RXJS INTRODUCTION

VIVEKANAND P V

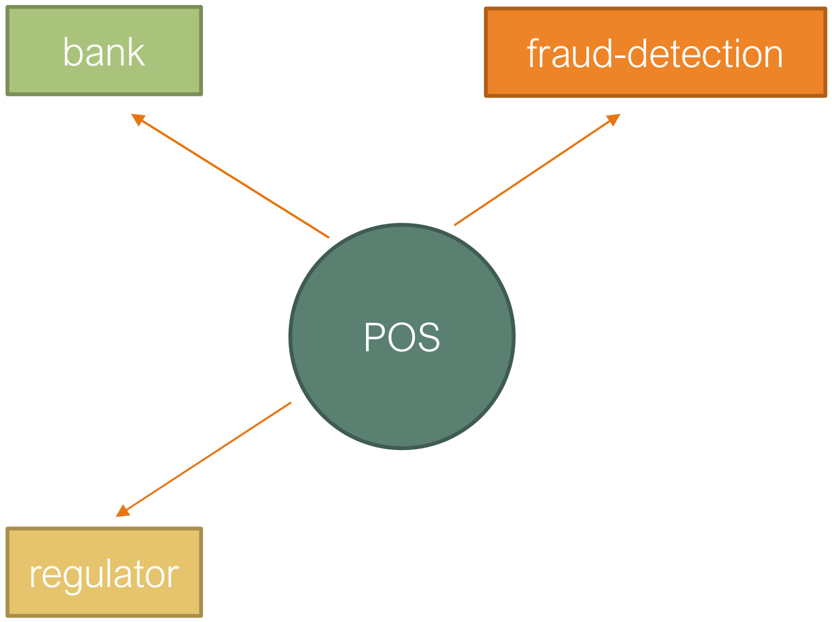
A SCENARIO

- Imagine a credit card is swiped at a POS.
- Issuing bank, regulatory authorities, fraud detection team, tax authorities, service aggregators would like to know this transaction.

How would you design the system?

A SIMPLE SOLUTION

- Let the POS call relevant methods on bank, regulator, etc.
- POS in charge of the whole process
- Any interested party has to be called by POS



DRAWBACKS

- Dependencies
- Inflexible system which is inextensible as well
- Doesn’t scale well

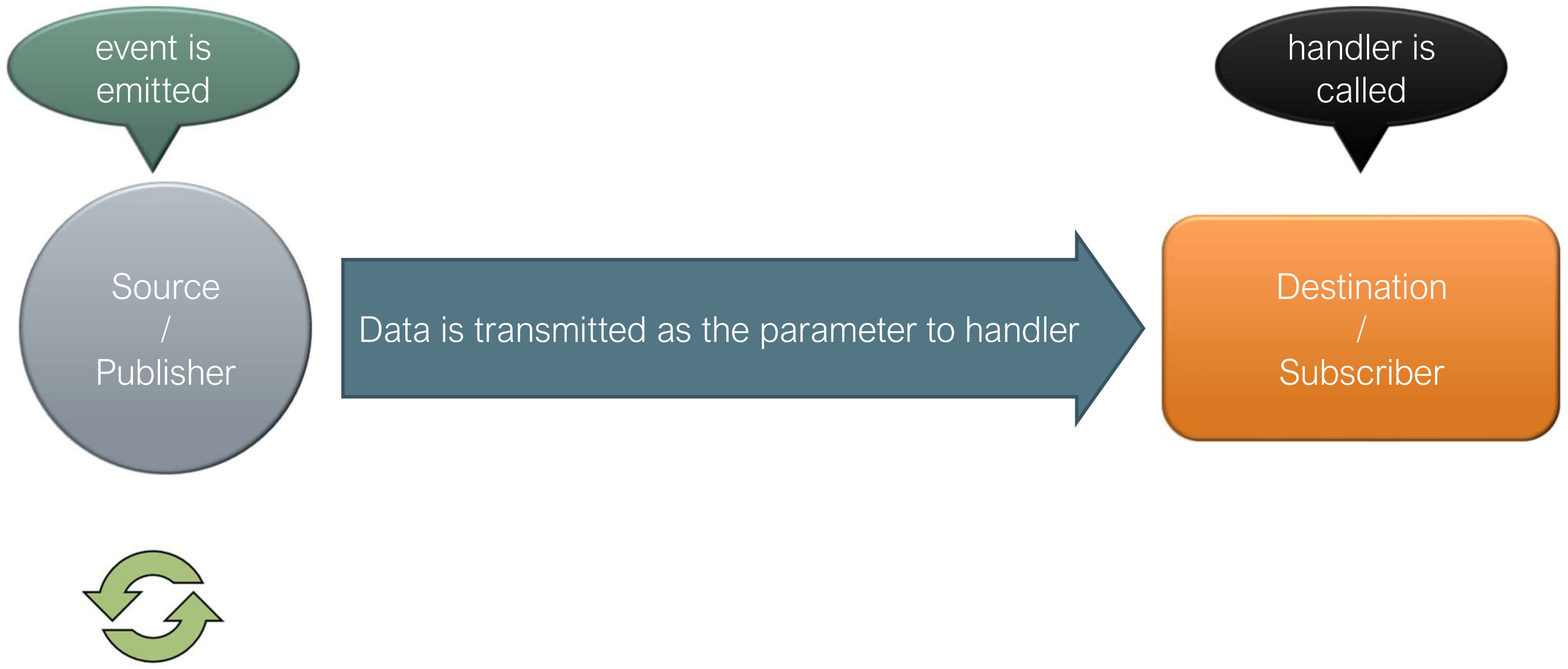
REACTIVE APPROACH

- POS publishes the card-swiped event
- Interested parties subscribe to the event by providing their handlers
- When the event is emitted, the attached handlers are called
- POS and interested parties are loosely coupled by highly coherent
- Interested parties react to the event, hence the name *reactive*

STREAMS: A PRACTICAL CASE

- Streams are the reactive channels for data propagation
- Publishing end keeps emitting the data
- Subscribing ends get the data as parameters in their handlers
- Provides a live communication, albeit one-way (source-destination)

A PICTORIAL REPRESENTATION



RXJS: WHY?

- Short for Reactive Extensions for JavaScript
- Asynchronous API that extends the observer design pattern
- Lots of useful operators
- Excellent documentation
- Most of Angular’s API is Observable-based
- A very powerful tool in state-management
- Knowledge of patterns can be transferred across languages

CORE CONCEPTS

- Observable is an asynchronous stream where data is emitted over time (streaming)
- Inherently secure, as the subscribers can in no way influence the source (read-only streams)
- Fluent-API like pipelining for observable streams is possible
- Error handling is much more flexible and configurable
- For every use, there is an operator, or one could be built by combining several

OBSERVABLES: HOW?

- Observables have to be subscribed (passive by default)
- Typically, the subscriber provides 3 callback functions: success, error, and complete (in that order)
- Forgetting to unsubscribe causes memory leak
- If the observable is used for data binding, automatic subscription can be opted with async pipe
- An observable is seldom created manually. Usually, observables are provided by the API or creational operators are used.

OBSERVABLES: PUBLISHER

- next(…) for new data emission
- error(…) in case of error
- complete() for completion

PLEASE NOTE

- Observable stops once either error(…) is emitted or complete() is called. First one wins.
- After stopping, further calls such as next(…), error(…), complete() are ignored.
- Some observables will never complete while some will emit only one value

OBSERVABLES: SUBSCRIBER

- There can be numerous subscribers to a single observable
- Every subscriber provides next, error, and complete call backs (in that order) for subscriptions
- When observable calls next, error, or complete, these callbacks provided by the subscriber are appropriately invoked. Thus data transmits as function parameters.

OBSERVER IN ACTION

```
let stream$ = new Observable<number>(observer => {
      // First data emits
      observer.next(1);

      // Second data emits
      observer.next(2);

      // Error. Observer comes to a halt
      observer.error(new Error('Unexpected error'));

      //  Following lines have no effect
      observer.next(5);

      observer.complete();
    });
```

SUBSCRIBER IN ACTION

```
stream$.subscribe(
      data => {
        //  called at next(...)
        //  ...
      },
      error => {
        //  called at error(...)
        //  ...
      },
      () => {
        //  called at complete()
        //  ...
      }
    );
```

EXTENSION OF AN OBSERVABLE

- On an existing observable, creating a pipe(…) which sets up an observable pipeline processing capability
- pipe(…) returns the final observable. This is analogous to the concept of middlewares.
- There are lots of operators that can be used to create the pipeline to achieve various needs

PIPELINING: A PRACTICAL CASE

```
const modifiedStream$ = stream$.pipe( //  create pipeline
      map(data => data * data),       //  transmitted number is squared
      first(val => val > 9)           //  take the first square > 9
    );

//  henceforth the subscribers to modifiedStream$ will get only one value
```

SOME USEFUL OPERATORS

- map
- first
- last
- tap
- concat
- delay
- debounce
- catchError
- throwError
- finalize
- zip
- distinctUntilChanged

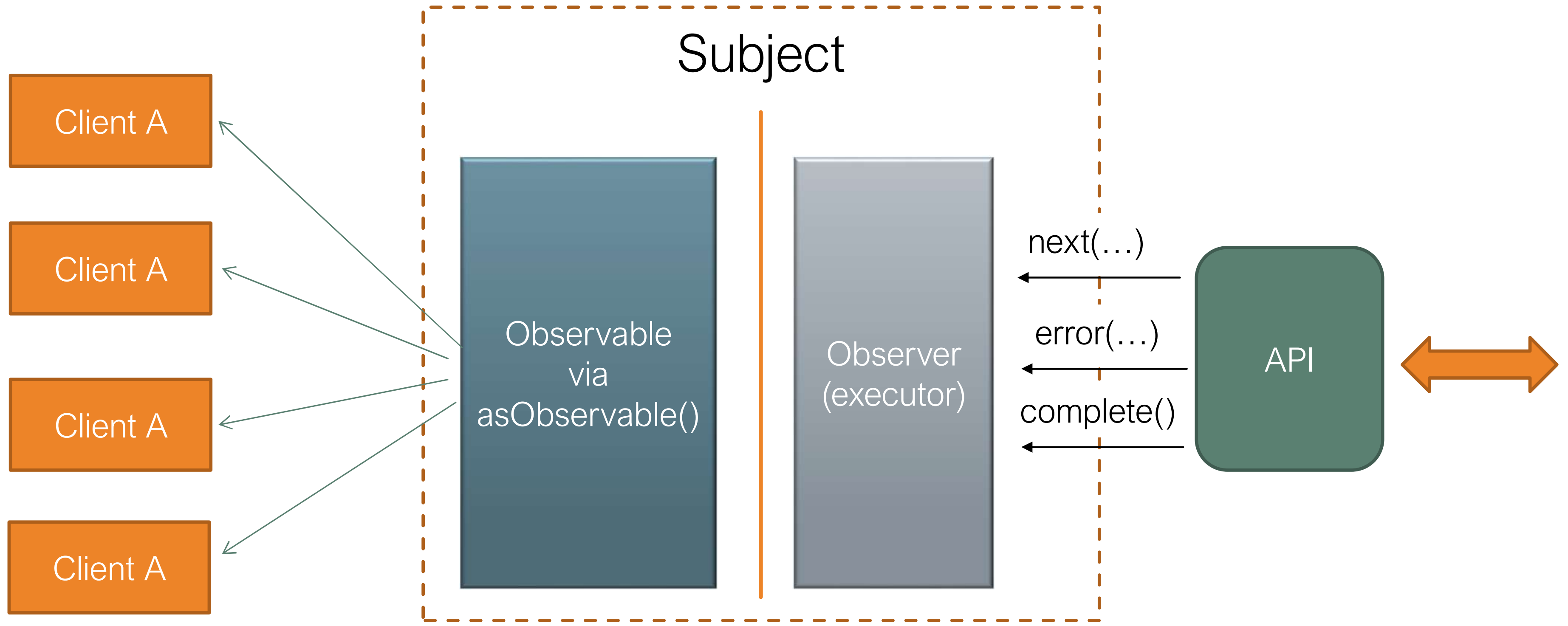
PLEASE BE AWARE

Only the observer, that is the executor inside the body of the observable can call next, error, or complete. Subscriber, or any external entity cannot enforce this. Thus the communication between observable and its subscribers is always one way.

SUBJECTS FOR STATE-MANAGEMENT

- Every subject has observable part as well as executor part (observer)
- Observable part is exposed to clients for subscription, while the wrapper API is built around the executor part
- Clients use this API so that the subject emits new value
- Do not expose the subject to clients

SUBJECTS: ANATOMY



FLAVOURS OF SUBJECTS

Type	Description
Subject<T>	Base subject. No memory. New subscriber gets value at subsequent next(…)
BehaviorSubject<T>	Most used. Every new subscriber gets the latest emitted value. Should be created with an initial value.
ReplaySubject<T>	Replays the data emitted so far to every new subscriber
AsyncSubject<T>	Does not emit until complete. At complete, the last emitted value will be transmitted to clients. Occasionally used. Could be used for caching http streams.

STORE PATTERN

- Abstract the state-management in a service
- Create a private subject
- Create the necessary API (CRUD?)
- Output endpoints are observable-based unless requirement states otherwise

STORE PATTERN: AUTHENTICATION USE CASE

```
export class AuthService {
  private authSubject = new BehaviorSubject<boolean>(false);

  loginStatus$ = this.authSubject.asObservable();

  constructor() {}

  login() {
    this.authSubject.next(true);
  }

  logout() {
    this.authSubject.next(false);
  }
}
```

FURTHER READING

- https://www.learnrxjs.io/
- http://reactivex.io/
- https://en.wikipedia.org/wiki/Reactive_programming
- https://github.com/ReactiveX/rxjs
- https://angular.io/guide/rx-library