RXJS AND ROUTING

LESSON 02

SWAFE–01

SINGLE-PAGE APPLICATIONS

OVERVIEW

- All application functionality exists in a single HTML page
- Browsers only render the parts that matters to the user
- Significantly improves the user experience

INDEX.HTML

```
1  <!doctype html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <title>RoutingBasic</title>
6    <base href="/">
7    <meta name="viewport" content="width=device-width, initial-scale=1">
8    <link rel="icon" type="image/x-icon" href="favicon.ico">
9  </head>
10 <body>
11   <app-root></app-root>
12 </body>
13 </html>
```

examples/lesson02-rxjs-and-routing/projects/router-basic/src/index.html

`<base href>`

- The router uses the browser's `history.pushState` for navigation
- The browser uses the `<base href>` to prefix relative URLs
- This is needed to be able to locate other resources
  - Media assets, such as images, audio, etc.
  - Stylesheets
  - Scripts, such as third-party libraries, etc.
- Will throw a `404` if misconfigured (look at where it tried to locate the files to fix it)

ROUTING

OVERVIEW

- A routing module exports a `RouterModule`
- The `Route` is a configuration interface that defines a single route
- The order of routes is important
  - The `Router` uses a first-match wins strategy
  - More specific routes should be placed above less specific
- Add the `RouterLink` directive to an element to link to a route
- You can define wildcard routes to handle user attempts to navigate to a non-existing route

ROUTING MODULE

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3  import { FirstComponent } from './first/first.component';
4  import { SecondComponent } from './second/second.component';
5
6  const routes: Routes = [{
7    path: 'first',
8    component: FirstComponent,
9  }, {
10   path: 'second',
11   component: SecondComponent,
12 },...];
13
14 @NgModule({
15   imports: [RouterModule.forRoot(routes)],
16   exports: [RouterModule]
17 })
18 export class AppRoutingModule { }
```

examples/lesson02-rxjs-and-routing/projects/router-basic/src/app/app-routing.component.ts

ROUTE INTERFACE

- `path` The path to match against
- `component` The component to instantiate when the path matches
- `outlet` A name of a `RouterOutlet` object where the component can be placed when the path matches
- `redirectTo` A URL to redirect to when the path matches
- `loadChildren` An object specifying lazy-loading child routes
- `loadComponent` An object specifying standalone component route

ROUTER OUTLETS

- Acts as a placeholder that Angular dynamically fills based on current router state
- Each outlet can have a unique name
- Named outlets can be targets of secondary routes
- Using named outlets and secondary routes, you can target multiple outlets in the same `RouterLink` directive

ROUTER OUTLETS

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3  ...
4  import { ThirdComponent } from './third/third.component';
5  import { FourthComponent } from './fourth/fourth.component';
6
7  const routes: Routes = [..., {
8    path: 'third',
9    component: ThirdComponent,
10 }, {
11   path: 'third',
12   component: FourthComponent,
13   outlet: 'feature'
14 }];
15
16 @NgModule({
17   imports: [RouterModule.forRoot(routes)],
18   exports: [RouterModule]
19 })
```

examples/lesson02-rxjs-and-routing/projects/routing-basic/src/app/app-routing.module.ts

ROUTER OUTLETS

```
<nav>
  <ul>
    <li><a routerLink="first">First</a></li>
    <li><a routerLink="second">Second</a></li>
    <li><a [routerLink]="[{outlets: { primary: ['third'], feature: ['third']}}]
  </ul>
</nav>
<router-outlet></router-outlet>
<router-outlet name="feature"></router-outlet>
```

examples/lesson02-rxjs-and-routing/projects/routing-basic/src/app/app.component.html

WILDCARD ROUTES

- A well-functiong application should gracefully handle when users attempt to navigate to a part of your application that does not exist
- A wildcard route should be placed last because it matches any URL

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3  import { PageNotFoundComponent } from './page-not-found/page-not-found.compone
4
5  const routes: Routes = [{
6    ...
7  }, {
8    path: '**',
9    component: PageNotFoundComponent,
10 }];
11 @NgModule({
12   imports: [RouterModule.forRoot(routes)],
13   exports: [RouterModule]
14 })
15 export class AppRoutingModule { }
```

examples/lesson02-rxjs-and-routing/projects/routing-advanced/src/app/app-routing.module.ts

LAZY-LOADING MODULES

OVERVIEW

- Modules are eagerly loaded by default
- Configure routes to only load when needed
  - Rather than loading all modules when the application launches
- It is also possible to pre-load parts of the application in the background to improve user experience
- Lazy-loading keeps the initial bundle sizes smaller
  - Which helps to reduce load times (and thereby improve user experience)

IMPORTS AND ROUTE CONFIGURATION

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3  import { PageNotFoundComponent } from './page-not-found/page-not-found.compone
4
5  const routes: Routes = [{
6    path: 'one',
7    loadChildren: () => import('./feature-one/feature-one.module').then(m => m.F
8  }, {
9    path: 'two',
10   loadChildren: () => import('./feature-two/feature-two.module').then(m => m.F
11 }, {
12   path: 'three',
13   loadChildren: () => import('./feature-three/feature-three.module').then(m =>
14 }, {
15   path: '**',
16   component: PageNotFoundComponent,
17 }];
18 @NgModule({
19   imports: [RouterModule.forRoot(routes)],
```

examples/lesson02-rxjs-and-routing/projects/routing-advanced/src/app/app-routing.module.ts

CHUNK FILES

ROUTE GUARDS

OVERVIEW

- Prevents clients from navigating to parts of an application without authorization
- The router supports the following guard interfaces are available in Angular
  - `CanActivate` decide if route can be activated
  - `CanActivateChild` decide if a child route can be activated
  - `CanDeactivate` decide if a route can be deactivated
  - `CanLoad` decide if children can be loaded
- A guard can return synchronously (`boolean`) or asynchronously (`Observable` / `Promise`)

EXAMPLE – AUTHGUARD

```
1  import { Injectable } from '@angular/core';
2  import { ActivatedRouteSnapshot, CanActivate, CanLoad, Route, RouterStateSnaps
3  import { Observable } from 'rxjs';
4  import { tap } from 'rxjs/operators';
5  import { AuthService } from './auth.service';
6
7  @Injectable({
8    providedIn: 'root'
9  })
10 export class AuthGuard implements CanActivate, CanLoad {
11   constructor(private auth: AuthService) { }
12
13   canActivate(
14     route: ActivatedRouteSnapshot,
15     state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boole
16       return this._checkAuthState(state.url)
17   }
18
19   canLoad(
```

examples/lesson02-rxjs-and-routing/projects/route-guards/src/app/auth.guard.ts

EXAMPLE – AUTHGUARD

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3  import { AnotherPublicComponent } from './another-public/another-public.compon
4  import { AuthGuard } from './auth.guard';
5  import { PublicComponent } from './public/public.component';
6
7  const routes: Routes = [{
8    path: '',
9    pathMatch: 'full',
10   children: [{
11     path: '',
12     component: PublicComponent
13   }, {
14     path: '',
15     component: AnotherPublicComponent,
16     outlet: 'another'
17   }]
18 }, {
19   path: 'restricted',
```

examples/lesson02-rxjs-and-routing/projects/route-guards/src/app/app-routing.module.ts

ROUTE GUARD USAGE

- Route guards (or services) should not be used for verification

  Check your tokens in a trusted environment (backend)

- You can apply any number of guards to a path
  - When one returns `false`, the entire navigation will cancel
  - Pending guards that have not completed will be cancelled

REACTIVE PROGRAMMING

OVERVIEW

- A declarative programming paradigm concerned with data streams and the propagation of change
- Data stream
  - Sequence of elements made available over time
- Asynchronous and event-based

REACTIVEX

- An API for asynchronous programming with observable streams
- A combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming
- Created by Dutch computer scientist Erik Meijer in 2009
- Initially designed to be used with .NET technology (Rx.NET), but have since been ported to multiple languages
    - Java (RxJava), JavaScript (RxJS), Swift (RxSwift), Kotlin (RxKotlin), Python (RxPY)

FEATURES

- Functional Avoid intricate stateful programs, clean input/output functions over observable streams
- Less is more Operators reduce verbose implementations into a few lines of code
- Async error handling Mechanisms for handling errors in asynchronous computations
- Concurrency made easy observables and schedulers abstract away low-level threading and synchronization

RXJS

OVERVIEW

- Consumer The code that is subscribing to an Observable
- Observer The manifestation of a Consumer, that may have some (or all) handlers
- Producer Any entity that is the source of values
- Subscription A contract where a consumer is observing values pushed by a producer
- Observable A template for connecting an Observer, as a Consumer, to a Producer, via a subscribe action, resulting in a Subscription

HOT VS. COLD OBSERVABLES

- Cold observables can only have one subscription
  - Do not create the value before it is subscribed to
  - Creates a new producer for each subscription
  - Always unicast: One producer observed by one consumer
- Hot observables can have multiple subscription
  - The producer is created outside of the context of a subscription
  - Most likely multicast: One producer is observed by multiple observers
  - `Subject` is a special observable that allows multicasting
    - Can have new values pushed to it (add more values after creation)
    - Must be created before adding/pushing new values
  - `BehaviorSubject` is a `Subject` with the notion of current value
    - A `Subject` with the concept of a current value

BEHAVIORSUBJECT

```
1  import { Injectable } from '@angular/core';
2  import { BehaviorSubject, Observable } from 'rxjs';
3
4  @Injectable({
5    providedIn: 'root'
6  })
7  export class SharedService {
8
9    data$: BehaviorSubject<Billionaire | null> = new BehaviorSubject(null)
10
11   getBillionaire(): Observable<Billionaire> {
12     return this.data$
13   }
14
15   pushBillionaire(billionaire: Billionaire) {
16     this.data$.next(billionaire)
17   }
18 }
```

examples/lesson02-rxjs-and-routing/projects/rxjs/src/app/shared.service.ts

RXJS OPERATORS

OVERVIEW

- RxJS operators allow complex asynchronous code to be composed in a declarative manner
- Operators are, simply speaking, just functions
- There are two kinds of operators:
  - Pipeable operators takes an `Observable` as input and generates a new `Observable` as output
  - Creation operators are standalone functions that creates new `Observable` objects
- Marble diagrams are used to explain how operators work
  - Many operators are related to time
  - Include the input `Observable` object(s), the operator and the output `Observable`

OPERATOR CATEGORIES

- There are various categories
  - Creating and composing streams
  - Iterating through the values in a stream
  - Mapping values to different types
  - Filtering data from streams
  - Utility for handling errors and debug
- There are over 100 operators, but some are used more than others

SELECTED OPERATORS

- Creation create `Observable` objects from data and events
  - of
  - from
- Combination combine multiple `Observable` objects
  - combineLastest
- Transformation transform source data into new types
  - map
  - switchMap
- Filtering filter values emitted from `Observable` objects
  - filter
  - take
  - debounceTime
- Utility operators that perform helpful operations
  - tap

OF

```
1  import { of } from 'rxjs';
2
3  of(1, 2, 3)
4  .subscribe(
5    next => console.log('next:', next),
6    err => console.log('error:', err),
7    () => console.log('the end'),
8  );
9
10 // Logs:
11 // next: 1
12 // next: 2
13 // next: 3
14 // the end
```

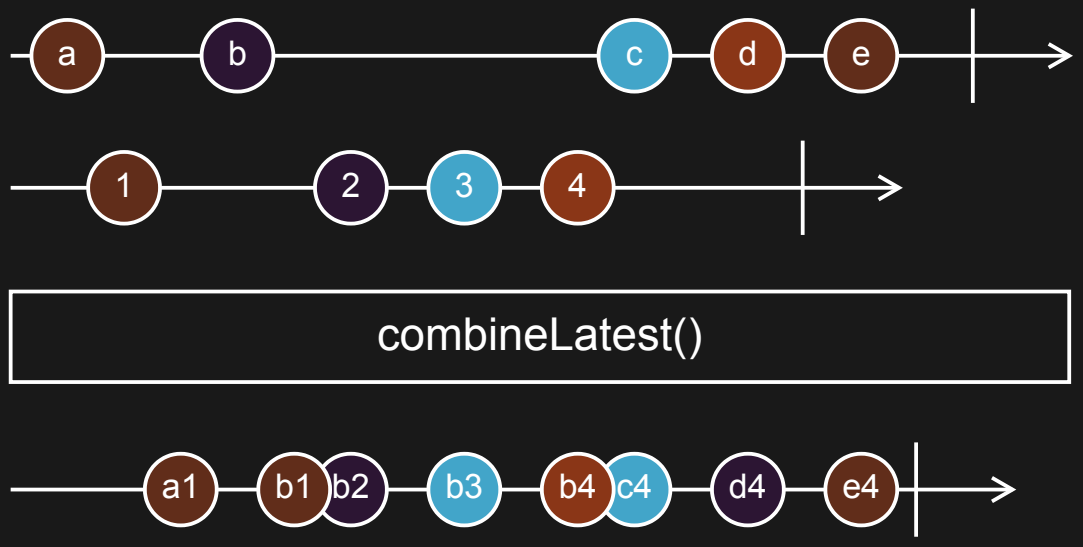
FROM

```
1  import { from } from 'rxjs';
2
3  const array = [10, 20, 30];
4  const result = from(array);
5
6  result.subscribe(x => console.log(x));
7
8  // Logs:
9  // 10
10 // 20
11 // 30
```

COMBINELATEST

Get latest values of multiple Observables

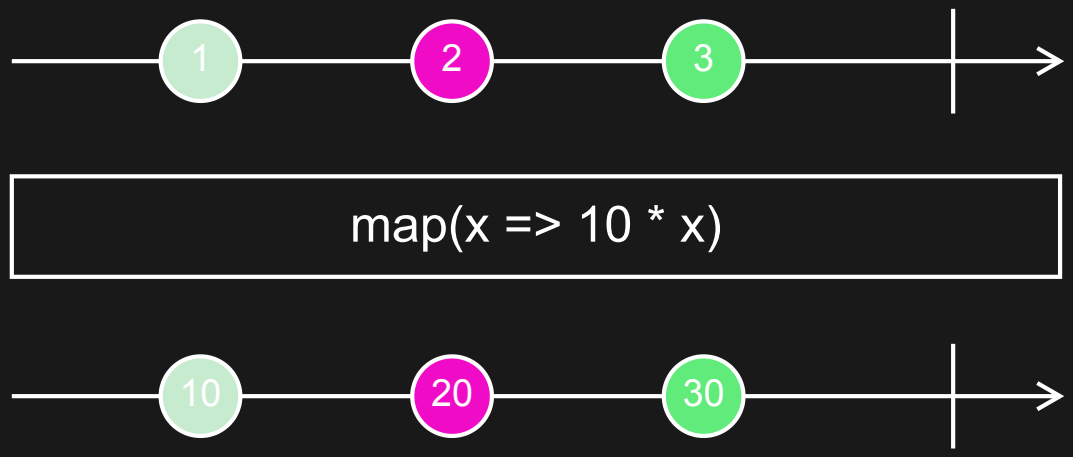
```
1 import { combineLatest, timer } from 'rxjs';
2
3 const firstTimer = timer(0, 1000); // emit 0, 1, 2... after every second, start
4 const secondTimer = timer(500, 1000); // emit 0, 1, 2... after every second, st
5 const combinedTimers = combineLatest([firstTimer, secondTimer]);
6 combinedTimers.subscribe(value => console.log(value));
```



MAP

Transform an input to a new output

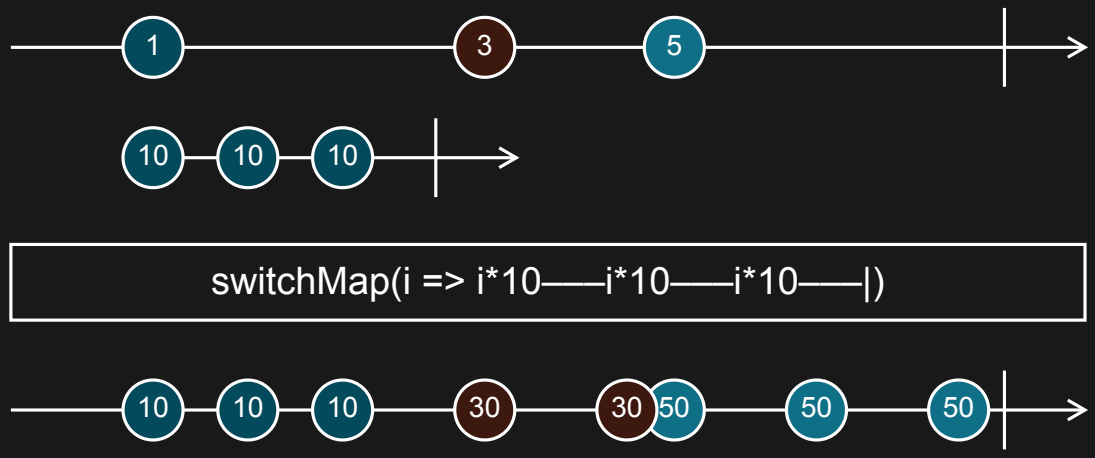
```
1 import { fromEvent } from 'rxjs';
2 import { map } from 'rxjs/operators';
3
4 const clicks = fromEvent(document, 'click');
5 const positions = clicks.pipe(map(ev => ev.clientX));
6 positions.subscribe(x => console.log(x));
```



SWITCHMAP

Switch from one Observable to another

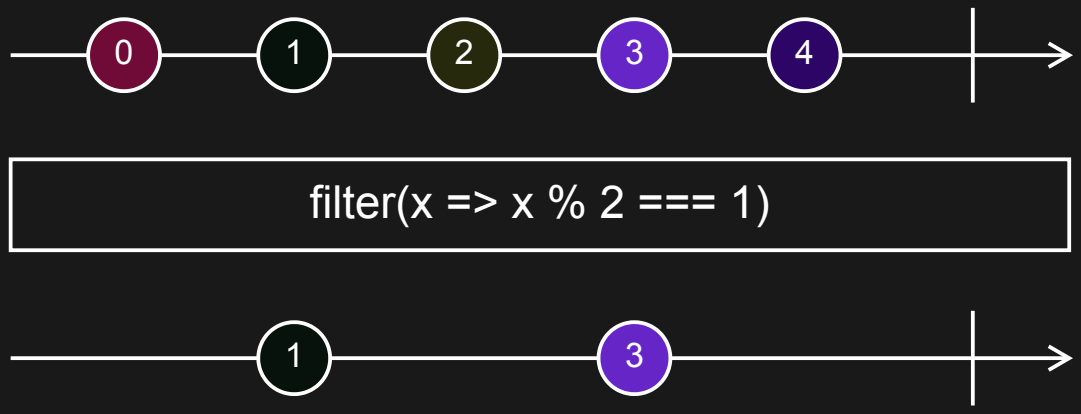
```
1 import { fromEvent, interval } from 'rxjs';
2 import { switchMap } from 'rxjs/operators';
3
4 const clicks = fromEvent(document, 'click');
5 const result = clicks.pipe(switchMap((ev) => interval(1000)));
6 result.subscribe(x => console.log(x));
```



FILTER

Control which values are emitted based on condition

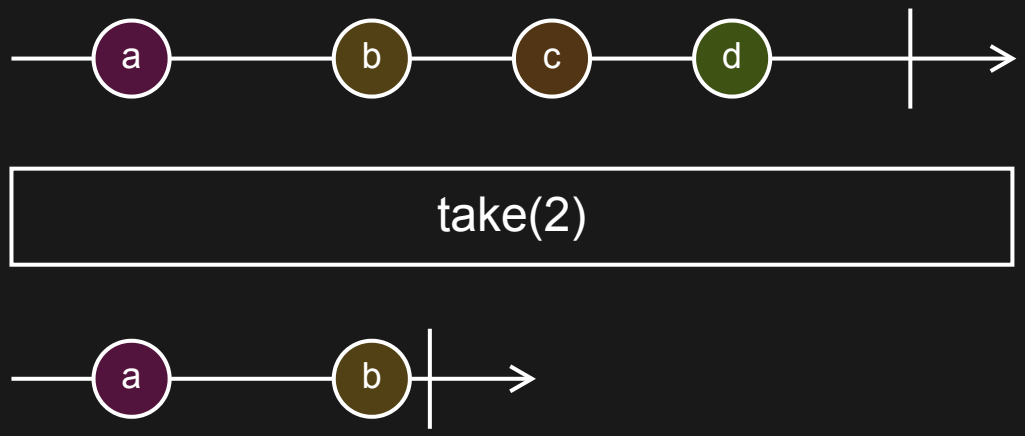
```
1 import { fromEvent } from 'rxjs';
2 import { filter } from 'rxjs/operators';
3
4 const clicks = fromEvent(document, 'click');
5 const clicksOnDivs = clicks.pipe(filter(ev => ev.target.tagName === 'DIV'));
6 clicksOnDivs.subscribe(x => console.log(x));
```



TAKE

Emits only the first count values emitted by the source Observable

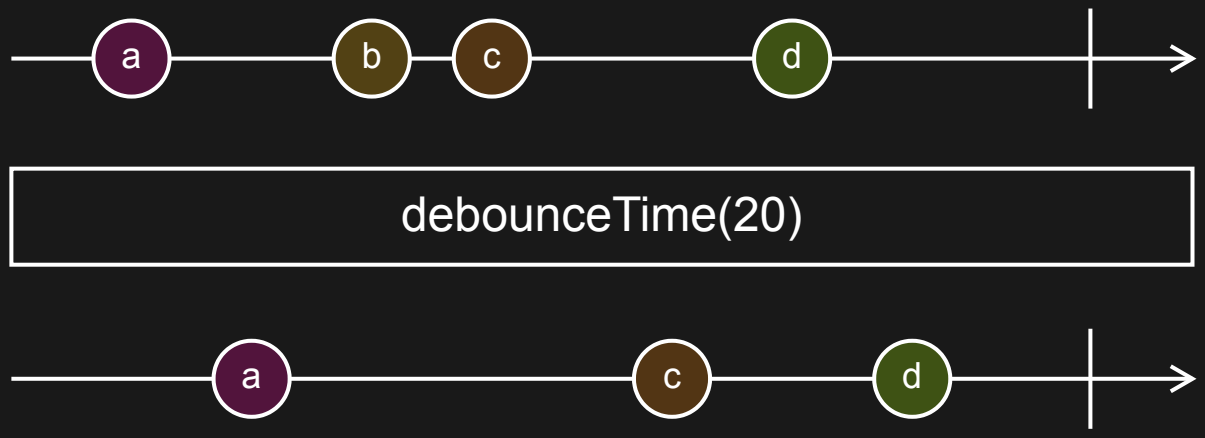
```
1 import { interval } from 'rxjs';
2 import { take } from 'rxjs/operators';
3
4 const intervalCount = interval(1000);
5 const takeFive = intervalCount.pipe(take(5));
6 takeFive.subscribe(x => console.log(x));
```



DEBOUNCETIME

Emits notification after time span and no new source emissions

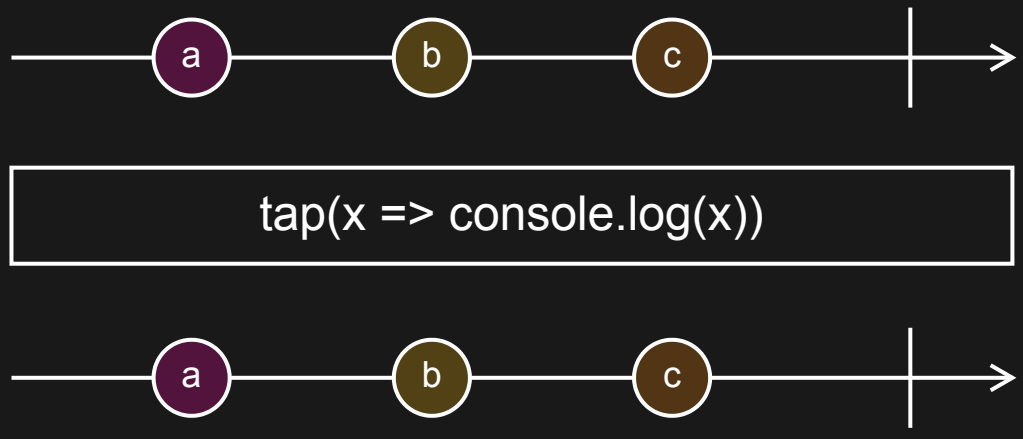
```
1 import { fromEvent } from 'rxjs';
2 import { debounceTime } from 'rxjs/operators';
3
4 const clicks = fromEvent(document, 'click');
5 const result = clicks.pipe(debounceTime(1000));
6 result.subscribe(x => console.log(x));
```



TAP

Trigger side effects from inside the observable pipe

```
1 import { of } from 'rxjs';
2 import { tap, map } from 'rxjs/operators';
3
4 of(Math.random()).pipe(
5   tap(console.log),
6   map(n => n > 0.5 ? 'big' : 'small')
7 ).subscribe(console.log);
```



OBSERVABLES IN ANGULAR

OVERVIEW

- Transmitting data between components
- HTTP
- Router
- Reactive forms

ASYNCPIPE

- The async pipe subscribes to an `Observable` or `Promise`
- Returns the latest value emitted
- Marks components to be checked for changes
- Unsubscribes automatically upon component destruction

SERVICE

```
1  import { Injectable } from '@angular/core';
2  import { interval, Observable, of } from 'rxjs';
3  import { switchMap, tap } from 'rxjs/operators'
4  import { User, USERS } from './user.type';
5
6  @Injectable({
7    providedIn: 'root'
8  })
9  export class DataService {
10
11   list: number[] = []
12
13   getData(): Observable<number> {
14     return interval(1000)
15   }
16
17   getList(): Observable<number[]> {
18     return this.getData().pipe(
19       tap(value => this.list.push(value)),
```

examples/lesson02-rxjs-and-routing/projects/async-pipe/src/app/data.service.ts

CLASS

```
1  import { Component, OnInit } from '@angular/core';
2  import { Observable } from 'rxjs';
3  import { DataService } from './data.service';
4  import { User } from './user.type';
5
6  @Component({
7    selector: 'app-root',
8    templateUrl: './app.component.html',
9    styleUrls: ['./app.component.scss']
10 })
11 export class AppComponent implements OnInit {
12   data$: Observable<number>
13   dataList$: Observable<number[]>
14   dataUser$: Observable<User>
15
16   constructor(private service: DataService) {
17     this.data$ = this.service.getData()
18     this.dataList$ = this.service.getList()
19     this.dataUser$ = this.service.getObjects()
```

examples/lesson02-rxjs-and-routing/projects/async-pipe/src/app/app.component.ts

TEMPLATE

```
<p>{{ data$ | async }}</p>
<ul>
  <li *ngFor="let value of dataList$ | async">{{value}}</li>
</ul>
<div *ngIf="dataUser$ | async as user">
  <p>{{ user.name }} {{ user.company }}</p>
  <p>{{ user | json }}</p>
</div>
```

examples/lesson02-rxjs-and-routing/projects/async-pipe/src/app/app.component.html

ACCESSING ROUTE INFORMATION

- Often, applications want to pass information from one component to another
- The `ActivatedRoute` interface can be used to extract information
  - The `snapshot` property contains information about a route at a particular moment in time
  - The `paramMap` property is an `Observable` that contains all required and optional parameters specific to the route
- Inject `ActivatedRoute` in the constructor of the components that need to access route information

ACCESSING ROUTE INFORMATION

```
1  import { Component, Inject, OnInit } from '@angular/core';
2  import { ActivatedRoute, ParamMap } from '@angular/router';
3  import { Observable } from 'rxjs';
4  import { tap } from 'rxjs/operators';
5  import { Device, DEVICES } from '../device.type';
6
7  @Component({
8    selector: 'app-device',
9    templateUrl: './device.component.html',
10   styleUrls: ['./device.component.scss']
11 })
12 export class DeviceComponent implements OnInit {
13
14   device: Device
15   paramMap$: Observable<ParamMap>
16
17   constructor(private activatedRoute: ActivatedRoute, @Inject(DEVICES) private
18
19   ngOnInit(): void {
```

examples/lesson02-rxjs-and-routing/projects/routing-advanced/src/app/feature-three/device/device.component.ts

WRAP-UP

- Routing in Angular
  - Routing modules
  - Directives
  - Lazy-loading modules
  - Guards
- Reactive programming
  - Streams and events
  - Operators
- Observables in Angular