RxJS – Reactive Approach

Imperative Vs Declarative

1. Imperative programming is what we having being doing from a while, for example when we write a code in c or c sharp for detecting & print odd number Between 1 to 100. So the pseudo code will be as follows
    a. Loop through numbers between 1 to 100
    b. Check is it divisible by 2.
    c. If it is divisible by 2 then don't print that number
    d. If it is not divisible by 2 then print that number as odd number
2. The above example, there is a problem statement and a solution for that problem but to find the solution we need to follow some steps and that my friend is imperative way of detecting and printing odd numbers between 1 to 100.
3. We tell the computer what needs to be done to achieve the result is Imperative programming.
4. Declarative way or programming is we tell the computer what we want, for example when a problem statement is given such as detect & print odd number btw 1 to 100. The pseudo code would be as follows
    a. Calling PrintOddNumbers(from, to) where from 1 and to would be 100
5. So as you can see that we told the computer to provide us what we want and it just worked.
6. Don't be surprise if under the PrintOddNumber method you will find imperatively written code.
7. Declarative way prompts elegant, sugar coated way of writing things, it hide the complexity of code for example PrintOddNumber method has hidden the complexity of the looping and number should be divisible by 2 by the users, it only states that PrintOddNumber just provide me from & to and I will do the heavy lifting.
8. One more example of declarative way is asking question to ChatGPT and ChatGPT provides you the answer but the complexity of searching the question is hidden inside the ChatGPT's algorithm
9. So why declarative way of writing code because it will simple, human readable, efficient, modular & maintainable.

Declarative way for RxJS – Angular Style

1. When we write Observable.subscribe(), we have converted the Observables into imperative style of writing code

```
ngOnInit(): void {
  this.postsSubscription =  this.postService.post_data_with_category.subscribe(data => {
      this.posts = data;
  })
}
```

2. Understand Change Detection Strategy & Declarative way in Angular.
    a. If Change Detection Strategy is default then the full page will get loaded if any variable is changed this might lead to performance issue in bigger application.
    b. If you have parent component & child component then if any change done in child component will load both parent and child component.
    c. Some time it is needed but sometimes it is not required so we have to use Change detection strategy as OnPush.

d. When we add Change Detection Strategy as OnPush then if changes occurred in child component will not trigger load event of parent component.
e. The component detect changes when a class variable is changed by event or user input but when we use OnPush Strategy it stops doing that so we need to write the code in declarative way
f. One way to detect changes for OnPush strategy, is by calling **detectChanges** method in **ChangeDetectorRef** wherever we need to info the UI that some changes have be performed.
g. And another way is to write declarative style of coding.

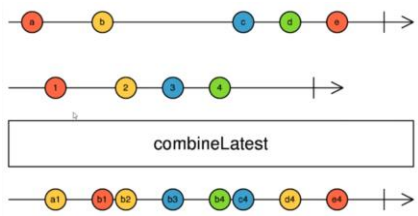
3. Understand why to use unsubscribe for Http request
    a. onNgInit function has a http call request & you don't have a unsubscribe to that http call if you change the page/component, it will not stop till it gets completed.
    b. So if you want to stop the http call request if page is destroyed then onNgDestroy we need to unsubscribe the http call request.
4. Since Observables are lazy until and unless we subscribe the observable it will not work.
    a. So now we have a problem that if we write Observable.subscribe({}) we are writing imperative way of programming.
    b. To avoid this we have a pipe called as **async** as we have done in below example it will take care of subscribe and unsubscribe of observable.

```
  styleUrls: ['./declarative-post.component.scss'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class DeclarativePostComponent {
  dPost$ = this.dPostService.post_data;

  constructor(private dPostService: DeclarativePostService) {
  }
}
```

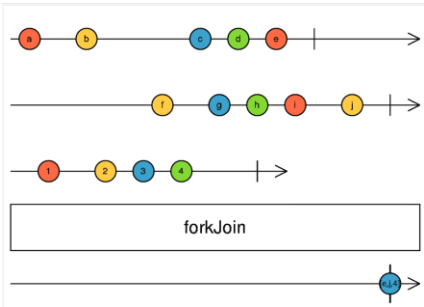
```
<tbody *ngIf="dPost$ | async as posts">
  <tr *ngFor="let post of posts">
    <td>{{post.id}}</td>
    <td>{{post.title}}</td>
    <td>{{post.description}}</td>
    <td>{{post.categoryName}}</td>
  </tr>
</tbody>
```

```
@Injectable({
  providedIn: 'root'
})
export class DeclarativePostService {

  constructor(private http: HttpClient) { }

  get post_data(){
    return this.http
            .get<{[id: string]: IPost}>('https://project-rxjs-default-rtdb.firebaseio.com/posts.json')
            .pipe(map(posts => {
              let postData: IPost[] = [];
              for (let id in posts) {
                postData.push({ ...posts[id], id})
              }
              return postData;
            }));
  }
}
```

5. Use of **combineLatest** & **forkJoin**
    a. **combineLatest –** It combine two or more observables but for combineLatest to work it is necessary that all the observable should at least emit one value first. After that it will provide the latest value whenever one of the observable will emit the data.



    b. **forkJoin** – It combine two or more observables but it will not emit any observable until all the observable have finished emitting the data.



    c. E.g. combineLatest – Just replace keyword “combineLatest” with “forkJoin” it work as expected.

```
  get post_with_category() {
    return combineLatest([
      this.post_data,
      this.dCatergoryService.category_data,
    ]).pipe(
      map(([posts, categories]) => {
        return posts.map((post) => {
          return {
            ...post,
            categoryName: categories.find((category) => category.id == post.categoryId)
              ?.title,
          } as IPost;
        })
      })
    );
  }
}
```

6. When we have used observables and Change Strategy as OnPush
    a. Assume, on change of dropdown we need to change the data in the grid. If we use normal approach of assigning the class variable to the selected item in the onChange method, the change will not trigger and in turn the data will not be filter accordingly.

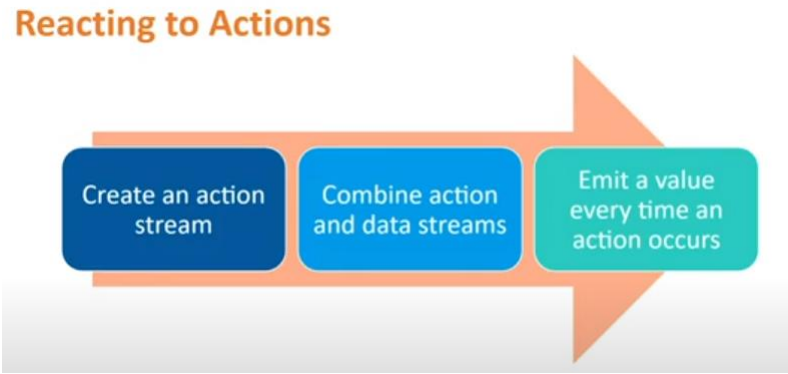
```
export class DeclarativePostComponent {
  selectedCategoryId = '';

  dPost$ = this.dPostService.post_with_category;
  dCategory$ = this.dCategoryService.category_data;

  dFilterData$ = this.dPost$.pipe(
    map((posts) => {
      return posts.filter((post) =>
        this.selectedCategoryId
          ? post.categoryId === this.selectedCategoryId
          : true
      );
    })
  );
```

```
  onCategoryChange(event: Event) {
    let selectedCategoryId = (event.target as HTMLSelectElement).value;
    this.selectedCategoryId = selectedCategoryId;
    console.log(this.selectedCategoryId);
  }

  constructor(
    private dPostService: DeclarativePostService,
    private dCategoryService: DeclarativeCategoryService
  ) {}
}
```

```
<div class="row">
  <div class="col-md-4">
    <select class="form-select" (change)="onCategoryChange($event)">
      <option value="">Select Caterogy</option>
      <option [value]="cat.id" *ngFor="let cat of dCategory$ | async">
        {{cat.title}}
      </option>
    </select>
  </div>
```

b. Component won't detect changes automatically we need to do it via **ChangeDetectorRef** or Convert the variable which will be changing into observable

7. Data Stream Vs Action Stream
    a. When observable stream emits data and gets completed it can be termed as **Data Stream**. It won't re-execute again and again E.g. http requests
    b. When Observable stream emits a data whenever action occurs it can be termed as **Action stream.** It will be always **live** listening for the changes. E.g. user selecting option in dropdown list or user inputting data in textbox.
        i. The streams which are always active and listening for the changes.
        ii. It will never be completed.
8. Combining the Data Stream and Action Stream i.e. basically reacting to user actions.
    a. To combine data stream and action stream we can use combineLatest Operator.
    b. combineLatest will execute whenever action or data observable emits the data.
    c. Since data stream gets completed after emitting the data but action stream is live and it never gets completed. So whenever any action is emitted the combineLatest will be executed and we get our result as observable.



9. Convert User actions to Action observable stream by using fromEvent or Subject or BehaviourSubject.
    a. fromEvent – fromEvent(eventTarget or nativeElement, 'action name(focus,blur,keyup,click etc.)')
        i. e.g.

```
scroll() {
  const source = fromEvent(window, 'scroll');
  source.subscribe(val => console.log(val));
}
```

    b. Subject or BehaviourSubject
        i. Observables are unicast whereas the subjects are multicast i.e. multiple subscriber can share one stream.
        ii. Subject and BehaviourSubject acts both ways i.e. Observer (next, error and completed) and Observable.
        iii. BehaviourSubject can hold a one default value, when it is subscribed, it emits that value immediately. A Subject doesn't hold a value.
        iv. Subject doesn't have a default value, so when it is subscribed it doesn't emits the value since it doesn't have one. so for the subject to trigger and emit the values we need to use this next function e.g. <subjectName>.next(<value>);

10. Demo of combineLatest , BehaviourSubject

```
export class DeclarativePostComponent {
  selectedCatergorySubject = new BehaviorSubject<string>('');
  selectedCatergoryAction$ = this.selectedCatergorySubject.asObservable();

  selectedCategoryId = '';

  dPost$ = this.dPostService.post_with_category;
  dCategory$ = this.dCategoryService.category_data;

  dFilterData$ = combineLatest([
    this.dPost$,
    this.selectedCatergoryAction$,
  ]).pipe(
    map(([posts, selectedCategoryId]) => {
      return posts.filter((post) => {
        return selectedCategoryId ? post.categoryId == selectedCategoryId : true;
      });
    })
  );

  onCategoryChange(event: Event) {
    let selectedCategoryId = (event.target as HTMLSelectElement).value;
    this.selectedCatergorySubject.next(selectedCategoryId);
  }

  constructor(
    private dPostService: DeclarativePostService,
    private dCategoryService: DeclarativeCategoryService
  ) {}
}
```

11. Triggering Changes from component to another component via action stream.
    a. So as we have seen we need Behavior Subject or Subject or fromEvent to convert user select/input into Action stream.
    b. When we declare the Behavior Subject in a component then it can be used for that component only i.e. DropDown Changes which can be used for filtering data in the same component.
    c. On user click or select you need to change another component then we need to declare the Action Stream Observable in some common area and use the output of the change in the required component. i.e. On Click of a button/href, we need to change the data in another component
    d. E.g. we declared the action stream in services and passed the data to the services for filtering or finding the data from one component and in another component we used the filter data via service.

```
<div class="list-group">
  <a href="#"
    class="list-group-item list-group-item-action disabled" aria-current="true">
    List of Catergory
  </a>
  <a href="#"
    (click)="onSelectedPost(post, $event)"
    class="list-group-item list-group-item-action" *ngFor="let post of post$ | async">
    {{post.title}} | {{post.categoryName}}
  </a>
</div>
```

```
export class AltPostsComponent {
  post$ = this.postService?.post_with_category.pipe(map((posts) => {
    return posts.filter(post => post.categoryName !== undefined)
  }));

  constructor(private postService: DeclarativePostService) {
  }

  onSelectedPost(post: IPost, event: Event){
    event.preventDefault();
    post.id && this.postService.selectPost(post.id)
  }
}
```

```
@Component({
  selector: 'app-single-post',
  templateUrl: './single-post.component.html',
  styleUrls: ['./single-post.component.scss'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class SinglePostComponent {
  post$ = this.postService.post;
  constructor(private postService: DeclarativePostService) { }
}
```

```
export class DeclarativePostService {
  private selectedPostSubject: BehaviorSubject<string> =
    new BehaviorSubject<string>('');
  selectedPostAction$ = this.selectedPostSubject.asObservable();

  constructor(
    private http: HttpClient,
    private dCatergoryService: DeclarativeCategoryService
  ) {}

  selectPost(postId: string) {
    console.log(postId);
    this.selectedPostSubject.next(postId);
  }

  get post() {
    return combineLatest([
      this.post_with_category,
      this.selectedPostAction$,
    ]).pipe(
      map(([posts, selectedId]) => {
        return posts.find((post) => post.id === selectedId) as IPost;
      })
    );
  }
```

12. Error handling – Via Action Stream Observable & catchError Operators
    a. So normally error handling was done via subscribe i.e. inside the observer Object

```
post$ = this.postService.post.subscribe({
  next : (data) => {},
  error: (error) => { /* showing error logic */ }
});
```

    b. But since we don't want to write imperative code we need to use catchError operator and Action Stream observables.
    c. So let's see how we do it when we deal with some http request and that request has some issue.
        i. Adding catchError in the pipe operator of http request.

```
get post_with_category() {
  return combineLatest([
    this.post_data,
    this.dCatergoryService.category_data,
  ]).pipe(
    map(([posts, categories]) => {
      return posts.map((post) => {
        return {
          ...post,
          categoryName: categories.find(
            (category) => category.id == post.categoryId
          )?.title,
        } as IPost;
      });
    }),
    catchError(this.handleError)
  );
}
```

```
handleError(error: Error){
  // Write your handle logic here
  return throwError(() => {
    return 'something is not right, try after some time.'
  })
}
```

        ii. Adding catchError operator where we were are consuming the http Requests observable. –

```
@Component({
  selector: 'app-single-post',
  templateUrl: './single-post.component.html',
  styleUrls: ['./single-post.component.scss'],
})
export class SinglePostComponent {
  errorMessage = ''
  post$ = this.postService.post
          .pipe(catchError((error) => {
            this.errorMessage = error;
            return EMPTY;
          }));
```

```
Go to component
<div class="row" *ngIf="errorMessage">
  <div class="col-md-12">
    {{errorMessage}}
  </div>
</div>
<div *ngIf="post$ | async as post">
  <div>Id : {{post.id}}</div>
  <div>Title : {{post.title}}</div>
```

We have not added changeDetection Strategy (OnPush) yet. so we will be able to view the error(s). In case we add onPush strategy then we won't we able to see the error message.

        iii. Add changeDetection Strategy OnPush & Implement Action Stream Observable.

```
@Component({
  selector: 'app-single-post',
  templateUrl: './single-post.component.html',
  styleUrls: ['./single-post.component.scss'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class SinglePostComponent {
  errorSubject = new BehaviorSubject<string>('');
  errorMessageAction$ = this.errorSubject.asObservable();
  post$ = this.postService.post
          .pipe(catchError((error) => {
            this.errorSubject.next(error);
            return EMPTY;
          }));
  constructor(private postService: DeclarativePostService)
}
```

```
Go to component
<div class="row" *ngIf="errorMessageAction$ | async as errorMessage">
  <div class="col-md-12">
    {{errorMessage}}
  </div>
</div>
<div *ngIf="post$ | async as post">
  <div>Id : {{post.id}}</div>
  <div>Title : {{post.title}}</div>
  <div>Description : {{post.description}}</div>
  <div>Category Name : {{post.categoryName}}</div>
</div>
```

        iv. So this how we handled error for http request in declarative manner.

13. Caching operator for Observable – for better response – share() and shareReplay() operator.
    a. Cache Operator like share() and shareReplay() are used for minimize repeated API calls to the server.
    b. share() keeps the count of subscriber and when subscriber counts reaches to zero then share will unsubscribe from the source observable and resets its inner observable(the subject).
        i. Late subscriber will trigger a new subscription to the source observable i.e. a new execution of the source observable.

```
import { defer, delay, of, share, shareReplay, tap } from 'rxjs';

console.clear();

const source$ = defer(() => of(Math.round(Math.random() * 100))).pipe(
  tap((x) => console.log('Processing: ', x)),
  delay(1000),
  // if you remove share then
  // each subscription will have its own execution of the source observable
  share()
);

source$.subscribe((x) => console.log('subscription 1: ', x));
source$.subscribe((x) => console.log('subscription 2: ', x));

setTimeout(
  // this subscription arrives late to the party.
  () => source$.subscribe((x) => console.log('subscription 3: ', x)),
  3500
);
```

```
Console was cleared
Processing: 57
subscription 1: 57
subscription 2: 57
Processing: 92
subscription 3: 92
```

        ii. So as you can see the "Processing: 0" emitted data **57.** So when a late subscriber joined the party the old emitted data **57** was erased/gone/flushed etc. and it triggered the new subscription "Processing: 1" emitted data **92.**
        iii. If you need the old data **57** also then we need to use shareReplay().
    c. shareReplay() doesn't keeps the count of subscriber by default, it won't be able to un-subscribe to the source Observable ever. Unless we use **refCount** option.
        i. So if you don't want memory leak in you code, you can use shareReplay() as shown below.

```
shareReplay({ bufferSize: 1, refCount: true })
```

        ii. If late subscriber subscribes to the source observable late then the old value is maintained and **bufferSize** option is used to maintain how many data needs to be cached.
        iii. E.g.

```
import { defer, delay, of, share, shareReplay, tap } from 'rxjs';

console.clear();

const source$ = defer(() => of(Math.round(Math.random() * 100))).pipe(
  tap((x) => console.log('Processing: ', x)),
  delay(1000),
  // if you remove shareReplay then
  // each subscription will have its own execution of the source observable
  shareReplay({ bufferSize: 1, refCount: true })
);

source$.subscribe((x) => console.log('subscription 1: ', x));
source$.subscribe((x) => console.log('subscription 2: ', x));

setTimeout(
  // this subscription arrives late to the party.
  () => source$.subscribe((x) => console.log('subscription 3: ', x)),
  3500
);
```

```
Console was cleared
Processing: 30
subscription 1: 30
subscription 2: 30
subscription 3: 30
```

d. In **Angular**, share() and shareReplay() behaves a bit differently when Observable is subscribe in the template using **async** pipe operator inside ***ngIf** directive then the **refCount** might reach to **zero** if it is **unsubscribes** automatically which causes it to trigger a new execution of the source Observable.

e. We need to create observable variable as shown below for share() and shareReply() to work.

```
category_data$ = this.http
                .get<{ [id: string]: ICategory }>(
                  'https://project-rxjs-default-rtdb.firebaseio.com/categories.json'
                )
                .pipe(
                  map((categories) => {
                    let categoryData: ICategory[] = [];
                    for (let id in categories) {
                      categoryData.push({ ...categories[id], id });
                    }
                    return categoryData;
                  }),
                  catchError(this.handleError),
                  shareReplay(1)
                );
```

```
get category_data() {
  return this.http
    .get<{ [id: string]: ICategory }>(
      'https://project-rxjs-default-rtdb.firebaseio.com/categories.json'
    )
    .pipe(
      map((categories) => {
        let categoryData: ICategory[] = [];
        for (let id in categories) {
          categoryData.push({ ...categories[id], id });
        }
        return categoryData;
      }),
      catchError(this.handleError),
      shareReplay(1)
    );
}
```

14. Avoid multiple async in the html template. Try to use only one async per template.

a. Multiple async pipe in the html template makes the page to load slower. In simple terms this can lead to performance issue.

b. If you call async pipe inside a *ngFor directive such as below example it will lead to performance issue.

```
<div class="list-group" *ngIf="post$ | async as posts">
  <a href="#"
    class="list-group-item list-group-item-action disabled" aria-current="true">
    List of Catergory
  </a>
  <a href="#"
    (click)="onSelectedPost(post, $event)"
    class="list-group-item list-group-item-action"
    [class]="{active : post.id === (selectedPost$ | async)?.id}"
    *ngFor="let post of posts">
    {{post.title}} | {{post.categoryName}}
  </a>
</div>
```

c. Subscribe count of async pipe is equal to the number of item present in the list and Hence it leads to performance issue.

d. We can negate this by using viewmodel observable at component level.

```
<div class="row" *ngIf="viewModel$ | async as viewModel">
  <div class="col-md-4">
    <div class="list-group">
      <a href="#"
        class="list-group-item list-group-item-action disabled"
        aria-current="true">
        List of Catergory
      </a>
      <a href="#"
        (click)="onSelectedPost(post, $event)"
        class="list-group-item list-group-item-action"
        [class]="{active : post.id === viewModel.selectedPost.id}"
        *ngFor="let post of viewModel.posts">
        {{post.title}} | {{post.categoryName}}
      </a>
    </div>
  </div>
  <div class="col-md-8">
    <app-single-post></app-single-post>
  </div>
```

```
selectedPost$ = this.postService.post$;

post$ = this.postService?.post_with_category$.pipe(
  tap((posts) => {
    posts[0].id && this.postService.selectPost(posts[0].id)
    this.loaderService.hideLoader();
  }),
  map((posts) => {
  return posts.filter(post => post.categoryName !== undefined)
}));

viewModel$ = combineLatest([this.post$, this.selectedPost$])
             .pipe(map(([posts, selectedPost]) =>  {
               return {posts, selectedPost}
             }))

constructor(private postService: DeclarativePostService,
            private loaderService:LoaderService) {
}
```

15. Adding Post Value to Html DOM.
    a. Example

In Declartive-Post Services

```
private postCRUDSubject = new Subject<CRUDAction<IPost>>();
postCRUDAction$ = this.postCRUDSubject.asObservable();
```

```
addPost(post: IPost) {
  this.postCRUDSubject.next({ action: 'add', data: post });
}
```

```
all_post$ = merge(
  this.post_with_category$,
  this.postCRUDAction$
  .pipe(map((data) => {
    return [data.data];
  }))
).pipe(
  scan((posts, value) => {
    return [...posts, ...value];
  }, [] as IPost[])
);
```

Add-Post Component.ts- Button Click

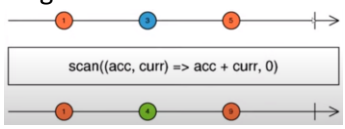
```
onAddPost(){
  let purePost = this.postFormGroup.value as IPost;

  let post = {
    ...purePost,
    categoryName : ''
  } as IPost

  this.postService.addPost(post);
}
```

```
export interface CRUDAction<T>{
  action : 'add' | 'update' | 'delete';
  data : T;
}
```

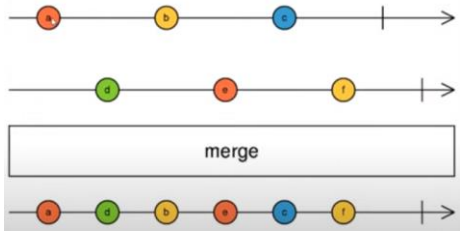
Alt-Posts Component.ts (all_post$)

```
post$ = this.postService?.all_post$.pipe(
  tap((posts) => {
    posts[0].id && this.postService.selectPost(posts[0].id)
    this.loaderService.hideLoader();
  }),
  map((posts) => {
    return posts.filter(post => post.categoryName !== undefined)
}));
```

    b. Scan operator comes under Transformation Operator
        i. Marble Diagram



        ii. For the first operation, it uses the seed data and after the first operation the result is saved in the accumulator.
        iii. Dry Run of the marble diagram
            1. Run 1) Accumulator = 0 & Current = 1 i.e. 0 + 1 = 1.
                a. Result is saved as accumulator.
            2. Run 2) Accumulator = 1 & Current = 3 i.e. 1 + 3 = 4
            3. Run 3) Accumulator = 4 & Current = 5 i.e. 5 + 4 = 9
            4. So this is how scan operator works it can be used to maintain the previously emitted data and perform some action/logic with it.
    c. Merge operator
        i. It concurrently emits all values provided by the input source.
        ii. It stops emitting the data once any of the source observable come across or cause an error.
        iii. Marble Diagram

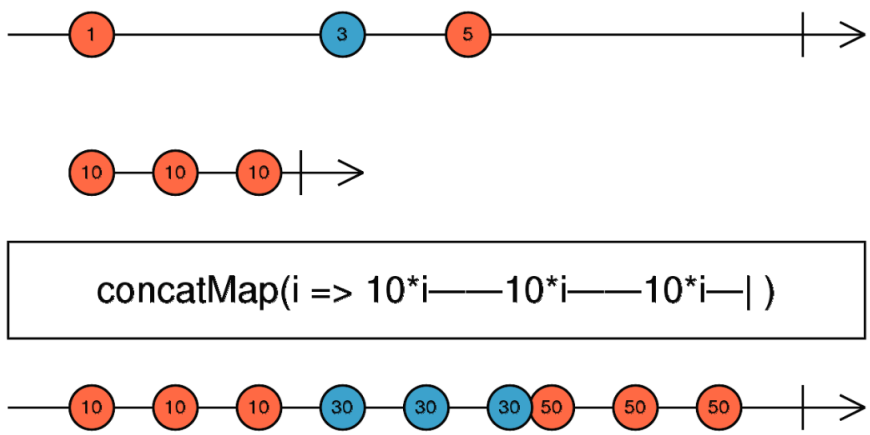


16. Saving Data in Database aka Firebase
    a. E.g. Code for saving the data in the database

```
savePost(postAction: CRUDAction<IPost>) {
  if (postAction.action === 'add') {
    return this.addPostToServer(postAction.data);
  }
  return of(postAction.data);
}

addPostToServer(post: IPost) {
  return this.http.post<{ name: string }>(this.url, post).pipe(
    map((id) => {
      return {
        ...post,
        id: id.name,
      };
    })
  );
}

modifyPosts(posts: IPost[],
            value: IPost[] | CRUDAction<IPost>): IPost[] {
  if(!(value instanceof Array)){
    if(value.action === 'add'){
      return [...posts, value.data]
    }
  } else {
    return value;
  }

  return posts;
}
```

```
all_post$ = merge(
  this.post_with_category$,
  this.postCRUDAction$.pipe(
    concatMap((postAction) =>
      this.savePost(postAction).pipe(
        map((post) => ({
          ...postAction,
          data: post,
        }))
      )
    )
  )
).pipe(
  scan((posts, value) => {
    return this.modifyPosts(posts, value);
  }, [] as IPost[])
);
```

    b. concatMap operator is a flattening operator
        i. Marble Diagram



        ii. It is combination of map and concatAll operator.
        iii. So it emits the value in a sequential manner.
        iv. If order is important then we should use the concatMap operator.
        v. concatMap allows to execute all requests in a sequence – only once the pervious request has been completed, a new request is initiated via subscription.

vi. E.g.

```
const urls = [
   'https://api.mocki.io/v1/0350b5d5',
   'https://api.mocki.io/v1/ce5f60e2'
];

from(urls).pipe(
   concatMap((url) => {
       return fromFetch(url);
   })
).subscribe((response) => console.log(response.status));
```

vii. So url[0] will be initiated and processed and after the response is printed in the console the second url[1] will be initiated i.e. subscribed and processed and then response is printed .

viii. If error occurs while emitting or processing the value then it will stop propagating the data. We can use catchError operator to handle the error and pass EMPTY observable.

17. Use concatMap to trigger the observable, process and extract data from the observable

a. E.g. – Adding category name in the post via categories observable.

```
savePost(postAction: CRUDAction<IPost>) {
  if (postAction.action === 'add') {
    return this.addPostToServer(postAction.data).pipe(
      concatMap(post =>
        this.dCatergoryService.category_data$.pipe(
          map((categories) => {
            return {
              ...post,
              categoryName: categories.find((x) => x.id == post.categoryId)?.title
            };
          })
        )
      )
    );
  }
  return of(postAction.data);
}
```

b. concatMap can be used to trigger an observable inside the pipe.

18. How to populate the formGroup without subscribe.
    a. E.g.

```
export class UpdatePostComponent {

  post$ = this.postService
            .post$
            .pipe(tap(postData => {
                postData && this.postFormGroup.patchValue(postData);
            }));

  postFormGroup = this.createPostFormGroup();

  categories$ = this.catergoryService.category_data$;
```

```
createPostFormGroup() {
  return this.formBuilder.group({
    title: new FormControl<string | null>(null,{
      nonNullable: true,
      validators: [Validators.required]
    }),
    description: new FormControl<string>(''),
    categoryId: new FormControl<string | null>('',{
      validators: [Validators.required]
    }),
  });
}
```

```
Go to component
<div>
  <h3>Update Post</h3>
<div *ngIf="post$ | async">
  <form [formGroup]="postFormGroup" (submit)="onUpdatePost()">
    <div class="my-2">
      <label>Title</label>
      <input title="Title" type="text" class="form-control" formControlName="title"/>
    </div>
```

    b. Tap operator is used to perform action outside of the observable. We can hide, show and perform logging, setValue or patchValues of the formGroup.

```
export class SinglePostComponent {
  showUpdatePost = false;
  errorSubject = new BehaviorSubject<string>('');
  errorMessageAction$ = this.errorSubject.asObservable();
  post$ = this.postService.post$
+           .pipe(
+             tap(() => this.showUpdatePost = false),
+             catchError((error) => {
              this.errorSubject.next(error);
              return EMPTY;
            }));
  constructor(private postService: DeclarativePostService) { }

  onUpdatePost(){
    this.showUpdatePost = true;
  }
```

19. Update the data in database in angular using http patch and rxjs

a. E.g. component level changes

```
<div *ngIf="post$ | async">
  <form [formGroup]="postFormGroup" (submit)="onUpdatePost()">
    <div class="my-2 visually-hidden">
      <label>Id</label>
      <input title="Id" type="text" class="form-control" formControlName="id"/>
    </div>
    <div class="my-2">
```

```
createPostFormGroup() {
  return this.formBuilder.group({
    id: new FormControl<string | null>(null),
    title: new FormControl<string | null>(null,{
      nonNullable: true,
      validators: [Validators.required]
    }),
    description: new FormControl<string>(''),
    categoryId: new FormControl<string | null>('',{
      validators: [Validators.required]
    }),
  });
}

onUpdatePost(){
  let purePost = this.postFormGroup.value as IPost;

  this.postService.updatePost(purePost);
}
```

b. Service Level changes

```
updatePost(post: IPost) {
  this.postCRUDSubject.next({ action: 'update', data: post });
}
```

```
savePost(postAction: CRUDAction<IPost>) {
  let postObservable$! : Observable<IPost>;

  if (postAction.action === 'add') {
    postObservable$! = this.addPostToServer(postAction.data)
  }

  if (postAction.action === 'update') {
    postObservable$! = this.updatePostToServer(postAction.data)
  }

  return postObservable$.pipe(
    concatMap(post =>
      this.dCatergoryService.category_data$.pipe(
        map((categories) => {
          return {
            ...post,
            categoryName: categories
                          .find((x) => x.id == post.categoryId)?.title
          };
        })
      )
    )
  );
}
```

```
updatePostToServer(post: IPost) {
  return this.http.patch<IPost>(this.postPatchDeleteURL+`${post.id}.json`, post);
}

modifyPosts(posts: IPost[], value: IPost[] | CRUDAction<IPost>): IPost[] {
  if (!(value instanceof Array)) {
    if (value.action === 'add') {
      return [...posts, value.data];
    }
    if (value.action === 'update') {
      return posts.map(post =>
        post.id === value.data.id ? value.data : post);
    }
  } else {
    return value;
  }

  return posts;
}
```

20. Delete the data using http delete and rxjs.

a. E.g. Component Level Changes

```
<div class="col-md-4">
  <button class="btn btn-warning" (click)="onUpdatePost()">Update Post</button>
        
  <button class=" btn btn-danger" (click)="onDeletePost(post)">Delete Post</button>
</div>
```

```
onDeletePost(post: IPost){
  if(confirm('Are you sure you want to delete the data?')){
    this.postService.deletePost(post)
  }
}
```

b. Service Level changes

```
deletePost(post: IPost) {
  this.postCRUDSubject.next({ action: 'delete', data: post });
}
```

```
savePost(postAction: CRUDAction<IPost>) {
  let postObservable$! : Observable<IPost>;

  if (postAction.action === 'add') {
    postObservable$! = this.addPostToServer(postAction.data)
  }

  if (postAction.action === 'update') {
    postObservable$! = this.updatePostToServer(postAction.data)
  }

  if (postAction.action === 'delete') {
    return this.deletePostToServer(postAction.data)
              .pipe(map(post => postAction.data))
  }

  return postObservable$.pipe(
    concatMap(post =>
      this.dCatergoryService.category_data$.pipe(
        map((categories) => {
          return {
            ...post,
            categoryName: categories
                          .find((x) =>
                                x.id == post.categoryId)?.title
          };
        })
      )
    )
  );
}
```

```
deletePostToServer(post: IPost) {
  return this.http.delete(this.postPatchDeleteURL+`${post.id}.json`);
}

modifyPosts(posts: IPost[], value: IPost[] | CRUDAction<IPost>): IPost[] {
  if (!(value instanceof Array)) {
    if (value.action === 'add') {
      return [...posts, value.data];
    }
    if (value.action === 'update') {
      return posts.map(post =>
        post.id === value.data.id ? value.data : post);
    }
    if (value.action === 'delete') {
      return posts.filter(post => post.id !== value.data.id);
    }
  } else {
    return value;
  }
  return posts;
}
```

21. Getting Route Param Data and Populating the Form using it.
    a. E.g. In UI, it is necessary to using async pipe on vm$

```
postFormGroup = this.createPostFormGroup();

categories$ = this.catergoryService.category_data$;

selectedPost$ = this.router.paramMap.pipe(map(paramMaps => {
  let id = paramMaps.get('id');
  this.postService.selectPost(id + '');
  return id;
}))

post$ = this.postService.post$.pipe(tap(postData => {
  postData && this.postFormGroup.patchValue(postData);
}))

vm$ = combineLatest([this.selectedPost$, this.post$]);

constructor(
  private formBuilder: FormBuilder,
  private catergoryService: DeclarativeCategoryService,
  private postService: DeclarativePostService,
  private router : ActivatedRoute
) {}
```

```
<div *ngIf=" vm$ | async as vm">
  <form [formGroup]="postFormGroup" (submit)="onPostSubmit()">
    <div class="my-2 visually-hidden">
      <label>Id</label>
      <input title="Id" type="text" class="form-control" formControlName="id"/>
    </div>
    <div class="my-2">
      <label>Title</label>
      <input title="Title" type="text" class="form-control" formControlName="title"/>
    </div>
    <div class="my-2">
      <label>Description</label>
      <textarea class="form-control" title="Description" formControlName="description"
    </div>
```

22. Add and Update the Data using Service
    a. E.g.

UI Side changes

```
<div *ngIf=" vm$ | async as vm">
  <h3>{{postId ? 'Update Post' : 'Add Post'}}</h3>
```

```
<button type="submit" class="btn btn-primary" [disabled]="validateForm()">
  {{postId ? 'Update Post' : 'Add Post'}}
</button>
```

post-form.component.ts

```
postId : string| null = null;

postFormGroup = this.createPostFormGroup();

categories$ = this.catergoryService.category_data$;

selectedPost$ = this.router.paramMap.pipe(map(paramMaps => {
  let id = paramMaps.get('id');
  if(id){
    this.postId = id;
  }
  this.postService.selectPost(id + '');
  return id;
}))
```

```
onPostSubmit() {
  let purePost = this.postFormGroup.value as IPost;
  if(this.postId){
    this.postService.updatePost(purePost);
  } else {
    this.postService.addPost(purePost);
  }
}
```

DeclartivePost.component.ts

```
dPost$ = this.dPostService.post_with_category$;
dCategory$ = this.dCategoryService.category_data$
            .pipe( catchError((error) => {
              this.errorSubject.next(error);
              return EMPTY;
            }));
```

```
dPost$ = this.dPostService.all_post$;
dCategory$ = this.dCategoryService.category_data$
            .pipe( catchError((error) => {
              this.errorSubject.next(error);
              return EMPTY;
            }));
```

23. Adding Global Notification Services for Showing Success and Error Messages

a. E.g.

```
export class NotificationService {
  private successMessageSubject: Subject<string> = new Subject<string>();
  successMessageAction$ = this.successMessageSubject.asObservable();

  private errorMessageSubject: Subject<string> = new Subject<string>();
  errorMessageAction$ = this.errorMessageSubject.asObservable();

  setSuccessMessage(message: string) {
    this.successMessageSubject.next(message);
  }

  setErrorMessage(message: string) {
    this.errorMessageSubject.next(message);
  }

  clearSuccessMessage() {
    this.setSuccessMessage('');
  }

  clearErrorMessage() {
    this.setErrorMessage('');
  }

  clearAllMessage() {
    this.clearSuccessMessage();
    this.clearErrorMessage();
  }

  constructor() {}
}
```

24. Use the Notification Service

a. E.g.

app.component.html

```
<div class="container">
  <div class="alert alert-success"
       *ngIf="successMessage$ | async as successMessage">
       {{successMessage}}
  </div>
  <div class="alert alert-danger"
       *ngIf="errorMessage$ | async as errorMessage">
       {{errorMessage}}
  </div>
</div>
```

app.component.ts

```
export class AppComponent {
  title = 'reactive-practs';
  showLoader$ = this.loaderService.loadingAction$;
  errorMessage$ = this.notificationService
                    .errorMessageAction$.pipe(
                      tap((message)=>{
                        message && this.hideTheMessage()
                      })
                    );
  successMessage$ = this.notificationService
                      .successMessageAction$.pipe(
                        tap((message)=>{
                          message && this.hideTheMessage()
                        })
                      );

  hideTheMessage(){
    setTimeout(
      () => this.notificationService.clearAllMessage(),
      4000
    )
  }
```

DeclarativePostService

```
savePost(postAction: CRUDAction<IPost>) {
  let postObservable$! : Observable<IPost>;

  if (postAction.action === 'add') {
    postObservable$! = this.addPostToServer(postAction.data)
                        .pipe(tap((post) => {
                          this.notificationService
                            .setSuccessMessage('Post added successfully.')
                        }))
  }

  if (postAction.action === 'update') {
    postObservable$! = this.updatePostToServer(postAction.data)
                        .pipe(tap((post) => {
                          this.notificationService
                            .setSuccessMessage('Post updated successfully.')
                        }))
  }

  if (postAction.action === 'delete') {
    return this.deletePostToServer(postAction.data)
              .pipe(map(post => postAction.data),
                    tap((post) => {
                      this.notificationService
                        .setSuccessMessage('Post updated successfully.')
                    }))
  }
```

25. Routing back to Details page.
    a. E.g.

declarative-post.service.ts

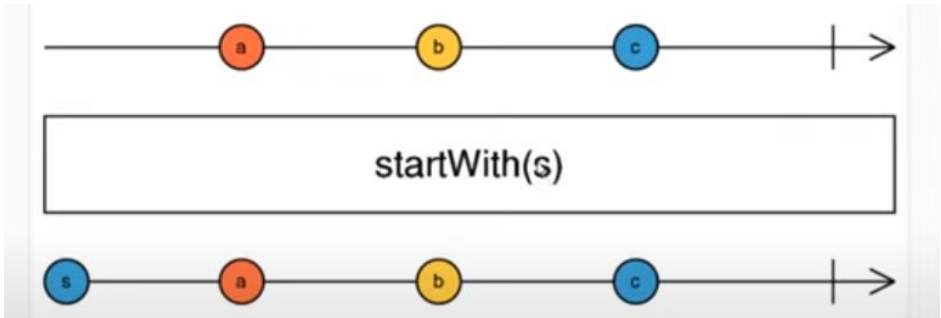
```
private postCRUDNotification = new Subject<boolean>();
postCRUDNotificationAction$ = this.postCRUDNotification.asObservable();
```

```
savePost(postAction: CRUDAction<IPost>) {
  let postObservable$! : Observable<IPost>;

  if (postAction.action === 'add') {
    postObservable$! = this.addPostToServer(postAction.data)
                        .pipe(tap((post) => {
                          this.notificationService
                            .setSuccessMessage('Post added successfully.')
                          this.postCRUDNotification.next(true);
                        }))
  }

  if (postAction.action === 'update') {
    postObservable$! = this.updatePostToServer(postAction.data)
                        .pipe(tap((post) => {
                          this.notificationService
                            .setSuccessMessage('Post updated successfully.')
                            this.postCRUDNotification.next(true);
                        }))
  }

  if (postAction.action === 'delete') {
    return this.deletePostToServer(postAction.data)
              .pipe(map(post => postAction.data),
                    tap((post) => {
                      this.notificationService
                        .setSuccessMessage('Post updated successfully.')
                        this.postCRUDNotification.next(true);
                    }))
```

post-form.component.ts

```
notification$ = this.postService
                  .postCRUDNotificationAction$
                  .pipe(
                    startWith(false),
                    tap(message => {
                      message && this.router
                          .navigateByUrl('/declarative-post')
                    })
                  );

vm$ = combineLatest([this.selectedPost$,
                     this.post$,
                     this.notification$]);

constructor(
  private formBuilder: FormBuilder,
  private catergoryService: DeclarativeCategoryService,
  private postService: DeclarativePostService,
  private router: Router,
  private route : ActivatedRoute
) {}
```

    b. startWith Operator –
        i. Marble Diagram



        ii. As you can see in the above marble diagram startWith is supplied with “s” to the observable and afterwards the observable started to get the data “a”, “b” & “c”. So the data emitted was “s”, “a”, “b” & “c”.
        iii. Sometimes, we must use the observable in combineLatest but combineLatest starts emitting the values once all the observable has emitted once. So to initialize the observable with empty value or initial value, we can use startWith operator.

26. Notification for Error Messages.
    a. E.g.

post-form.component.ts

```
post$ = this.postService.post$.pipe(tap(postData => {
  postData && this.postFormGroup.patchValue(postData);
}), catchError(error => {
  this.notificationService.setErrorMessage(error);
  // postCRUDSubject in the declartive-post.service
  // stops working it will not invoke the all_post$
  // in-turn you will not able to see the any error
  // msgs. Hence I redirected it to details page
  // as workaround.
  this.router.navigateByUrl('/declarative-post')
  return EMPTY;
}))
```

declartive-post.service.ts

```
all_post$ = merge(
  this.post_with_category$,
  this.postCRUDAction$.pipe(
    concatMap((postAction) =>
      this.savePost(postAction).pipe(
        map((post) => ({
            ...postAction,
            data: post,
        }))
      )
    )
  )
).pipe(
  scan((posts, value) => {
    return this.modifyPosts(posts, value);
  }, [] as IPost[]),
  shareReplay({ bufferSize: 1, refCount: true }),
  catchError(this.handleError)
);
```

declartive-post.service.ts

```
savePost(postAction: CRUDAction<IPost>) {
  let postObservable$! : Observable<IPost>;

  if (postAction.action === 'add') {
    postObservable$! = this.addPostToServer(postAction.data)
                        .pipe(tap((post) => {
                          this.notificationService
                            .setSuccessMessage('Post added successfully.')
                          this.postCRUDNotification.next(true);
                        }), catchError(this.handleError))
  }

  if (postAction.action === 'update') {
    postObservable$! = this.updatePostToServer(postAction.data)
                        .pipe(tap((post) => {
                          this.notificationService
                            .setSuccessMessage('Post updated successfully.')
                            this.postCRUDNotification.next(true);
                        }), catchError(this.handleError))
  }

  if (postAction.action === 'delete') {
    return this.deletePostToServer(postAction.data)
              .pipe(map(post => postAction.data),
                    tap((post) => {
                      this.notificationService
                        .setSuccessMessage('Post deleted successfully.')
                        this.postCRUDNotification.next(true);
                    }), catchError(this.handleError))
  }
```

Added catchError operator where ever neccessary.

27. T
28. T

Reference:

- Leela Web Dev – YouTube Channel
- Images – Computer Baba.
- NG-MY-Conference - Deborah Kurata
- https://www.bitovi.com/blog/always-know-when-to-use-share-vs.-sharereplay