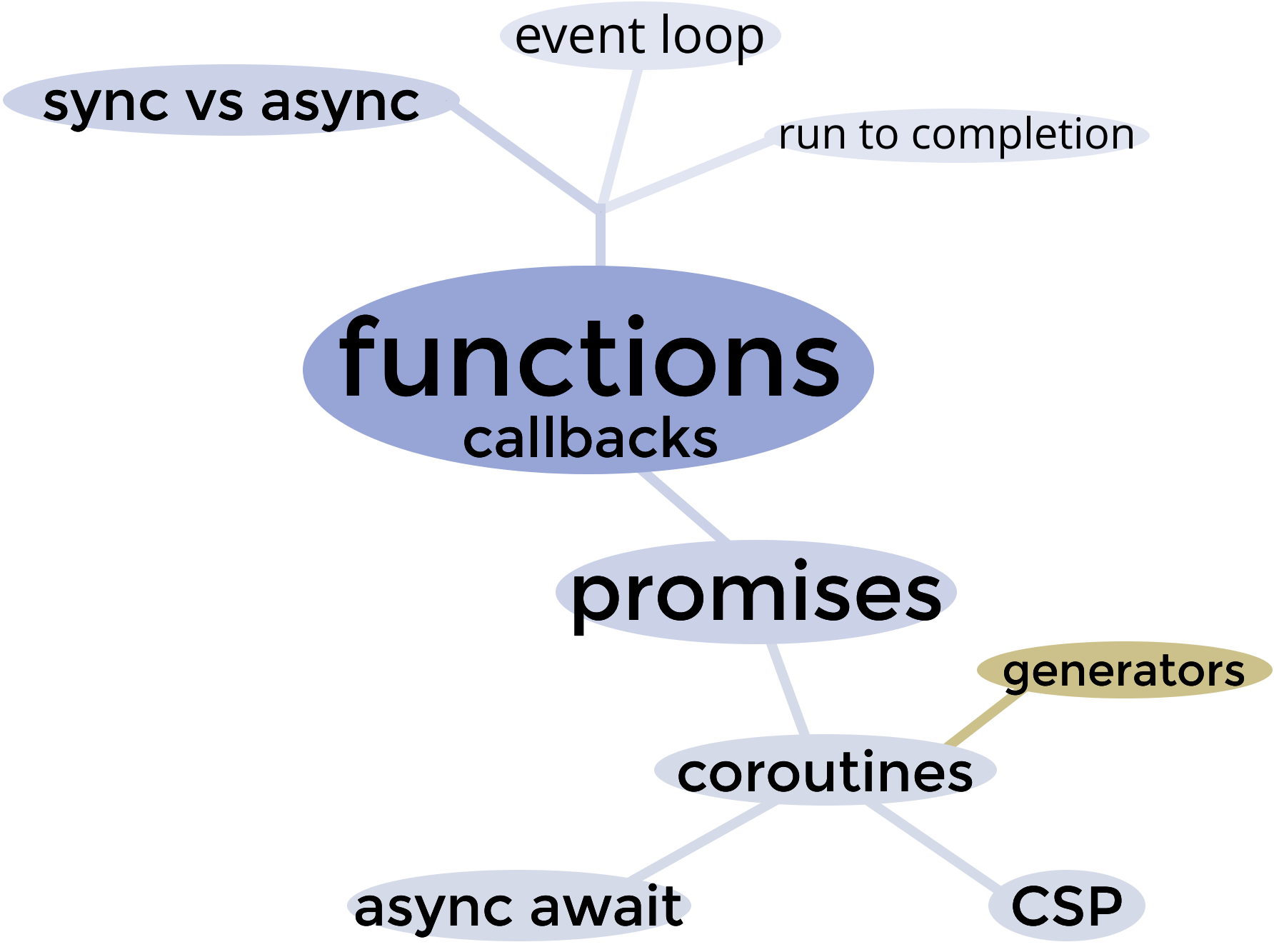

event loop
sync vs async
run to completion
functions

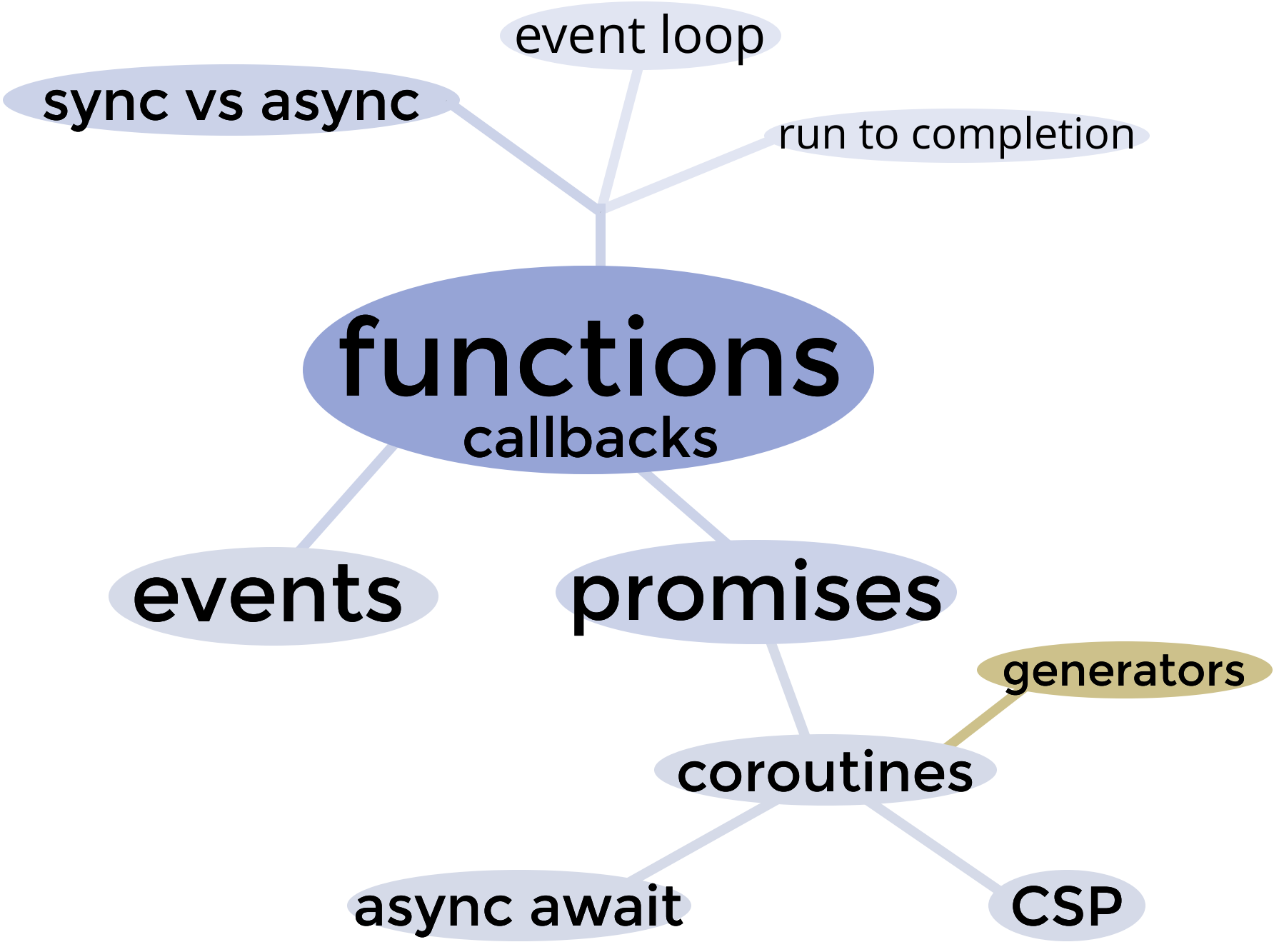

event loop
sync vs async
run to completion
functions
callbacks

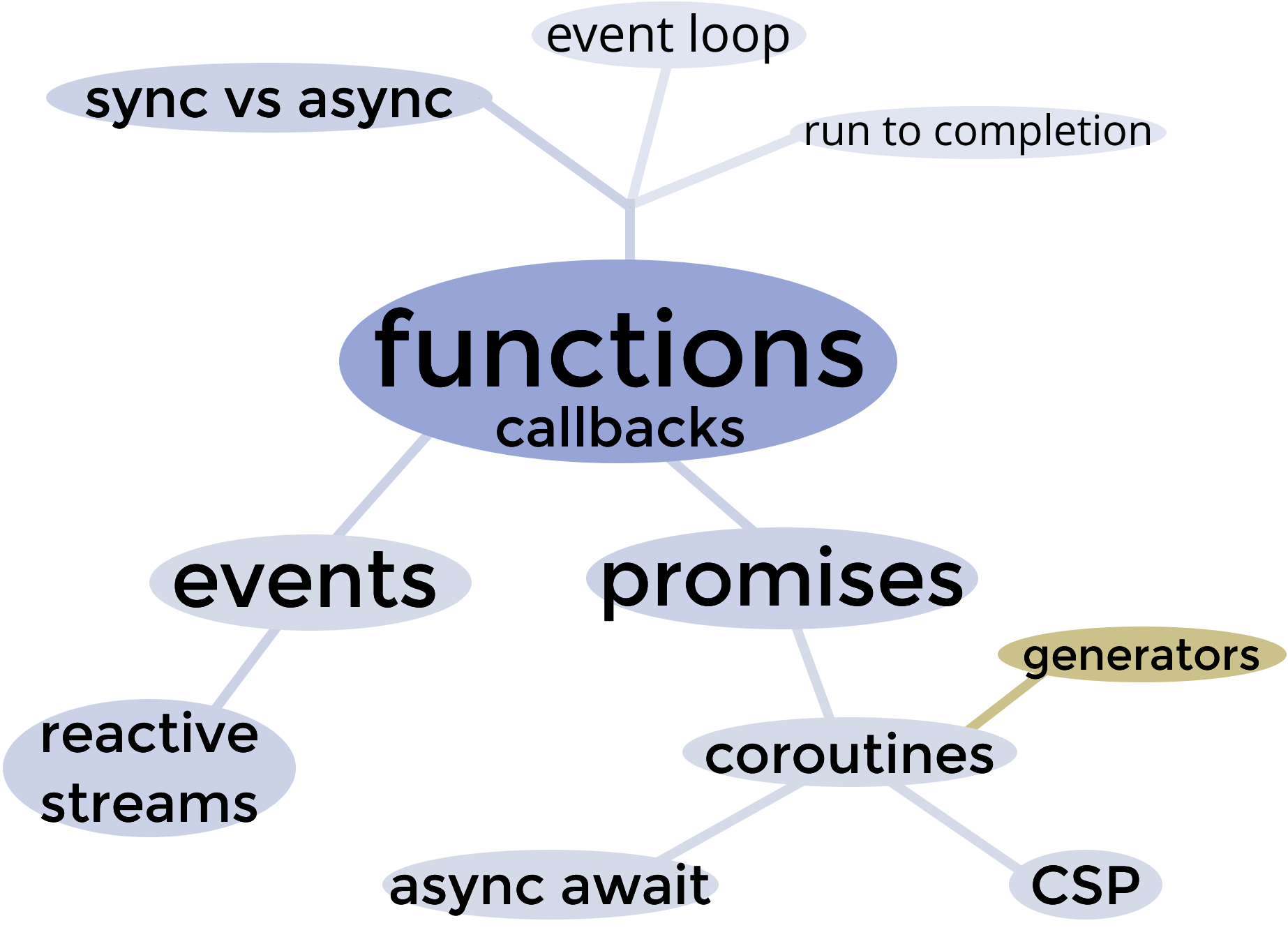

event loop
sync vs async
run to completion
functions
callbacks
promises


event loop
sync vs async
run to completion
functions
callbacks
promises
generators
coroutines


event loop
sync vs async
run to completion
functions
callbacks
promises
generators
coroutines
CSP


event loop
sync vs async
run to completion
functions
callbacks
promises
generators
coroutines
async await
CSP


event loop
sync vs async
run to completion
functions
callbacks
events
promises
generators
coroutines
async await
CSP


event loop
sync vs async
run to completion
functions
callbacks
events
promises
generators
reactive
streams
coroutines
async await
CSP

reactive streams

are difficult

not just a tool

a whole **programming paradigm**

debugging

reactivity

fundamental concepts

fundamental concepts

- reactive vs ~~imperative~~

fundamental concepts

- reactive vs ~~imperative~~
- streams

fundamental concepts

- reactive vs ~~imperative~~
- streams
- push vs ~~pull,~~
  inversion of control

fundamental concepts

- reactive vs ~~imperative~~
- streams
- push vs ~~pull,~~
  inversion of control
- lazy by default

fundamental concepts

- reactive vs ~~imperative~~
- streams
- push vs ~~pull,~~
  inversion of control
- lazy by default
- sync or async

fundamental concepts

- reactive vs ~~imperative~~
- streams
- push vs ~~pull,~~
  inversion of control
- lazy by default
- sync or async
- completed or opened

fundamental concepts

- reactive vs ~~imperative~~
- streams
- push vs ~~pull,~~
  inversion of control
- lazy by default
- sync or async
- completed or opened
- immutable

fundamental concepts

- reactive vs ~~imperative~~
- streams
- push vs ~~pull,~~
  inversion of control
- lazy by default
- sync or async
- completed or opened
- immutable
- hot or cold

why

"Why should I consider adopting RP?"

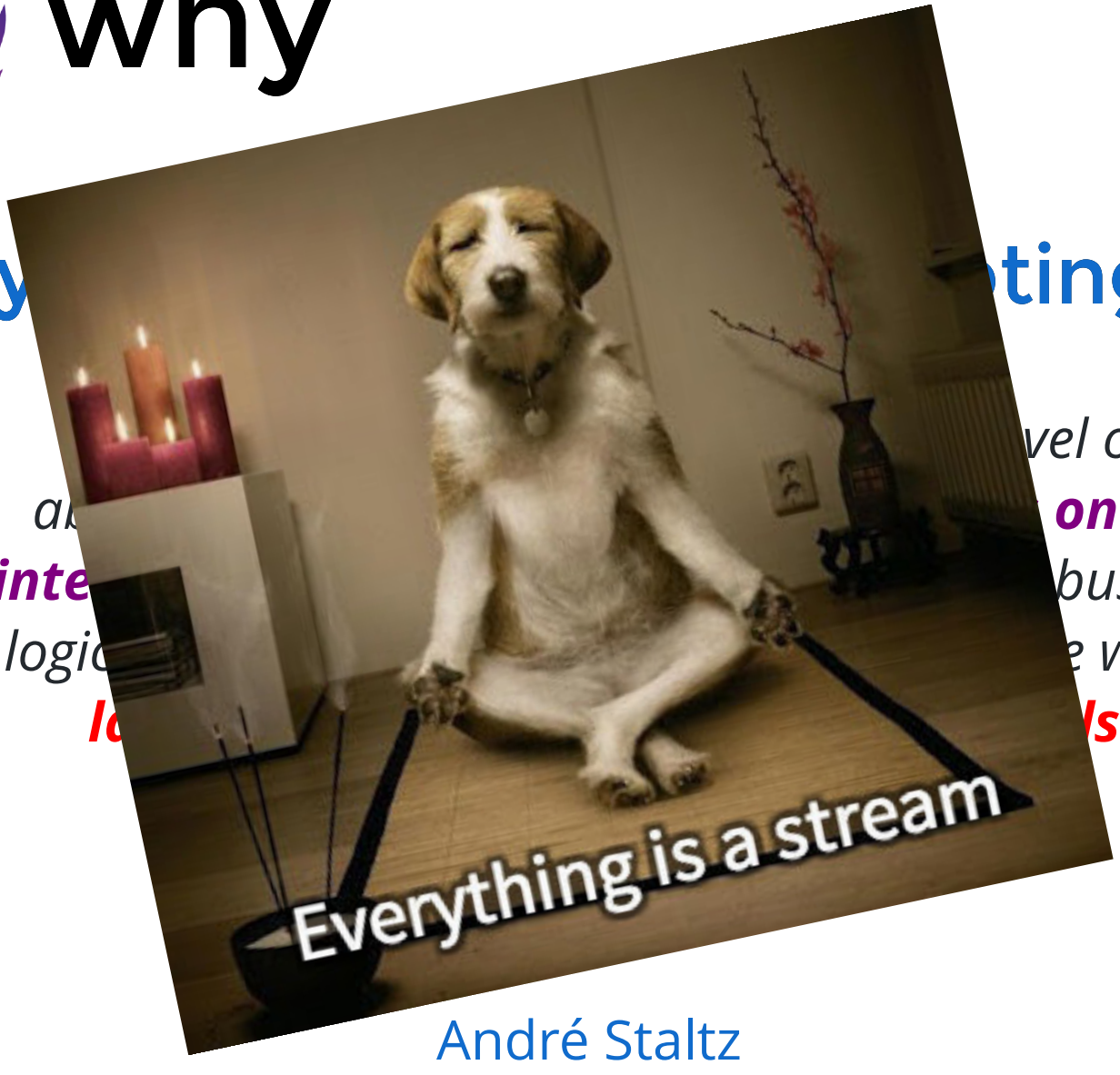
*" Reactive Programming raises the level of abstraction of your code so you can **focus on the interdependence of events** that define the business logic, **rather than** having to constantly fiddle with a **large amount of implementation details** .*
Code in RP will likely be more concise.
[...]

André Staltz

why

"Why ting RP?"

vel of
a ***on the***
inte *business*
logic *e with a*
l ***ls*** .



André Staltz

what REACTIVE is not

```
let rate = 3.94;
let amount = 1000;
let exchange = amount / rate; // 253.80

rate = 3.97;
exchange // DESYNC, sync manually!
```

what REACTIVE is not

```
let rate = 3.94;
let amount = 1000;
let exchange = amount / rate; // 253.80

rate = 3.97;
exchange // DESYNC, sync manually!
```

- imperative
- implementation details all around
- state incosistency = bugs!

what REACTIVE is

```
let rate$ = 3.94;
let amount$ = 1000;
let exchange$ = amount$ / rate$; // 253.

rate$ = 3.97;
exchange$ // 251.89
```

what REACTIVE is

```
let rate$ = 3.94;
let amount$ = 1000;
let exchange$ = amount$ / rate$; // 253.

rate$ = 3.97;
exchange$ // 251.89
```

- programming reactions (interdependence of events)
- automatic data flow
- state handled for us (IoC)

what REACTIVE is

```
let rate$ = 3.94;
let amount$ = 1000;
let exchange$ = amount$ / rate$; // 253.

rate$ = 3.97;
exchange$ // 251.89
```

^ simplified for now

- programming reactions (interdependence of events)
- automatic data flow
- state handled for us (IoC)

each **variable** ,
whose value changes
throughout program
runtime, might be
represented **as a stream**

Don’t call us, we’ll call you

The Hollywood Principle

Don’t call us, we’ll call you

IoC ?

IoC ?

*" **inversion of control (IoC)** is a design principle in which custom-written portions of a computer program receive the flow of control from a generic framework*

IoC ?

*" **inversion of control (IoC)** is a design principle in which custom-written portions of a computer program <u>receive the flow of control from a generic framework</u>*

IoC in RxJS

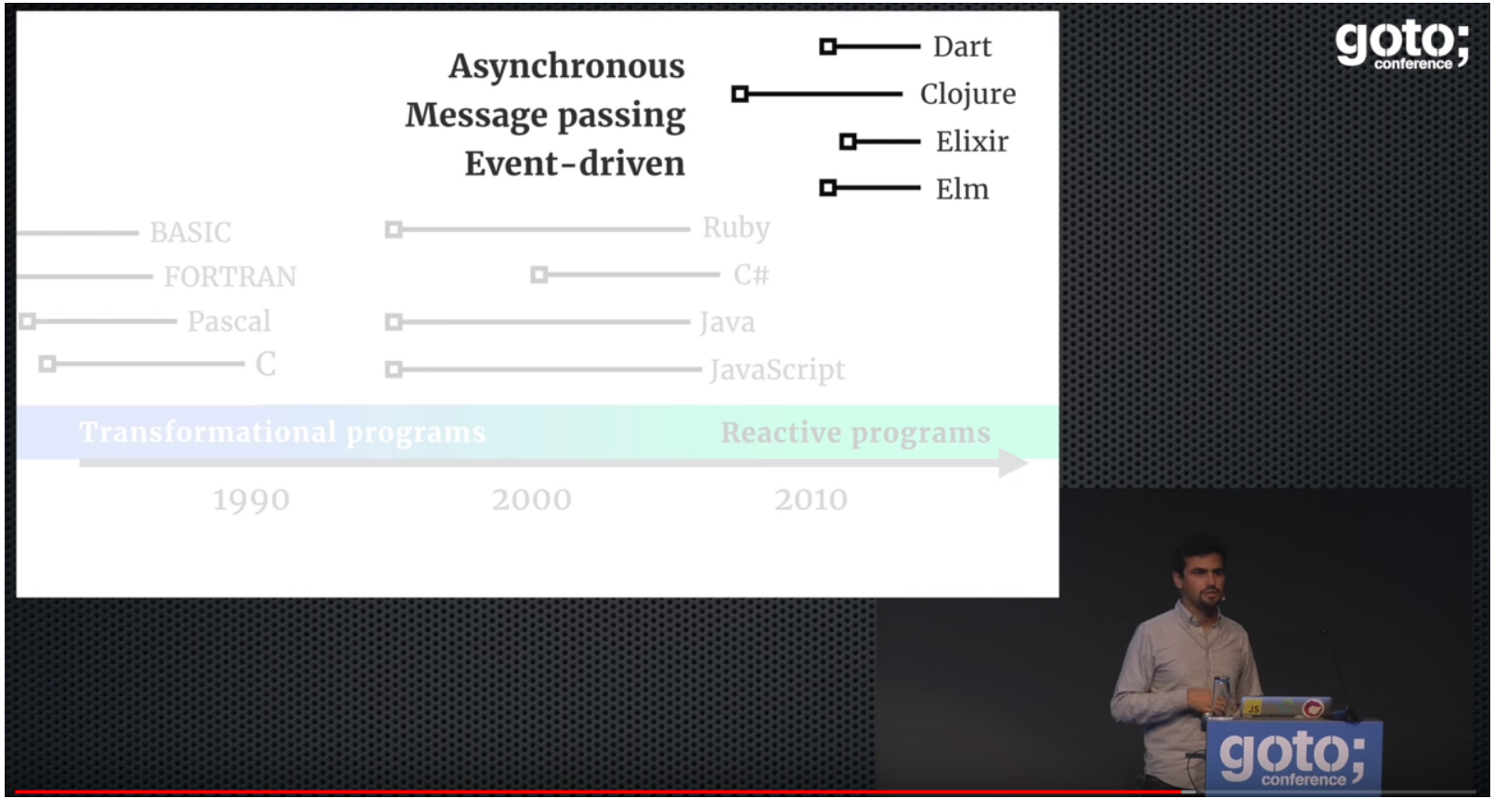
changed from a distance

```
// In the controller.js file
model.set("activity", {
  "name": "training",
  "date": new Date()
});
```

change observed from a distance

```
// ui-component.js
activity$.subscribe(activity => {
  doSomething(activity);
});
```

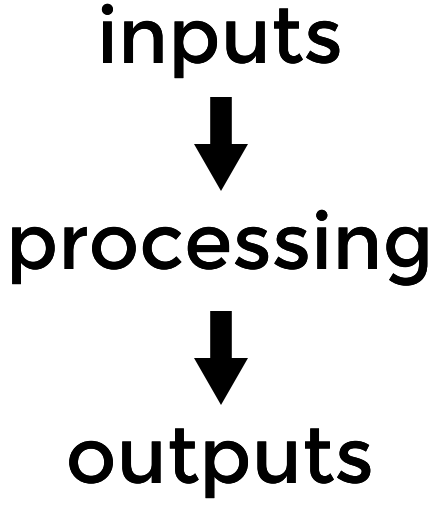
going super-high level...



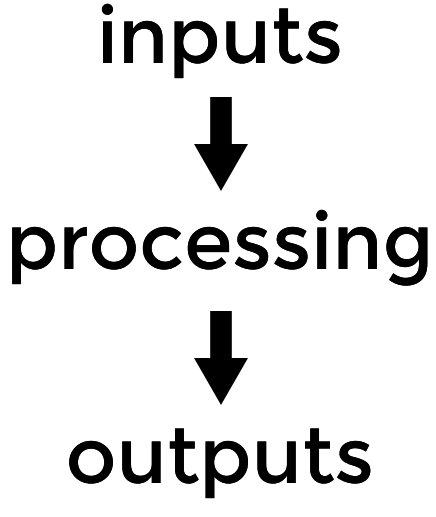
- 📹 André Staltz:
  The Return of Stream I/O

“ ***A transformational system*** *repeatedly waits for all its inputs to arrive, carries out some processing, and outputs the results when the processing is done.*

" ***A transformational system*** *repeatedly waits for all its inputs to arrive, carries out some processing, and outputs the results when the processing is done.*



" ***A transformational system*** *repeatedly waits for all its inputs to arrive, carries out some processing, and outputs the results when the processing is done.*

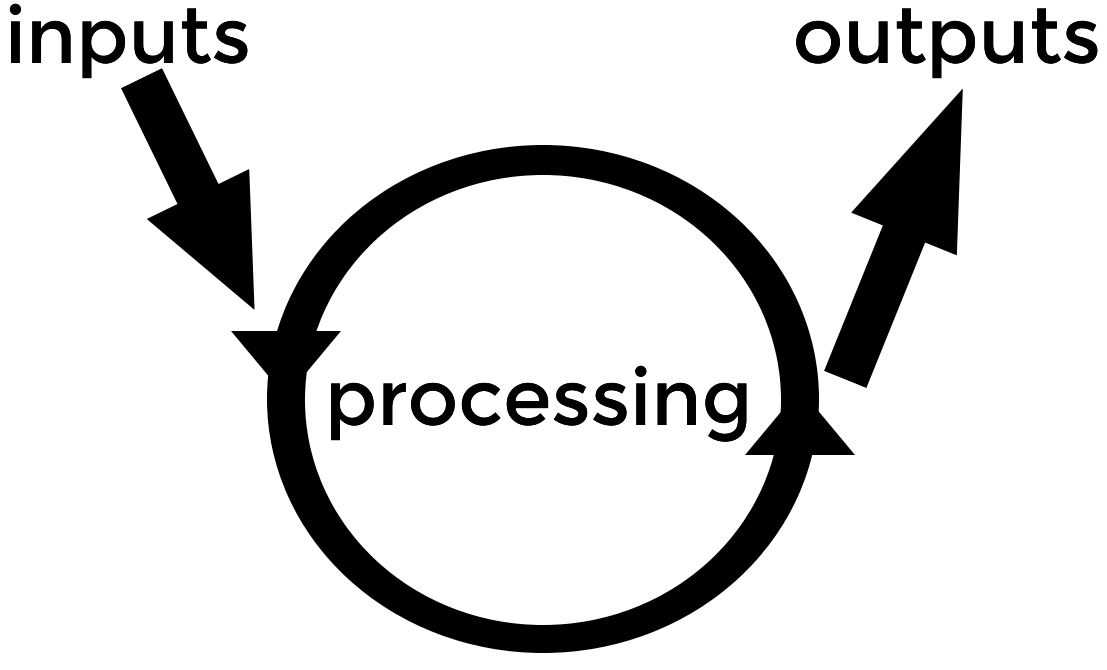


examples:

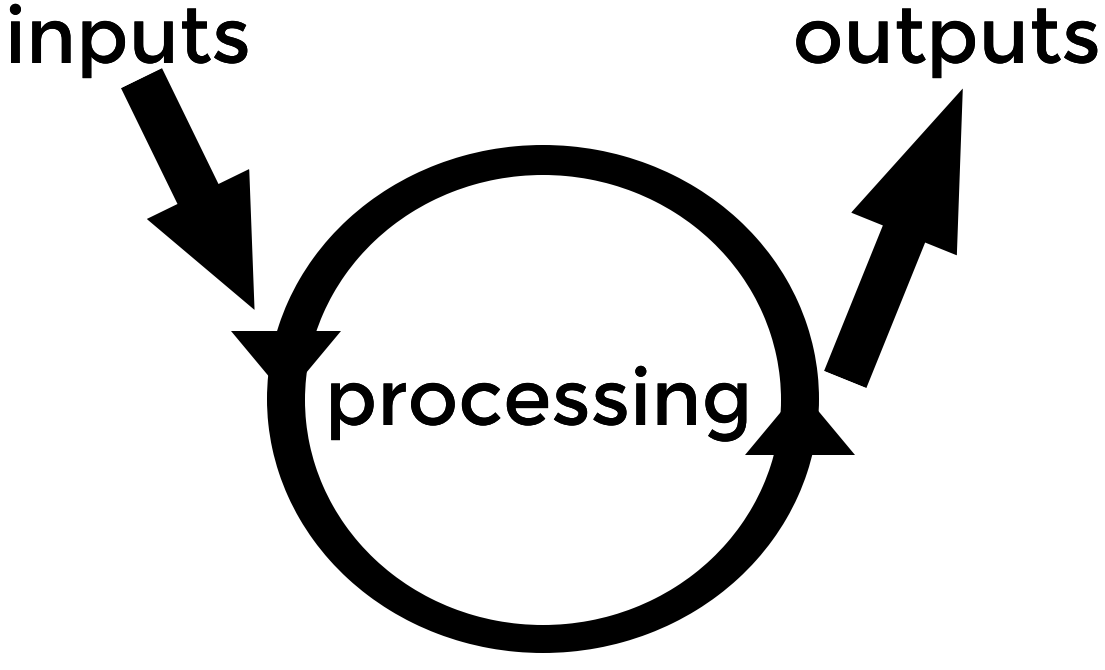
- compiler
- UNIX commands
- *file-in, file-out* programs

*" **A reactive system** continuously interacts with its environment, using inputs and outputs that are either continuous in time or discrete. The inputs and outputs are often asynchronous, meaning that they may arrive or change values unpredictably at any point in time.*

*" **A reactive system** continuously interacts with its environment, using inputs and outputs that are either continuous in time or discrete. The inputs and outputs are often asynchronous, meaning that they may arrive or change values unpredictably at any point in time.*



*" **A reactive system** continuously interacts with its environment, using inputs and outputs that are either continuous in time or discrete. The inputs and outputs are often asynchronous, meaning that they may arrive or change values unpredictably at any point in time.*



examples:

- real-time systems
- user interfaces
- Websites
- Servers

transformational

`tsc`

(typescript compiler)

reactive

`tsserver`

(typescript server)

Array

- entirely in-memory
- available upfront

Array

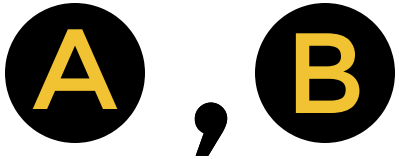
- entirely in-memory
- available upfront

Stream

Array

- entirely in-memory
- available upfront

Stream



- items pushed over time
- growing collection
- don't know when they arrive

Array

- entirely in-memory
- available upfront

Stream

- items pushed over time
- growing collection
- don't know when they arrive

Array

- entirely in-memory
- available upfront

Stream

- items pushed over time
- growing collection
- don't know when they arrive

Array

- entirely in-memory
- available upfront

Stream

- items pushed over time
- growing collection
- don't know when they arrive

Array

- entirely in-memory
- available upfront

Stream

- items pushed over time
- growing collection
- don't know when they arrive

Array - data

- entirely in-memory
- available upfront

Stream

- items pushed over time
- growing collection
- don't know when they arrive

Array - data

- entirely in-memory
- available upfront

Stream - data & time

- items pushed over time
- growing collection
- don't know when they arrive


Everything is a stream

- business data
  - HTTP, WebSockets

- business data
  - HTTP, WebSockets
- business event
  - incomming msg

- business data
  - HTTP, WebSockets
- business event
  - incomming msg
- time passed
  - timeouts, intervals
  - refresh every 30s

- business data
  - HTTP, WebSockets
- business event
  - incomming msg
- time passed
  - timeouts, intervals
  - refresh every 30s
- UI event
  - clicks, changes, checks, mouse*

- business data
  - HTTP, WebSockets
- business event
  - incomming msg
- time passed
  - timeouts, intervals
  - refresh every 30s
- UI event
  - clicks, changes, checks, mouse*
- intentions
  - refresh now
  - request to load data

real life examples

think of real-life
examples of streams

real life examples

think of real-life examples of streams

think of examples of stream processing **pipelines**

real life examples

think of real-life examples of streams

think of examples of stream processing **pipelines**

home: think of examples

tomorrow: discussion

data, time, memory

comparison

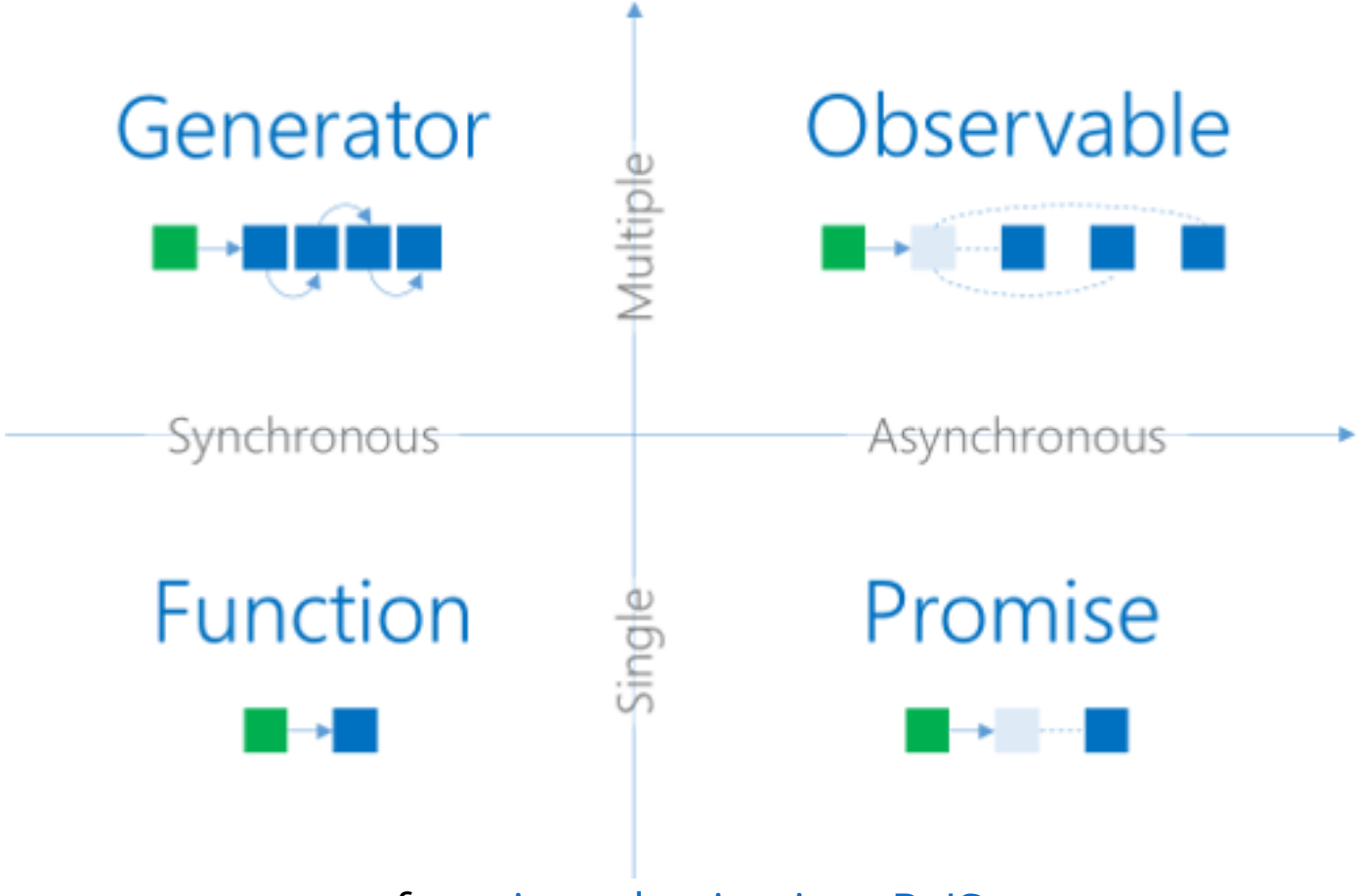
	Singular	Plural
Spatial	Value	Iterable<Value>
Temporal	Promise<Value>	Observable<Value>

from Kris Kowal's

General Theory of Reactivity

data, time, memory

comparison



from introduction into RxJS

reactive streams

an **observable** is
an attachment to
an **event stream**
that produces a **promise**
for **each event**

treating events as collections

Async Data Streams

Async Data Streams

- will happen in future
- mainly asynchronous (using only sync observables makes no sense)

Async Data Streams

- will happen in future
- mainly asynchronous (using only sync observables makes no sense)

- raw information
- objects, events, user interactions, etc.

Async Data Streams

- will happen in future
- mainly asynchronous (using only sync observables makes no sense)

- raw information
- objects, events, user interactions, etc.

- values made over time
- from many various sources

```
.map(e => e * 2)
```

```
[1, 2, 3].map(e => e * 2)
```

```
[1, 2, 3].map(e => e * 2)
```

```
stream$.map(e => e * 2)
```

```
[1, 2, 3].map(e => e * 2)
```

- all available

```
stream$.map(e => e * 2)
```

```
[1, 2, 3].map(e => e * 2)
```

- all available

```
stream$.map(e => e * 2)
```

- not necessarily exist

```
[1, 2, 3].map(e => e * 2)
```

- all available
- in-memory

```
stream$.map(e => e * 2)
```

- not necessarily exist

```
[1, 2, 3].map(e => e * 2)
```

- all available
- in-memory

```
stream$.map(e => e * 2)
```

- not necessarily exist
- sync or async, doesn't matter

```
[1, 2, 3].map(e => e * 2)
```

- all available
- in-memory

```
stream$.map(e => e * 2)
```

- not necessarily exist
- sync or async, doesn't matter
- don't know when they arrive

```
[1, 2, 3].map(e => e * 2)
```

- all available
- in-memory

```
stream$.map(e => e * 2)
.subscribe(
  value => console.log(`value: ${value}`),
  ...
);
```

- not necessarily exist
- sync or async, doesn't matter
- don't know when they arrive
- push instead of pull

reactive streams

3 fundamental pieces

- data source
- processing items
- observers

reactive streams

```
const click$ = Rx.Observable.fromEvent(document, 'click');
```

reactive streams

```
const click$ = Rx.Observable.fromEvent(document, 'click')
.map(e => ({
  x: e.clientX,
  y: e.clientY
}))
.filter(e => e.x < document.body.clientWidth/2);
```

reactive streams

```
const click$ = Rx.Observable.fromEvent(document, 'click')
.map(e => ({
  x: e.clientX,
  y: e.clientY
}))
.filter(e => e.x < document.body.clientWidth/2);

click$.subscribe(
  value => console.log(`value: ${value}`),
  e => console.warn(`error: ${e}`),
  () => console.log('completed')
);
```

reactive streams

observer

```
const click$ = Rx.Observable.fromEvent(document, 'click')
.map(e => ({
  x: e.clientX,
  y: e.clientY
}))
.filter(e => e.x < document.body.clientWidth/2);

click$.subscribe(
  value => console.log(`value: ${value}`),
  e => console.warn(`error: ${e}`),
  () => console.log('completed')
);
```

reactive streams

observer + iterator

```
const click$ = Rx.Observable.fromEvent(document, 'click')
.map(e => ({
  x: e.clientX,
  y: e.clientY
}))
.filter(e => e.x < document.body.clientWidth/2);

click$.subscribe(
  value => console.log(`value: ${value}`),
  e => console.warn(`error: ${e}`),
  () => console.log('completed')
);
```

reactive streams

observer + iterator + FP

```
const click$ = Rx.Observable.fromEvent(document, 'click')
.map(e => ({
  x: e.clientX,
  y: e.clientY
}))
.filter(e => e.x < document.body.clientWidth/2);

click$.subscribe(
  value => console.log(`value: ${value}`),
  e => console.warn(`error: ${e}`),
  () => console.log('completed')
);
```

why *not* promises

- collection *growing over time*
- item *changing over time*
- bulk processing
  (map 5XHRs to 1 result)
- cancellation

explain yourself

- reactive vs ~~imperative~~
- streams
- push vs ~~pull,~~
  inversion of control
- lazy by default
- sync or async
- completed or opened
- immutable
- hot or cold

usage

“ I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.

- Abraham Maslow

what kind of applications does RxJS fit?

code

reactivex.io

cross-platform

RxGo

RxPHP

RxPY

RxCpp

RxJava

Rx.rb

RxKotlin

UniRx

Rx.NET

RxLua

RxScala

RxClojure

RxDart

RxSwift

RxGroovy

visualisation

//jsfiddle.net/0n33fru7/embedded/result/

https://jsfiddle.net/0n33fru7/

visualisation

```
var src$ = Rx.Observable.interval(1000)
  .map(e => e*2 + 1);

src$
  .subscribe(updateActual);

src$
  .rotateAndSubscribe(row0);

src$
  .map(e => e * 2)
  .delay(333)
  .rotateAndSubscribe(row1);

src$
  .startWith(10, 20)
  .map(e => e * 3)
  .delay(666)
  .rotateAndSubscribe(row2);
```

n33fru7/embedded/result/

https://jsfiddle.net/0n33fru7/

npm install rxjs --save

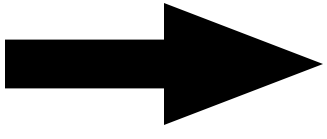
```
node_modules/
  rxjs/
    (TypeScript .d.ts files)
    (Module files)
    bundles/
      Rx.js
      Rx.min.js
      Rx.min.js.map
    src/
      (TypeScript files)
```

docs: reactivex.io/rxjs/

be careful for **breaking changes**

github ReactiveX/rxjs
CHANGELOG.md

- fromArray
- fromIterable
- fromCallback
- fromNodeCallback



Rx.Observable.from

RxJS v5

rewritten in 

conforms to *ES8* proposal

sandbox jsfiddle

jsfiddle demo:

jsfiddle.net/r3wnfLL6/

//jsfiddle.net/r3wnfLL6/
embedded/result/

creating observables

```
// LEGACY Rx.Observable.return(1)
// LEGACY Rx.Observable.just(1)
Rx.Observable.of(3, 4, 5, 6)
Rx.Observable.from([3, 4, 5, 6])
Rx.Observable.range(3, 4)
Rx.Observable.timer(miliseconds)
Rx.Observable.interval(miliseconds)
Rx.Observable.fromEvent(element, 'event')
Rx.Observable.fromPromise(promise)
Rx.Observable.defer(
   functionReturningObservable)
Rx.Observable.create(observer => {
  // manipulate each observer manually
})
```

Single

implemented in RxJava, RxGroovy, RxScala

```
Observable(onSuccess, onError)
```

why is this ignored in RxJS?

The $ convention

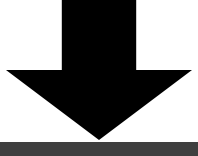
imperative collection (state)

```
users: User[];
roles: Role[];
departments: Department[];
```

The $ convention

imperative collection (state)

```
users: User[];
roles: Role[];
departments: Department[];
```



```
users$: Observable<User[]>;
roles$: Observable<Role[]>;
departments$: Observable<Department[]>;
```

reactive stream

marble diagrams

marble diagrams

```
//ASCII Marble Diagram

----0----1----2----3---->   Observable.interval(1000);
----1----2----3|            Observable.fromArray([1,2,3]);
----#                       Observable.of(1,2).do(x => throw '#'

---> is the timeline
0, 1, 2, 3 are emitted values
# is an error
| is the 'completed' signal
```

Marbles Presentation goes here...

each operator creates a new observable (!)

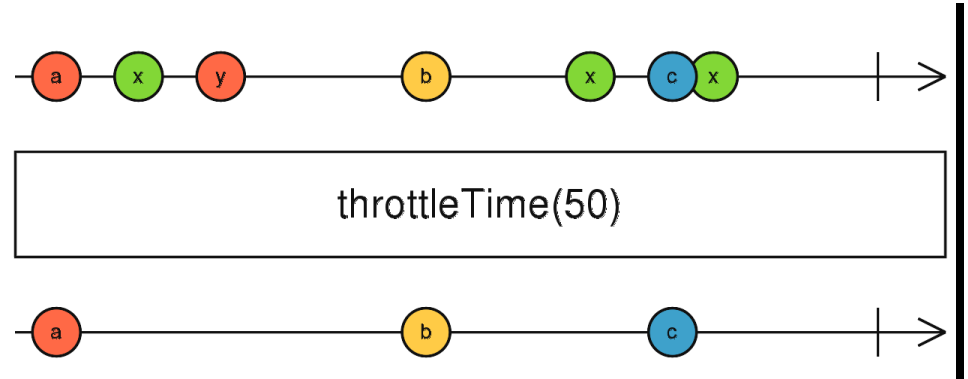
- not subscribed yet
- functional composition

operators quiz

- throttle, throttleTime
- debounce, debounceTime

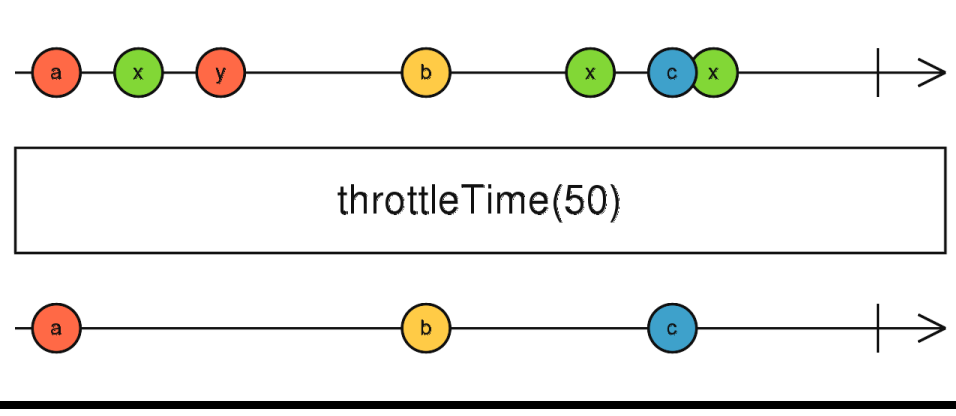
operators quiz

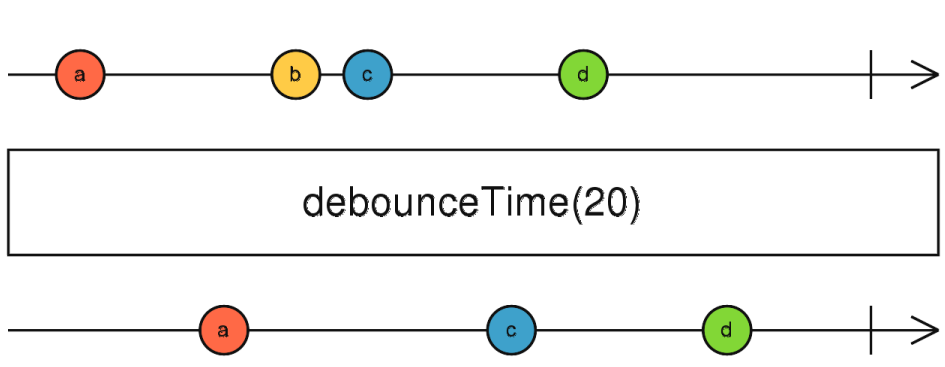
- throttle, throttleTime
- debounce, debounceTime



operators quiz

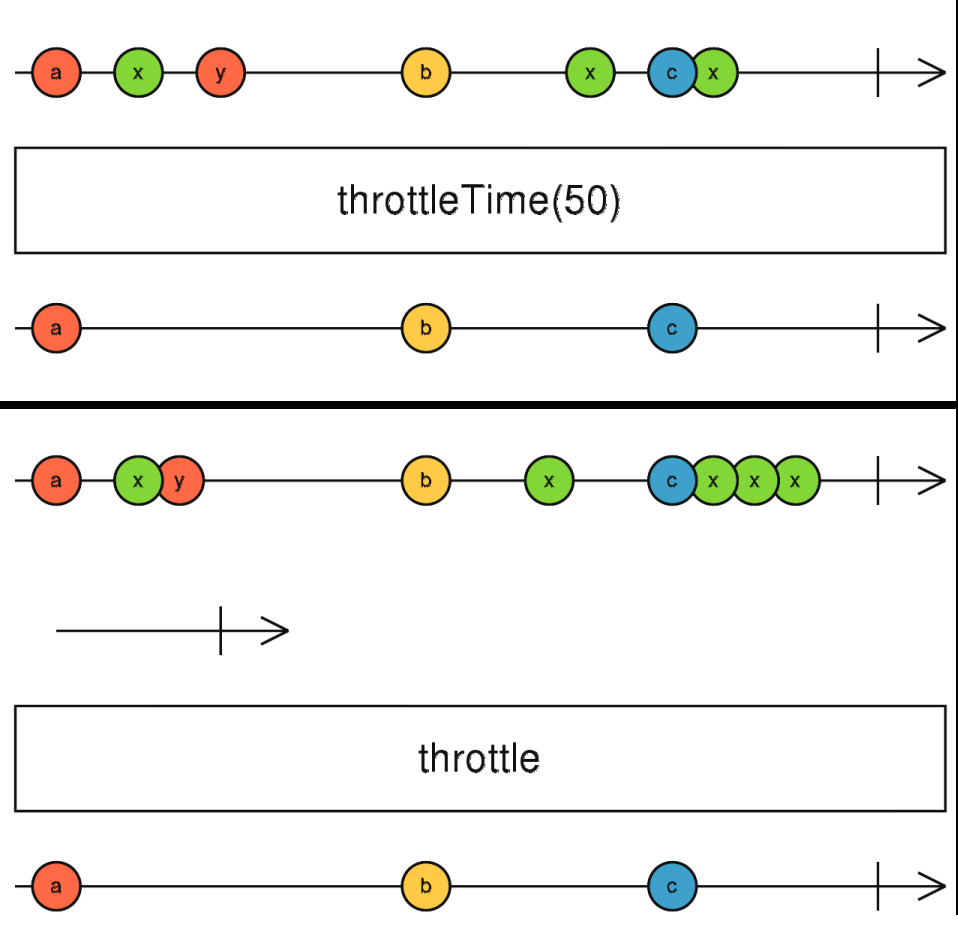
- throttle, throttleTime
- debounce, debounceTime





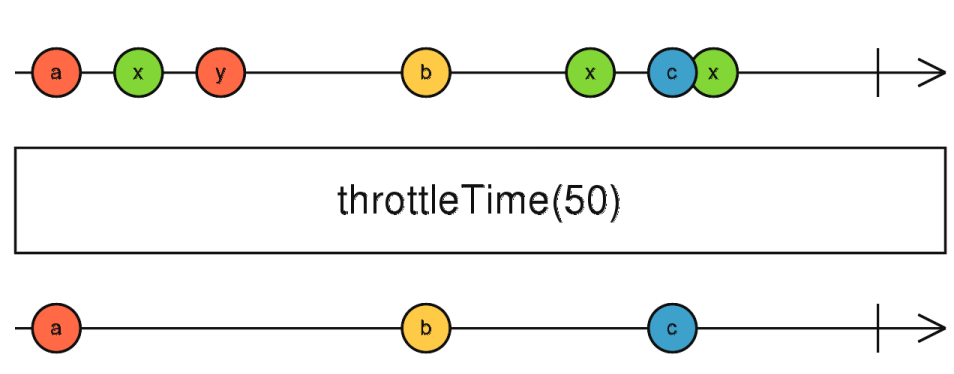
operators quiz

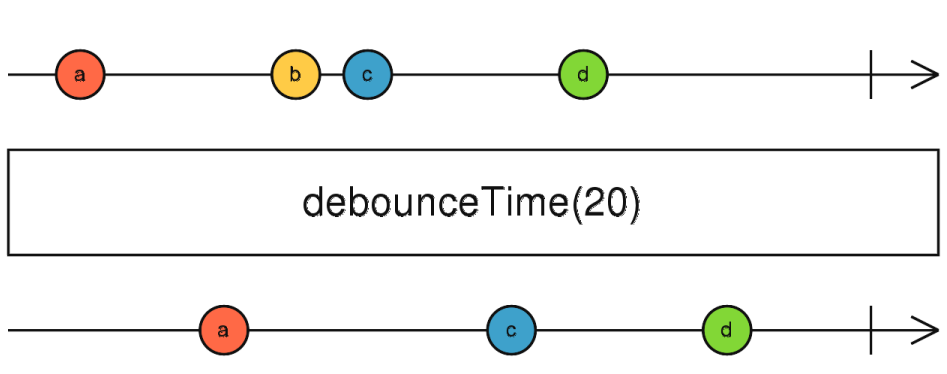
- throttle, throttleTime
- debounce, debounceTime

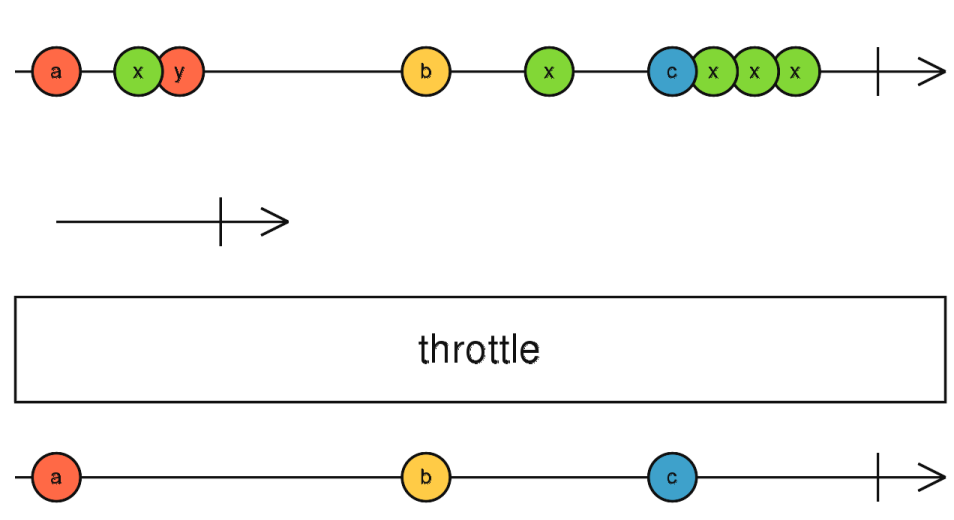


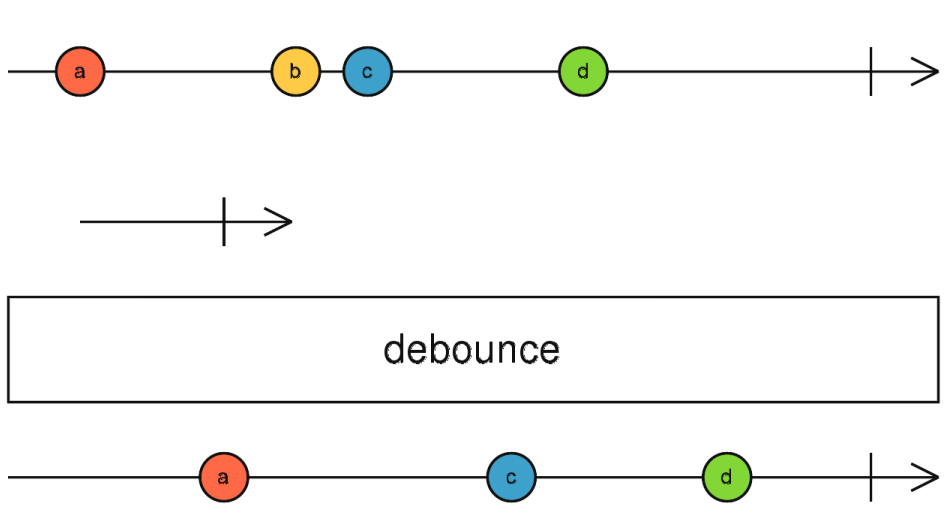
operators quiz

- throttle, throttleTime
- debounce, debounceTime









operators quiz

operators quiz

- map vs mapTo

operators quiz

- map vs mapTo
- reduce vs scan

operators quiz

- map vs mapTo
- reduce vs scan
- map vs flatMap

operators quiz

- map vs mapTo
- reduce vs scan
- map vs flatMap
- concat vs merge

operators quiz

- map vs mapTo
- reduce vs scan
- map vs flatMap
- concat vs merge
- zip vs combineLatest vs withLatestFrom

operators quiz

- map vs mapTo
- reduce vs scan
- map vs flatMap
- concat vs merge
- zip vs combineLatest vs withLatestFrom
- delay vs interval vs timer

operators quiz

- map vs mapTo
- reduce vs scan
- map vs flatMap
- concat vs merge
- zip vs combineLatest vs withLatestFrom
- delay vs interval vs timer
- delay vs delayWhen

operators quiz

- flatMap
- switchMap
- concatMap
- exhaustMap

flattening operators

- **flatMap / mergeMap**
  - concurrency
- **switchMap**
  - no concurrency, cancelling
- **concatMap**
  - no concurrency, no cancelling
- **exhaustMap**
  - no concurrency, ingoring

observers (subscribers)

observers (subscribers)

functional API

```
let source$ = Rx.Observable.of(1, 2, 3);

source$.subscribe(
  value => console.log(`value: ${value}`),
  e => console.error(`error: ${e}`),
  () => console.info('completed')
);
```

demo

```
value: 1
value: 2
value: 3
completed
```

observers (subscribers)

functional API

```
let source$ = Rx.Observable.of(1, 2, 3);

source$.subscribe(
  value => console.log(`value: ${value}`),
  e => console.error(`error: ${e}`),
  () => console.info('completed')
);
```

demo

```
value: 1
value: 2
value: 3
completed
```

```
let source$ = Rx.Observable.of(1, 2, 3);

source$.subscribe(
  console.log,
  console.error,
  _ => console.info('completed')
);
```

demo

observers (subscribers)

object literal API

```
let source$ = Rx.Observable.of(1, 2, 3);

source$.subscribe({
  next(value){
    console.log(`value: ${value}`);
  },
  error(e){
    console.warn(`error: ${e}`);
  },
  complete(){
    console.log('completed');
  }
});
```

demo

```
value: 1
value: 2
value: 3
completed
```

observers (subscribers)

class API

```
let source$ = Rx.Observable.of(1, 2, 3);

class Subscriber {
  next(value){
    console.log(`value: ${value}`);
  }
  error(e){
    console.error(`error: ${e}`);
  }
  complete(){
    console.info('completed');
  }
}

source$.subscribe(new Subscriber());
```

```
value: 1
value: 2
value: 3
completed
```

demo

unsubscribe

```
let source$ = Rx.Observable.interval(1000);

let subscription = source$.subscribe(
  value => console.log(`value: ${value}`),
  e => console.error(`error: ${e}`),
  () => console.info('completed')
);

// value: 0
// value: 1
// value: 2
subscription.unsubscribe();
```

complete your streams instead of unsubscribing!

cleaning up

```
let stream$ = new Rx.Observable.create((observer) =
  let i = 0;
  let id = setInterval(() => {
    observer.next(i++);
  }, 100)

  return function(){
    clearInterval(id);
  }
});

let subscription = stream$.subscribe(
  console.log
);

setTimeout(() => {
  // indirectly invoking clean up
  subscription.unsubscribe()
}, 5000);
```

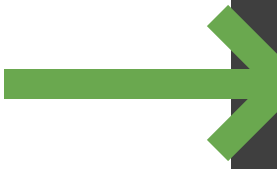
cleaning up

```
let stream$ = new Rx.Observable.create((observer) =
  let i = 0;
  let id = setInterval(() => {
    observer.next(i++);
  }, 100)

  return function(){
    clearInterval(id);
  }
});

let subscription = stream$.subscribe(
  console.log
);

setTimeout(() => {
  // indirectly invoking clean up
  subscription.unsubscribe()
}, 5000);
```



no need to unsubscribe completed streams

guest starring async pipe

```
@Component({
  selector: 'async-component',
  template: `<code>promise|async</code>
             <code>stream|async</code>`
})
export class AsyncComponent {
  private promise: Promise<any>;
  private stream: Observable<any>;

  constructor(){
    this.promise = new Promise((res, rej) => {
      setTimeout(() => res('item-p'), 1000)
    });

    this.stream = Observable.of('item-s')
      .delay(1000)
  }
}
```

guest starring
async pipe

```
@Component({
  selector: 'async-component',
  template: `<code>promise|async</code>
             <code>stream|async</code>`
})
export class AsyncComponent {
  private promise: Promise<any>;
  private stream: Observable<any>;

  constructor(){
    this.promise = new Promise((res, rej) => {
      setTimeout(() => res('item-p'), 1000)
    });

    this.stream = Observable.of('item-s')
      .delay(1000)
  }
}
```

subscribed after ngOnInit

exercises warmup

Waves - exercises

JSfiddle:
https://goo.gl/WxwTuD

github gist:
https://goo.gl/p7kiF9

Waves - exercises

//jsfiddle.net/tomasz_ducin/av3dpuj1/3/embedded/result/

Timer - exercises

JSfiddle:
https://goo.gl/WkZHd2

github gist:
https://goo.gl/J3P6nH

Timer - exercises

//jsfiddle.net/tomasz_ducin/t77unkco/embedded/result/

JSfiddle:
https://goo.gl/WkZHd2

github gist:
https://goo.gl/J3P6nH

autocompleter

jsfiddle demo:

https://goo.gl/xPC9i3

//jsfiddle.net/tomasz_ducin/ohp3wewy/1/embedded/result/

gotchas

gotchas

```
const source$ = Rx.Observable.interval(500)
  .reduce( (aggr, item) => aggr + item, 0 )

source$.subscribe(
  console.log,
  console.error,
  _ => console.info('completed')
)
```

gotchas

```
const source$ = Rx.Observable.interval(500)
  .reduce( (aggr, item) => aggr + item, 0 )

source$.subscribe(
  console.log,
  console.error,
  _ => console.info('completed')
)
```

use scan instead of reduce (waiting for completion)

gotchas

```
const source$ = Rx.Observable
  .concat(
    Rx.Observable.interval(1000),
    Rx.Observable.of('John', 'Lennon', 'died')
  )
  .delay(5000)

source$.subscribe(
  console.log,
  console.error,
  _ => console.info('completed')
)
```

gotchas

```
const source$ = Rx.Observable
  .concat(
    Rx.Observable.interval(1000),
    Rx.Observable.of('John', 'Lennon', 'died')
  )
  .delay(5000)

source$.subscribe(
  console.log,
  console.error,
  _ => console.info('completed')
)
```

use merge instead of concat (waiting for completion)

let & pipe

let

accessing the whole observable not just items one by one

sequences of operators - **reused**

```
// REUSABLE!
const retryThreeTimes = obs =>
    obs.retry(3)
    .catch(_ => Rx.Observable.of('ERROR!'));

const fetchData = Rx.Observable.of('api/users')
  .mergeMap(url => examplePromise(url))
  .let(retryThreeTimes);

fetchData.subscribe(...);
```

RxJS 5.5 pipe

functional composition

instead of prototypal inheritance

```
const source$ = Observable.range(0, 10)

const filterOutEvens = filter(x => x % 2)
const doubleBy = x => map(value => value * x);
const sum = reduce((acc, next) => acc + next, 0);

source$.pipe(
  filterOutEvens,
  doubleBy(2),
  sum
).subscribe(...);
```

link, link

RxJS 5.5 pipe

functional composition

instead of prototypal inheritance

```
const source$ = Observable.range(0, 10)

const filterOutEvens = filter(x => x % 2)
const doubleBy = x => map(value => value * x);
const sum = reduce((acc, next) => acc + next, 0);

source$.pipe(
  filterOutEvens,
  doubleBy(2),
  sum
).subscribe(...);
```

link, link

```
const complicatedLogic = pipe(
  filterOutEvens,
  doubleBy(2),
  sum
);

source$
  .let(complicatedLogic)
  .subscribe(...);
```

RxJS 5.5 changes

1. do -> tap
2. catch -> catchError
3. switch -> switchAll
4. finally -> finalize

why pipes?

dot-chained operators considered **harmful** :

1. imported in 1 place -> affects everyone, because prototype gets modified
2. not tree-shakeable (webpack, rollup, etc.)
3. unused variables linting impossible
4. custom operators are easier to implement with functional composition

HOO

HOO

Higher Order Functions

- takes 1 or more functions as arguments
- returns a function as a result

HOO

Higher Order Functions

- takes 1 or more functions as arguments
- returns a function as a result

Higher Order Components

- takes 1 or more components as arguments
- returns a component as a result

HOO

Higher Order Functions

- takes 1 or more functions as arguments
- returns a function as a result

Higher Order Components

- takes 1 or more components as arguments
- returns a component as a result

H igher O rder O bservables

- an observable of observables
- many inner observables, 1 outer observable

HOO

Higher Order Functions

`Fn(Fn<T>()) -> Result<T>`

Higher Order Components

`Component(Component) -> Component`

Higher Order Observables

`Observable<`


LET'S PLAY A
GAME
quickmeme.com

LET'S PLAY A
GAME


LET'S PLAY A
concatMap
GAME


LET'S PLAY A
concatMap
switchMap
GAME


LET'S PLAY A
concatMap
switchMap
mergeMap
GAME


LET'S PLAY A
concatMap
switchMap
mergeMap
exhaustMap
GAME

promises vs observables

promise vs observable

promise API

promise.then(**onSuccess**, [**onFailure**])
promise.catch(**onFailure**)

observable API

observable$.subscribe(
onNext , **onError**, **onCompleted**)

promise vs observable

promises

promise.then cannot be undone once registered

observables

observable.subscribe can be unsubscribed

promise vs observable

promises

are one-time operations (disposable)

observables

are of multiple use

André Staltz in 🇫🇮
@andrestaltz

Promises:
- Always async
- 1 value
- Eager
- Cannot be cancelled

Observables:
- Sync or async
- 0 or many values
- Lazy
- Can be cancelled

Zobacz tłumaczenie

PODANYCH DALEJ 146 POLUBIENIA 262

observable ➡ promise

- emits last item before completion (similat to AsyncSubject)
- no emit if not completed
- reject if error

```
let sourceOf$ = Rx.Observable.of(42)

let sourceInterval$ =
  Rx.Observable.interval(1000)
  .take(5);

sourceOf$
  .toPromise()
  .then(d => console.log('of', d))
sourceInterval$
  .toPromise()
  .then(d => console.log('interval', d))
```

jsfiddle demo

observable ➡ promise

- emits last item before completion (similat to AsyncSubject)
- no emit if not completed
- reject if error

```
let sourceOf$ = Rx.Observable.of(42)

let sourceInterval$ =
  Rx.Observable.interval(1000)
  .take(5);

sourceOf$
  .toPromise()
  .then(d => console.log('of', d))
sourceInterval$
  .toPromise()
  .then(d => console.log('interval', d))
```

jsfiddle demo

```
// immediately
of 42

// all ignored
// interval 0
// interval 1
// interval 2
// interval 3

// last item
// completes
interval 4
```

observable ➡ promise

```
let sourceOf$ = Rx.Observable.of(42)

let sourceInterval$ =
  Rx.Observable.interval(1000);
  // never completes !!!

sourceOf$
  .toPromise()
  .then(d => console.log('of', d))
sourceInterval$
  .toPromise()
  .then(d => console.log('interval', d))
```

jsfiddle demo

observable ➡ promise

```
let sourceOf$ = Rx.Observable.of(42)

let sourceInterval$ =
  Rx.Observable.interval(1000);
  // never completes !!!

sourceOf$
  .toPromise()
  .then(d => console.log('of', d))
sourceInterval$
  .toPromise()
  .then(d => console.log('interval', d))
```

```
// immediately
of 42

// all ignored
// interval 0
// interval 1
// interval 2
// interval 3
// interval 4
// interval 5
// ...
// never ends
```

jsfiddle demo

promise ➡ observable

- one-item stream
- emit & completed, if resolved
- error, if rejected

```
var source$ = new Rx.Observable.fromPromise(p);
source$.subscribe(new Subscriber('A'));
```

jsfiddle demo

observables + promises

does it make
sense to mix APIs?

reactive streams vs generators

generators:

reactive streams:

reactive streams vs generators

generators: lazy/imperative

reactive streams: lazy/IoC

built on top of RxJS

guest starring

as hard-dependency, HTTP

guest starring

as hard-dependency, HTTP

```
getItems(onNext, onError) {
  this.http
    .get("/items")
    .map(response => response.json())
    .retry(2)
    .subscribe(onNext, onError)
}
```

HTTP calls

are *cold* observables

- each subscriber will get its own producer, i.e. will make a separate call

guest starring

as ***EPIC*** in redux-observable, Netflix

guest starring

as ***EPIC*** in redux-observable, Netflix

```
function (action$: Observable<Action>, store: Store):
    Observable<Action>;
```

take a stream of actions, return a stream of actions

actions in, actions out

epics

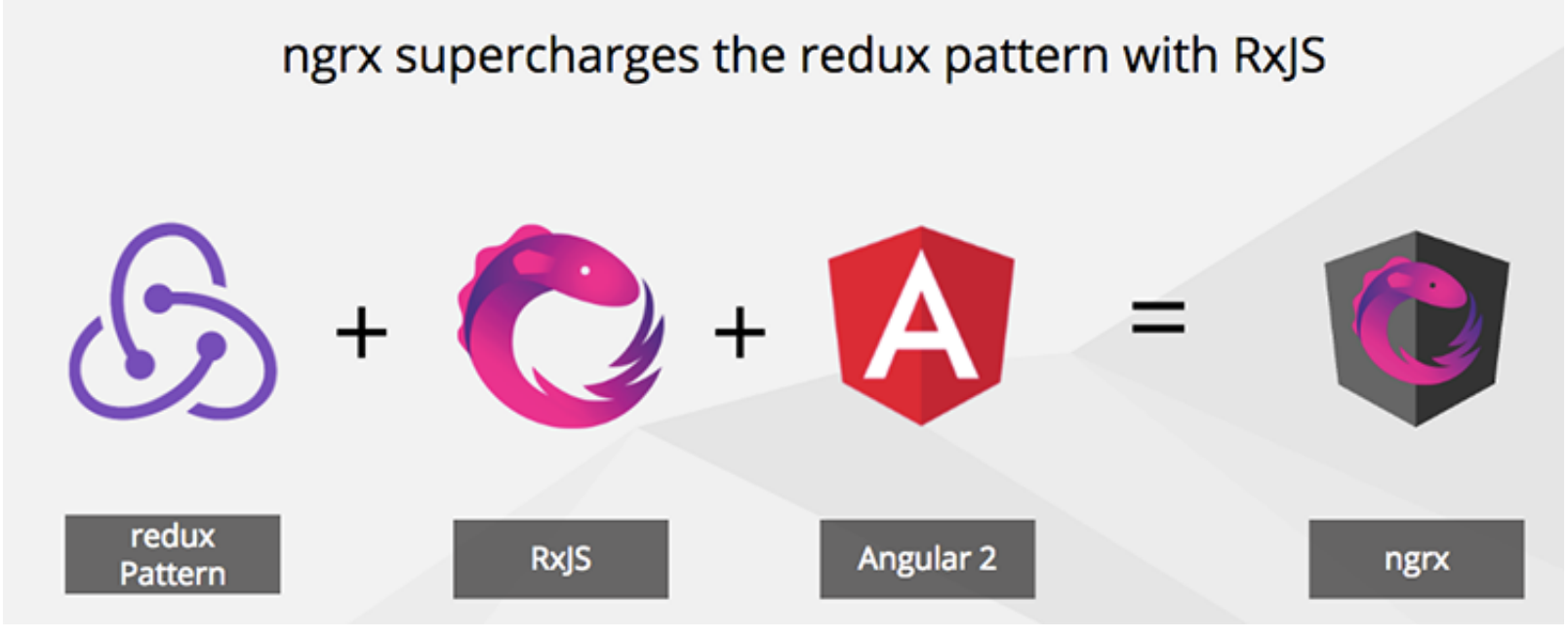
actions in, actions out

```
function (action$: Observable<Action>, store: Store):
    Observable<Action>;
```

The actions you emit will be immediately dispatched through the normal **`store.dispatch()`** , so under the hood redux-observable effectively does:

```
epic(action$, store)
.subscribe(store.dispatch)
```

guest starring



@ngrx/store

just redux

the store is also an observable

guest starring

JS +  =

as an ECMAScript a built-in (proposal)

Stage 1 Draft (November 20th, 2017)

```
.next()
.error()
.complete()
```

```
.subscribe()
.unsubscribe()
```

```
Observable.of()
Observable.from()
```

hot n'cold

quiz

```
randomItem$ = Observable.of()
  .map(() => Math.random())

randomItem$.subscribe(console.log) // sub A
randomItem$.subscribe(console.log) // sub B
```

hot n'cold

```
// promise gets started
var promise = $.ajax('/users/1');

// promise used by 1st consumer
promise.then(callback1);

// same promise used by another consumer
promise.then(callback2);
```

```
// observable is just a function composition
var source$ = new Rx.Observable.interval(500)

// separate instance for subscriber A
source$.subscribe(new Subscriber('A'))

// and separate for consumer B
source$.subscribe(new Subscriber('B'))
```

hot n'cold

```
// promise gets started
var promise = $.ajax('/users/1');

// promise used by 1st consumer
promise.then(callback1);

// same promise used by another consumer
promise.then(callback2);
```

promises are greedy and always hot

shared among all .then calls

observables are cold and lazy by default

separate for each sub

```
// observable is just a function composition
var source$ = new Rx.Observable.interval(500)

// separate instance for subscriber A
source$.subscribe(new Subscriber('A'))

// and separate for consumer B
source$.subscribe(new Subscriber('B'))
```

hot n'cold

cold = emitting separate items for
separate subscribers
(create new producer for each consumer),
the default

hot = emitting same items
for all subscribers
(single producer for all consumers)

hot n'cold

hot n'cold

hot

event (click) observables are hot by default (emit, even if no subscribers)

hot n'cold

hot

event (click) observables are hot by default (emit, even if no subscribers)

cold

Http observables are cold by default in angular

start running on subscription

hot n'cold

hot **warm** **cold**

event (click) observables are hot by default (emit, even if no subscribers)

Http observables are cold by default in angular

start running on subscription

hot n'cold

cold

```
let source$ = Rx.Observable.create(observer =>
  observer.next(Date.now())
);

source$.subscribe(v =>
  console.log("1st subscriber: " + v));

source$.subscribe(v =>
  console.log("2nd subscriber: " + v)
```

```
"1st subscriber: 1496782641047"
"2nd subscriber: 1496782641048"
```

hot n'cold

`.publish()` shares value producer

warm

```
let source$ = Rx.Observable.create(observer =>
  observer.next(Date.now())
).publish();

source$.subscribe(v =>
  console.log("1st subscriber: " + v));

source$.subscribe(v =>
  console.log("2nd subscriber: " + v)

source$.connect();
```

```
"1st subscriber: 1496782641047"
"2nd subscriber: 1496782641047"
```

hot n'cold

`.connect()` actually subscribes to it

hot

```
let source$ = Rx.Observable.create(observer =>
  observer.next(Date.now())
).publish();
obs.connect();

source$.subscribe(v =>
  console.log("1st subscriber: " + v));

source$.subscribe(v =>
  console.log("2nd subscriber: " + v)
```

```
(no output)
```

hot n'cold

`.refCount()` keeps active subscriptions

warm

subscribes when first subscriber joins,
unsubscribes when all subscribers are gone

```
stream$.publish().refCount()
```

same as

```
stream$.share()
```

hot cache stream

`.publishReplay() = .publish() + ReplaySubject`

warm

```
this.http.get(`${API_URL}/user/geo/countries`)
.map((res) => res.json());
.publishReplay()
.refCount();
```

first subscriber triggers request, shares as long as 1 subscriber is here

hot

```
this.http.get(`${API_URL}/user/geo/countries`)
.map((res) => res.json());
.publishReplay()
.connect();
```

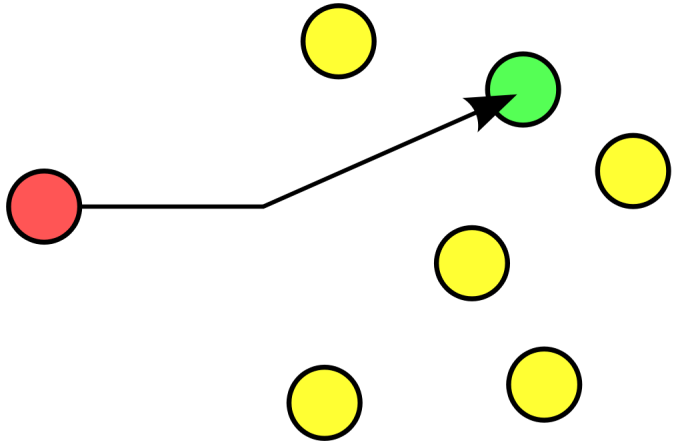
fires request immediately, keeps data shared until exists

~~hot n'cold~~

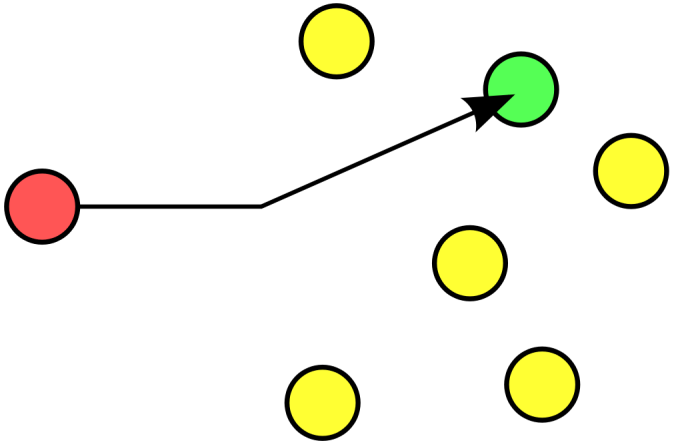
subjects

1 pub - n sub

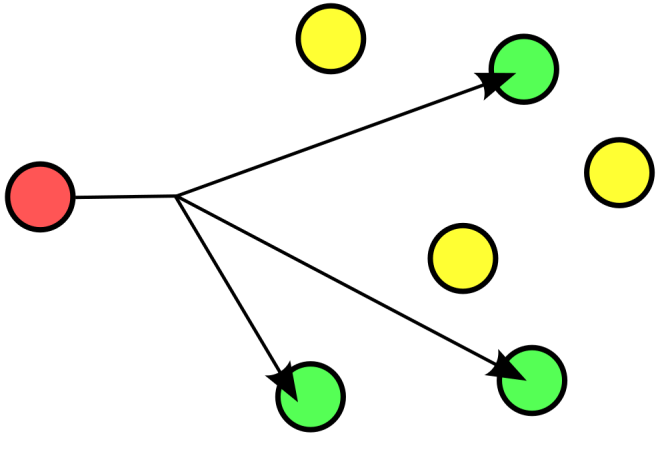
each ***observable execution***
has only one observer



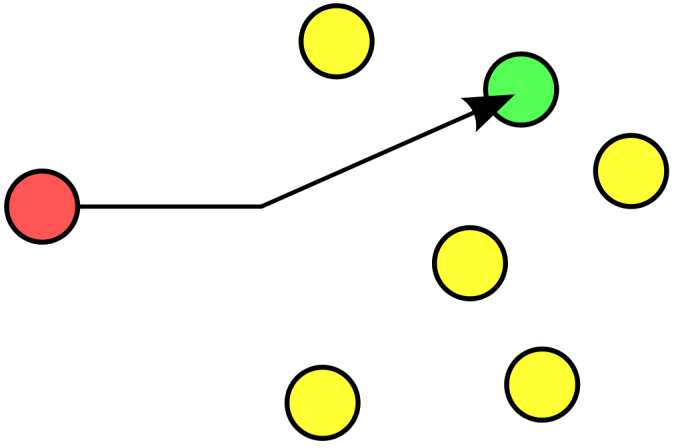
unicast



unicast

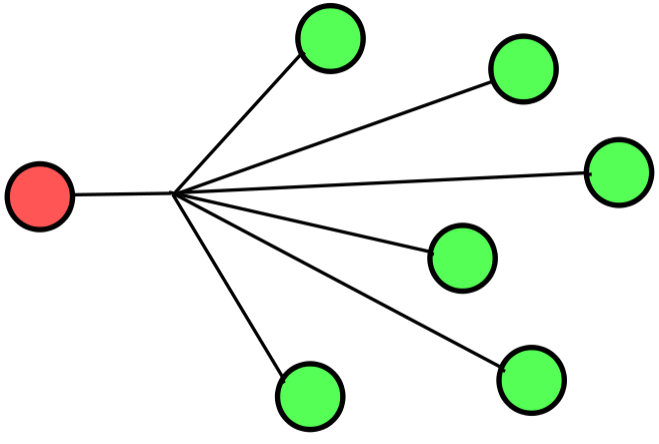


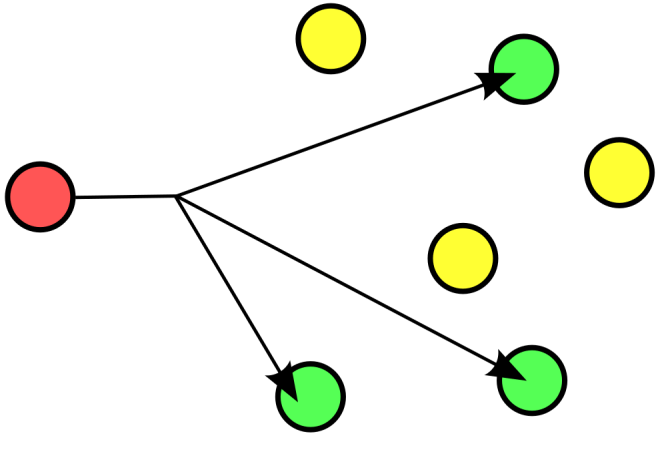
multicast



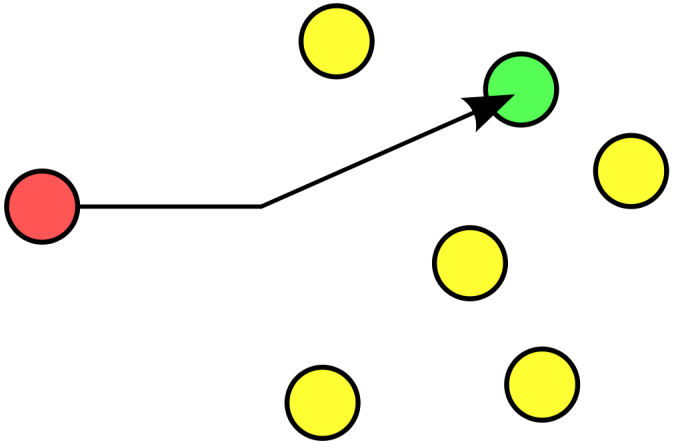
unicast

broadcast



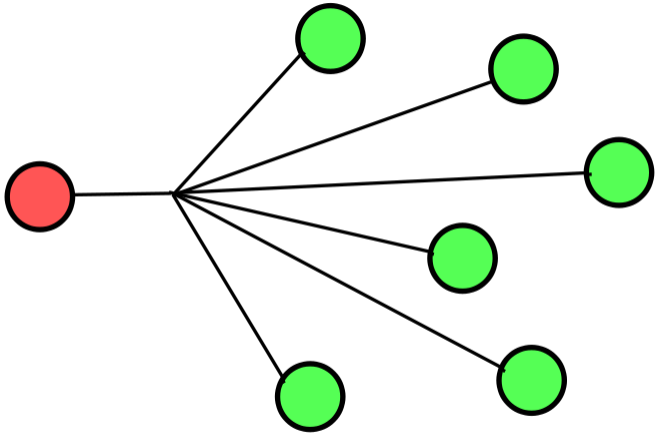


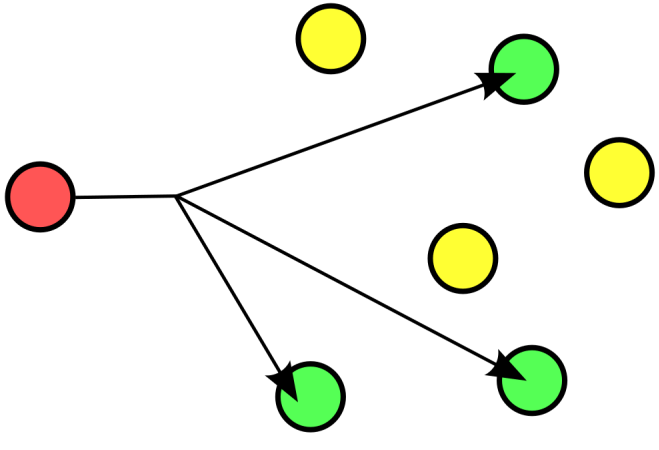
multicast



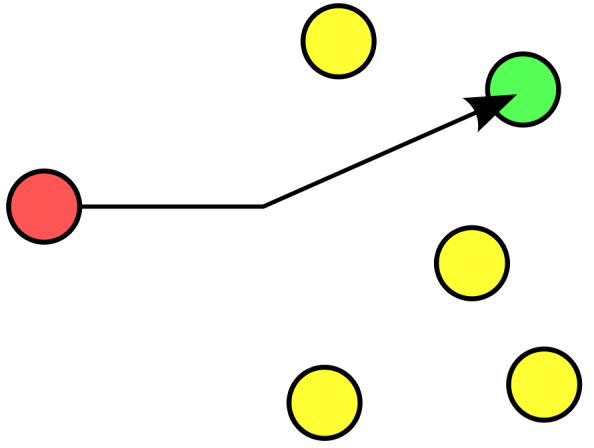
unicast

~~broadcast~~





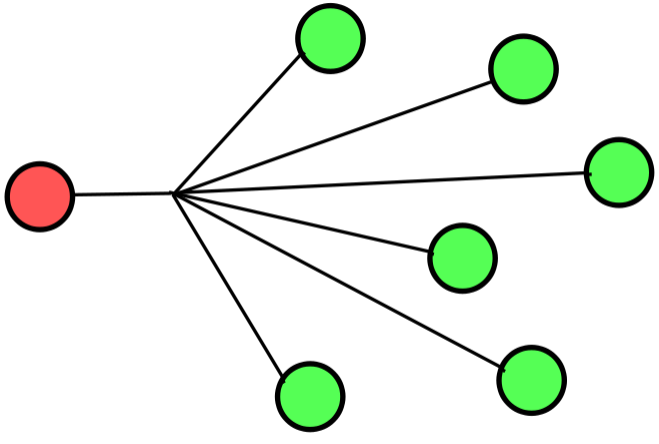
multicast

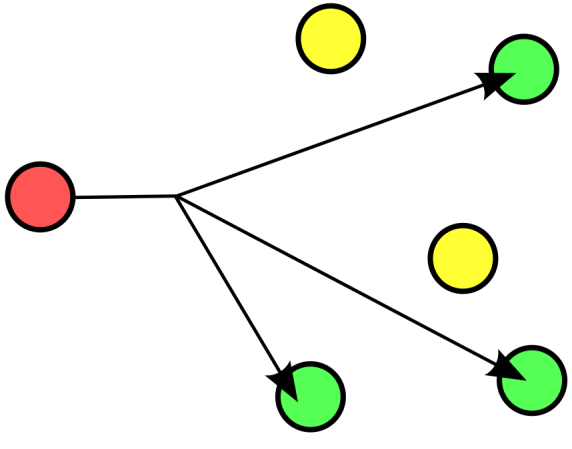


unicast

RxJS observables

~~broadcast~~





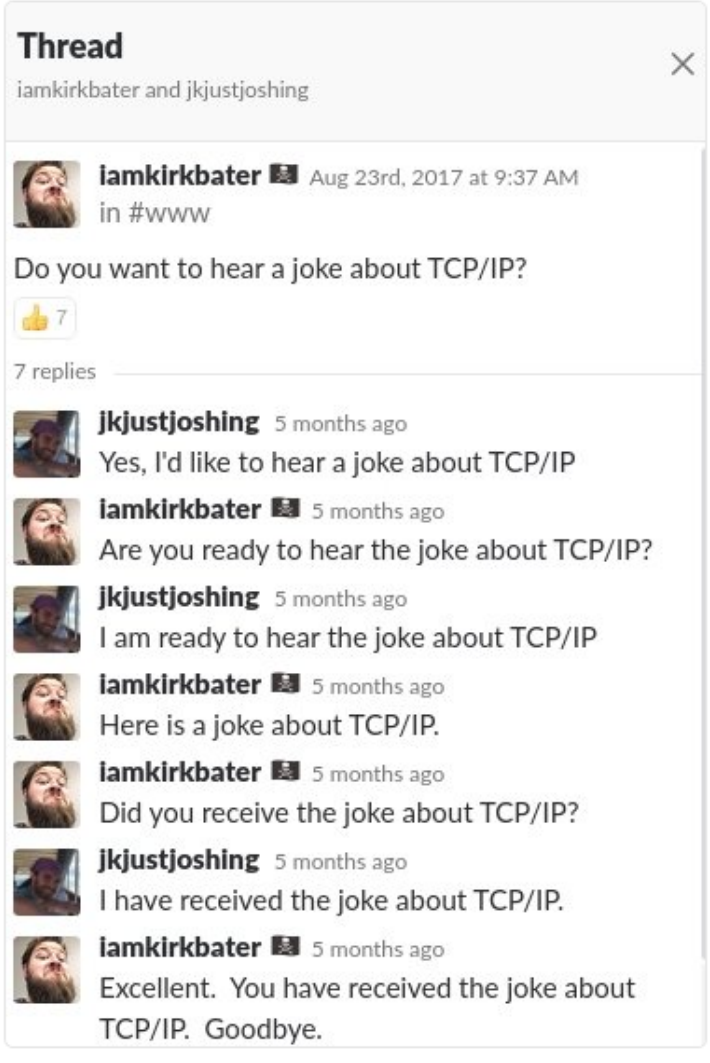
multicast

RxJS subjects

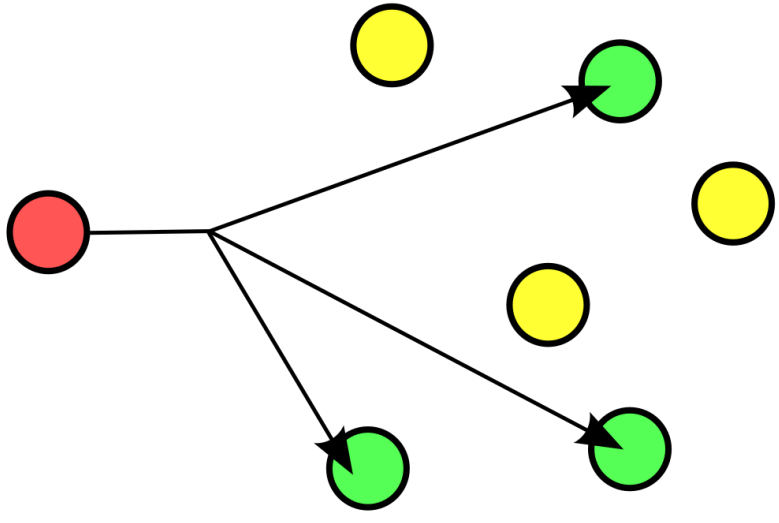
Kirk Bater
@KirkBater

Follow

This image is a TCP/IP Joke. This tweet is a UDP joke. I don't care if you get it.

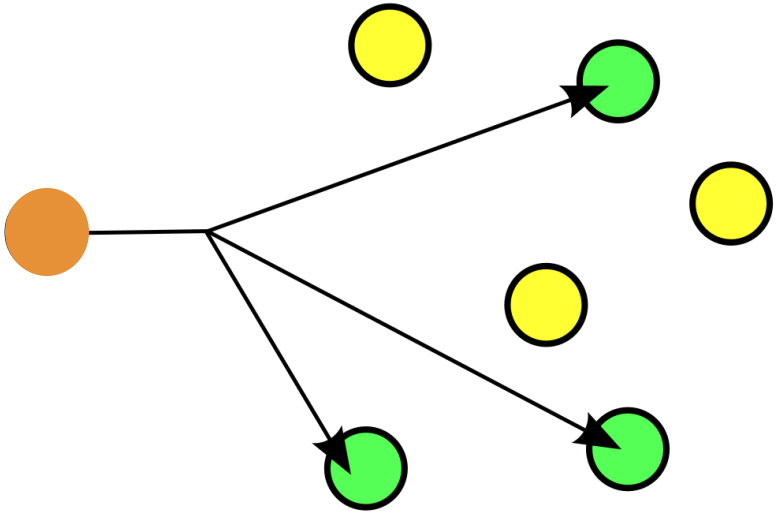


6:01 PM - 17 Jan 2018



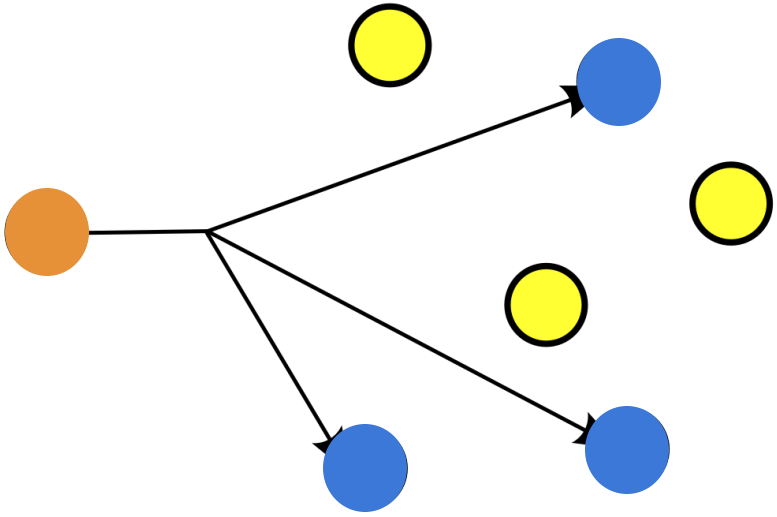
subject

```
let subject$ = Rx.Subject();
```



subject

```
let subject$ = Rx.Subject();
```

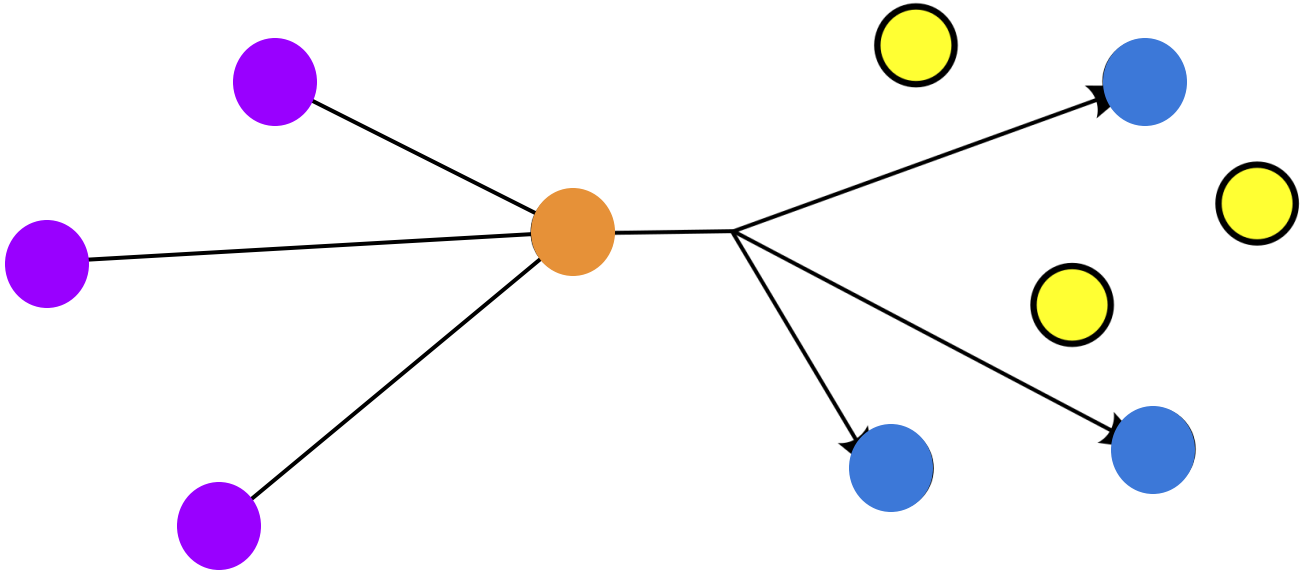


consumers

```
subject$.subscribe(consumerA);
subject$.subscribe(consumerB);
subject$.subscribe(consumerC);
```

subject

```
let subject$ = Rx.Subject();
```



sources

```
sourceA$.subscribe(subject$);
sourceB$.subscribe(subject$);
sourceC$.subscribe(subject$);
```

consumers

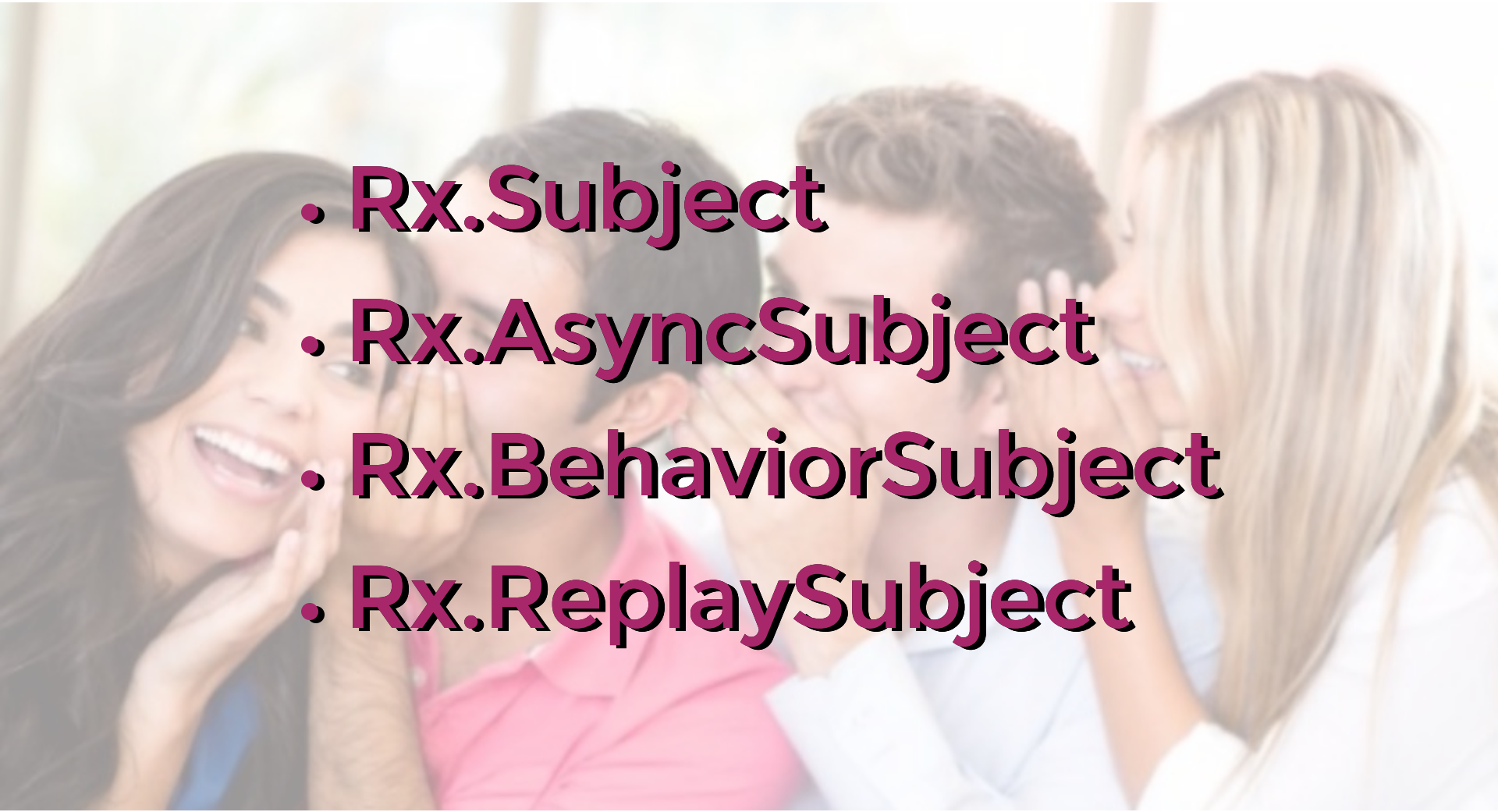
```
subject$.subscribe(consumerA);
subject$.subscribe(consumerB);
subject$.subscribe(consumerC);
```

Subjects

proxy, reemitter

Subjects

proxy, reemitter



- Rx.Subject
- Rx.AsyncSubject
- Rx.BehaviorSubject
- Rx.ReplaySubject

Subjects

subject = an observable AND observer at the same time (a proxy)

```
next([value])
error([error message])
complete()
subscribe()
unsubscribe()
```

Subjects

all subject types can:

```
// manually emit an item
subject$.next(item);

// re-emit items from original source
original$.subscribe(subject$);

// be subscribed by a consumer
subject$.subscribe(consumer);
```

subject types differ in *what their consumers get* , when subscribed

Rx.Subject

just a proxy

- both an observable and an observer (inherits from both)
- reemits, reerrors - at exact time of original item
- multiple input items -> multiple output items (as long as subscribed)

Rx.Subject

```
var subject$ = new Rx.Subject();

var i = 0;
var handle = setInterval(() => {
  console.log(i);
  subject$.next(i);
  if (++i > 5) {
    subject$.complete();
    clearInterval(handle);
  }
}, 500);

subject$.subscribe(new Subscriber('A'))
setTimeout(() => {
  subject$.subscribe(new Subscriber('B'))
}, 1500);
```

jsfiddle demo

Rx.Subject

```
var subject$ = new Rx.Subject();

var i = 0;
var handle = setInterval(() => {
  console.log(i);
  subject$.next(i);
  if (++i > 5) {
    subject$.complete();
    clearInterval(handle);
  }
}, 500);

subject$.subscribe(new Subscriber('A'))
setTimeout(() => {
  subject$.subscribe(new Subscriber('B'));
}, 1500);
```

```
0
A value: 0
1
A value: 1
2
A value: 2
3
A value: 3
B value: 3
4
A value: 4
B value: 4
5
A value: 5
B value: 5

A completed
B completed
```

jsfiddle demo

Rx.Subject

```
let sourceA$ = Rx.Observable.interval(500)
  .map(e => 'A'+e);
let sourceB$ = Rx.Observable.interval(1000)
  .map(e => 'B'+e);

const proxy = new Rx.Subject();
// re-emit someone else's items
sourceA$.subscribe(proxy);
sourceB$.subscribe(proxy);

// other parties can subscribe to a proxy
proxy.subscribe(console.log);

// manually emit
proxy.next(3);
```

multiple sources

Rx.Subject

```
let sourceA$ = Rx.Observable.interval(500)
  .take(10)
  .map(e => 'A'+e);
let sourceB$ = Rx.Observable.interval(1000)
  .map(e => 'B'+e);

const proxy = new Rx.Subject();
// re-emit someone else's items
sourceA$.subscribe(proxy);
sourceB$.subscribe(proxy);

// other parties can subscribe to a proxy
proxy.subscribe(console.log);

// manually emit
proxy.next(3);
```

source completion completes subject

Rx.AsyncSubject

the last item before shutdown

promise-ish

- the last item emitted before onCompleted
- nothing emitted if not completed yet
- error emitted (onError called on subscribers) if error happened
- multiple input items -> at most 1 output item
- the last item available for future subscribers

Rx.AsyncSubject

```
var subject$ = new Rx.AsyncSubject();

var i = 0;
var handle = setInterval(() => {
  subject$.next(i);
  if (++i > 3) {
    subject$.complete();
    clearInterval(handle);
  }
}, 500);

subject$.subscribe(new Subscriber('A'))
setTimeout(() => {
  subject$.subscribe(new Subscriber('B'))
}, 3000);
```

jsfiddle demo

Rx.AsyncSubject

```
var subject$ = new Rx.AsyncSubject();

var i = 0;
var handle = setInterval(() => {
  subject$.next(i);
  if (++i > 3) {
    subject$.complete();
    clearInterval(handle);
  }
}, 500);

subject$.subscribe(new Subscriber('A'))
setTimeout(() => {
  subject$.subscribe(new Subscriber('B'))
}, 3000);
```

```
A value: 3
A completed
B value: 3
B completed
```

jsfiddle demo

Rx.BehaviorSubject

the most up-to-date item

- represents a single *thing*, for each new subscriber - this value is emitted
- if the *thing* gets a new value
  - it's emitted to all existing subscribers
  - future subscribers don't get historical data
- the *thing* started with initial value
- no- *thing* after completed

Rx.BehaviorSubject

```
var subject$ = new Rx.BehaviorSubject(42);
// initial value, but no subscribers

subject$.next(43);

subject$.subscribe(new Subscriber('A'));

subject$.next(44);
subject$.next(45);
subject$.subscribe(new Subscriber('B'));

subject$.next(56);

subject$.complete();
subject$.subscribe(new Subscriber('C'));
```

jsfiddle demo

Rx.BehaviorSubject

```
var subject$ = new Rx.BehaviorSubject(42);
// initial value, but no subscribers

subject$.next(43);

subject$.subscribe(new Subscriber('A'));

subject$.next(44);
subject$.next(45);
subject$.subscribe(new Subscriber('B'));

subject$.next(56);

subject$.complete();
subject$.subscribe(new Subscriber('C'));
```

jsfiddle demo

```
A value: 43

A value: 44

A value: 45
B value: 45

A value: 56
B value: 56

A completed
B completed
C completed
```

Rx.ReplaySubject

replay the whole... buffer

- replays the whole sequence
- also for future subscribers
- sequence size might be limited to buffer size and/or time length

Rx.ReplaySubject

```
// buffer size
var subject$ = new Rx.ReplaySubject(2);

subject$.next(43);

subject$.subscribe(new Subscriber('A'));

subject$.next(44);
subject$.next(45);
subject$.subscribe(new Subscriber('B'));

subject$.next(56);

subject$.complete();
subject$.subscribe(new Subscriber('C'));
```

jsfiddle demo

Rx.ReplaySubject

```
// buffer size
var subject$ = new Rx.ReplaySubject(2);

subject$.next(43);

subject$.subscribe(new Subscriber('A'));

subject$.next(44);
subject$.next(45);
subject$.subscribe(new Subscriber('B'));

subject$.next(56);

subject$.complete();
subject$.subscribe(new Subscriber('C'));
```

```
A value: 43
A value: 44
A value: 45

B value: 44
B value: 45

A value: 56
B value: 56

A completed
B completed

C value: 45
C value: 56
C completed
```

jsfiddle demo

guest starring

```
class EventEmitter<T> extends Subject<T> {
    constructor(isAsync?: boolean);
    emit(value?: T): void;
    subscribe(next?: any, error?: any, complete?: any): any
}
```

guest starring

```
@Component({
  selector: 'page-size',
  template: `elements per page:
  <button *ngFor="let size of sizes" (click)="setSize(size)">
    {{size}}
  </button>`
})
export class PageSizeComponent {
  private sizes = [5, 10, 25, 50, 100];
  @Output() sizeChanged = new EventEmitter();

  setSize(newSize){
    this.sizeChanged.emit(newSize);
  }
}
```

guest starring

```
@Component({
  selector: 'page-size',
  template: `elements per page:
  <button *ngFor="let size of sizes" (click)="setSize(size)">
    {{size}}
  </button>`
})
export class PageSizeComponent {
  private sizes = [5, 10, 25, 50, 100];
  @Output() sizeChanged = new EventEmitter();

  setSize(newSize){
    this.sizeChanged.emit(newSize);
  }
}
```

```
// parent component view:
<page-size (sizeChanged)="handleSizeChanged($event)"></page-si
```

schedulers

schedulers

centralized dispatchers to control concurrency

- Immediate
- Timeout
- RequestAnimationFrame

errors & debugging

error handling

onError

.retry

~~.catch~~ / .catchError

~~.finally~~ / .finalize

debugging

debugging

- no debugger support yet

debugging

- no debugger support yet
- `obs$.do(x => console.log(x)) / tap`

  `obs$.do(console.log) / tap`

debugging

- no debugger support yet
- `obs$.do(x => console.log(x)) / tap`

  `obs$.do(console.log) / tap`
- `obs$.materialize, obs$.dematerialize`

debugging

- no debugger support yet
- `obs$.do(x => console.log(x)) / tap`

  `obs$.do(console.log) / tap`
- `obs$.materialize, obs$.dematerialize`
- marble diagrams

jsfiddle interval demo

//jsfiddle.net/0n33fru7/embedded/result/

stream serialization

materialize

dematerialize

- serialize for later usage
  - backpressure
  - later replay
- debugging, error finding

antipatterns

subscribe inside subscribe

```
function loadAJAX(url){
  // streams of items in AJAX list
  return Observable...
}

var subscription = click$.subscribe(
  e => loadAJAX('horses.json').subscribe(...)
  console.error,
  _ => console.info('completed')
);
```

there are better ways to do that

(flatMap in above case)

subscribe inside subscribe

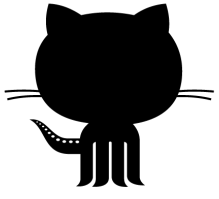
```
function loadAJAX(url){
  // streams of items in AJAX list
  return Observable...
}

var subscription = click$.subscribe(
  e => loadAJAX('horses.json').subscribe(...)
  console.error,
  _ => console.info('completed')
);
```

there are better ways to do that

(flatMap in above case)

exercise setup



https://github.com/ducin-public/async-currency-exchange

hands-on exercise

- `npm install mock-rest-api`
- **run the api:**

  `$ mock-rest-api`
- **git clone following repo:**

`https://github.com/ducin-public/asynchronous-javascript-training`