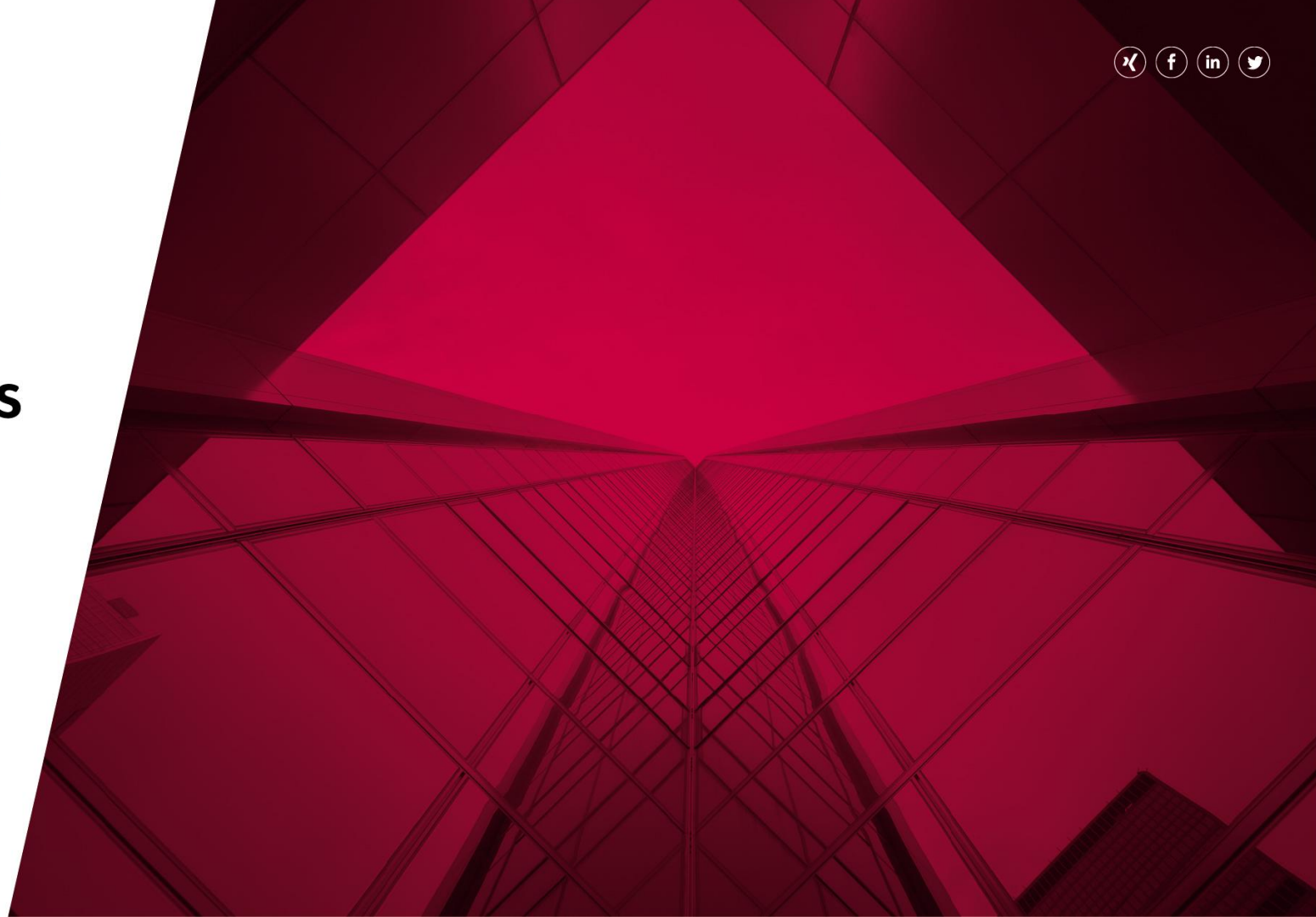


ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Agenda

- Dependency Injection - Services
- Intro RxJS - Observables
- Subscription handling / Error Handling
  - Async pipe
  - Unsubscribing
- Rxjs Operators
  - Nested Pipes
  - Handling concurrent API calls

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Services

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Services

- “Local Singletons”
- Data-Model-Layer of our application
- May be injected via Dependency Injection (DI)
- Two roles:
  - Provide methods or streams of data to subscribe to
  - Provide operations to modify data

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Services - Example

<code>

Services are the Data-Model-Layer of our application

```
@Injectable({
  providedIn: 'root',
})
export class BookDataService {
  private books = [{...}, {...}, {...}];

  getBooks() {
    return this.books;
  }
}
```

Services - Example

<code>

With `providedIn:'root'` the service is registered globally

```
@Injectable({
  providedIn: 'root',
})
export class BookDataService {
  private books = [{...}, {...}, {...}];

  getBooks() {
    return this.books;
  }
}
```

Services - Example

<code>

They define an API to interact with them

```
@Injectable({
  providedIn: 'root',
})
export class BookDataService {
  private books = [{...}, {...}, {...}];

  getBooks() {
    return this.books;
  }
}
```

Services

<code>

Create a service explicit for a module with the providers array

```
@NgModule({
  providers: [
    BookDataService
  ]
})

@Component({ … })
export BookListComponent {
  constructor(private bookData: BookDataService) {}
}
```

ARCHITECTS

INSIDE KNOWLEDGE

Services

<code>

Create a service instance for a component and its children

```
@Component({
  // ...
  providers: [BookDataService]
})
export class BookListComponent {
  constructor(private bookData: BookDataService) {}
}
```

Services

<code>

Create a service instance for a component and its children

```
@Component({
  // ...
})
export class BookListComponent {
  constructor(private bookData: BookDataService) {}
}
```

ARCHITECTS

INSIDE KNOWLEDGE

Dependency Injection

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Dependency Injection - Why

- Keep component classes clean
- Better testable code
- Easy replacement of services

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Without Dependency Injection

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Dependency Injection

You have to create instances on your own.

```
@Component({ … })
class BookListComponent {
  private bookDataService;
  constructor() {
    this.bookDataService = new BookDataService();
  }
}
```

With Dependency Injection

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Dependency Injection

Dependency Injection is also called **Inversion of control**.

The injector has control over service instantiation.

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Dependency Injection

<code>

Injector is responsible for creating instances.

```
@NgModule({
  providers: [BookDataService],
})
export class AppModule { }


@Component({})
class BookListComponent {
  constructor(private bookDataService: BookDataService) {}
}
```

ARCHITECTS

INSIDE KNOWLEDGE

Dependency Injection

<code>

Injector is responsible for creating instances.

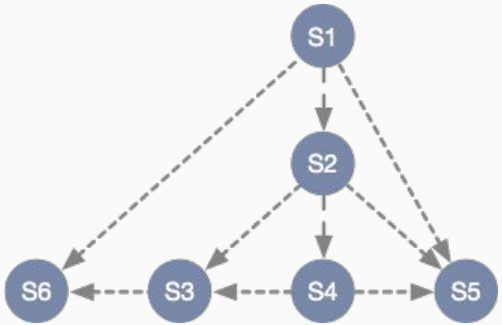
```
@Component({
  providers: [BookDataService]
})
class BookListComponent {
  constructor(private bookDataService: BookDataService) {}
}
```

ARCHITECTS

INSIDE KNOWLEDGE

Dependency Injection

- Services can have dependencies, too
- Injects service instances created in a component, where service is used!
- Watch out for dependency cycles!




```
service 'S1', (S2, S5, S6)
service 'S2', (S3, S4, S5)
service 'S3', (S6)
service 'S4', (S3, S5)
service 'S5', ()
service 'S6', ()
```

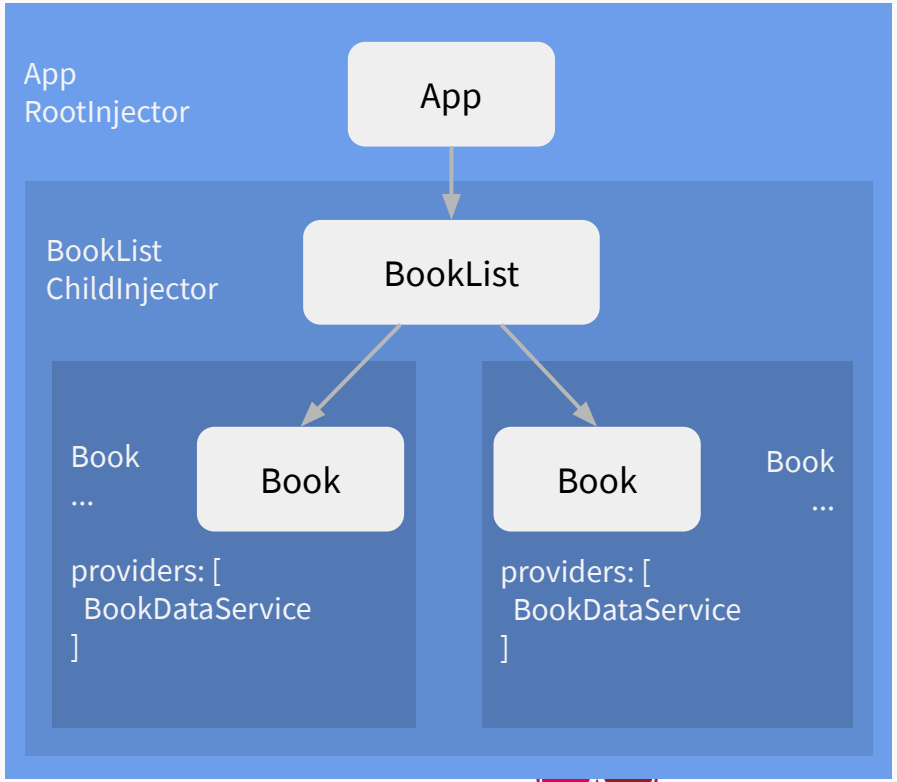
Dependency Injection

- Based on the type of a class
- An instance is available for all child components, too

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Dependency Injection

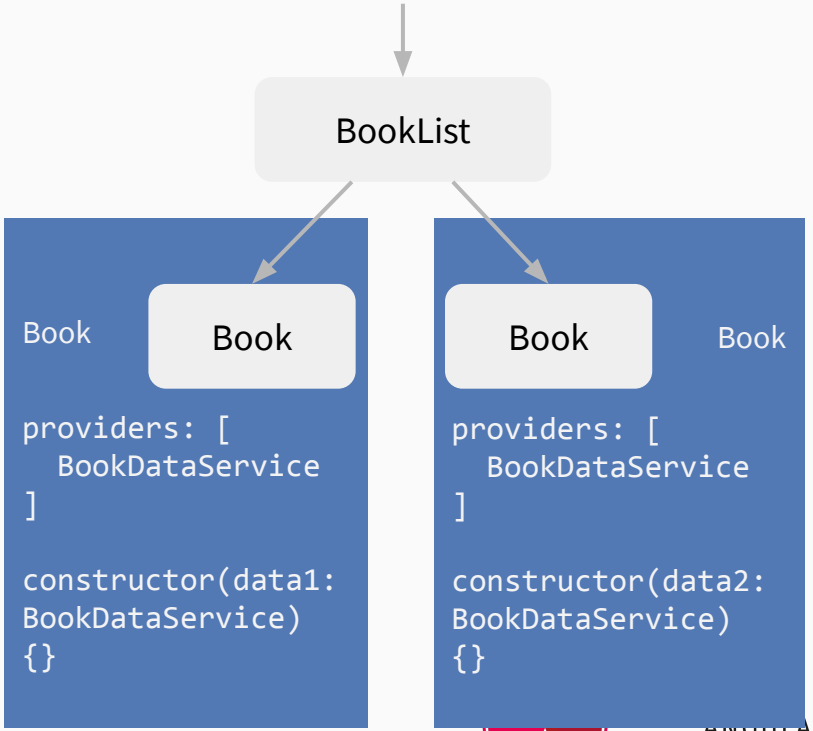
- **Injector** per component
- Each component has an own injector
- Base injector = **RootInjector**
- Each nested component has a **ChildInjector**



ARCHITECTS
INSIDE KNOWLEDGE

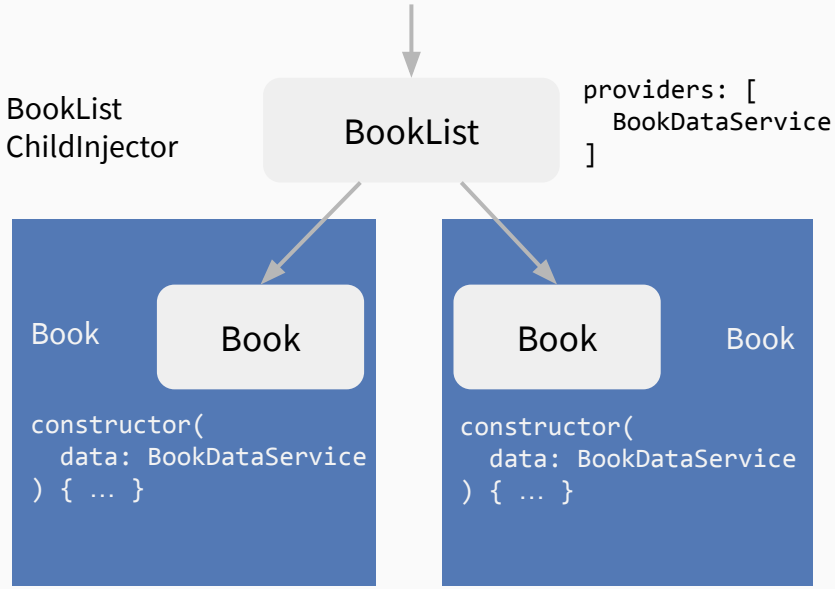
Dependency Injection

- New service instance for each BookComponent



Dependency Injection

- Share one service instance
- Create instance in parent component BookList
- Only inject service in BookComponent → no providers!



ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

Dependency Injection - @Injectable()

- Annotation of classes that use DI
- Metadata to compile the type-information to the ES5 code
- Without an annotation you lose the type information

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Task 1

Create a BookData service

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Workshop
Angular RxJS

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

RxJS Versions

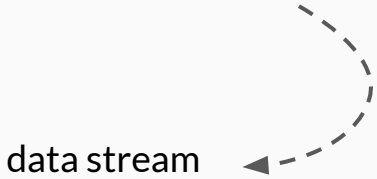
Stable 🏭

7.0.0

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Observable & Promise

	Single	Multiple
Pull	Function	Iterator
Push	Promise	Observable



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

RxJS Versions

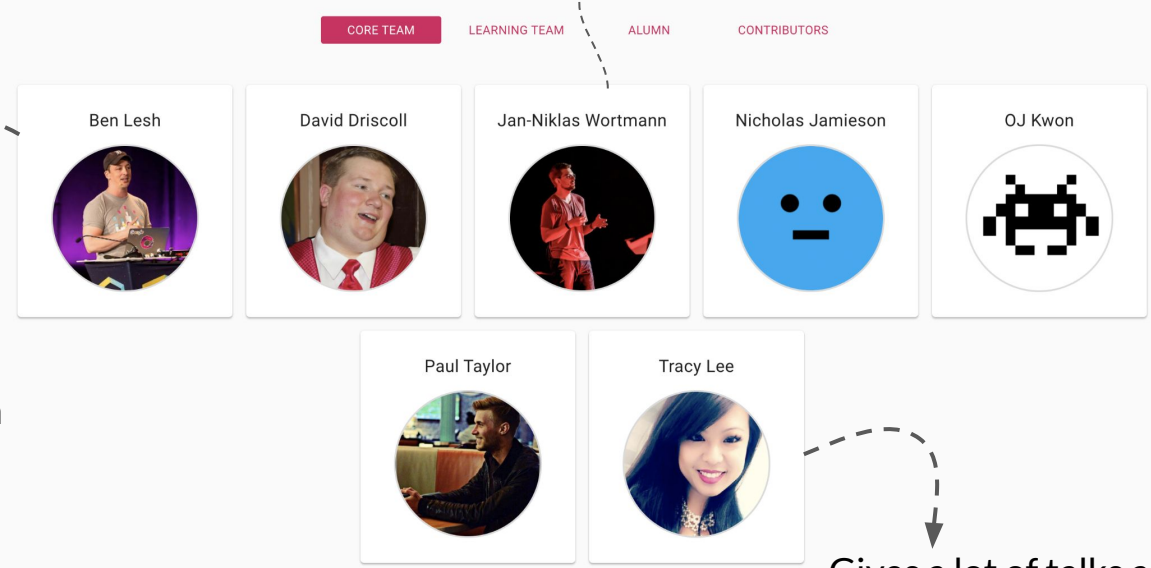
Stable 🏭

7.0.0

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

RxJS Core team

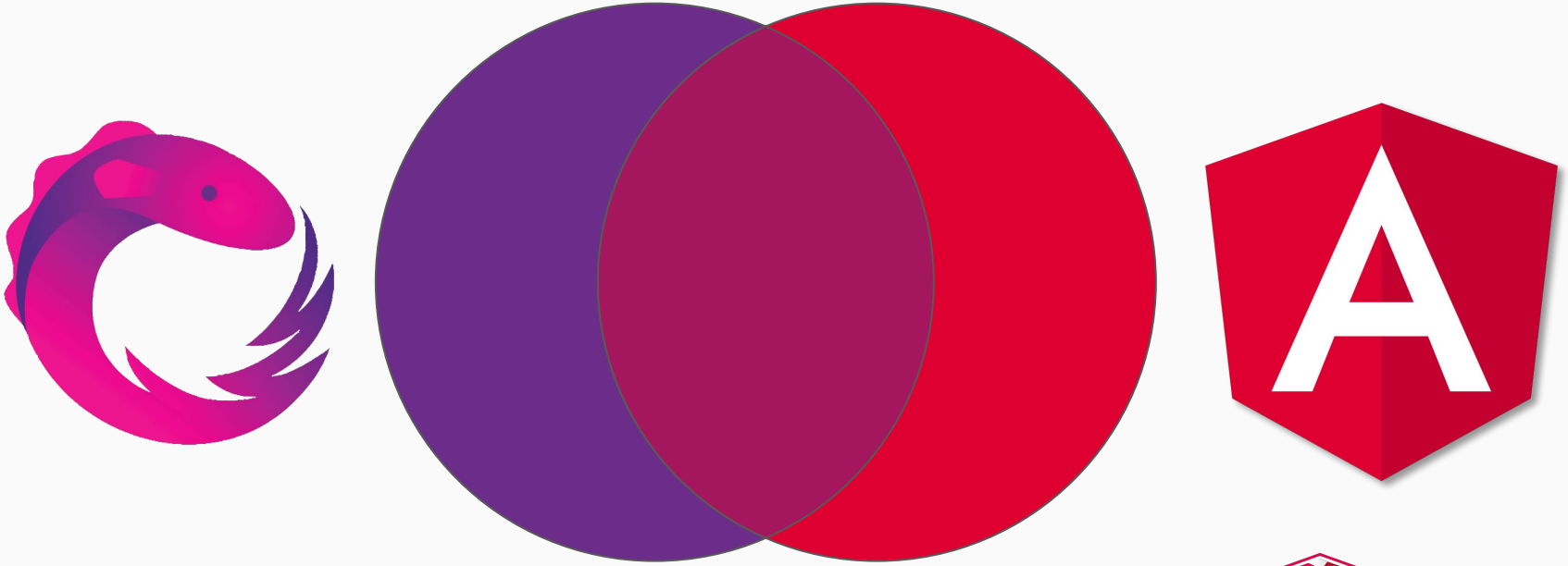
Core Team Member from Germany 🤗



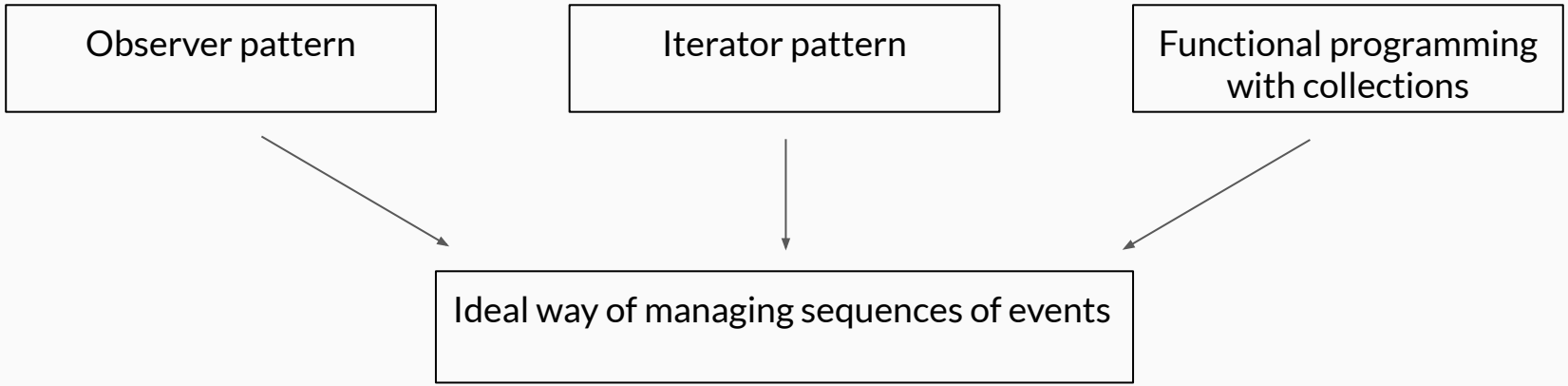
Team lead Did the whole rewrite from JS to TS.

Gives a lot of talks about RxJS, runs Workshops & spreads the word about RxJS all over the world.

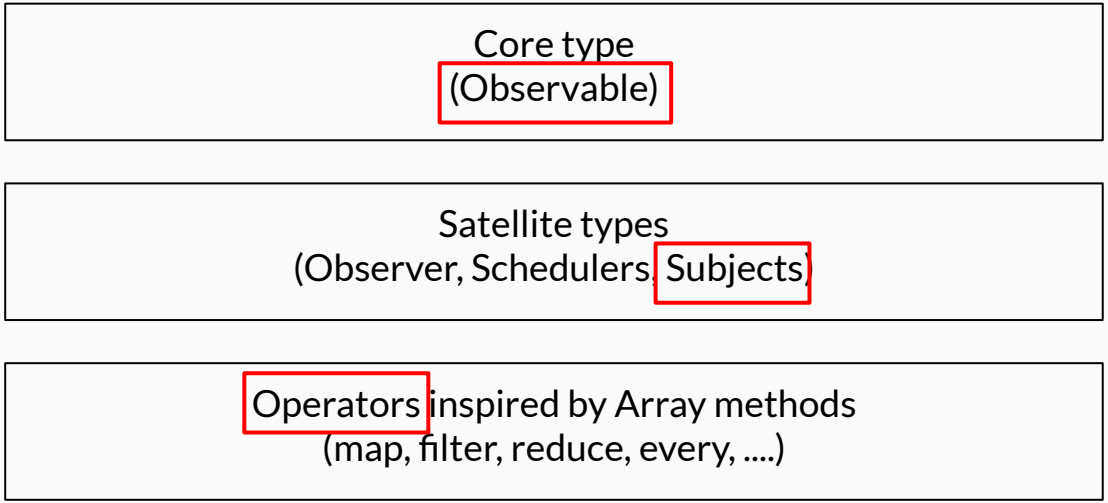
Intersection with Angular community



Patterns & paradigms in RxJS



Content of RxJS

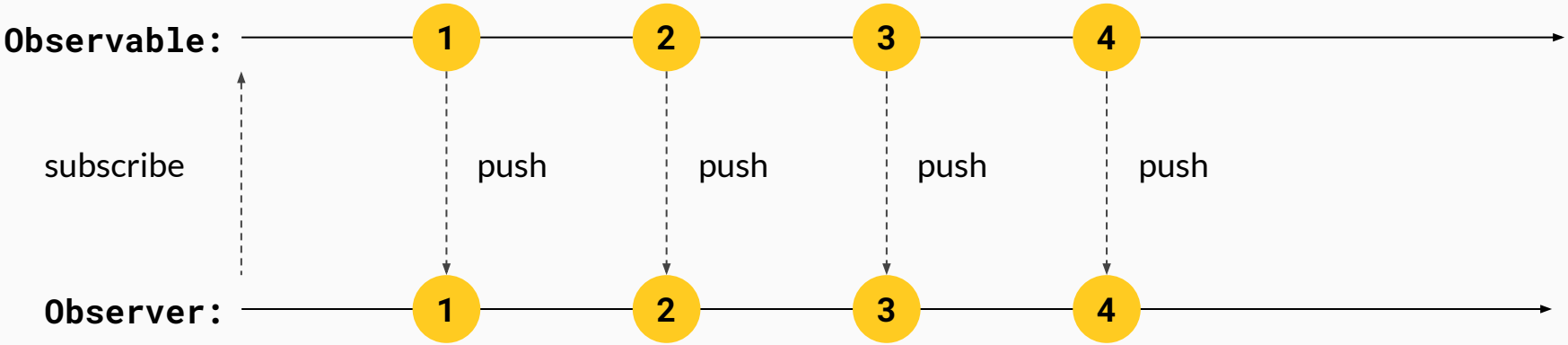


ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

What is an observable?

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Observable & observer



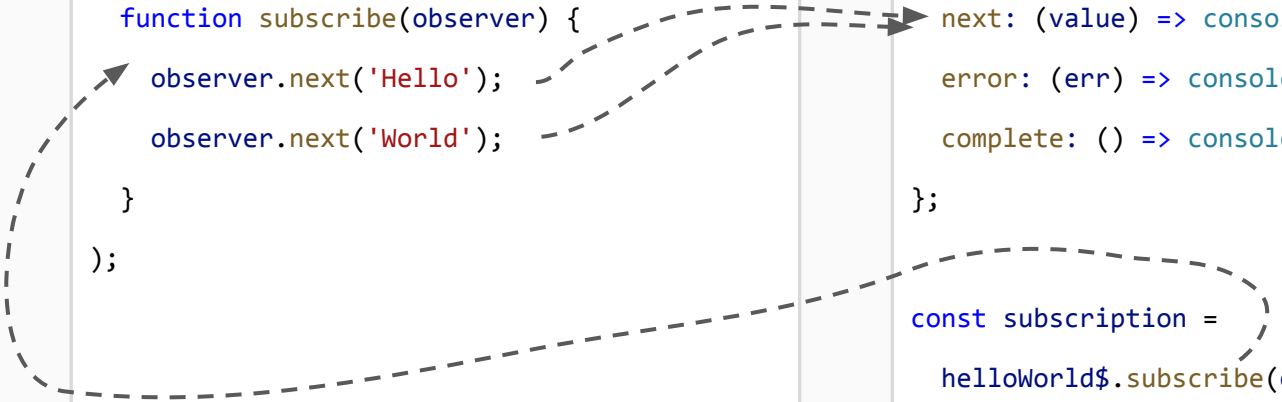
Create & subscribe to observable

hello-world.ts

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

const helloWorld$ = new Observable(
  function subscribe(observer) {
    observer.next('Hello');
    observer.next('World');
  }
);
```

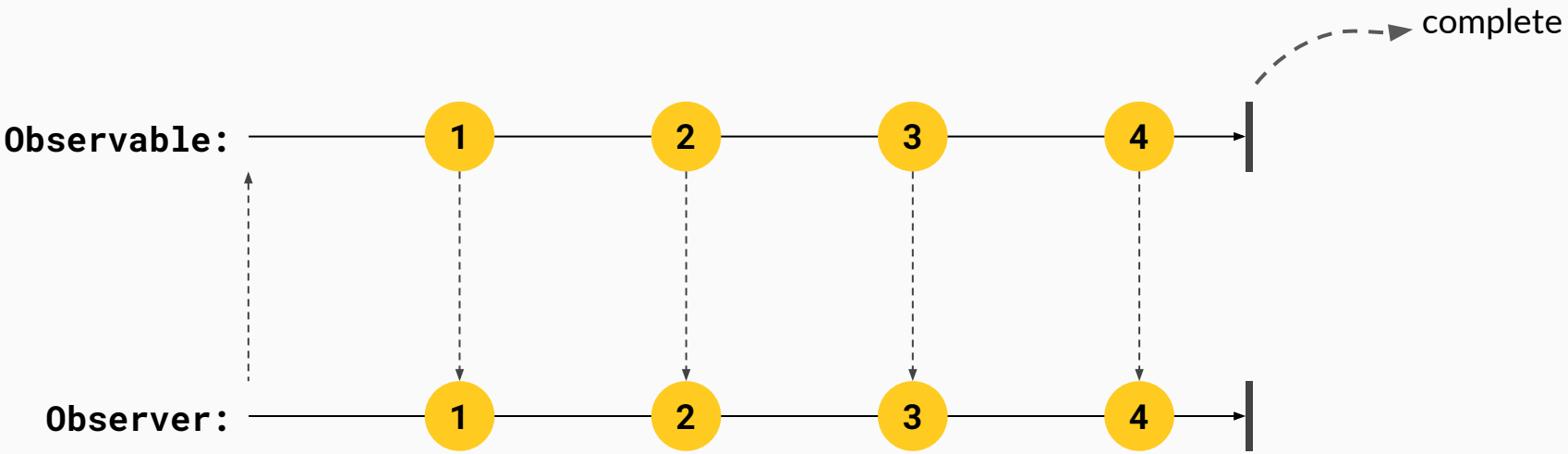
hello-world.ts

```ts
// ...

const observer = {
  next: (value) => console.log(value),
  error: (err) => console.error(err),
  complete: () => console.log('done')
};

const subscription =
  helloWorld$.subscribe(observer);
```



ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

Complete



Complete

hello-world.ts

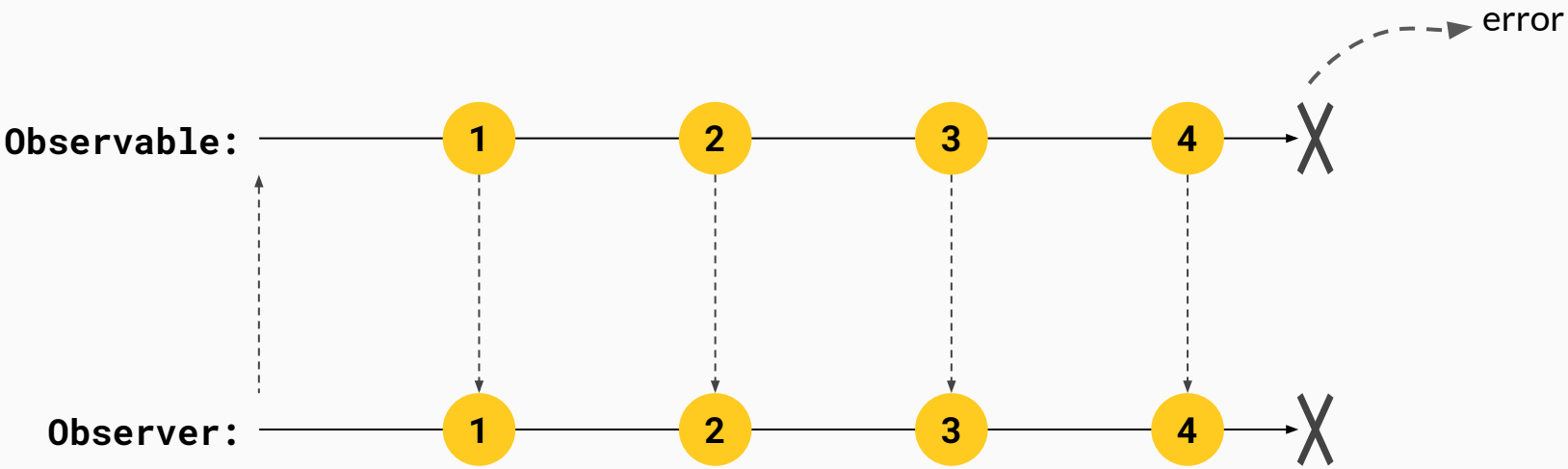
```ts
import { Observable } from 'rxjs';

const helloWorld$ = new Observable(
  function subscribe(observer) {
    observer.next('Hello');
    observer.next('World');
    observer.complete();
  }
);
```

hello-world.ts

```ts
// ...

const observer = {
  next: (value) => console.log(value),
  error: (err) => console.error(err),
  complete: () => console.log('done')
};

const subscription =
  helloWorld$.subscribe(observer);
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Error



Error

hello-world.ts

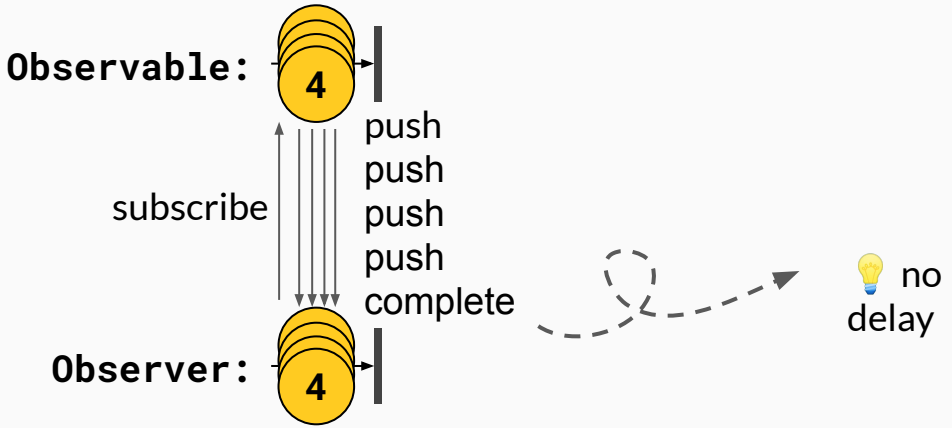
```ts
import { Observable } from 'rxjs';

const helloWorld$ = new Observable(
  function subscribe(observer) {
    observer.next('Hello');
    observer.next('World');
    observer.error(new Error('fail'));
  }
);
```

hello-world.ts

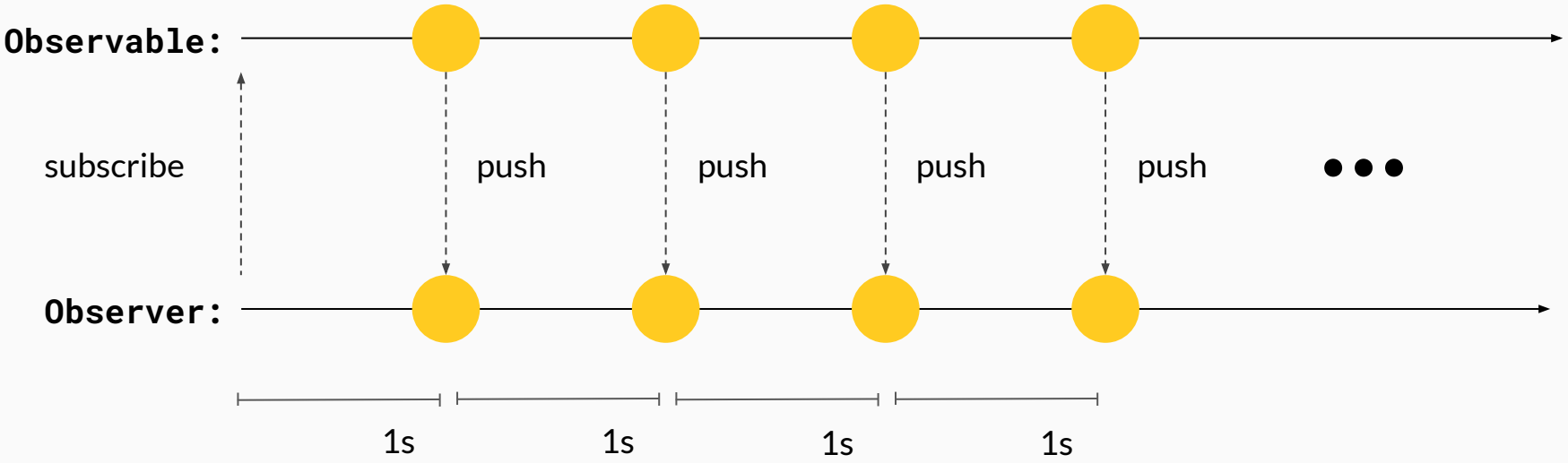
```ts
// ...

const observer = {
  next: (value) => console.log(value),
  error: (err) => console.error(err),
  complete: () => console.log('done')
};

const subscription$ =
  helloWorld.subscribe(observer);
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

What it looks like



Let’s add a delay



Task 2

Create an Observable

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Observables - cancellation

```
const subscription = observable.subscribe(...)

subscription.unsubscribe()
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Operator

- Operators are functions
- allow complex asynchronous code to be easily composed in a declarative manner.

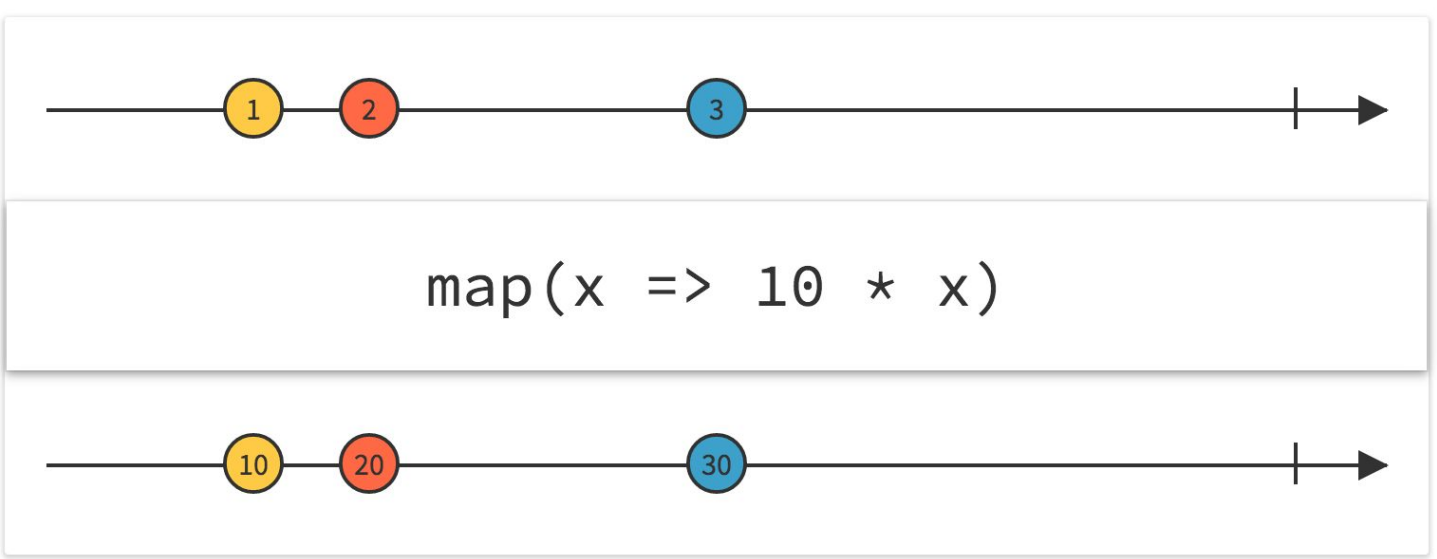
```
observableInstance.pipe(operator()).
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

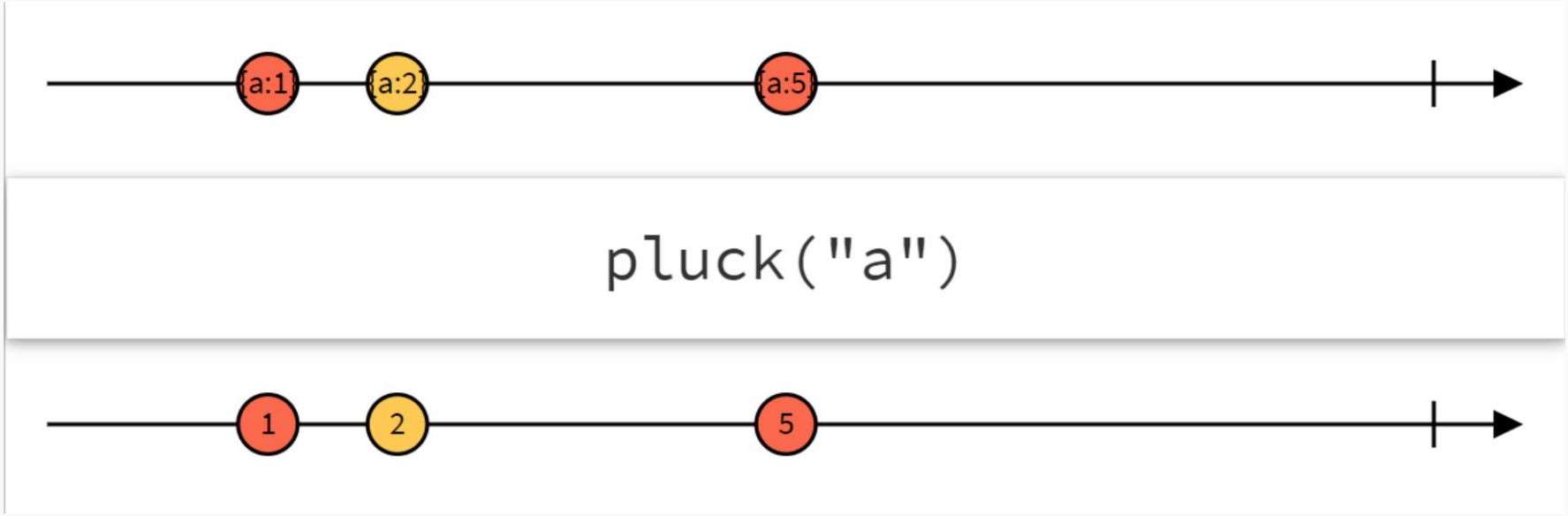
Transformation Operators

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

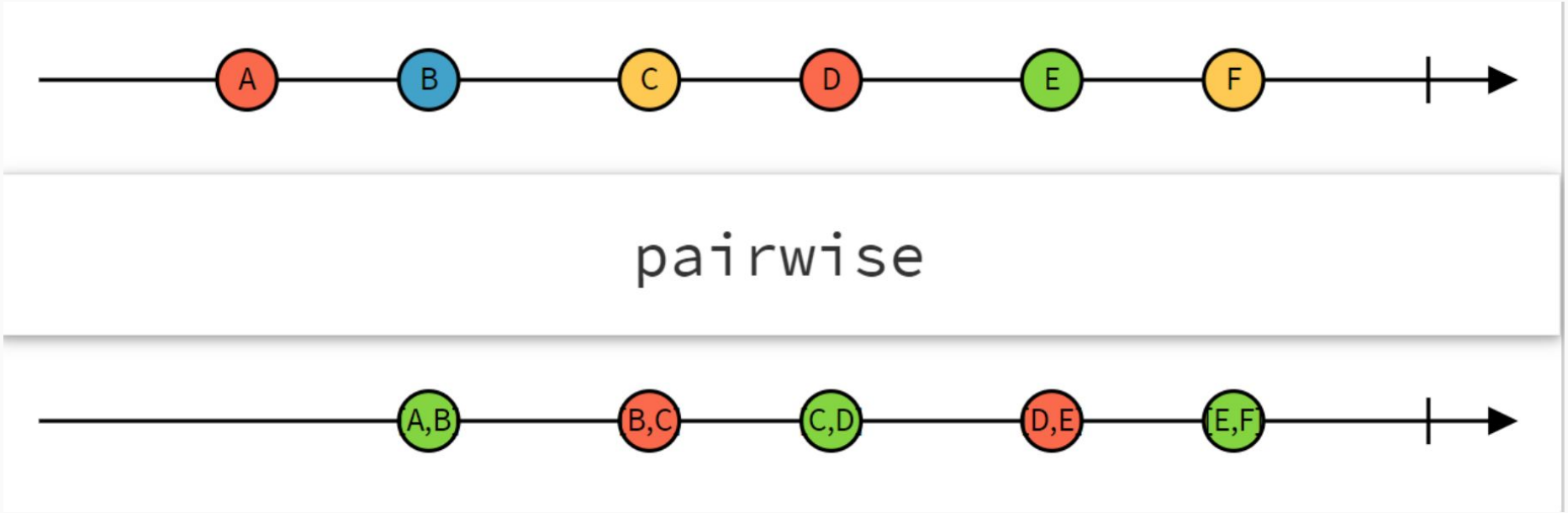
.map()



.pluck()



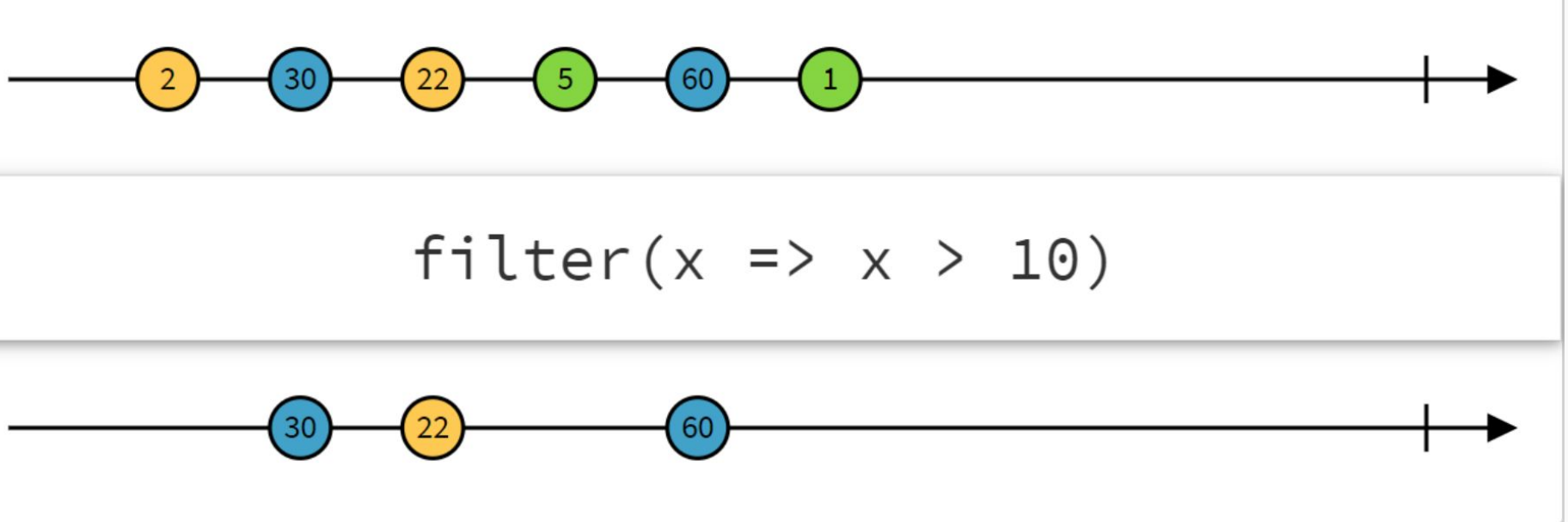
.pairwise()



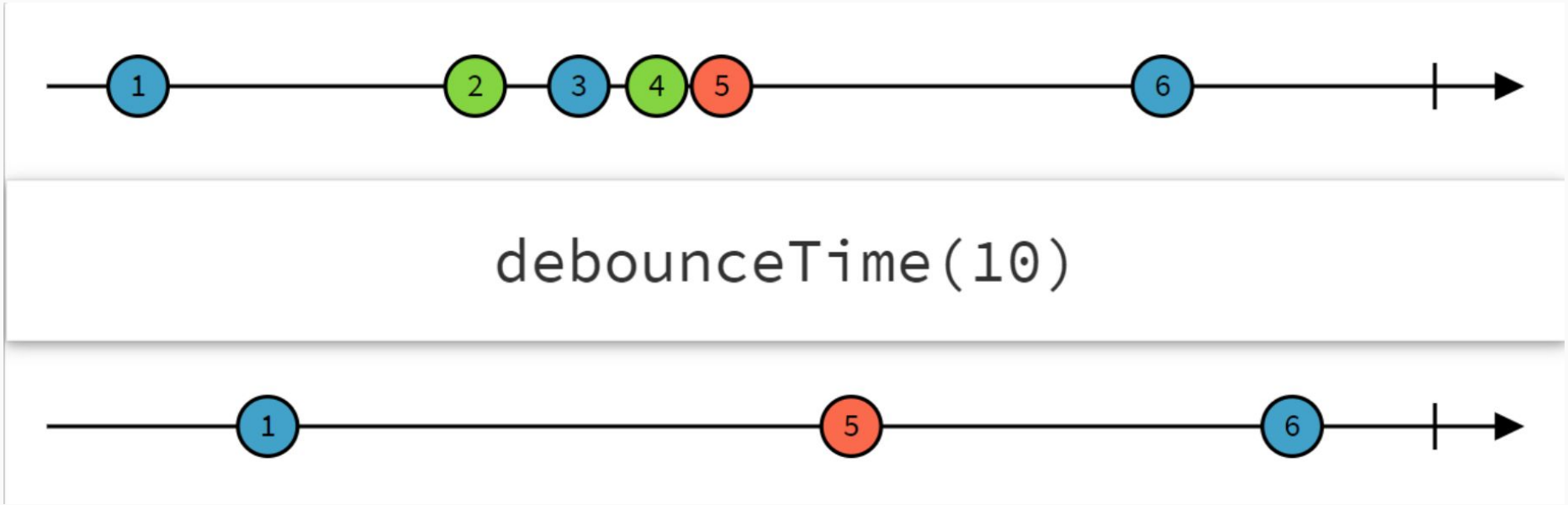
Filtering Operators

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

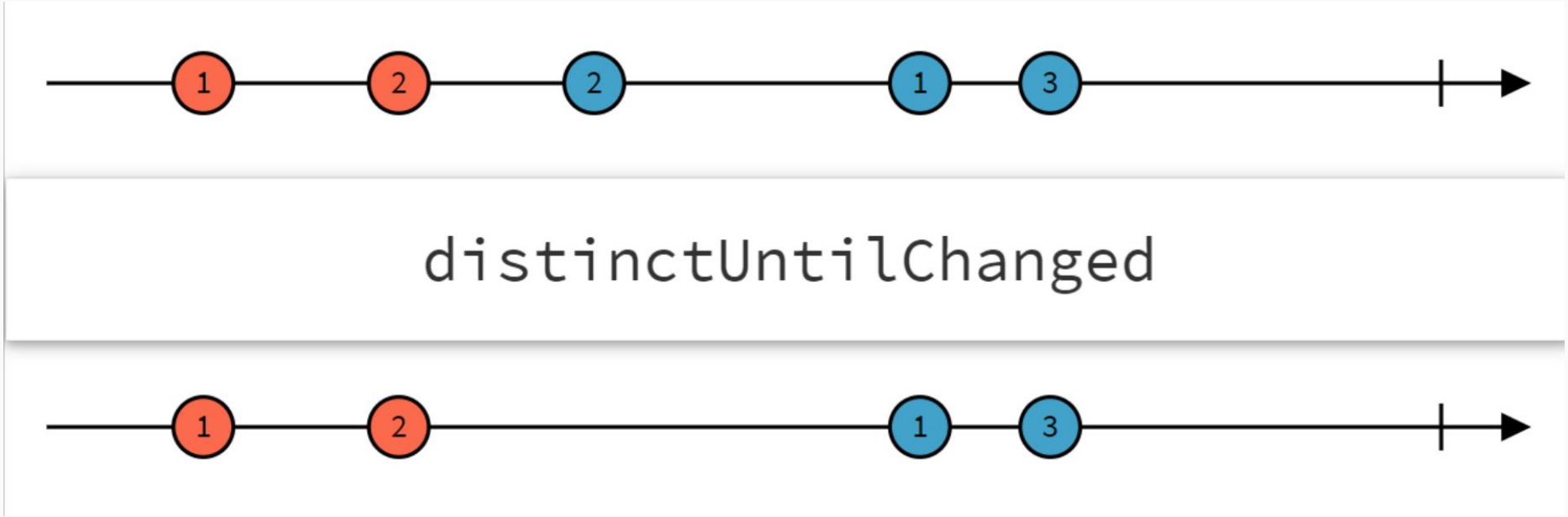
.filter()



.debounceTime(n: milliseconds)



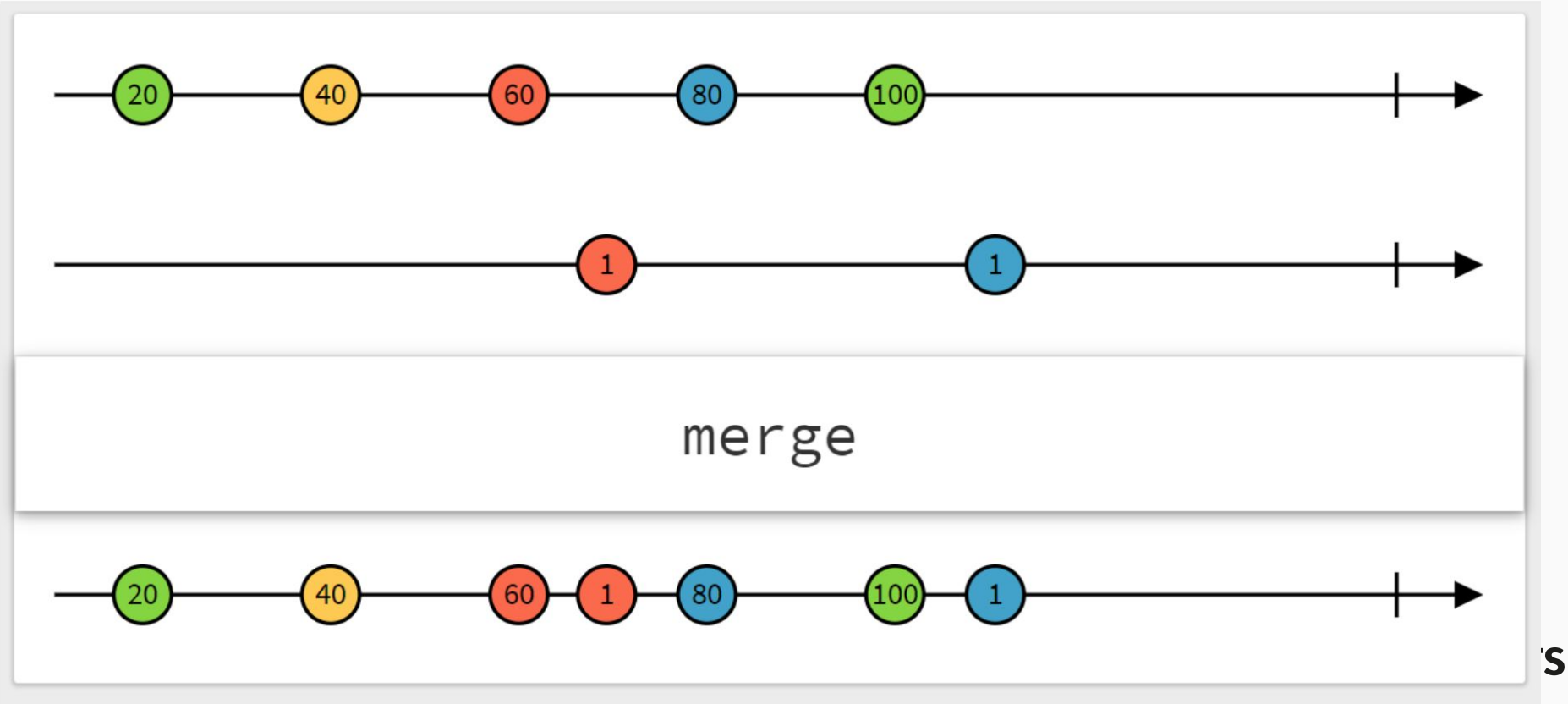
distinctUntilChanged()



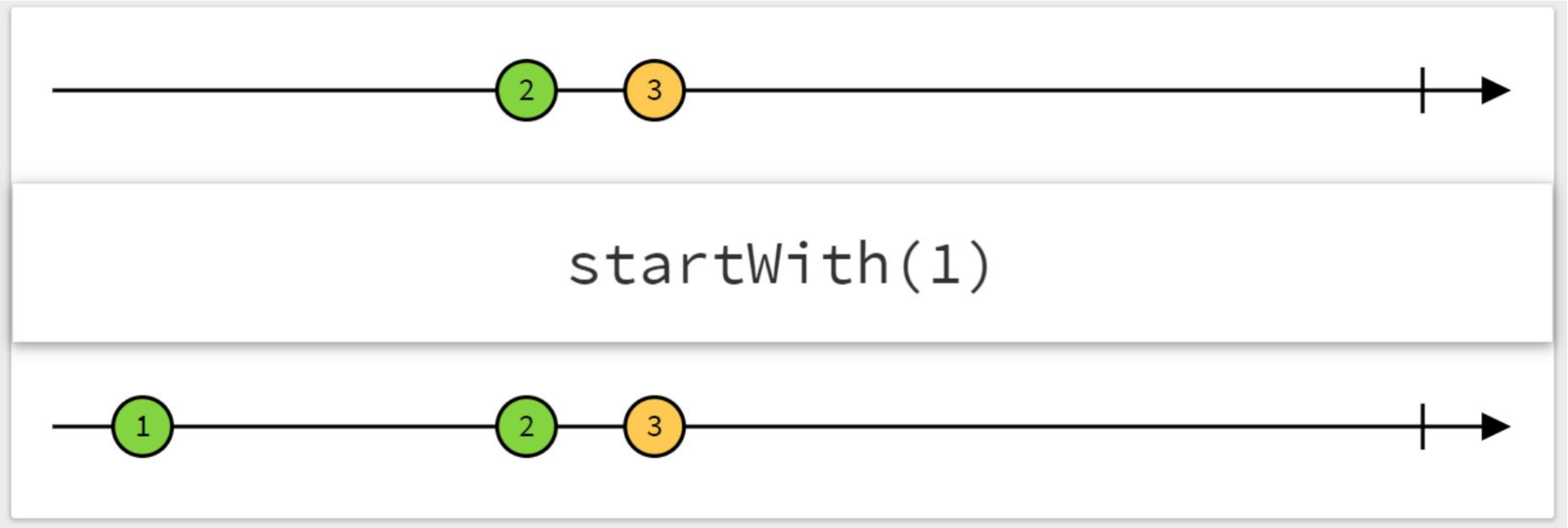
Combination Operators

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

merge()



startWith()



Error Handling

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Operators for Error Handling

- catchError
- Retry
- retryWhen
- throwError

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Observable Creation Operators

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Observables creation helpers

- of(value1, value2, …)
- from(promise/iterable/observable)
- fromEvent(item, eventName)
- Angular HttpClient
- Many more

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

You are usually not creating

your own observables!

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Observables - subscribing

Without subscribing an observable will not be fired

```
.subscribe(nextFn, errorFn, completeFn)
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Unsubscribe!!!

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Need to unsubscribe!!!

<code>

```
BookComponent implements OnInit, OnDestroy {
Subscription: Subscription;
    ngOnInit() {
         subscription = this.bookData
          .getBooks()
          .subscribe(books => this.books = books);
    }

    ngOnDestroy() {
        subscription.unsubscribe()
    }
}
```

ARCHITECTS
INSIDE KNOWLEDGE

HttpClient

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Using the HttpClient

- Basic HTTP handling
- `import {HttpClientModule} from '@angular/common/http'`
- `import {HttpClient} from '@angular/common/http'`
- Provides methods for
  - GET
  - PUT
  - POST
  - DELETE

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Http service

<code>

HttpClientModule has to be imported

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  ...
})
```

HttpClient Interface

Name	Parameter	Returnvalue
get	url, options?	Observable<TPayload>
post	url, body, options?	Observable<TPayload>
put	url, body, options?	Observable<TPayload>
delete	url, options?	Observable<TPayload>
patch	url, body, options?	Observable<TPayload>
head	url, options?	Observable<TPayload>
request	Request, options?	Observable<TPayload>

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

HttpClient usage

HttpClient functions return response observables

```
import { HttpClient } from '@angular/common/http';
...
  constructor(private http: HttpClient) {}

  getBooks() {
    return this.http.get<Book[]>(this.baseUrl)
  }
...
```

HttpClient

- Returns an observable
- Expects data in JSON format

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

HttpClient - Full response

<code>

Use observe: 'response' to get the full response

```
http
  .get<Book[]>('/books', {observe: 'response'})
  .subscribe(resp => {
    console.log(resp.headers.get('X-Custom-Header'));
    console.log(resp.body);
  });
```

HttpClient service

<code>

Subscribe to service observable in a component

```
constructor(private bookData: BookDataService) {
  this.bookData
      .getBooks()
      .subscribe(books => this.books = books);
}
```

ARCHITECTS

INSIDE KNOWLEDGE

Task 3

Load data from local API

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Async Pipe

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

The AsyncPipe accepts a Promise or Observable as input and subscribes to the input automatically, eventually returning the emitted values.

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Async Pipe

- Subscribe to Observable
- UnSubscribe on component destruction
- Built-In Pipe
- Simple use: {{ books$ | async }}

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Async Pipe

<code>

For every async a new subscription is made. Try to minimize use of async

```
<li *ngFor="let book of books$ | async">
  {{book.title}}
</li>

<span>{{ (books$ | async).length }}
```

Two subscriptions created.
Could cause performance issues

ARCHITECTS
INSIDE KNOWLEDGE

Async Pipe

<code>

Finnish Notation. Naming observables with an $ suffix

```
<li *ngFor="let book of books$ | async">
  {{book.title}}
</li>
```

Task 4

Use the async pipe

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Angular Subjects

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Helps to manage the state of your application

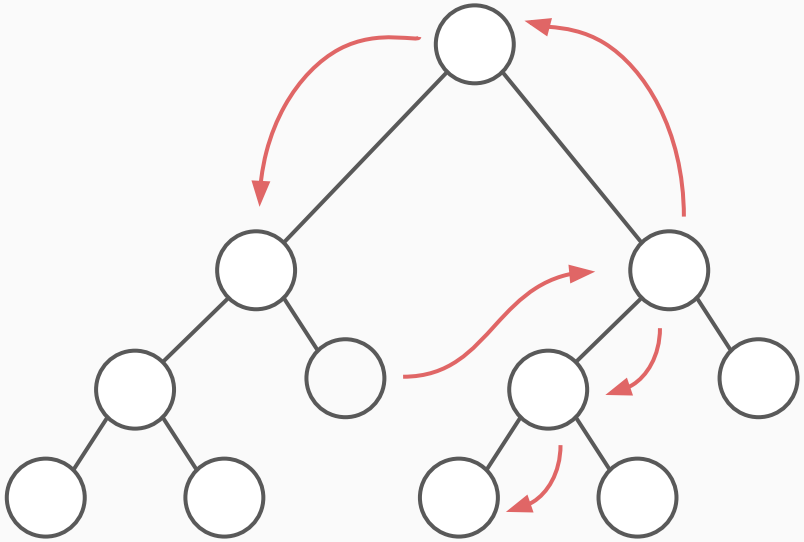
ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Why?

- Unidirectional data flow
- Predictable state changes and rendering
- Helping you application to be more “reactive”

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Why?



This is how we manage state at the moment:

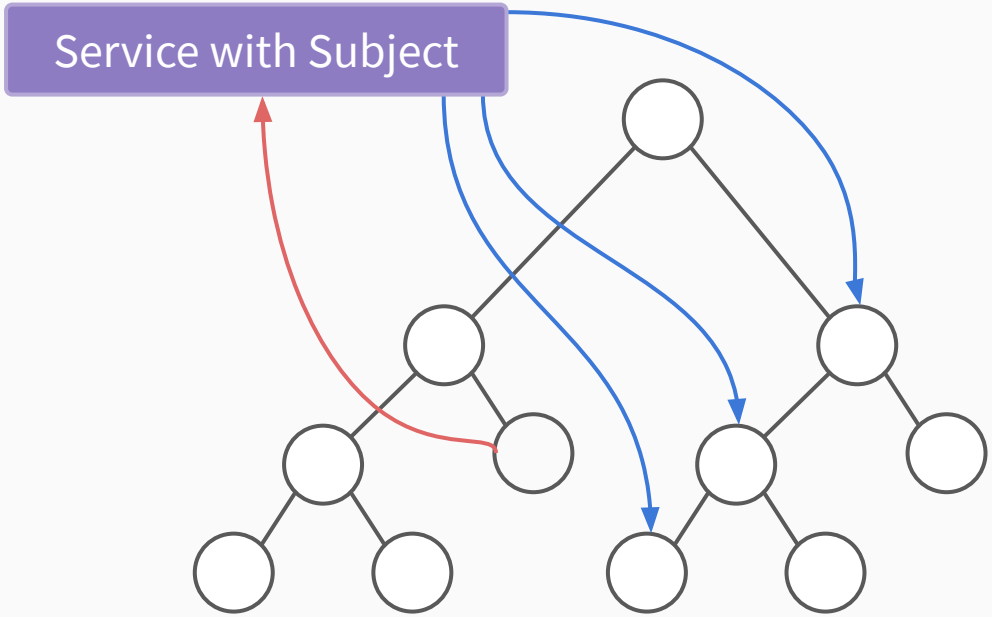
@Input()
@Output()

State management with Subjects

- Subjects are Observables but also Observers themselves
- Components can subscribe to Subjects
- Subjects **can emit data** too

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

State management with Subjects



Everything is dispatched from and to **one global store**

Creating a Subject

<code>

```
let subject = new Subject<string>();

// We subscribe to the subject
subject.subscribe((data) => {
 console.log(`Hello ${data}`)
});


subject.next('Angular');
// Hello Angular
```

ARCHITECTS
INSIDE KNOWLEDGE

Task 5

Create a HeaderService with a Subject

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Subjects are multicast

<code>

```
let subject = new Subject<string>();
subject.subscribe((data) => {
 console.log(`Subscriber 1 received ${data}`);
});

subject.subscribe((data) => {
 console.log(`Subscriber 2 received ${data}`);
});

subject.next('Hello Angular');

// Subscriber 1 received Hello Angular
// Subscriber 2 received Hello Angular
```

ARCHITECTS
INSIDE KNOWLEDGE

Don’t expose Subjects directly !!!

- Subscribers will be able to “mess up” with your Subjects
- Return an Observable:

```
private subject = new Subject<string>();

observable$ = this.subject.asObservable();
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Task 6

Change Headertitle on BookListItem-Click

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Using Subjects to unsubscribe

- We need to unsubscribe of all Subscriptions (otherwise we might geht memory leaks)
- But that can get really messy:

```
subsription1 = observable1$.subscribe((data) => {});
subsription2 = observable2$.subscribe((data) => {});
subsription3 = observable3$.subscribe((data) => {});
subsription4 = observable4$.subscribe((data) => {});

//ngOnDestroy:
subsription1.unsubscribe()
subsription2.unsubscribe()
subsription3.unsubscribe()
subsription4.unsubscribe()
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Using Subjects to unsubscribe

<code>

```
let destroy$ = new Subject<boolean>();

this.apiService.getObservable().pipe(
     takeUntil(this.destroy$)
)
.subscribe((data) => {
  ...
});


ngOnDestroy() {
     this.destroy$.next(true)
}
```

ARCHITECTS

INSIDE KNOWLEDGE

Other Subjects?

- A simple subject is not keeping the state
- Subscribers of subjects after value was emitted are not getting it

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

BehaviourSubject

- BehaviourSubject always stores the last emitted Value
- It needs a default Value to

```
private behaviourSubject = new BehaviourSubject<string>('default');
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

ReplaySubject

- ReplaySubjects always stores the last emitted Value**s**
- It needs the amount of Values it should store

```
private replaySubject = new ReplaySubject<string>(11);
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Flattening Operators

concatMap(), mergeMap(), switchMap(), and exhaustMap().

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

switchMap ⭐

- Switching to a new “inner” observable
  - The previous observable is cancelled
- Maintains only one inner subscription at a time
- Previous requests will be cancelled if source emits quickly enough

Stackblitz Example

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

mergeMap / flatMap ⭐

- flatMap is an alias for mergeMap
- Allows multiple inner subscriptions
- Requests that should not be canceled

Stackblitz Example

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

mergeMap / flatMap ⭐

<code>

```
book$: Observable<{}>;
 constructor(private http: HttpClient) {}

 ngOnInit() {
   this.book$ = this.http
     .get('/api/book/11')
     .pipe(flatMap(book => this.http.get(book.authorId)));
 }
```

ARCHITECTS
INSIDE KNOWLEDGE

concatMap ⭐

- Same like mergeMap but take care about the order
- Does not subscribe to the next observable until the previous completes

Stackblitz Example

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

exhaustMap

- The first emitted value will be mapped
- While this inner observable is still active all other emitted values will be ignored

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Combination Operators

combineLatest(), withLatestFrom(), concat(), forkJoin() and zip()

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

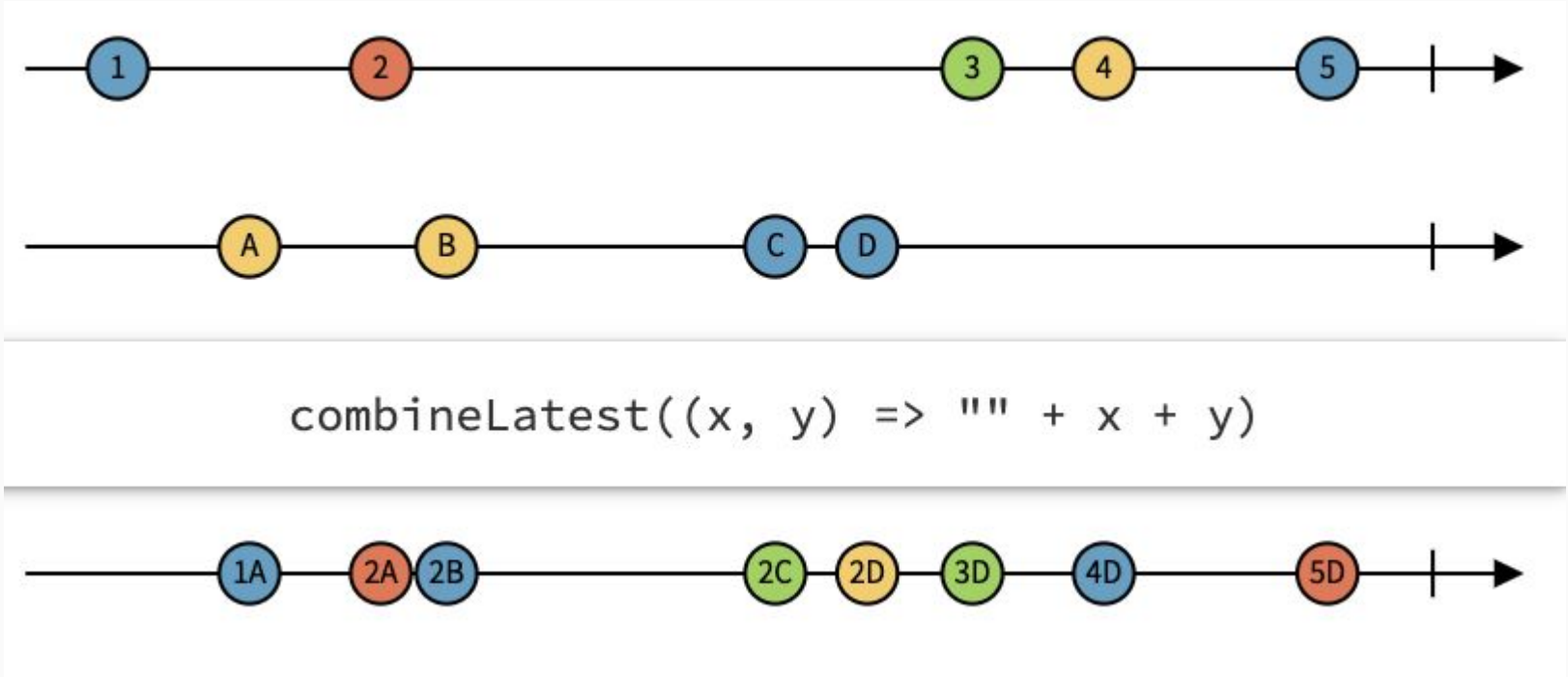
combineLatest ⭐

- Used for long-lived observables relying on each other
- Will not emit an initial value until each observable emitted

Stackblitz Example

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

combineLatest ⭐



withLatestFrom ⭐

- Also provide the last value from another observable
- Both sources must emit at least one value

Stackblitz Example

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

concat ⭐

- Transaction based
- just if previous Observables completes the next subscription starts
- When one source never completes, other observables never run

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

concat ⭐

<code>

```
concat(
    of(1, 2, 3),
    // subscribed after first completes
    of(4, 5, 6),
    // subscribed after second completes
    of(7, 8, 9)
    )
.subscribe(console.log);

// log: 1, 2, 3, 4, 5, 6, 7, 8, 9
```

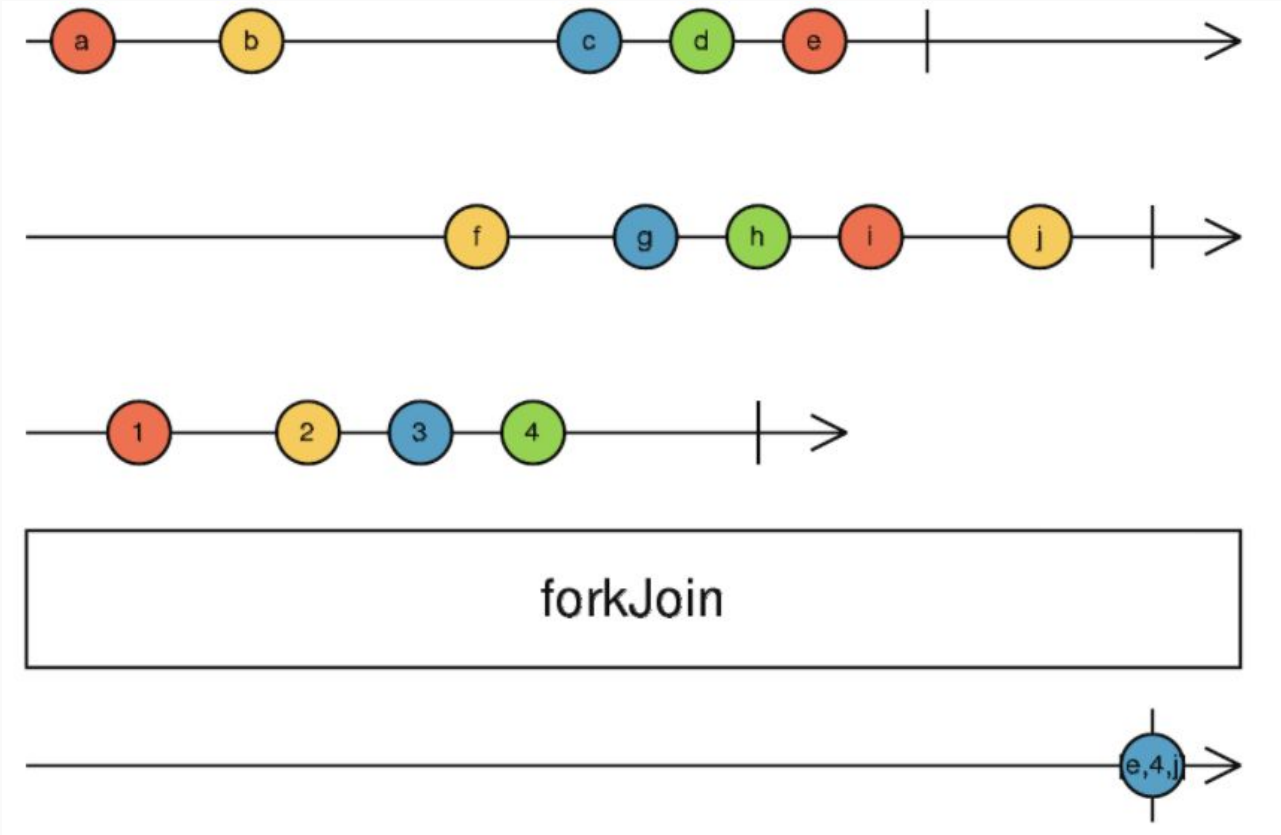
ARCHITECTS
INSIDE KNOWLEDGE

forkJoin

- Similar to Promise.all()
- Takes a list of Observables and executes them in parallel
- Waits for every Observable to emit a value and then emit a single value
- Emits the last emitted value from after all inner observables completed

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

forkJoin



zip

- Combines values together from multiple source Observables
- Doesn’t start to emit until each inner observable has at least one value
- Emits as long as emitted values can be collected from all inner observables

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Good example

```
const you$ = ['Cola Zero', 'Margherita Pizza', 'Tiramisu'];
const mario$ = ['Sprite', 'Carbonara Pizza', 'Fruits salad'];
const luigi$ = ['Pepsi', 'Quattro Formaggi Pizza', 'Ice cream'];

const waiter$ = zip(
  from(you$),
  from(mario$),
  from(luigi$)
);

waiter$.subscribe(
  next => console.log(next),
  error => console.log(error),
  () => console.log('completed!')
);

// ["Cola Zero", "Sprite", "Pepsi"]
// ["Margherita Pizza", "Carbonara Pizza", "Quattro Formaggi Pizza"]
// ["Tiramisu", "Fruits salad", "Ice cream"]
// completed!
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Bad Example

```
const you$ = ['Cola Zero', 'Margherita Pizza', 'Tiramisu'];
const girlfriend$ = ['Sprite'];

const waiter$ = zip(
  from(you$),
  from(girlfriend$)
);

waiter$.subscribe(
  next => console.log(next),
  error => console.log(error),
  () => console.log('completed!')
);

// ["Cola Zero", "Sprite"]
// completed!
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Sending Http Requests

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Multiple parallel HTTP requests with forkJoin

```
  forkJoin([
    this.http.get(`https://api/1/`),
    this.http.get(`https://api/2/`),
    this.http.get(`https://api/3/`),
  ])
    .subscribe(console.log)

// [resultObject1, resultObject2, resultObject3]
```

The forkJoin issue

- The order will be preserved but if one request is delayed all the others have to wait
- If any of the requests fails, it will fail for the whole collection

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Multiple parallel HTTP requests with mergeMap

```
getPokemons(pokeIds: number[]): Observable<Item> {
  return from(pokeIds).pipe(
    mergeMap(pokeIds => this.http.get<Pokemon>(`pokemon/${id}`))
  );
}
```

ARCHITECTS
INSIDE KNOWLEDGE

Sequential HTTP requests [0]

<code>

```
this.http.get('url').pipe(
    concatMap(result1 => this.http.get('url', result1))
    concatMap(result2 => this.http.get('url', result2))
    concatMap(result3 => this.http.get('url', result3))
 ).subscribe();
```

ARCHITECTS
INSIDE KNOWLEDGE

Sequential HTTP requests [1]

<code>

```
this.http.get(`https://pokemon.com/api/pokemon/11`)
    .pipe(
      switchMap(response =>
        forkJoin(response.enemies.map(url => this.http.get(url)))
        )
    ).subscribe(console.log)


// [{ name: “Zapdos” },{ name: “Lugia” },{ name: “Suicune” },...]
```