

Angular HTTP Client, JSONP, CORS, RxJS

Trayan Iliev

IPT – Intellectual Products & Technologies
e-mail: tiliev@iproduct.org
web: <http://www.iproduct.org>

Oracle®, Java™ and JavaScript™ are trademarks or registered trademarks of Oracle and/or
Microsoft .NET, Visual Studio and Visual Studio Code are trademarks of Microsoft Corp

Agenda

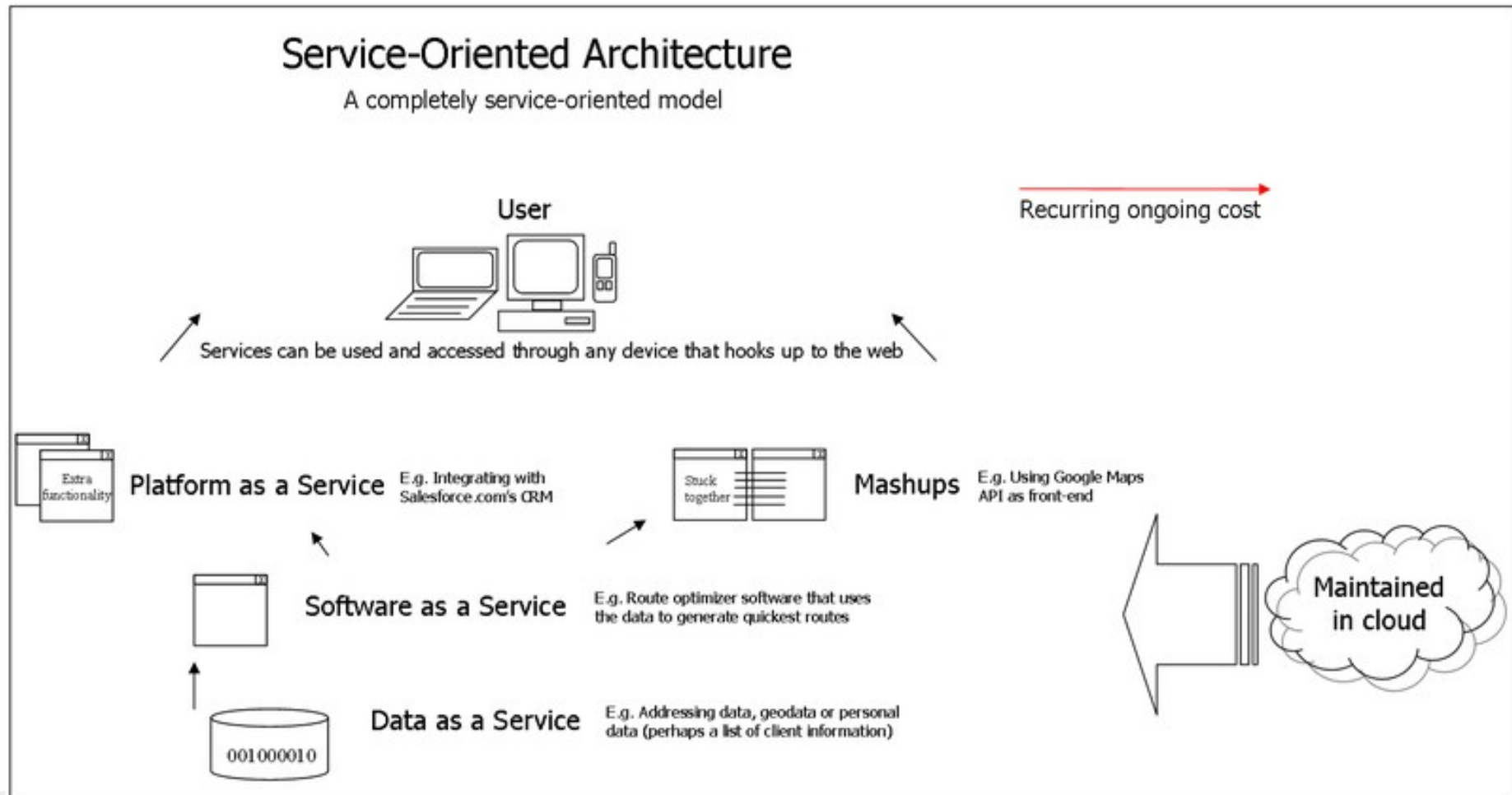
1. Service Oriented Architecture (SOA) & REST
2. Web Standards and Protocols (HTTP, URI, MIME, JSON)
3. REpresentational State Transfer (REST) architectural style – advantages and main constraints
4. RESTful services + JSON. HATEOAS
5. Same origin policy and CORS.
6. Building simple JSON-based CRUD REST(-like) application
7. Sample services with HTTP / HttpClient Angular modules
8. RxJS -composing functional operators
9. JSONP Wiki search Angular demo

Where is The Code?

Angular and TypeScript Web App Development code
is available @GitHub:

<https://github.com/iproduct/course-angular>

Service Oriented Architecture (SOA)



Source: http://en.wikipedia.org/wiki/File:SOA_Detailed_Diagram.png,
Author: JamesLWilliams2010, License: Creative Commons Attribution 3.0 Unported

REST Architecture

According to **Roy Fielding** [Architectural Styles and the Design of Network-based Software Architectures, 2000]:

- Client-Server
- Stateless
- Uniform Interface:
 - Identification of resources
 - Manipulation of resources through representations
 - Self-descriptive messages
 - Hypermedia as the engine of application state (HATEOAS)
- Layered System
- Code on Demand (optional)

Representational State Transfer (REST) [1]

- REpresentational State Transfer (REST) is an architecture for accessing distributed hypermedia web-services
- The resources are identified by URIs and are accessed and manipulated using an HTTP interface base methods (GET, POST, PUT, DELETE, OPTIONS, HEAD, PATCH)
- Information is exchanged using representations of these resources
- Lightweight alternative to SOAP+WSDL -> HTTP + Any representation format (e.g. JavaScript Object Notation – JSON)

Representational State Transfer (REST) [2]

- Identification of resources – URIs
- Representation of resources – e.g. HTML, XML, JSON, etc.
- Manipulation of resources through these representations
- Self-descriptive messages - Internet media type (**MIME type**) provides enough information to describe how to process the message. Responses also explicitly indicate their **cacheability**.
- Hypermedia as the engine of application state (aka **HATEOAS**)
- Application contracts are expressed as **media types** and [semantic] link relations (**rel** attribute - **RFC5988**, "Web Linking")

[Source: http://en.wikipedia.org/wiki/Representational_state_transfer]

Simple Example: URLs + HTTP Methods

Uniform Resource Locator (URL)	GET	PUT	POST	DELETE
Collection, such as http://api.example.com/comments/	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	Delete the entire collection.
Element, such as http://api.example.com/comments/11	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	Replace the addressed member of the collection, or if it does not exist, create it.	Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

Source: https://en.wikipedia.org/wiki/Representational_state_transfer

Advantages of REST

- **Scalability of component interactions** – through layering the client server-communication and enabling load-balancing, shared caching, security policy enforcement;
- **Generality of interfaces** – allowing simplicity, reliability, security and improved visibility by intermediaries, easy configuration, robustness, and greater efficiency by fully utilizing the capabilities of HTTP protocol;
- **Independent development and evolution of components**, dynamic evolvability of services, without breaking existing clients.
- **Fault tolerant, Recoverable, Secure, Loosely coupled**

Richardson's Maturity Model of Web Services

According to **Leonard Richardson** [Talk at QCon, 2008 - <http://www.crummy.com/writing/speaking/2008-QCon/act3.html>]:

- **Level 0 – POX:** Single URI (XML-RPC, SOAP)
- **Level 1 – Resources:** Many URIs, Single Verb (URI Tunneling)
- **Level 2 – HTTP Verbs:** Many URIs, Many Verbs (CRUD – e.g Amazon S3)
- **Level 3 – Hypermedia Links Control the Application State = HATEOAS (Hypertext As The Engine Of Application State) == **truely** RESTful Services**

Hypermedia As The Engine Of Application State (HATEOAS) – New Link Header (RFC 5988) Example

Content-Length →1656

Content-Type →application/json

Link →<<http://localhost:8080/polling/resources/polls/629>>; **rel="prev"**;
type="application/json"; title="Previous poll",
<<http://localhost:8080/polling/resources/polls/632>>; **rel="next"**;
type="application/json"; title="Next poll",
<<http://localhost:8080/polling/resources/polls>>; **rel="collection"**;
type="application/json"; title="Polls collection",
<<http://localhost:8080/polling/resources/polls>>; **rel="collection up"**;
type="application/json"; title="Self link",
<<http://localhost:8080/polling/resources/polls/630>>; **rel="self"**

Web Application Description Language (WADL)

- XML-based file format providing machine-readable description of HTTP-based web application resources – typically RESTful web services
- WADL is a W3C Member Submission
 - Multiple resources
 - Inter-connections between resources
 - HTTP methods that can be applied accessing each resource
 - Expected inputs, outputs and their data-type formats
 - XML Schema data-type formats for representing the RESTful resources
- But WADL resource description is **static**

HTTP Request Structure

GET /context/Servlet HTTP/1.1

Host: *Client_Host_Name*

Header2: Header2_Data

...

HeaderN: HeaderN_Data

<Празен ред>

POST /context/Servlet HTTP/1.1

Host: *Client_Host_Name*

Header2: Header2_Data

...

HeaderN: HeaderN_Data

<Празен ред>

POST_Data

HTTP Response Structure

HTTP/1.1 200 OK

Content-Type: application/json

Header2: Header2_Data

...

HeaderN: HeaderN_Data

<Празен ред>

```
[ { "id":1,  
  "name":"Novelties in Java EE 7 ...",  
  "description":"The presentation is ...",  
  "created":"2014-05-10T12:37:59",  
  "modified":"2014-05-10T13:50:02",  
},  
 { "id":2,  
  "name":"Mobile Apps with HTML5 ...",  
  "description":"Building Mobile ...",  
  "created":"2014-05-10T12:40:01",  
  "modified":"2014-05-10T12:40:01",  
}]
```

Cross-Origin Resource Sharing (CORS)

- Позволява осъществяване на заявки за ресурси към домейни различни от този за извикващия скрипт, като едновременно предоставя възможност на сървъра да прецени към кои скриптове (от кои домейни – Origin) да връща ресурса и какъв тип заявки да разрешава (GET, POST)
- За да се осъществи това, когато заявката е с HTTP метод различен от GET се прави предварителна (preflight) OPTIONS заявка в отговор на която сървъра връща кои методи са достъпни за съответния Origin и съответния ресурс

Нови заглавни части на HTTP при реализация на CORS

- HTTP GET заявка

GET /crossDomainResource/ HTTP/1.1

Referer: <http://sample.com/crossDomainMashup/>

Origin: <http://sample.com>

- HTTP GET отговор

Access-Control-Allow-Origin: <http://sample.com>

Content-Type: application/xml

Нови заглавни части на HTTP при реализация на POST заявки при CORS

- HTTP OPTIONS preflight request

OPTIONS /crossDomainPOSTResource/ HTTP/1.1

Origin: http://sample.com

Access-Control-Request-Method: POST

Access-Control-Request-Headers: MYHEADER

- HTTP response

HTTP/1.1 200 OK

Access-Control-Allow-Origin: http://sample.com

Access-Control-Allow-Methods: POST, GET, OPTIONS

Access-Control-Allow-Headers: MYHEADER

Access-Control-Max-Age: 864000

Fetch API

[https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API]

- The **Fetch API** provides an interface for fetching resources like XMLHttpRequest, but more powerful and flexible feature set.
- **Promise<Response> WorkerOrGlobalScope.fetch(input[, init])**
 - **input** - resource that you wish to fetch – url string or Request
 - **init** - custom settings that you want to apply to the request: **method**: (e.g., GET, POST), **headers**, **body** (Blob, BufferSource, FormData, URLSearchParams, or USVString), **mode**: (cors, no-cors, or same-origin), **credentials** (omit, same-origin, or include. to automatically send cookies this option must be provided), **cache**: (default, no-store, reload, no-cache, force-cache, or only-if-cached), **redirect** (follow, error or manual), **referrer** (default is client), **referrerPolicy**: (no-referrer, no-referrer-when-downgrade, origin, origin-when-cross-origin, unsafe-url), **integrity** (subresource integrity value of request)

Angular HttpClient – Importing All Rx Operators

```
//tslint:disable-next-line:import-blacklist
import 'rxjs/Rx';
const API_URL = 'http://localhost:4200/api/';

@Injectable()
export class BackendService {
  constructor(private http: HttpClient, private logger: LoggerService){}

  findAll<T extends Identifiable> (type: Type<T>): Promise<T[]> {
    const url = API_URL + this.getCollectionName(type)
    return this.http.get<T[]>(url)
      .map(productsResponse => productsResponse['data'])
      .do(products => this.logger.log(products))
      .toPromise<T[]>();
  }
}
```

HttpClient – Observable.prototype Patching

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/do';
const API_URL = 'http://localhost:4200/api/';

@Injectable()
export class BackendService {
  constructor(private http: HttpClient, private logger: LoggerService){}

  findAll<T extends Identifiable> (type: Type<T>): Promise<T[]> {
    const url = API_URL + this.getCollectionName(type)
    return this.http.get<T[]>(url)
      .map(productsResponse => productsResponse['data'])
      .do(products => this.logger.log(products))
      .toPromise<T[]>();
  }
}
```

HttpClient – Pipe & Lettable Operators

```
import { catchError, map, tap } from 'rxjs/operators';
const API_URL = 'http://localhost:4200/api/';

@Injectable()
export class BackendService {
  constructor(private http: HttpClient, private logger: LoggerService){}

  findAll<T extends Identifiable> (type: Type<T>): Promise<T[]> {
    const url = API_URL + this.getCollectionName(type)
    return this.http.get<T[]>(url)
      .pipe(
        map(productsResponse => productsResponse['data']),
        tap(products => this.logger.log(products))
      ).toPromise<T[]>();
  }
}
```

HttpClient – Type-checking the Responses


```
export interface CollectionResponse<T> {  
  data: Array<T>;  
}  
  
export interface IndividualResponse<T> {  
  data: T;  
}  
...  
findAll<T extends Identifiable> (type: Type<T>): Promise<T[]> {  
  const url = API_URL + this.getCollectionName(type)  
  return this.http.get<CollectionResponse<T>>(url)  
    .pipe(  
      map(productsResponse => productsResponse.data),  
      tap(products => this.logger.log(products))  
    ).toPromise();  
}  
...
```

Entyty Typed Services

```
@Injectable()
export class ProductService {

  constructor(private backend: BackendService) { }

  findAll() {
    return this.backend.findAll(Product);
  }
}
```



Consuming Component

```
@Component({  
  selector: 'ws-product-list',  
  ...  
})  
export class ProductListComponent implements OnInit {  
  products: Product[] = [];  
  constructor(public productService: ProductService) { }  
  
  ngOnInit() {  
    this.productService.findAll().then(products => {  
      console.log(products);  
      this.products = products;  
    }).catch(err => console.log('Error:', err));  
  }  
  ...  
}
```


HttpClient – Error Handling (1)

```
export interface CollectionResponse<T> {  
  data: Array<T>;  
}  
export interface IndividualResponse<T> {  
  data: T;  
}  
...  
findAll<T extends Identifiable> (type: Type<T>): Promise<T[]> {  
  const url = API_URL + this.getCollectionName(type)  
  return this.http.get<CollectionResponse<T>>(url)  
    .pipe(  
      map(productsResponse => productsResponse.data),  
      tap(products => this.logger.log(products)),  
      retry(3), // retry request if failed - up to 3 times  
      catchError(this.handleError)  
    ).toPromise();  
}  
...
```

HttpClient – Error Handling (2)

```
private handleError(error: HttpResponse) {  
  if (error.error instanceof ErrorEvent) {  
    // Client-side or network error  
    console.error('Client-side error:', error.error.message);  
  } else {  
    // Backend unsuccessful status code.  
    console.error(  
      `Backend returned code ${error.status}, ` +  
      `body was: ${JSON.stringify(error.error)},`  
      `message was: ${JSON.stringify(error.message)}`);  
  }  
  // return ErrorObservable with a user-facing error message  
  return new ErrorObservable(  
    'There was a problem with backend service. Try again later.');
```

Angular 4.3+ HttpClient API: Setup

```
@Injectable()
export class BackendPromiseHttpService implements BackendPromiseService
{

    Constructor(
        @Inject(API_BASE_URL) private baseUrl: string,
        private http: HttpClient,
        private logger: LoggerService
    ) {}

    ...
}
```

Angular 4.3+ HttpClient API: POST Entity

```
public add<T extends Identifiable>(type: Type<T>, item: T): Promise<T> {  
    return this.http.post<T>(this.baseUrl + '/' + this.getCollectionName(type), item)  
        .catch(error => {  
            return Observable.throw(  
                new ApplicationError<T>(this.getErrorMessage(error), type, item.id, item));  
            }).toPromise();  
        }  
}
```

Angular 4.3+ HttpClient API: PUT (Edit) Entity

```
public edit<T extends Identifiable>(type: Type<T>, item: T): Promise<T>
{
    return this.http.put<T>(this.baseUrl + '/' +
        this.getCollectionName(type) + '/' + item.id, item)
        .catch(error => {
            return Observable.throw(
                new ApplicationError<T>(
                    this.getErrorMessage(error), type, item.id, item));
        }).toPromise();
}
```

Angular 4.3+ HttpClient: DELETE Entity

```
public delete<T extends Identifiable>(type: Type<T>, id: IdentityType) {  
    return this.http.delete<T>(this.baseUrl + '/' +  
                                this.getCollectionName(type) + '/' + id)  
        .catch(error => {  
            return Observable.throw(  
                new ApplicationError<T>(this.getErrorMessage(error), type, id));  
            }).toPromise();  
        }  
}
```

Reactive Programming. Functional Programming

- **Reactive Programming [Wikipedia]:** a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow. **Ex: $a := b + c$**
- **Functional Programming [Wikipedia]:** a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm. Eliminating side effects can make it much easier to understand and predict the program behavior. **Ex: `bookStream.map(book => book.author)`**

Functional Reactive Programming

- **Functional Reactive Programming (FRP)** [Wikipedia]: a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter). FRP has been used for programming graphical user interfaces (GUIs), robotics, and music, aiming to simplify these problems by explicitly modeling time. **Ex. (RxJS):**

```
const Observable = require('rxjs').Observable;
Observable.from(['Reactive', 'Extensions', 'JavaScript'])
  .take(2).map(s => `${s}: on ${new Date()}`)
  .subscribe(s => console.log(s));
```

Result: *Reactive: on Sat Apr 29 2017 20:00:39 GMT+0300*
Extensions: on Sat Apr 29 2017 20:00:39 GMT+0300

Definitions of Reactive Programming

- Microsoft® opens source polyglot project **ReactiveX** (Reactive Extensions) [<http://reactivex.io>]:

Rx = Observables + LINQ + Schedulers :)

- Supported Languages – Java: RxJava, JavaScript: RxJS, C#: Rx.NET, C#(Unity): UniRx, Scala: RxScala, Clojure: RxClojure, C++: RxCpp, Ruby: Rx.rb, Python: RxPY, Groovy: RxGroovy, JRuby: RxJRuby, Kotlin: RxKotlin, Swift: RxSwift
- ReactiveX for platforms and frameworks: RxNetty, RxAndroid, RxCocoa
- **ES7 Observable** Specification
[<https://github.com/tc39/proposal-observable>]

ES7 Observable Spec (RxJS 5)

```
interface Observable {  
  constructor(subscriber : SubscriberFunction);  
  subscribe(observer : Observer) : Subscription;  
  subscribe(onNext : Function,  
            onError? : Function,  
            onComplete? : Function) : Subscription;  
  [Symbol.observable]() : Observable;  
  static of(...items) : Observable;  
  static from(iterableOrObservable) : Observable;  
}
```

```
interface Subscription {  
  unsubscribe() : void;  
  get closed() : Boolean;  
}
```

RxJS – JS ReactiveX (Reactive Extensions)

[<http://reactivex.io/rxjs/>, <https://github.com/ReactiveX/rxjs>]

- **ReactiveX** is a **polyglot** library for composing asynchronous event streams (observable sequences).
- It extends the **observer pattern** by declarative composition of functional transformations on events streams (e.g. **map-filter-reduce**, etc.)
- Abstracts away low-level concerns like **concurrency, synchronization, and non-blocking I/O**.
- Follows the **next - error - completed** event flow
- Allows natural implementation of **Redux** design pattern
- Alternative (together with *promises*) for solving “callback hell” problem

RxJS Resources

ReactiveX - RxJS:

<http://reactivex.io/rxjs/>, <https://github.com/ReactiveX/rxjs>

RxMarbles:

<http://rxmarbles.com/>

RxJS Coans:

<https://github.com/Reactive-Extensions/RxJSKoans>

Learn RxJS:

<https://www.learnrxjs.io/>

RxJS – Observable Lifecycle

[<http://reactivex.io/rxjs/manual/overview.html>]

- **Creating** Observables
- **Subscribing** to Observables
- **Executing** the Observable
- **Disposing** Observables

Hot and Cold Event Streams

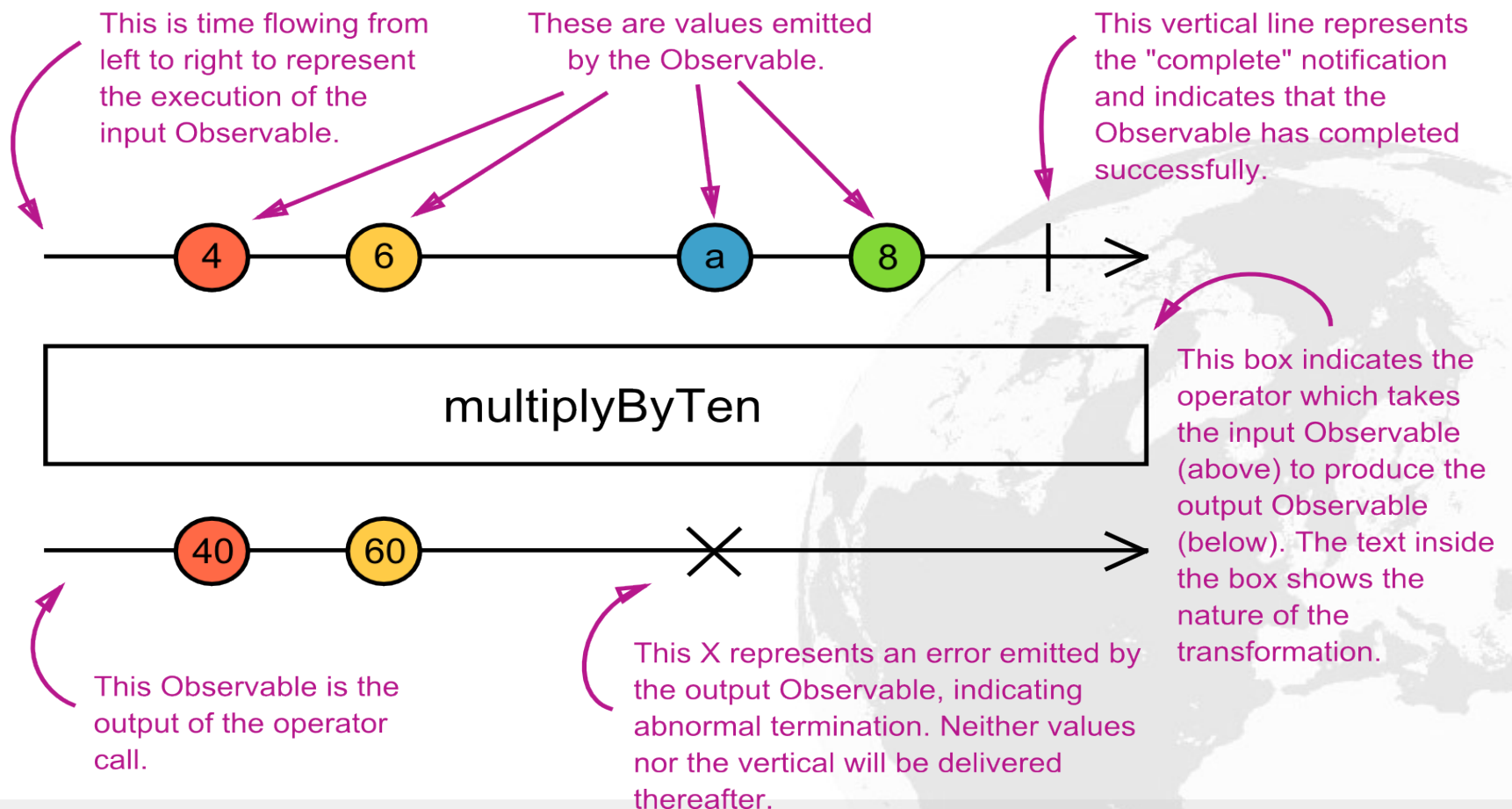
- **PULL-based (Cold Event Streams)** – Cold streams are streams that run their sequence when and if they are subscribed to. They present the sequence from the start to each subscriber.
- **PUSH-based (Hot Event Streams)** – Hot streams emit values independent of individual subscriptions. They have their own timeline and events occur whether someone is listening or not - e.g. mouse events. A mouse is generating events regardless of whether there is a subscription. When subscription is made observer receives current events as they happen.

RxJS – Hot (PUSH) Event Stream Publishers

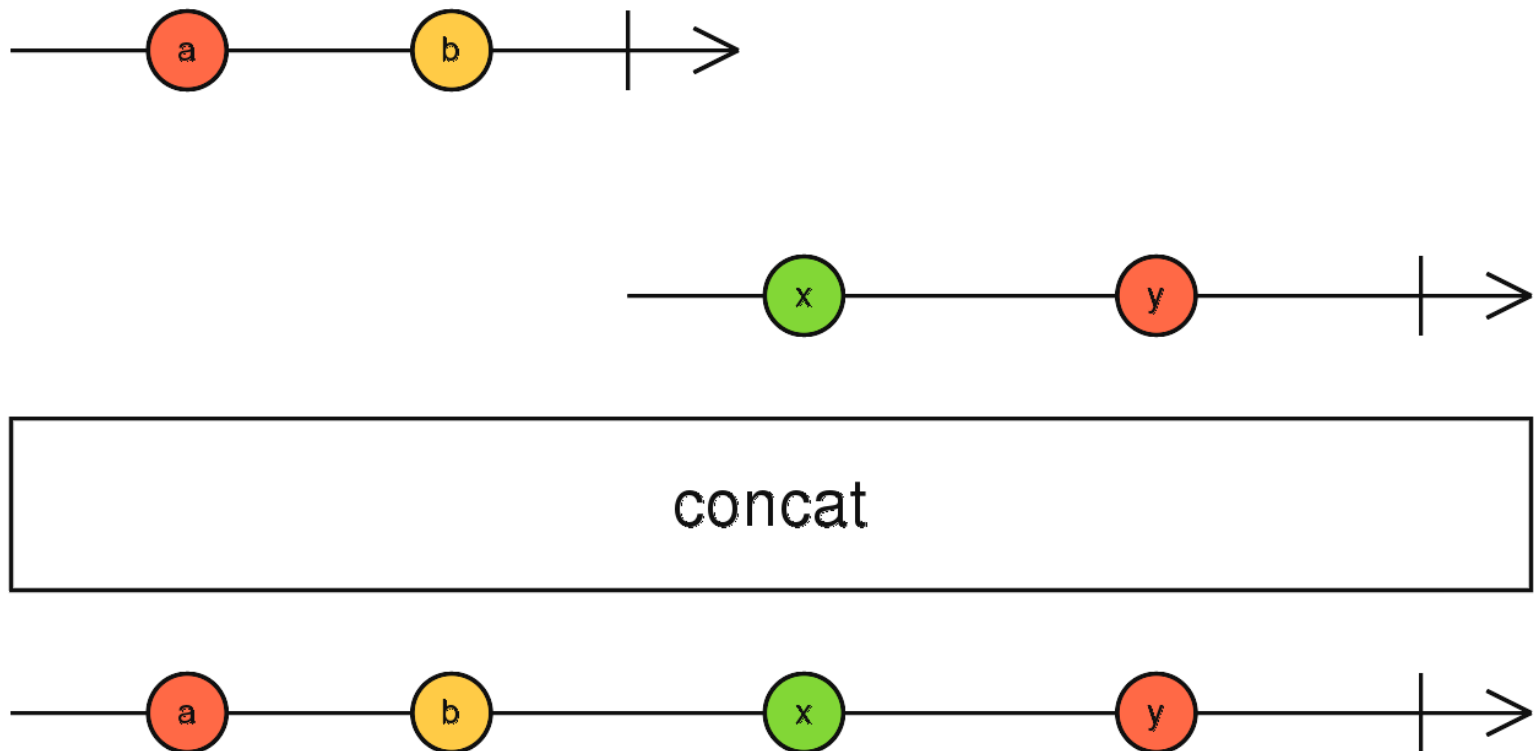
[<http://reactivex.io/rxjs/manual/overview.html#subject>]

- RxJS **Subject** = Observable allowing multicasting to many subscriber Observers
- **BehaviorSubject** – models a notion of “current value” over time
- **ReplaySubject** – allows to replay last N events for all subscriber observers
- **AsyncSubject** – only the last value is sent to its observers, and only when the execution completes

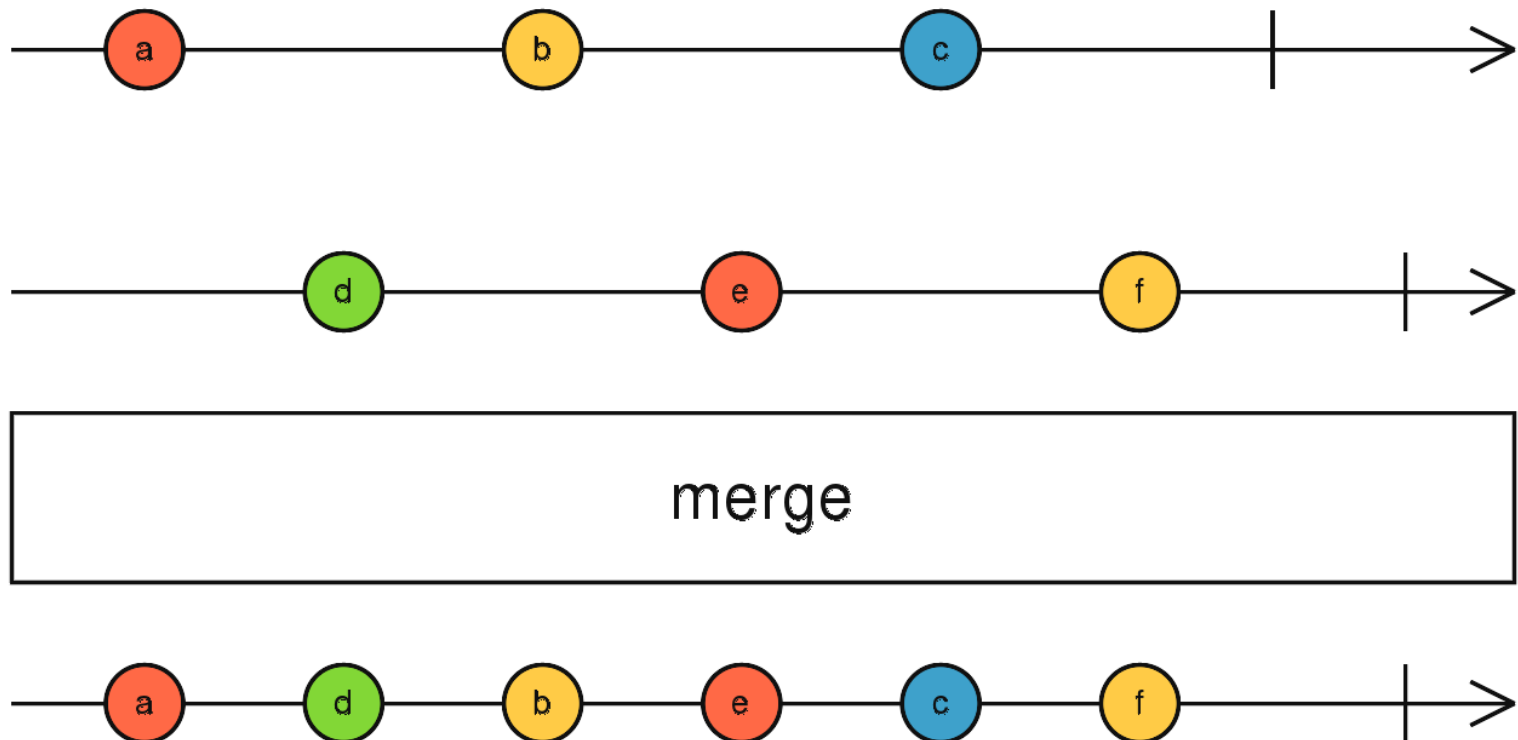
Anatomy of Rx Operator



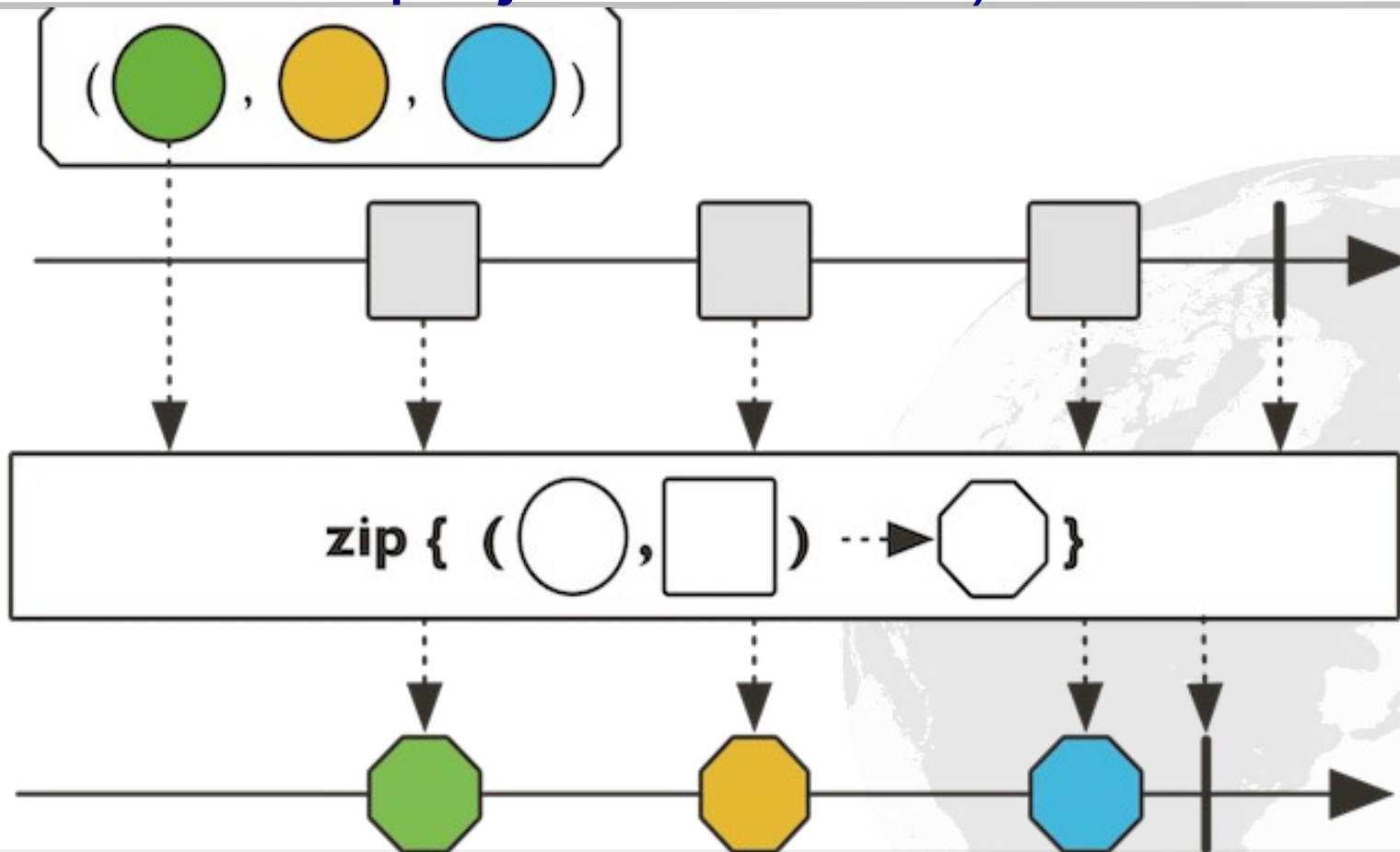
concat(in1: Observable, in2: Observable) :Observable



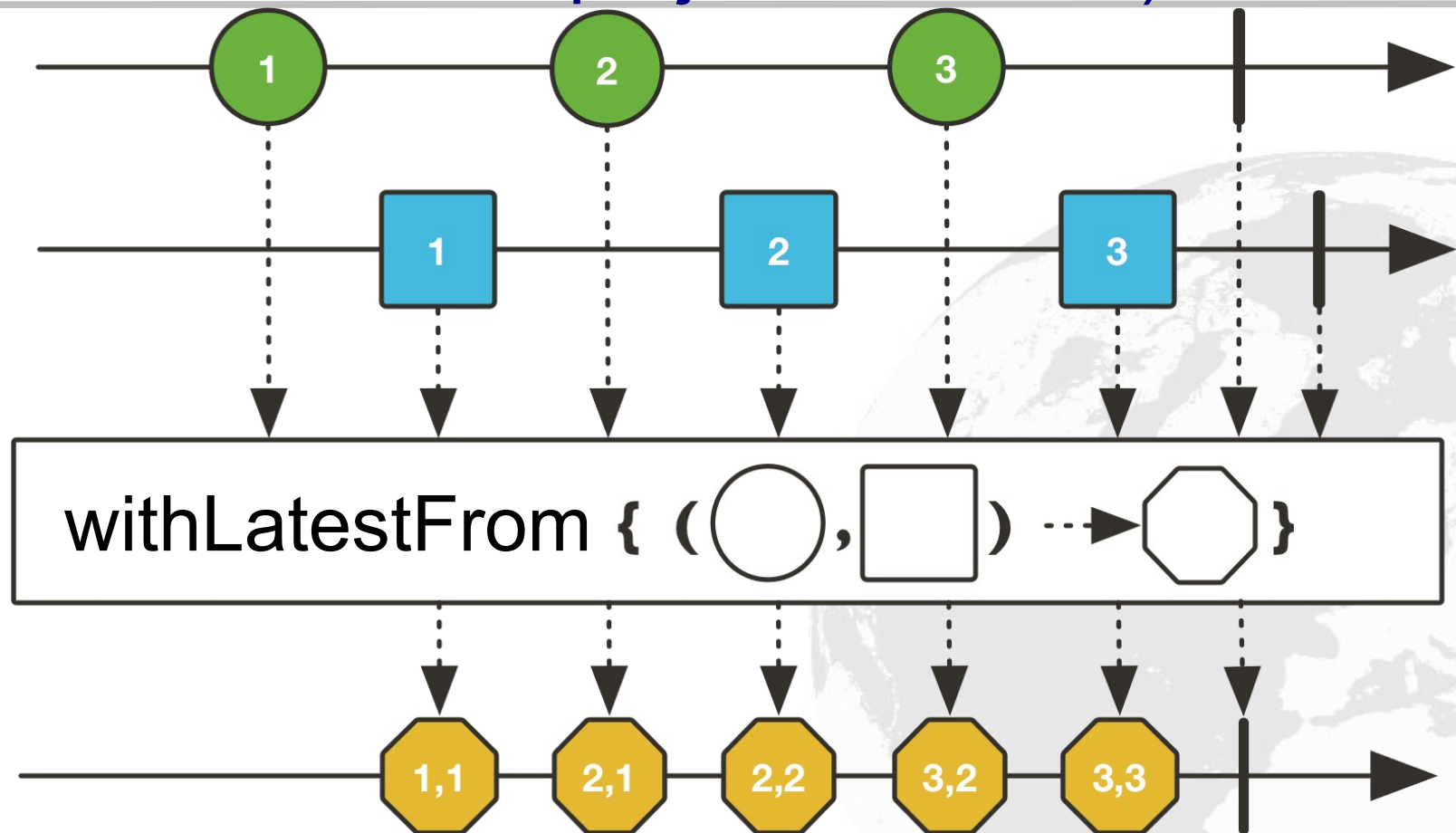
merge(inObservables: ...Observable) :Observable



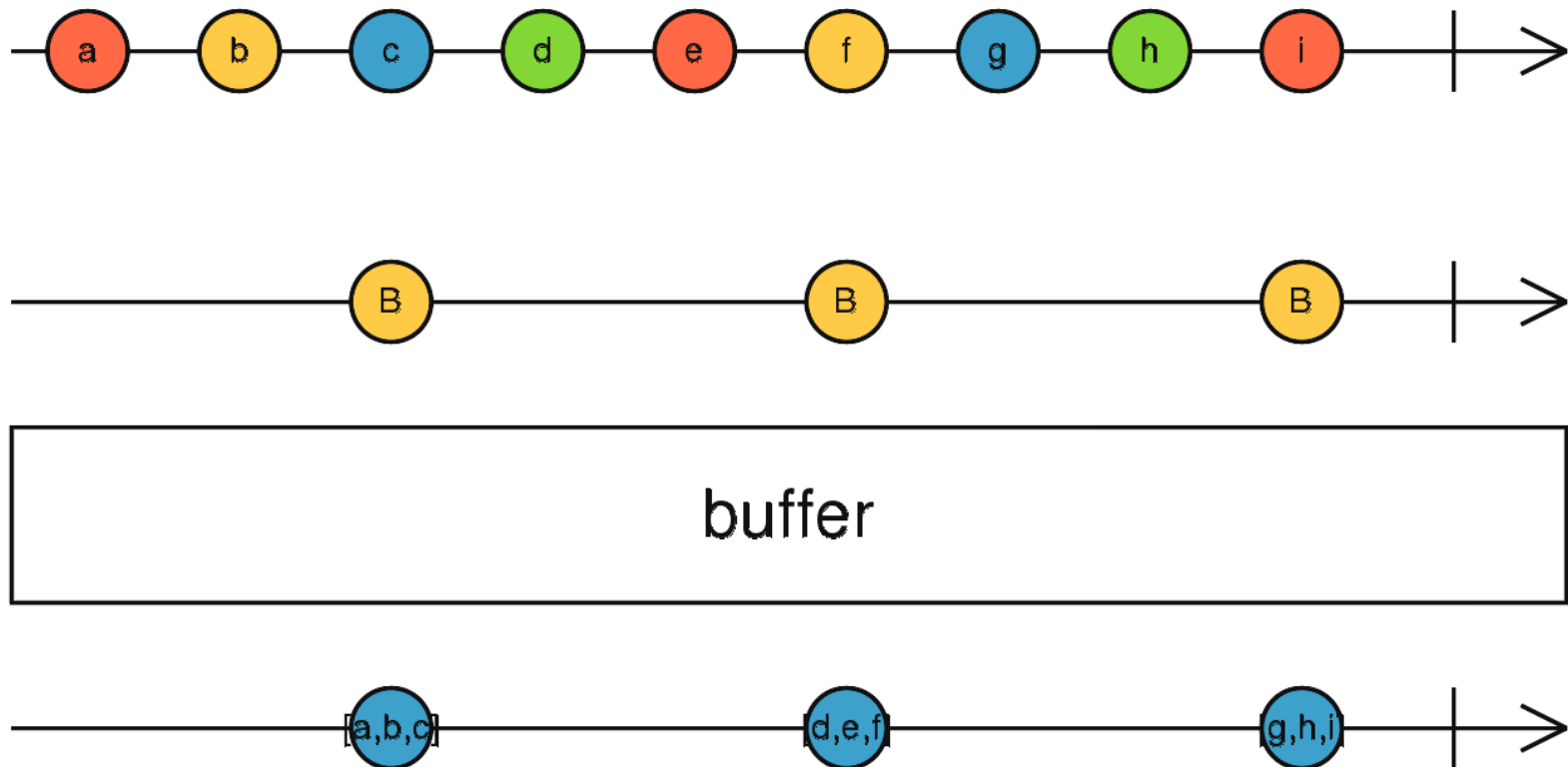
zip(observables: ...Observable, project: Function): Observable



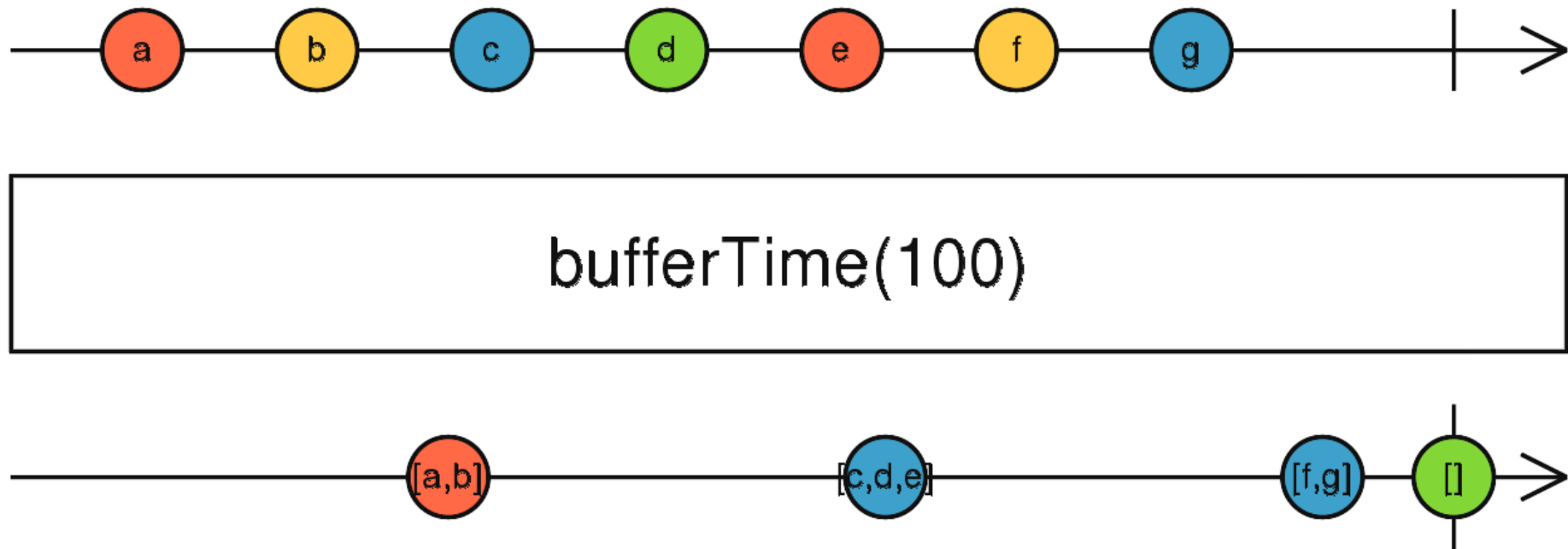
withLatestFrom(other: Observable, project: Function): Observable



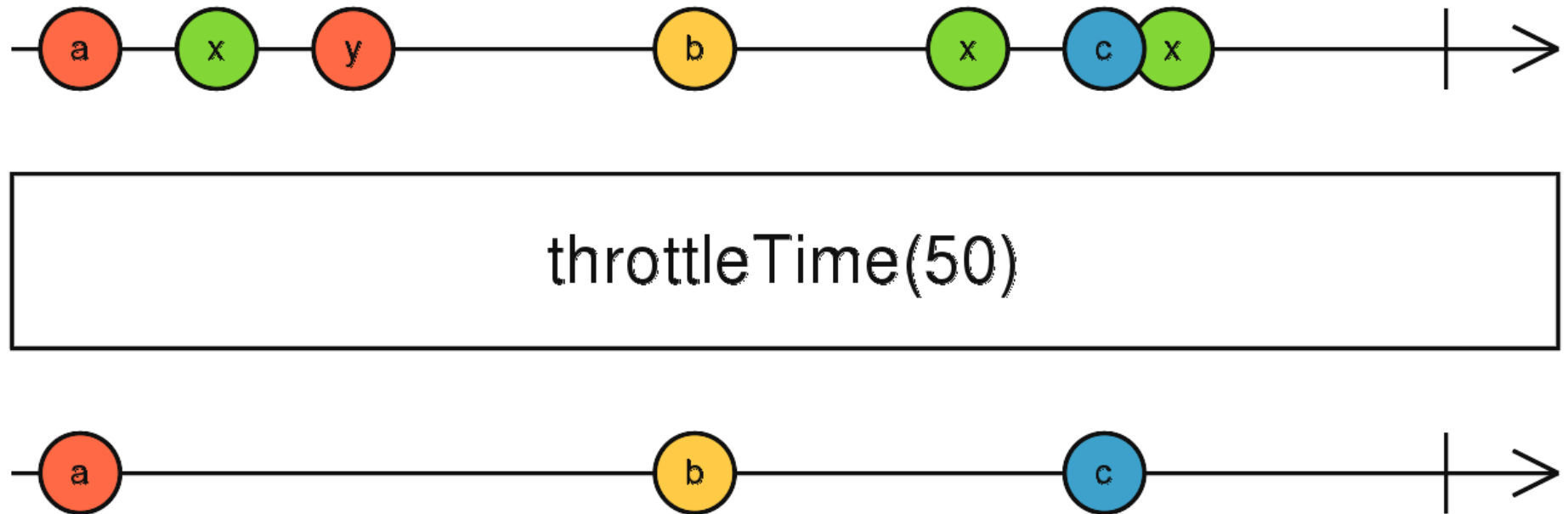
buffer(closingNotifier: Observable): Observable



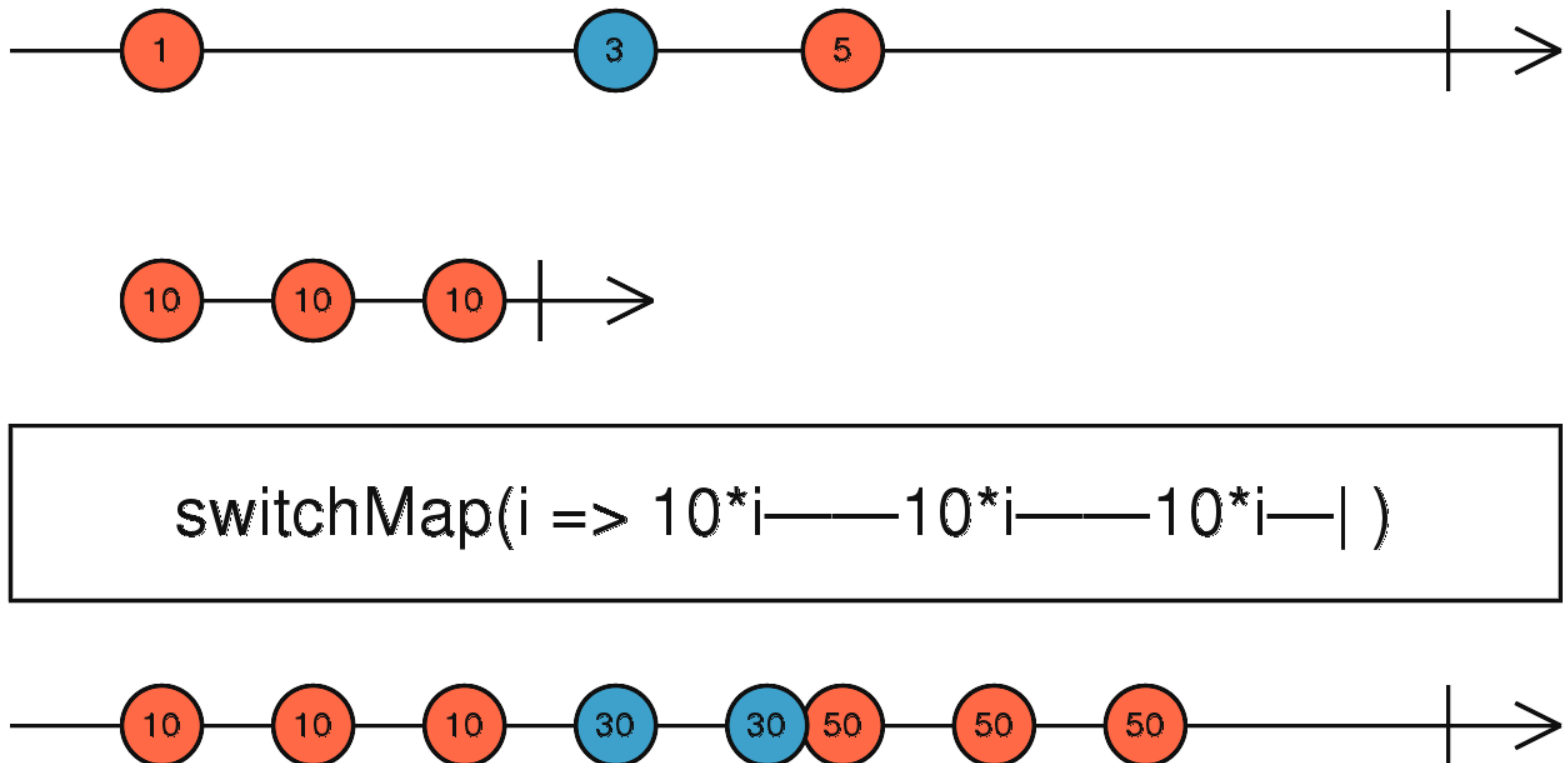
`bufferTime(time: number, creationInterval: number, maxSize: number, sched: Scheduler): Observable`



`throttleTime(duration: number, scheduler: Scheduler): Observable`



Example: switchMap()



Cross-Domain Requests Using JSONP (1)

[<https://angular.io/docs/ts/latest/guide/server-communication.html>]

```
import { Component } from '@angular/core';
import { Observable, Subject } from 'rxjs/Rx';
import { WikipediaService } from './wikipedia.service';
@Component({
  selector: 'my-wiki',
  template: `
    <div class="demo">
      <h1>Wikipedia Demo</h1>
      <p><i>Fetches after each keystroke</i></p>
      <input #term (keyup)="search(term.value)"/>
      <ul>
        <li *ngFor="let item of items | async">{{item}}</li>
      </ul>
    </div>`,
  providers: [WikipediaService]
})
```

Cross-Domain Requests Using JSONP (2)

[<https://angular.io/docs/ts/latest/guide/server-communication.html>]

```
export class WikiComponent {  
  private searchTermStream = new Subject<string>();  
  public items: Observable<string[]> = this.searchTermStream  
    .debounceTime(300)  
    .distinctUntilChanged()  
    .switchMap((term: string) => this.wikipediaService.search(term));  
  
  constructor(private wikipediaService: WikipediaService) { }  
  public search(term: string) { this.searchTermStream.next(term); }
```

Cross-Domain Requests Using JSONP (3)

[<https://angular.io/docs/ts/latest/guide/server-communication.html>]

```
import {Injectable} from '@angular/core';
import {Jsonp, Response, URLSearchParams} from '@angular/http';
@Injectable()
export class WikipediaService {
  constructor(private jsonp: Jsonp) { }
  public search(term: string) {
    let wikiUrl = 'http://en.wikipedia.org/w/api.php';
    let params = new URLSearchParams();
    params.set('search', term); // the user's search value
    params.set('action', 'opensearch');
    params.set('format', 'json');
    params.set('callback', 'JSONP_CALLBACK');
    return this.jsonp
      .get(wikiUrl, { search: params })
      .map((response: Response) => <string[]> response.json()[1]);
  }
}
```

Thanks for Your Attention!

Questions?

