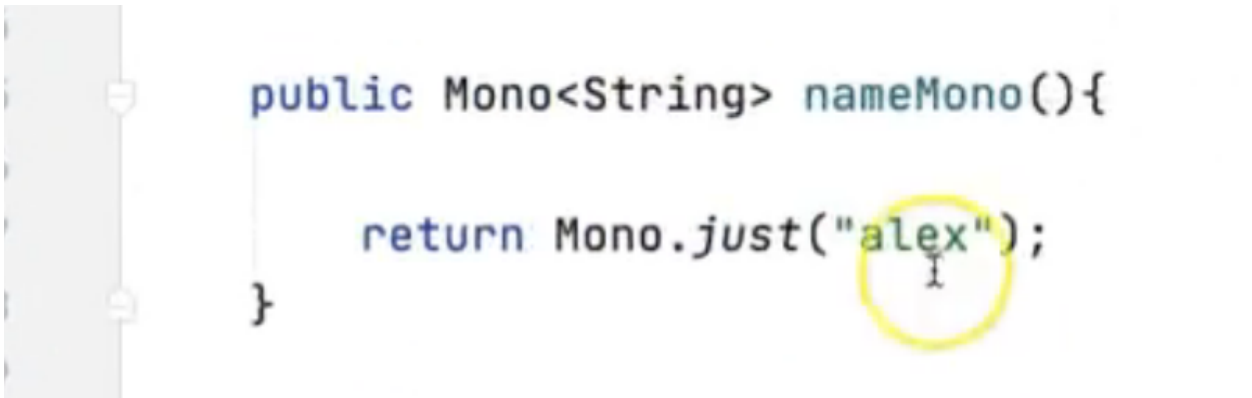
Reactive Programming dependencies :

```
dependencies {
    implementation("io.projectreactor:reactor-core:3.4.0")

    //log - logback
    implementation("ch.qos.logback:logback-classic:1.2.3")

    //reactor-debug-agent
    implementation('io.projectreactor:reactor-tools:3.4.0')

    //testing
    testImplementation("io.projectreactor:reactor-test:3.4.0")
    testImplementation('org.junit.jupiter:junit-jupiter:5.5.1')
    testImplementation("org.mockito:mockito-core:3.2.4")
    testImplementation("org.mockito:mockito-junit-jupiter:3.2.4")

}
```

The reactive library here is reactive-core. Spring webflux uses behind the screen for us. reactor-test to test non-blocking code.

```
public class FluxAndMonoGeneratorService {

    public Flux<String> namesFlux() {

        return Flux.fromIterable(List.of("alex", "ben","chloe")); // db or a remote service call
    }


    public static void main(String[] args) {

        FluxAndMonoGeneratorService fluxAndMonoGeneratorService = new FluxAndMonoGeneratorService();

        fluxAndMonoGeneratorService.namesFlux()
                .subscribe(name -> {
                    System.out.println("Name is : " + name);
                });

    }
}
```

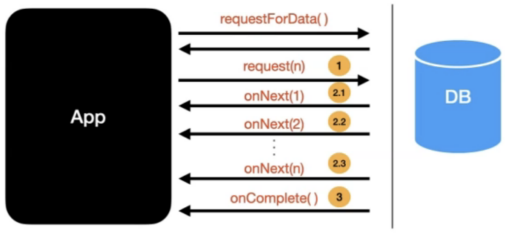


```
        fluxAndMonoGeneratorService.nameMono()
                .subscribe(name->{
                    System.out.println("Mono name is : " + name);
                });

    }
```

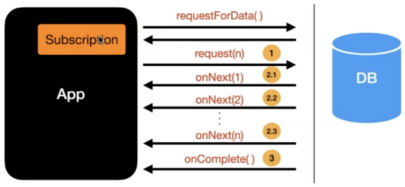
Until subscribe nothing will happen

```
> Task :FluxAndMonoGeneratorService.main()
04:33:46.755 [main] DEBUG reactor.util.Loggers$LoggerFactory - Using Slf4j logging framework
Name is : alex
Name is : ben
Name is : chloe
Mono name is : alex
```

Reactive Streams Events



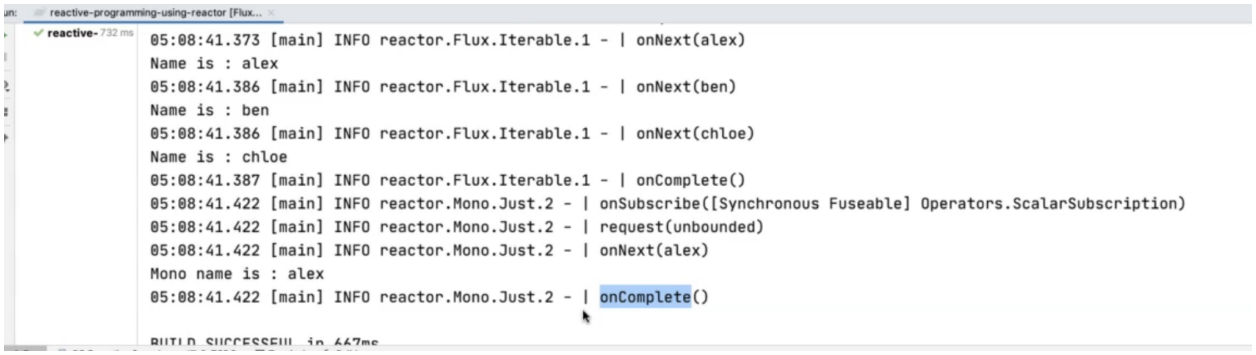
Reactive Streams Events



For the second call (onSubscribe) in the above diagram it will create a subscription object in the subscriber.

```java
public Flux<String> namesFlux() {

    return Flux.fromIterable(List.of("alex", "ben","chloe"))
            .log(); // db or a remote service call
}

public Mono<String> nameMono(){

    return Mono.just("alex");
}
```

Add log method for both namesFlux and nameMono to see data flow between subscriber and pusblisher



Flow of data for flux and mono

Use StepVerifier from reactive test to test non blocking code.

```
//then
StepVerifier.create(namesFlux)
        //.expectNext("alex", "ben","chloe")
        .expectNextCount(3)
        .verifyComplete();
}
```

```
//then
StepVerifier.create(namesFlux) StepVerifier.FirstStep<String>
        //.expectNext("alex", "ben","chloe")
        //.expectNextCount(3)
        .expectNext("alex") StepVerifier.Step<String>
        .expectNextCount(2)
        .verifyComplete();
```

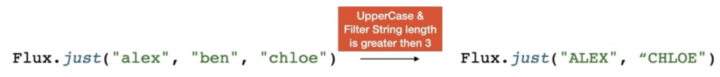
Combination of both exepctNext and expectNextCount.

The create function will take case of subscribe method on publisher

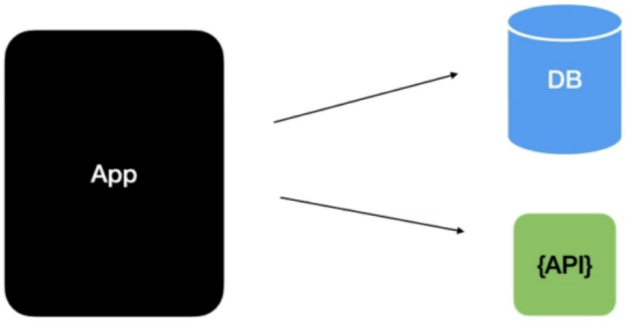
Transforming Data Using Operators in Project Reactor

Why Transform Data ?

- It is pretty common for apps to transform data from its original form



These are simple data transformations but in reality it is different when we subscribe data from DB and other API



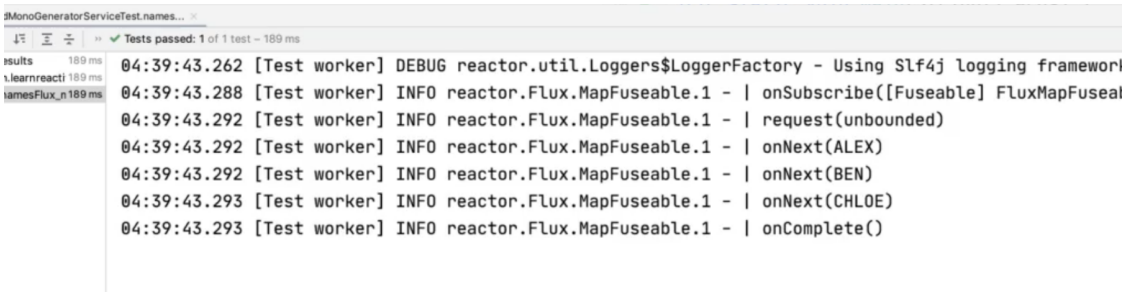
map() Operator

- Used to transform the element from one form to another in a Reactive Stream
- Similar to the map() operator in Streams API

```
20
21      public Flux<String> namesFlux_map() {
22
23          return Flux.fromIterable(List.of("alex", "ben","chloe"))
24                  .map(String::toUpperCase)
25                  //.map(s -> s.toUpperCase())
26                  .log(); // db or a remote service call
27      }
28
```

Line number 24 and and 25

```
27  @Test
28  void namesFlux_map() {
29      //given
30
31      //when
32      var namesFlux = fluxAndMonoGeneratorService.namesFlux_map();
33
34      //then
35      StepVerifier.create(namesFlux)
36              .expectNext("ALEX", "BEN", "CHLOE")
37              .verifyComplete();
38  }
39
```

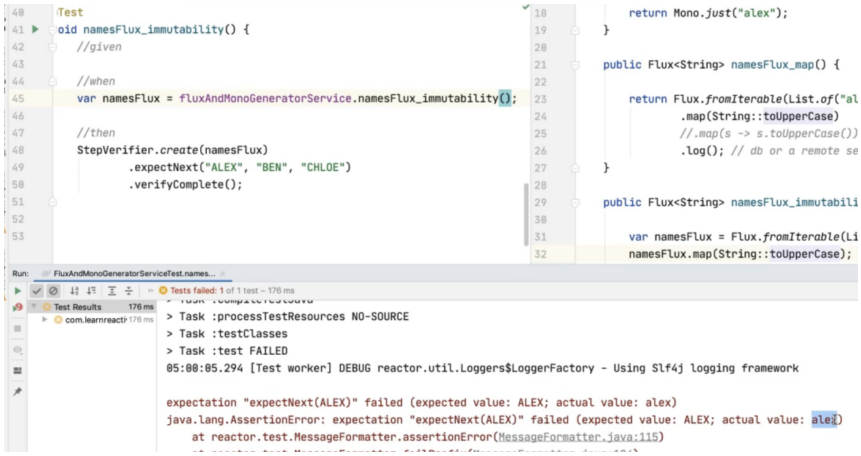


Reactive Streams are Immutable

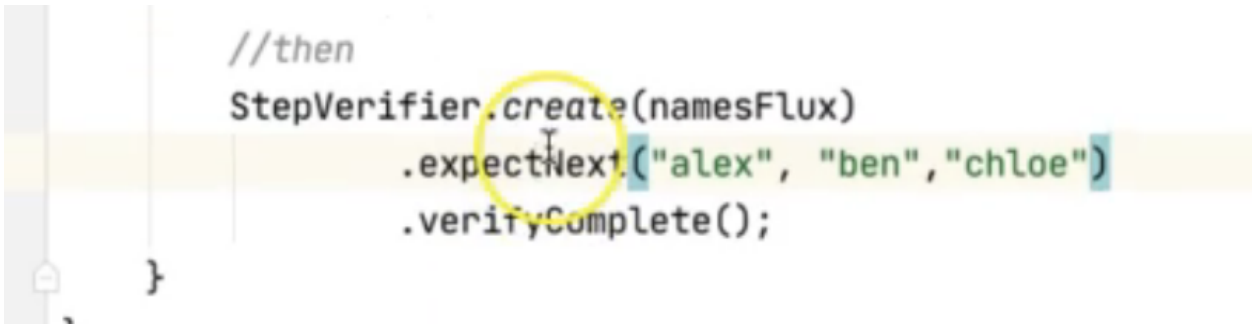
```
public Flux<String> namesFlux_immutability() {

    var namesFlux = Flux.fromIterable(List.of("alex", "ben","chloe"));
    namesFlux.map(String::toUpperCase);
    return namesFlux;
}
```

Reactive streams are immutable and they will not change the value of original data.

```
39
40      @Test
41      void namesFlux_immutability() {
42          //given
43
44          //when
45          var namesFlux = fluxAndMonoGeneratorService.namesFlux_immutability
46
47          //then
48          StepVerifier.create(namesFlux)
49                  .expectNext("ALEX", "BEN", "CHLOE")
50                  .verifyComplete();
51      }
52  }
```

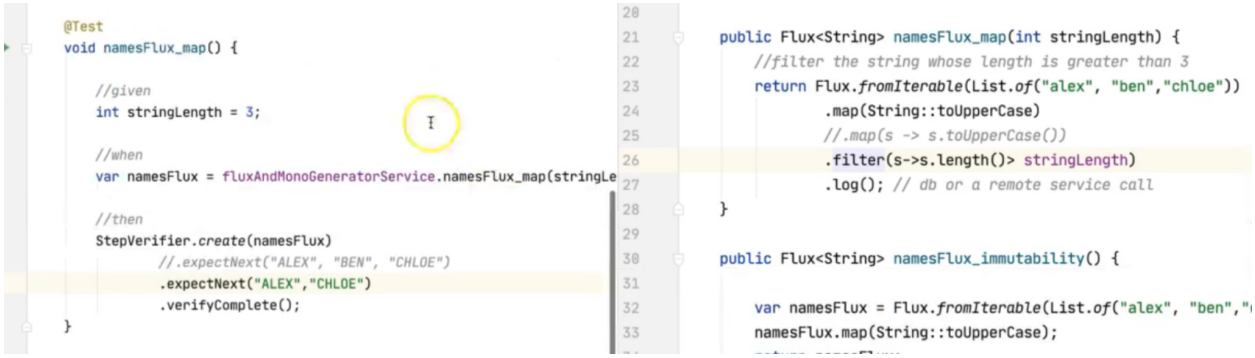


Test failed because of immutability. This will pass if we change expected data to lower case.



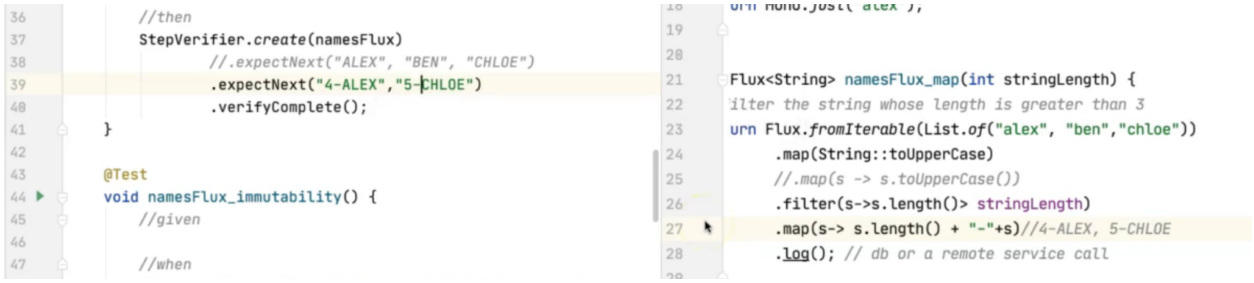
filter() Operator

- Used to filter elements in a Reactive Stream
- Similar to the filter() operator in Streams API



Right side actual code and left side test case.

Some changes to filtered values



```
        //then
        StepVerifier.create(namesFlux)
                //.expectNext("ALEX", "BEN", "CHLOE")
                .expectNext("4-ALEX","5-CHLOE")
                .verifyComplete();
    }

    @Test
    void namesFlux_immutability() {
        //given

        //when
```

```
Flux<String> namesFlux_map(int stringLength) {
ilter the string whose length is greater than 3
urn Flux.fromIterable(List.of("alex", "ben","chloe"))
     .map(String::toUpperCase)
     //.map(s -> s.toUpperCase())
     .filter(s->s.length()> stringLength)
     .map(s-> s.length() + "-"+s)//4-ALEX, 5-CHLOE
     .log(); // db or a remote service call
```

Left is test case and right is actual code

flatMap()

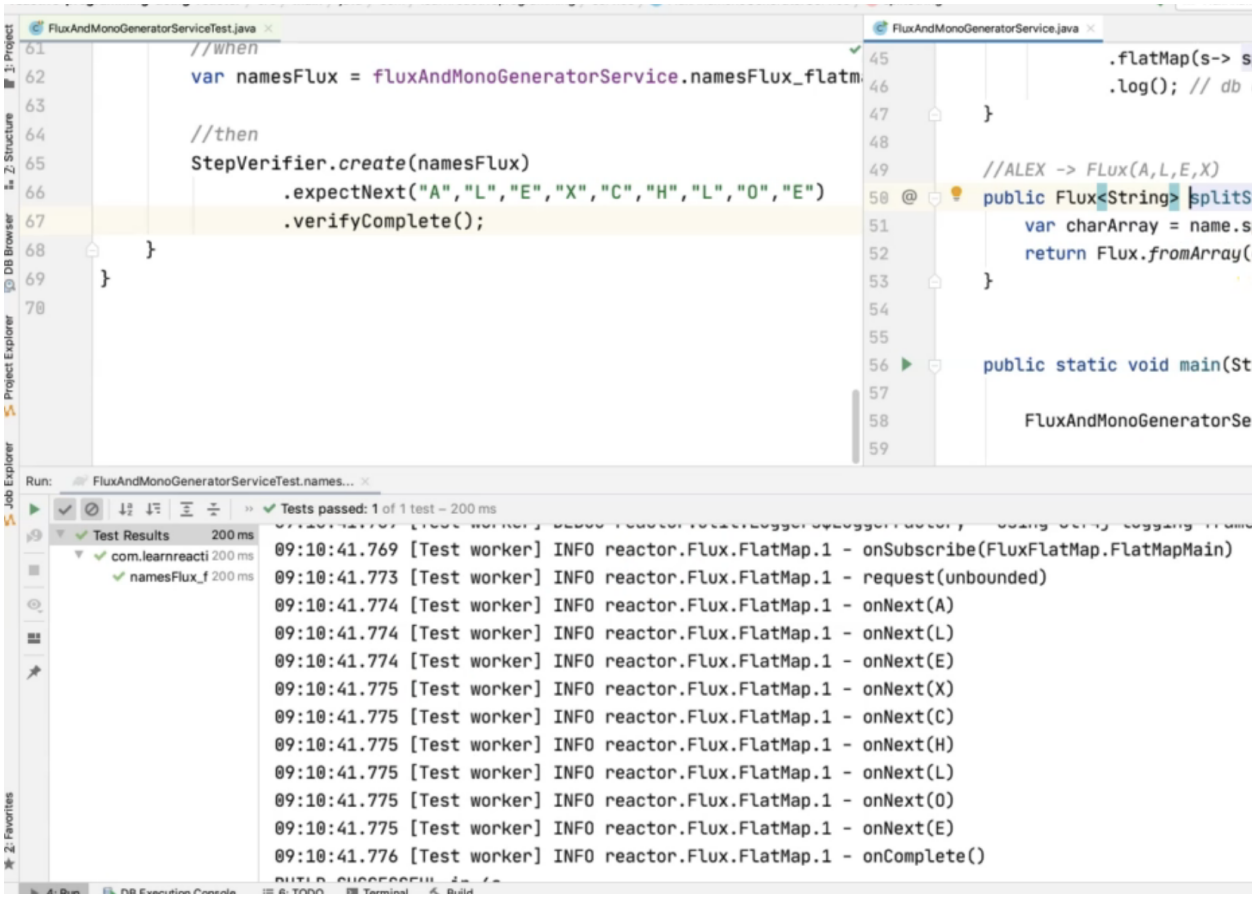
- Transforms one source element to a Flux of 1 to N elements

```
"ALEX" -> Flux.just("A", "L", "E", "X")
```

- Use it when the transformation returns a Reactive Type (Flux or Mono)

- Returns a Flux<Type>

```
public Flux<String> namesFlux_flatmap(int stringLength) {
    //filter the string whose length is greater than 3
    return Flux.fromIterable(List.of("alex", "ben","chloe"))
            .map(String::toUpperCase)
            //.map(s -> s.toUpperCase())
            .filter(s->s.length()> stringLength)
            // ALEX,CHLOE ->A, L, E, X, C, H, L , O, E
            .flatMap(s-> splitString(s)) // A,L,E,X,C,H,L,O,E
            .log(); // db or a remote service call
}

//ALEX -> FLux(A,L,E,X)
public Flux<String> splitString(String name){
    var charArray = name.split( regex: "");
    return Flux.fromArray(charArray);
}
```



Testcase for flatmap.

Asynchronous operations using flatmap.

Here order is not important, the outcome should match. See first chloe and followed by alex. Flatmap can’t be used incase of order is important.

map()

- One to One Transformation
- Does the simple transformation from T to V
- Used for simple synchronous transformations
- Does not support transformations that returns Publisher

flatmap()

- One to N Transformations
- Does more than just transformation. Subscribes to Flux or Mono that's part of the transformation and then flattens it and sends it downstream
- Used for asynchronous transformations
- Use it with transformations that returns Publisher

Last point publisher means flux or mono

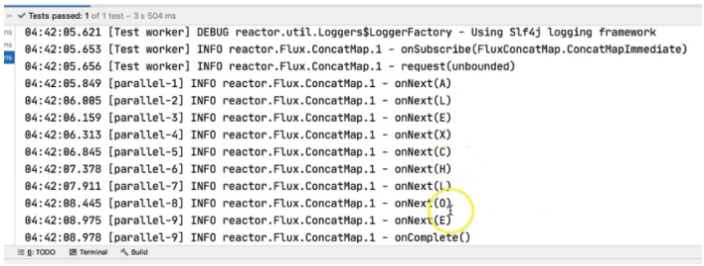
concatMap()

- Works similar to **flatMap()**
- Only difference is that **concatMap()** preserves the ordering sequence of the Reactive Streams.

Order sequencing maintained here unlink flatmap during asynchronous call.

```
    @Test
    void namesFlux_concatmap() {
        //given
        int stringLength = 3;

        //when
        var namesFlux = fluxAndMonoGeneratorService.namesFlux_conc

        //then
        StepVerifier.create(namesFlux)
                .expectNext("A","L","E","X","C","H","L","O","E")
                //.expectNextCount(9)
                .verifyComplete();
    }
}
```

```
                .log(); // db or a remote service call
    }

    public Flux<String> namesFlux_concatmap(int stringLength) {
        //filter the string whose length is greater than 3
        return Flux.fromIterable(List.of("alex", "ben","chloe"))
                .map(String::toUpperCase)
                //.map(s -> s.toUpperCase())
                .filter(s->s.length()> stringLength)
                // ALEX,CHLOE ->A, L, E, X, C, H, L , O, E
                .concatMap(s-> splitString_withDelay(s)) // A,L,E,X,C,H,L
                .log(); // db or a remote service call
    }

    //ALEX -> FLux(A,L,E,X)
    public Flux<String> splitString(String name){
        var charArray = name.split( regex: "");
        return Flux.fromArray(charArray);
```



Order issue fixed

Flatmap is faster where concatmap is slower

flatMap in Mono

- Use it when the transformation returns a Mono
- Returns a Mono<T>
- Use flatMap if the transformation involves making a REST API call or any kind of functionality that can be done asynchronously

```
private Mono<List<String>> splitStringMono(String s) {
    var charArray = s.split("");
    return  Mono.just(List.of(charArray))
            .delayElement(Duration.ofSeconds(1));
}
```

The above slide and below example are very important

```
47
48      public Mono<List<String>> namesMono_flatMap(int stirngLength) {
49          return Mono.just("alex")
50                  .map(String::toUpperCase)
51                  .filter(s-> s.length() > stirngLength)
52                  .flatMap(this::splitStringMono);  //Mono<List of A, L, E  X>
53
54      }
55
56      private Mono<List<String>> splitStringMono(String s) {
57          var charArray = s.split( regex: "");
58          var charList = List.of(charArray); //ALEX -> A, L, E, X
59          return Mono.just(charList);
60      }
```

Command + shift + t will create a test case

```java
@Test
void namesMono_flatMap() {
    //given
    int stringLength = 3;

    //when
    var value = fluxAndMonoGeneratorService.namesMono_flatMap(stringL

    //then
    StepVerifier.create(value)
            .expectNext(List.of("A","L","E","X"))
            .verifyComplete();
}
}
```

```java
            .filter(s-> s.length() > stringLength);

}

public Mono<List<String>> namesMono_flatMap(int stringLength) {
    return Mono.just("alex")
            .map(String::toUpperCase)
            .filter(s-> s.length() > stringLength)
            .flatMap(this::splitStringMono)
            .log();  //Mono<List of A, L, E  X>

}

private Mono<List<String>> splitStringMono(String s) {
    var charArray = s.split( regex: "");
    var charList = List.of(charArray); //ALEX -> A, L, E, X
    return Mono.just(charList);
}
```

flatMapMany( ) in Mono

- Works very similar to flatMap( )

```java
private Flux<String> splitString_withDelay(String name) {
    var delay = new Random().nextInt(1000);
    var charArray = name.split("");
    return Flux.fromArray(charArray)
            .delayElements(Duration.ofMillis(delay));
}
```

FlatmapMany will help to return flux when we apply on mono

```java
        //when
        var value = fluxAndMonoGeneratorService.namesMono_flatMapMany(str

        //then
        StepVerifier.create(value)
                .expectNext("A","L","E","X")
                .verifyComplete();
    }
}
```

```java
            .log();  //Mono<List of A, L, E  X>

}

public Flux<String> namesMono_flatMapMany(int stringLength) {
    return Mono.just("alex")
            .map(String::toUpperCase)
            .filter(s-> s.length() > stringLength)
            .flatMapMany(this::splitString)
            .log();  //Mono<List of A, L, E  X>

}
```

```
Run: FluxAndMonoGeneratorServiceTest.names...
Tests passed: 1 of 1 test – 216 ms
Test Results 216 ms
com.learnreactiv 216 ms
> Task :test
15:51:39.560 [Test worker] DEBUG reactor.util.Loggers$LoggerFactory - Using Slf4j logging framework
15:51:39.660 [Test worker] INFO reactor.Flux.MonoFlatMapMany.1 - onSubscribe(MonoFlatMapMany.FlatMapManyMain)
15:51:39.663 [Test worker] INFO reactor.Flux.MonoFlatMapMany.1 - request(unbounded)
15:51:39.664 [Test worker] INFO reactor.Flux.MonoFlatMapMany.1 - onNext(A)
15:51:39.664 [Test worker] INFO reactor.Flux.MonoFlatMapMany.1 - onNext(L)
15:51:39.664 [Test worker] INFO reactor.Flux.MonoFlatMapMany.1 - onNext(E)
15:51:39.665 [Test worker] INFO reactor.Flux.MonoFlatMapMany.1 - onNext(X)
15:51:39.665 [Test worker] INFO reactor.Flux.MonoFlatMapMany.1 - onComplete()
BUILD SUCCESSFUL in 1s
4 actionable tasks: 3 executed, 1 up-to-date
3:51:39 PM: Task execution finished ':test --tests "com.learnreactiveprogramming.service.FluxAndMonoGeneratorServiceTest.namesMono_flatM
```

transform()

- Used to transform from one type to another
- Accepts **Function Functional Interface**
  - Function Functional Interface got released as part of Java 8
  - Input - Publisher (Flux or Mono)
  - Output - Publisher (Flux or Mono)

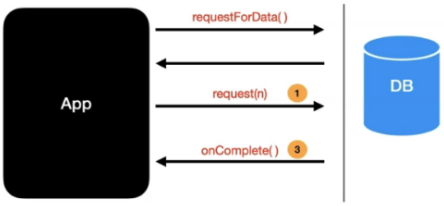
```
public Flux<String> namesFlux_transform(int stringLength) {
    //filter the string whose length is greater than 3

    Function<Flux<String>,Flux<String>> filterMap = name -> name.map(String::toUpperCase)
            .filter(s->s.length()> stringLength);

    return Flux.fromIterable(List.of("alex", "ben","chloe"))
            .transform(filterMap)
            .flatMap(s-> splitString(s)) // A,L,E,X,C,H,L,O,E
            .log(); // db or a remote service call
}
```

```
99
100     t
101     l namesFlux_transform() {
102     //given
103     int stringLength = 3;
104
105     //when
106     var namesFlux = fluxAndMonoGeneratorService.namesFlux_transform(st
107
108     //then
109     StepVerifier.create(namesFlux)
110             .expectNext("A","L","E","X","C","H","L","O","E")
111             .verifyComplete();
112
113
```

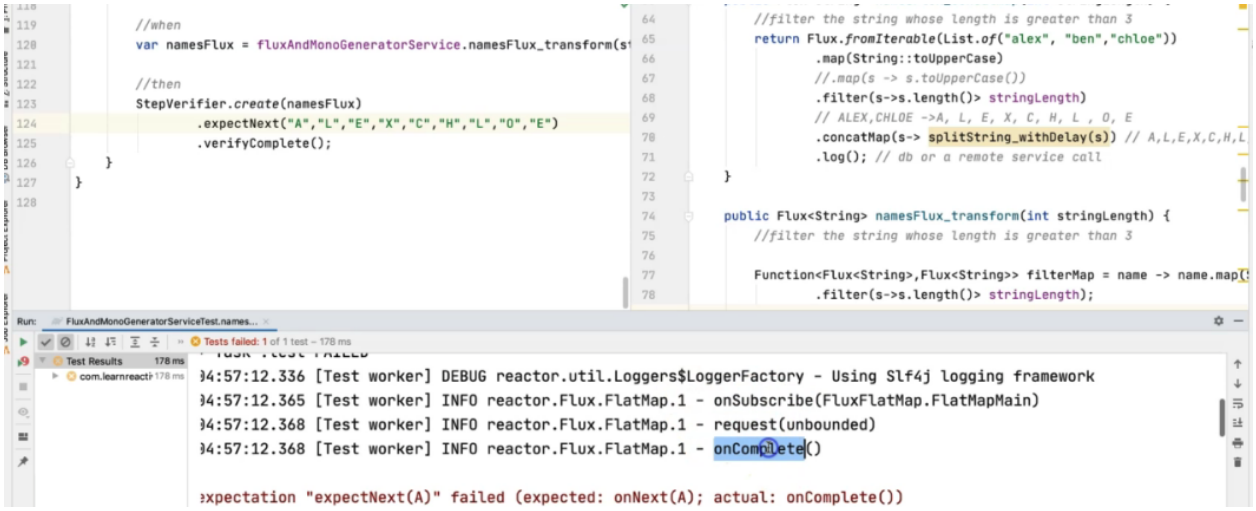
defaultIfEmpty() & switchIfEmpty()

defaultIfEmpty( ) & switchIfEmpty( )

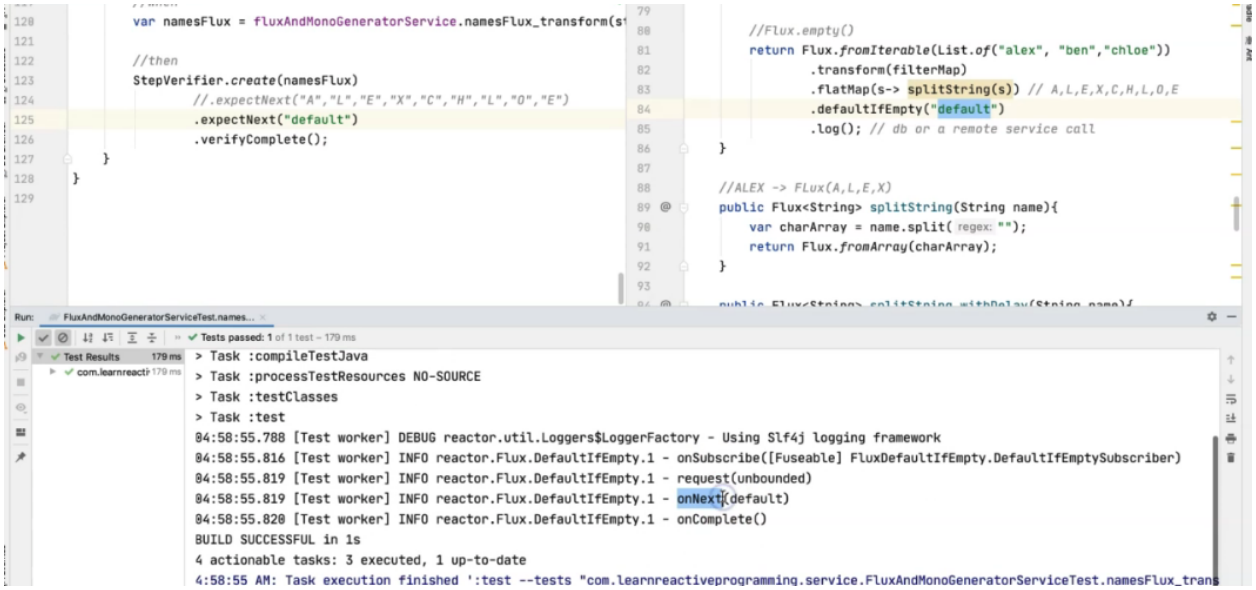
- Its not mandatory for a data source to emit data all the time



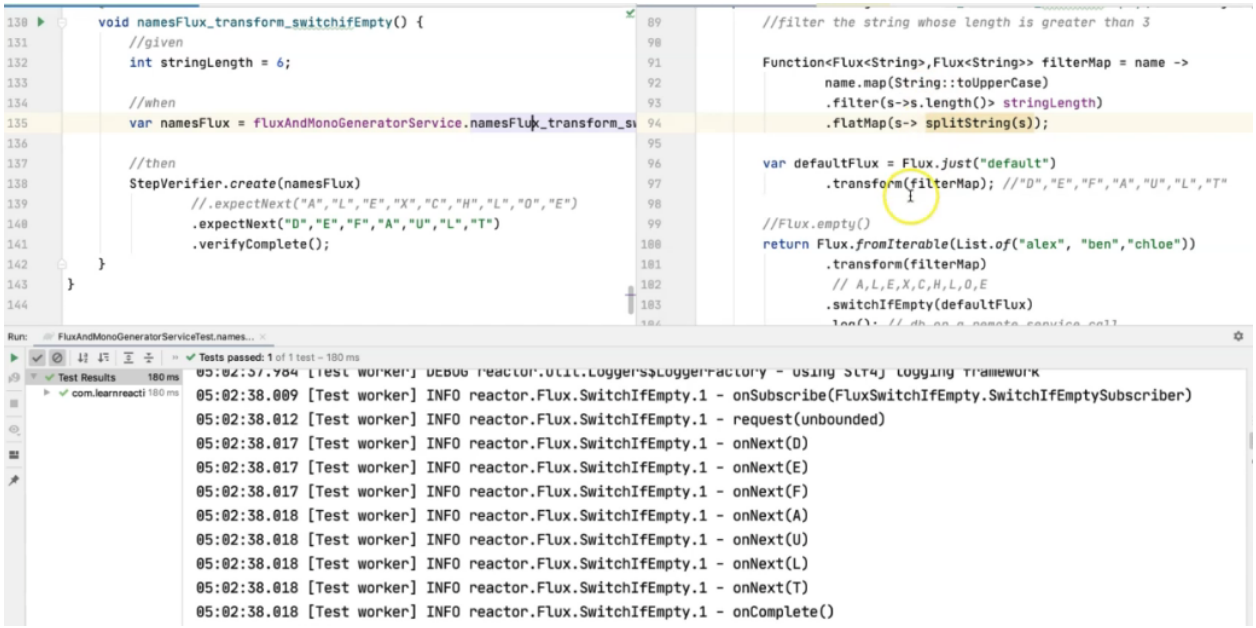
- We can use the defaultIfEmpty() or switchIfEmpty() operator to provide default values



Just pass a filter string with string length 6 and we will get an empty response. See the error message (last line)



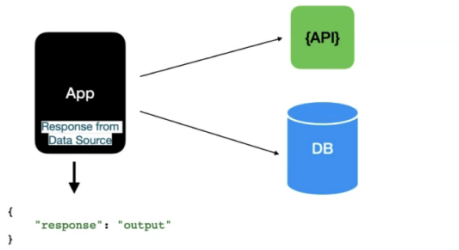
Notice line number 84



switchIfEmpty will produce publisher
defaultIfEmpty will produce the default type of its source type.

Combining Flux & Mono

Why combining Flux and Mono ?

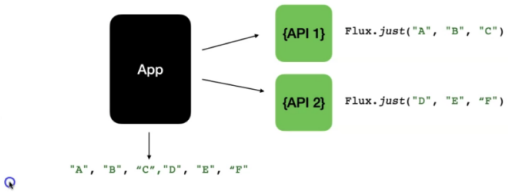


Most of the micro services work in this fashion only.

concat()
&
concatWith()

concat() & concatWith()

- Used to combine two reactive streams in to one



concat() & concatWith()

- Used to combine two reactive streams in to one
- Concatenation of Reactive Streams happens in a sequence
  - First one is subscribed first and completes
  - Second one is subscribed after that and then completes
- **concat( )** - static method in Flux
- **concatWith( )** - instance method in Flux and Mono
- Both of these operators works similarly

```
        //then
        StepVerifier.create(namesFlux)
                //.expectNext("A","L","E","X","C","H","L","O","E")
                .expectNext("D","E","F","A","U","L","T")
                .verifyComplete();
    }

    @Test
    void explore_concat() {
        //given

        //when
        var concatFlux = fluxAndMonoGeneratorService.explore_concat();

        //then
        StepVerifier.create(concatFlux)
                .expectNext("A","B","C","D","E","F")
                .verifyComplete();
    }
```

```
    }

    public Flux<String> explore_concat(){

        var abcFlux = Flux.just("A","B","C");

        var defFlux = Flux.just("D","E","F");

        return Flux.concat(abcFlux,defFlux);

    }

    //ALEX -> FLux(A,L,E,X)
    public Flux<String> splitString(String name){
        var charArray = name.split( regex: "");
        return Flux.fromArray(charArray);
    }
```

```
    public Flux<String> explore_concatwith(){

        var abcFlux = Flux.just("A","B","C");

        var defFlux = Flux.just("D","E","F");

        return abcFlux.concatWith(defFlux).log();

    }

    public Flux<String> explore_concatwith_mono(){

        var aMono = Mono.just("A");
        var bMono = Mono.just("B");

        return aMono.concatWith(bMono).log(); // A, B

    }
```

Line number 115 to 130 concatwith examples

merge( ) and mergeWith( ) are used to combine two publishers in to one.

merge()

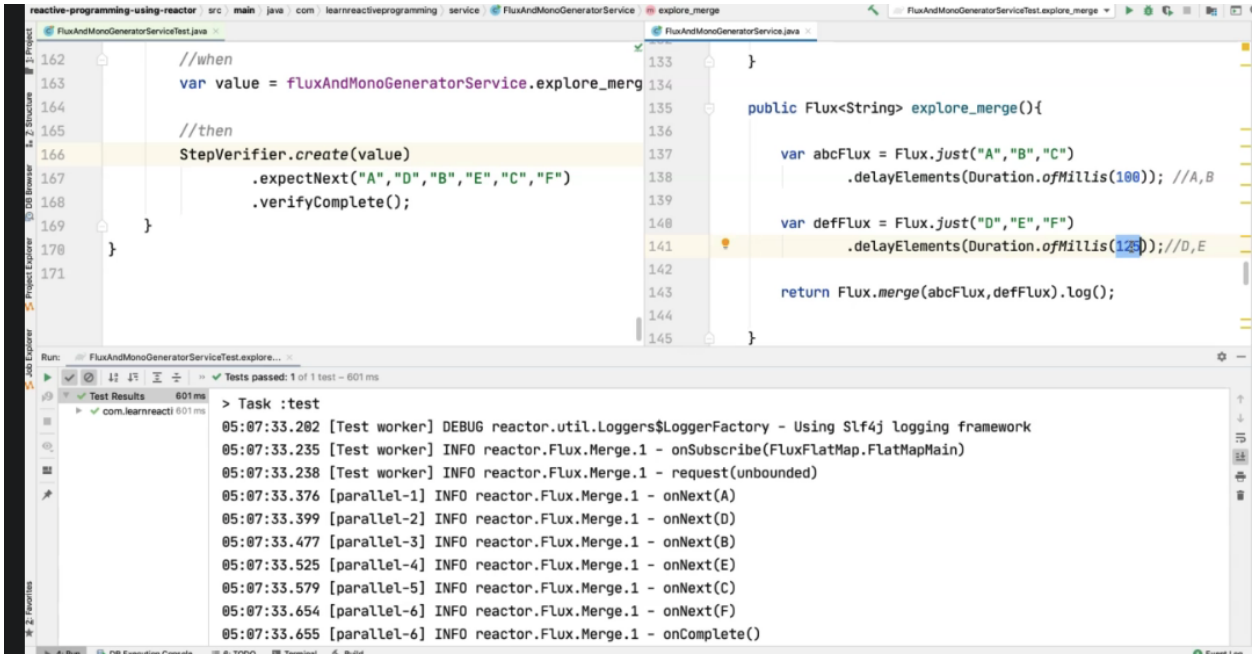
- merge() operator takes in two arguments

```
// "A", "D", "B", "E", "C", "F"
// Flux is subscribed early
public Flux<String> explore_merge() {

    var abcFlux = Flux.just("A", "B", "C")
            .delayElements(Duration.ofMillis(100))
            ;

    var defFlux = Flux.just("D", "E", "F")
            .delayElements(Duration.ofMillis(125))
            ;

    return Flux.merge(abcFlux,defFlux).log();

}
```

Merge will happen interleave fashion

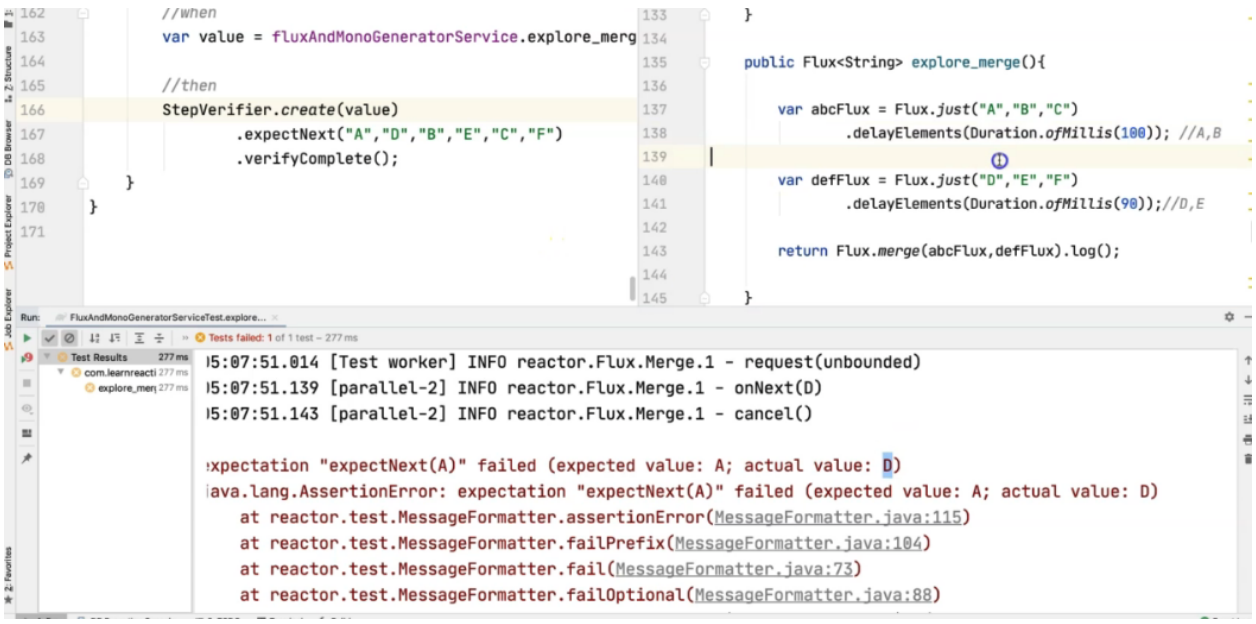
merge() & mergeWith()

- Both the publishers are subscribed at the same time
  - Publishers are subscribed eagerly and the merge happens in an interleaved fashion
  - concat() subscribes to the Publishers in a sequence
- **merge( )** - static method in Flux
- **mergeWith( )** - instance method in Flux and Mono
- Both of these operators works similarly



Merge will happen at 100 , 125, 200, 250 sequence

If we change the value we see vice versa value



Def flux executed first because of less delay time. That is why test got failed

```java
public Flux<String> explore_mergeWith(){

    var abcFlux = Flux.just("A","B","C")
            .delayElements(Duration.ofMillis(100)); //A,B

    var defFlux = Flux.just("D","E","F")
            .delayElements(Duration.ofMillis(125));//D,E

    //return Flux.merge(abcFlux,defFlux).log();
    return abcFlux.mergeWith(defFlux).log();
```

S (Ctrl+S for Win/Linux)

Event Log

mergeWith flux return example

```java
public Flux<String> explore_mergeWith_mono(){

    var aMono = Mono.just("A"); //A

    var bMono = Mono.just("B"); //B

    return aMono.mergeWith(bMono).log(); // A, B

}
```

Merge with mono return example

mergeSequential()

- Used to combine two Publishers (Flux) in to one
- Static method in Flux
- Both the publishers are subscribed at the same time
  - Publishers are subscribed eagerly
  - Even though the publishers are subscribed eagerly the merge happens in a sequence

```
                .verifyComplete();
    }

    @Test
    void explore_mergeSequential() {
        //given

        //when
        var value = fluxAndMonoGeneratorService.explore_merg

        //then
        StepVerifier.create(value)
                .expectNext("A","B","C","D","E","F")
                .verifyComplete();
    }
}
```

```
        return aMono.mergeWith(bMono).log(); // A, B

    }

    public Flux<String> explore_mergeSequential(){

        var abcFlux = Flux.just("A","B","C")
                .delayElements(Duration.ofMillis(100)); //A,B

        var defFlux = Flux.just("D","E","F")
                .delayElements(Duration.ofMillis(125));//D,E

        return Flux.mergeSequential(abcFlux,defFlux).log();
        // return abcFlux.mergeWith(defFlux).log();

    }
```

zip()

- Zips two publishers together in this example

```
// AD, BE, CF
public Flux<String> explore_zip() {

    var abcFlux = Flux.just("A", "B", "C");

    var defFlux = Flux.just("D", "E", "F");

    return Flux.zip(abcFlux, defFlux, (first, second) ->first + second );

}
```

AD BE CF

Third parameter is combinator lambda.

zip() & zipWith()

- zip()
  - Static method that's part of the Flux
  - Can be used to merge up-to 2 to 8 Publishers (Flux or Mono) in to one
- zipWith()
  - This is an instance method that's part of the Flux and Mono
  - Used to merge two Publishers in to one
- Publishers are subscribed eagerly
- Waits for all the Publishers involved in the transformation to emit one element
  - Continues until one publisher sends an OnComplete event

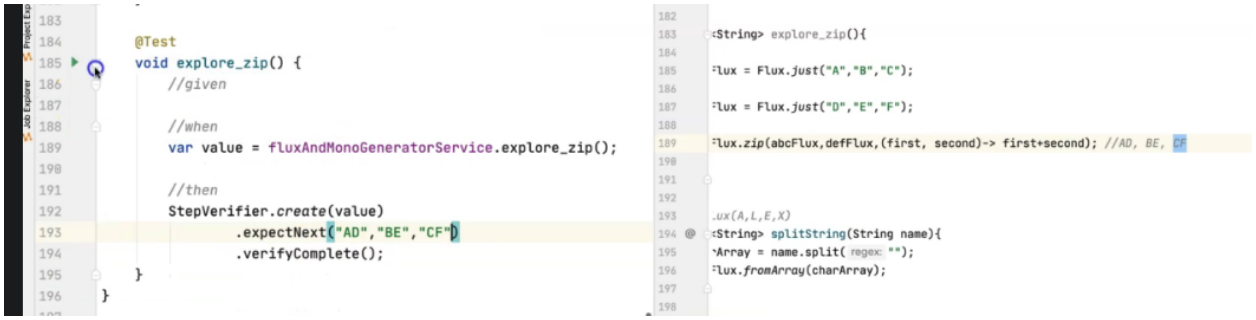
zip()

- Zips four publishers together in this example

```
// AD14, BE25, CF36
public Flux<String> explore_zip_1() {

    var abcFlux = Flux.just("A", "B", "C");
    var defFlux = Flux.just("D", "E", "F");
    var flux3 = Flux.just("1", "2", "3");
    var flux4 = Flux.just("4", "5", "6");

    return Flux.zip(abcFlux, defFlux,flux3,flux4)
            .map(t4 -> t4.getT1()+t4.getT2()+t4.getT3()+t4.getT4());

}
```

Tuple is returned before the map.

```
32
33      public Flux<String> explore_zip(){
34
35          var abcFlux = Flux.just("A","B","C");
36
37          var defFlux = Flux.just("D","E","F");
38
39          return Flux.zip(abcFlux,defFlux,(first, second)-> first+second); //AD, BE, CF
90
91      }
```



```java
@Test
void explore_zip() {
    //given

    //when
    var value = fluxAndMonoGeneratorService.explore_zip();

    //then
    StepVerifier.create(value)
            .expectNext("AD","BE","CF")
            .verifyComplete();
}
```

```java
<String> explore_zip(){

lux = Flux.just("A","B","C");

lux = Flux.just("D","E","F");

lux.zip(abcFlux,defFlux,(first, second)-> first+second); //AD, BE, CF



ux(A,L,E,X)
<String> splitString(String name){
Array = name.split( regex: "");
lux.fromArray(charArray);
```

```java
public static <T1, T2, T3, T4, T5, T6, T7, T8> Flux<Tuple8<T1, T2,
        Publisher<? extends T2> source2,
        Publisher<? extends T3> source3,
        Publisher<? extends T4> source4,
        Publisher<? extends T5> source5,
        Publisher<? extends T6> source6,
        Publisher<? extends T7> source7,
        Publisher<? extends T8> source8) {
    return zip(Tuples.fn8(), source1, source2, source3, source4, s
}
```

Tuple return type

```java
@Test
void explore_zip_1() {
    //given

    //when
    var value = fluxAndMonoGeneratorService.explore_zip_1();

    //then
    StepVerifier.create(value)
            .expectNext("AD14","BE25","CF36")
            .verifyComplete();
}
```

```java
}

public Flux<String> explore_zip_1(){

    var abcFlux = Flux.just("A","B","C");

    var defFlux = Flux.just("D","E","F");

    var _123Flux = Flux.just("1","2","3");
    var _456Flux = Flux.just("4","5","6");

    return Flux.zip(abcFlux,defFlux,_123Flux,_456Flux)
            .map(t4 -> t4.getT1()+t4.getT2()+t4.getT3()+t4.getT4())
            .log(); //AD14, BE25, CF36
```

```
288
289     public Flux<String> explore_zipWith(){
290
291         var abcFlux = Flux.just("A","B","C");
292
293         var defFlux = Flux.just("D","E","F");
294
295         return abcFlux.zipWith(defFlux, (first, second)-> first+second)
296                 .log(); //AD, BE, CF
297
298     }
299
300     public Mono<String> explore_mergeZipWith_mono(){
301
302         var aMono = Mono.just("A"); //A
303
304         var bMono = Mono.just("B"); //B
305
306         return aMono.zipWith(bMono)
307                 .map(t2-> t2.getT1()+t2.getT2()) //AB
308                 .log(); // A, B
309
310     }
```