**1. Introduction to RxJs**

6. What is RxJs? What Problem Does it Solve?

**RXJS - Reactive Extentions for JavaScript**

მის ძრავს წარმოადგენს ისეთი ასინქრონული ფუნქციები, როგორებიცაა: **setTimeout()** და **setInterval()** და მათ გარშემო სხვადასხვა ივენთები, რომლებიც სხვადასხვა სტრიმებს ქმნიან ამ ასინქრონული ფუნქციების გამოძახებით.

**1. Introduction to RxJs**

7. What is an RxJs Observable? A Simple Explanation

**rxjs**-ს გააჩნია ე.წ. დეკლარაციები, რომლებიც კონკრეტულ სტრიმს ადეკლარირებენ, თუმცა არ უშვებენ, აი მაგალითად როცა **interval(1000)**-ს ვწერთ, სანამ**.subscribe()**მეთოდს არ მივაბამთ, იგი არ გაეშვება, ანუ მხოლოდ დეკლარირებულია.

**$** - ნიშნის გამოყენება ცვლადის წინ, სადაც rxjs-ის დეკლარაცია ინახება, ნიშნავს, რომ იგი rxjs-ს ეკუთვნის.

**frinEvent()** პირველ არგუმენტად იღებს html ელემენტს, ხოლო მეორედ იღებს ივენთის სახეობას (მაგ. ქლიქი), და მას აქცევს ობზერვებლად.

რამდენიმე rxjs ოპერატორი **(იმპორტი: rxjs)**:

1. const interval$ = interval(1000);
2. interval$.subscribe(val => console.log("stream 1 " + val)); //ეგრევე ყოველ 1 წამში ერთხელ გაეშვება
4. const timer$ = timer(3000, 1000);
5. timer$.subscribe(val => console.log('timer stream 1 ' + val)) //3 წამის მერე ყოველ 1 წამში გაეშვება
7. const click$ = fromEvent(document, 'click')
8. click$.subscribe(evt => console.log(evt)) //ქლიქს უსმენს ბრაუზერში

**1. Introduction to RxJs**

8. 3 Core RxJs Concepts - Errors, Completion and Subscriptions

**rxjs**-ის **observable**-ს აქვს ასევე სხვა არგუმენტებიც, რომელბსაც **Observable Contract**-ს უწოდებენ, ესენია: **error**და **complete**, რომლებიც ლოგიკურად წარმოშობისას გაეშვებიან:

1. const click$ = fromEvent(document, 'click')
2. click$.subscribe(
3. evt => console.log(evt),
4. err => console.log(err),
5. ()=> console.log('completed')
6. )
7. }

**Observable**-ს ასევე გააჩნიე ე.წ. **unsubsucribe()** მეთოდი, როცა გვსურს რომ შევაჩერეთ რომელიმე სტრიმის მოქმედება.

ამისათვის თავად დეკლარაციის დასუბსქრაიბების ქოლი უნდა შევინახოთ ასევე ცვლადში და მაგალითად, კონკრეტულ მომენტში, ამ შემთხვევაში, 5 წამის მერე, შევაჩეროთ:

1. const interval$ = interval(1000);
3. const sub = interval$.subscribe(val => console.log("stream 1 " + val));
5. setTimeout(()=> {
7. sub.unsubscribe()
9. }, 5000)

**1. Introduction to RxJs**

9. Learn How Observables Work Under the Hood, Build Your Own HTTP Observable

იმისათვის, რომ ჩვენი საკუთარი Observable შევქმნათ, უნდა გამოვიყენოთ ბრძანება:**Observable.create(observer => {})**და იგი შევინახოთ ცვლადში, რათა სამომავლოდ სუბსქრაიბი შევძლოთ:

1. const http$ = Observable.create((observer) => {
2. fetch("https://chat-rooms-45cf5-default-rtdb.firebaseio.com/users.json")
3. .then((res) => {
4. return res.json();
5. })
6. .then((body) => {
7. observer.next(body);
8. observer.complete();
9. })
10. .catch((error) => {
11. observer.error(error);
12. });
13. });
15. http$.subscribe(
16. (res) => console.log(res),
17. noop, //rxjs ფუნქცია, რომელიც ნიშნავს - no operation;
18. () => console.log("completed")
19. );

როგორც ვხედავთ, თავად observer არგუმენტს, ლოგიკურად აქვს 3 მეთოდი: **.next(), .error() და .complete()**

**2. Essential RxJs Operators + Reactive Design**

10. What are RxJs Operators? Learn the Map Operator

პირველ რიგში, ობზერვებლის შექმნა გადამაქვს ცალკე **.ts** ფაილში, ცალკე ფუნქციაში და ამ ფაილის ამ ფუნქციას ვაიმპორტებ ჩემს კომპონენტში. შემდეგ კი ვიყენებ **.map()** ოპერატორს, რომელიც **.pipe()** მეთოდში ჯდება.

**map()**დაუვლის ყველა მნიშნველობას და შეცვლის ისე, როგორც მივუთითებთ.

მაგალითად:

1. const http$ = createHttpObservable(
2. "https://chat-rooms-45cf5-default-rtdb.firebaseio.com/users.json"
3. );
5. const users = http$.pipe(map((res) => Object.values(res)));
6. users.subscribe((res) => {
7. console.log(res);
8. });

**2. Essential RxJs Operators + Reactive Design**

11. Building Components with RxJs - Imperative Design

ამ დროს უკვე რეალურ თემფლეითში გადაგვყავს გამოძახებული ლოგიკა. ერთადერთი, რაც აქედან უნდა გავითვალისწინო არის ის, რომ მაქსიმალურად ვეცადო**.subscribe()**-ში არ შევქმნა ე.წ. ქოლბაქ-ჰელი, ანუ მანდ არ მქონდეს დიდი ლოგიკა გაწერილი.

**2. Essential RxJs Operators + Reactive Design**

12. Building Components with RxJs - Reactive Design

უკეთესი გზა იქნებოდა, თუ ჩვენს სტატიკურ ფროფერთიებს ობზერვებლებად ვაქცევდით და შემდეგ გავუტოლებდით უკვე გაგზავნილი რექვესტის ფუნქციის **pipe()**-ს სადაც .map() მეთოდი იქნებოდა და დააბრუნებდა უკვე სახეცვლილ დეითას. html-ში კი კონკრეტულ ფროფერთის **async**პაიპს მივაბამდით, რომელიც თავისთავში დაასუბსქრაიბებდა ჩვენს ფროფერთის (რომელიც უკვე ობზერვებლია) და მას მერე, რაც კომპონენტი წაიშლებოდა, unsubscribe()-საც გაუკეთებდა.

მაგალითი შემდეგში...

**2. Essential RxJs Operators + Reactive Design**

13. Sharing HTTP Responses with the shareReplay Operator

**ts:**

1. beginnersCourses$: Observable<Course[]>;
3. const http$: Observable<Course[]> = createHttpObservable(
4. "https://httprequeststudy-default-rtdb.firebaseio.com/server.json"
5. );
7. const courses$ = http$.pipe(map(courses => courses['courses']))
9. this.beginnersCourses$ = courses$.pipe(
10. map(
11. courses =>{
12. return courses['courses'].filter(course => course.category == 'BEGINNER')
13. }
14. )
15. )

**html:**

1. <courses-card-list
2. [courses]="beginnersCourses$ | async">
3. </courses-card-list>

აქ ნაჩვენები მაგალითი, **async**პაიპის მოქმედებიდან გამომდინარე და ასევე საჭიროებიდან, ორჯერ ასუბსქრაიბებს რექვესტს, რაც ორჯერ გზავნის მას. ეს არაა საუკეთესო მიდგომა, ამის გამოსასწორებლად გამოვიყენებთ ახალ **rxjs**ოპერატორს შემდეგში...

**2. Essential RxJs Operators + Reactive Design**

13. Sharing HTTP Responses with the shareReplay Operator

ამისათვის გამოიყენება პაიპი **shareReplay()** შემდეგნაირად:

1. const http$: Observable<Course[]> = createHttpObservable(
2. "https://httprequeststudy-default-rtdb.firebaseio.com/server.json"
3. );
5. const courses$ = http$.pipe(
6. tap(()=> {
7. console.log('http request executed')
8. }),
9. map(courses => courses['courses']),
10. shareReplay()
11. )
12. this.beginnersCourses$ = courses$.pipe(
13. map(
14. courses =>{
15. return courses.filter(course => course.category == 'BEGINNER')
16. }
17. )
18. )

**.tap()** ოპერატორი გამოიყენება საიდ ეფექტების პროდიუსისთვის, ანუ იგი არ ცვლის რექვესტს, ანუ არაფერს არ აბრუნებს, რაც სხვა ოპერატორს გადასცემდა დეითას.

**2. Essential RxJs Operators + Reactive Design**

15. Observable Concatenation - In-Depth Explanation

**rxjs**-ს აქვს ასეთი ოპერატორი: **concat()**, რომელიც რამდენიმე სტრიმს აერთიანებს წყვეტის გარეშე, ანუ რამდენიმე ობზერვებლს.

დოკუმენტაცია: *https://rxjs.dev/api/operators/concat*

**of()** წარმოადგენს **rxjs**-ის ფუნქციას, რომელში ჩასმული არგუმეტის შესრულების მერე ავტომატურად წყდება.

დოკუმენტაცია: *https://rxjs.dev/api/index/function/of*

საბოლოოდ კი სუბსქრაიბი ებმება **contact()** ფუნქციას.

ასევე მნიშვნელოვანია ვიცოდეთ, რომ თუ ერთი სოურსიდან გაშვებული ფუნქცია **complete**არ იქნება, მეორე სოურსი არ გაეშვება, ამიტომაც ვიყენებთ აქ**of()** ფუნქციას.

**კოდი:**

1. const source1$ = of(1,2,3)
2. const source2$ = of(4,5,6)
3. const source3$ = of(7,8,9)
4. const result$ = concat(source1$,source2$,source3$)
5. result$.subscribe(console.log)

**2. Essential RxJs Operators + Reactive Design**

16. Form Draft Pre-Save Example and the RxJs Filter Operator

ამ ლექციაში გავეცანით:

ანგულარის რეაქტივ ფორმს გააჩნია ობზერვებლი: **valueChanges**, რომელიც ყოველ ასოს დაწერაზე გამოიძახება, ანუ აკვირდება ცვლილებებს მნიშნველობაში.

ასევე გავეცანით **rxjs**-ის მეთოდს: **fromPromise()**, რომელიც არგუმენტად იღებს **Promise**ფუნქციას, როგორიცაა **fetch()** მაგალითად, და აქცევს ობზერვებლად.

დოკუმენტაცია: *იმპორტაცია 'rxjs/internal-compatibility'-დან*

ასევე filter() rxjs ოპერატორს, რომელიც ფილტრავს კონკრეტული ქონდიშენის მიხედვით და ისე ეწევა ემიტირებას.

დოკუმენტაცია: *https://rxjs.dev/api/operators/filter*

დანარჩენი შემდეგში...

**2. Essential RxJs Operators + Reactive Design**

17. The RxJs concatMap Operator - In-Depth Explanation and Practical Example

იმ შემთხვევაში, თუ გვსურს რქვესტები თანმიმდევრულად გაიგზავნონ და არა ერთდროულად და ქაოტურად, შეიძლება გამოვიყენოთ **rxjs**ოპერატორი: **concatMap()**. იგი თანმიმდევრულად უშვებს ობზერვებლს, ახდენს მასზე მოქმედებას ანუ ასუბსქრაიბებს კიდევაც.

დოკუმენტაცია: *https://rxjs.dev/api/operators/concatMap*

კოდი:

1. ngOnInit() {
2. this.form.valueChanges
3. .pipe(
4. filter(()=> this.form.valid),
5. concatMap(changes => this.saveCourse(changes))
6. )
7. .subscribe()
8. }
10. saveCourse(changes){
11. return fromPromise(fetch('https://httprequeststudy-default-rtdb.firebaseio.com/server/courses/' + this.course.id+ '.json', {
12. method: 'PUT',
13. body: JSON.stringify(changes),
14. headers: {
15. 'content-type': 'application/json'
16. }
17. }))
18. }

**2. Essential RxJs Operators + Reactive Design**

18. Understanding the merge Observable combination Strategy

**rxjs**-ის კიდევ ერთი საინტერესო ოპერატორია **merge()**, რომელიც არგუმენტად ობზერვებლებს იღებს, აერთიანებს მათ და ერთდროულად უშვებს.

დოკუმენტაცია:*https://rxjs.dev/api/index/function/merge*

კოდი:

1. const interval1$ = interval(1000);
2. const interval2$ = interval1$.pipe(map(val => 10\*val))
4. const result$ = merge(interval1$, interval2$)
5. result$.subscribe(console.log)

**2. Essential RxJs Operators + Reactive Design**

19. The RxJs mergeMap Operator - In-Depth Explanation

**concatMap()** შეგვიძლია ჩავანაცვლოთ **mergeMap()**-ით, თუმცა მოქმედება განსხვავებული იქნება: მერჯემაფი არ დაელოდება წინა რექვესტის დასრულებას და ისე გაგზავნის ახალს, მასთან ერთად, რაც საასურველი ჩვენს შემთხვევაში არაა, თუმცა ზოგში შესაძლოა იყოს.

დოკუმენტაცია: *https://rxjs.dev/api/operators/mergeMap*

კოდი:

1. ngOnInit() {
3. this.form.valueChanges
4. .pipe(
5. filter(()=> this.form.valid),
6. mergeMap(changes => this.saveCourse(changes))
7. )
8. .subscribe()
10. }

**2. Essential RxJs Operators + Reactive Design**

20. The RxJs exhaustMap Operator - In-Depth Explanation

**rxjs**-ის ოპერატორი **exhaustMap()** გვეხმარება იმაში, რომ სანამ რექვესტი არ დასრულდება, მანამ სხვა არ გაგზავნოს.

დოკუმენტაცია: *https://rxjs.dev/api/operators/exhaustMap*

კოდი:

1. fromEvent(this.saveButton.nativeElement, 'click').pipe(
2. exhaustMap(() => this.saveCourse(this.form.value))
3. )
4. .subscribe()

**2. Essential RxJs Operators + Reactive Design**

21. Unsubscription In Detail - Implementing a Cancellable HTTP Observable

**Fetch()**-ის აპი-ს აქვს ქენსელის მხარდაჭერა, ამისთვის იქ, სადაც fetch() მეთოდი არის გამოყენებული, უნდა გამოვიყენოთ ჩაშენებული ობიექტის ინსტანსი = new **AbortController()**, რომელსაც შევინახავთ ცვლადში და ამ ცვლადს ექნება ფროფერთი - signal, რომელსაც თავად fetch()-ს ჩავაწვდით მეორე არგუმენტად, როგორც ობიექტის ფროფერთი:, ხოლო მის განსახორციელებლად, ასევე ჩიგივე ცვლადში ჩაშენებულ **abort()** მეთოდს გამოვიყენებდით:

1. const controller = new AbortController();
2. const signal = controller.signal;
4. fetch(url, {signal})
5. .then((res) => {
6. return res.json();
7. })
8. controller.abort()

თუმცა იბზერვებლის შემთხვევაში, მისი გამოძახება გვსურს მაშინ, როცა ობზერვებლი დაანსუბსქრაიბდება. როცა ობზერვებლი ჩვენით გვაქვს შექმნილი, მაგ უბზერვებლს თუ რამე ფუნქციას დავაბრუნებინებთ, ეგ გამოიძახება, როცა ანსუბსქრაიბ ხდება, შესაბამისად, ანსუბსქრაიბისას გამოვიძახებთ **.abort()** მეთოდს:

1. return () => controller.abort()

**2. Essential RxJs Operators + Reactive Design**

22. Setting Up the Course Component

**switchMap()** ოპერატორი ხშირად გამოიყენება სერჩის ფუნქციის გაშვებისას. ასევე, სასარგებლო ოპერატორია **debounceTime()**, რომელიც არგუმენტად იღებს რიცხვს და იგი მილიწამებში იკითხება, რომელიც ნიშნავს, რომ მითითებულ მილიწამებში იგი სხვა ბრძანებას არ მიიღებს, ყოველ ჯერზე თავიდან გაეშვება ათვლა.

ასევე არსებობს **distinctUntilChanged(**), რომელიც იმ შემთხვევაში, თუ მნიშვნელობები იგივეა, მეორეჯერ აღარ გაუშვებს ფუნქციას:

1. fromEvent<any>(this.input.nativeElement, 'keyup').pipe(
2. map(event => event.target.value),
3. debounceTime(400),
4. distinctUntilChanged()
5. ).subscribe(console.log)

**2. Essential RxJs Operators + Reactive Design**

24. Finishing the Search Typeahead - The switchMap Operator

**switchMap()**ოპერატორი იმ შემთხვევაში, თუ ახალი მნიშვნელობა ჩაეწოდება იმ დროს, როცა წინა ობზერვებლს უსმენს, ამ წინას დააანსუბსქრაიბებს და ახალს დაასუბსქრაიბებს.

**3. RxJs Error Handling**

26. RxJs Error Handling - The Catch and Replace Error Handling Strategy

**catchError()** ოპერატორი გამოიყენება ერორის ჰენდლინგისთვის, მისი არგუმენტი იძახებს სხვა ობზერვებლს, როცა ერორი მოხდება, მაგალითად შეგვიძლია **of()** ფუნქცია გამოვიყენოთ ამისთვის:

1. const courses$ = http$.pipe(
2. map(courses => {
3. return courses['COURSES']
4. ),
5. shareReplay(),
6. catchError(err => of([
7. {
8. id: 6,
9. description: 'The Complete Typescript Course',
10. longDescription: "Complete Guide to Typescript From Scratch: Learn the language in-depth and use it to build a Node REST API.",
11. iconUrl: 'https://angular-academy.s3.amazonaws.com/thumbnails/typescript-2-small.png',
12. courseListIcon: 'https://angular-academy.s3.amazonaws.com/thumbnails/typescript-2-lesson.png',
13. category: 'BEGINNER'
14. }
15. ]))
16. )

**3. RxJs Error Handling**

27. The Catch and Rethrow RxJs Error Handling Strategy and the finalize Operator

**throwError()** ფუნქცია ისმება **catchError**()-ში, ანუ ბრუნდება და იგი ემიტირების გარეშე აგდებს ერორს.

**finalize()** ოპერატორი ეშვება ორივე შემთხვევაში: ერორია თუ შესრულდა ობზერვებლი.

**3. RxJs Error Handling**

28. The Retry RxJs Error Handling Strategy.screenflow

**retryWhen()** ოპერატორი ერორის შემთხვევაში თავიდან დაასუბსქრაიბებს ობზერვებლს, ხოლო იმისათვის, რომ ყოველ ჯერზე ეგრევე არ დაასუბსქრაიბოს, თაიმერი უნდა გავუწეროთ, ამისთვის გამოიყენება ორი ოპერატორი: **delayWhen()** და **timer()**:

1. retryWhen(errors => errors.pipe(
2. delayWhen(() => timer(200))
3. ))

ან უბრალოდ **delay()** ოპერატორის გამოყენებით, თუმცა პირველი ჯობს, რამდენადაც ყოველი 2 წამის მერე გააშვება და არა 2 წამის მერე:

1. retryWhen(errors => errors.pipe(
2. delay(2000)
3. ))

**3. RxJs Error Handling**

29. The startWith RxJs Operator - Simplifying the Course Component

**startWidth()**ოპერატორი შეამოწმებს მნიშვნელობას და მაგის მიხედვით გაუშვებს მომდევნო ოპერატორებს:

1. this.lessons$ = fromEvent<any>(
2. this.input.nativeElement,"keyup").
3. pipe(
4. map((event) => event.target.value),
5. startWith(''),
6. debounceTime(400),
7. distinctUntilChanged(),
8. switchMap((search) => this.loadLessons(search))
9. );

**3. RxJs Error Handling**

30. RxJs Throttling vs Debouncing - Understand the Differences

**throttle()** ოპერატორი იღებს ფუნქციას არგუმენტად და შემდეგნაირად მუშაობს: ეს ფუნქცია აბრუნებს ინტერვალს, სადაც ეთითება ის მილიწამები, რა დროის შემდეგაც უნდა დააემიტოს მნიშნველობა, მაგალითად თუ მითითებული დრო არაა გასული წერის დაწყებიდან, პირველი იმიტის შემდეგი ვალიუ არ გაეშვება.

1. throttle(() => interval(500))

**throttleTime()** ოპერატორი ზემოთ დაწერილის შემოკლებული ვერსიაა:

1. throttleTime(500)

**4. Building a RxJs Custom Operator**

33. The RxJs forkJoin Operator - In-Depth Explanation and Practical Example

**forkJoin()**ფუნქცია არგუმენტებად იღებს ობზერვებლებს და ყველა ობზერვებლის დაბრუნებულ მნიშნველობას ინახავს, თუმცა მას შემდეგ, რაც ყველა ობზერვებლი დასრულდება. თუ რომელიმე არ დასრულდება, არც ერთი არ შეინახება.

**5. RxJs Subjects and the Store Pattern**

35. What are RxJs Subjects? A Simple Explanation

**Subject()** თავის თავში წარმაოდგენს ობზერვერსაც და ობზერვებლსაც. რამდენადაც არ გვუსრ, რომ საბჯექტის ობზერვერის თვისებას, ანუ**next()**და ა.შ. სხვა ნაწილიდან იყოს მისაწვდომი, ამისთვის აღმოცემულ საბჯეცტს ცვლადში შევინახავთ და მივაბამთ მეთოდს -**.asObservable(). იგი თუ დასუბსქრაიბებულია, subject()-ში დანექსთებულ ინფოს ეგეც გამოიტანს.**

1. const subject = new Subject();
2. const series1$ = subject.asObservable()

**5. RxJs Subjects and the Store Pattern**

36. BehaviorSubject In Detail - When to Use it and Why?

**BehaviorSubject()**-ის შემთხვევაში, იგი ყოველთვის შეინახავს რაიმე მნიშვნელობას, სუბსქრაიბის დასრულების შემდეგაც კი - მასში შეინახება ბოლო ვალიუ. მას აუცილებლად უნდა ჰქონდეს არგუმენტში ინიციალ ველიუ.

1. const subject = new BehaviorSubject(0);
2. const series1$ = subject.asObservable()
3. series1$.subscribe(console.log);
5. subject.next(1)
6. subject.next(2)
7. subject.next(3)
9. setTimeout(()=> {
10. series1$.subscribe(val => console.log('late'))
11. subject.next(4)
12. })

**behaviorSubject()**-ის სრული ფუნქციონირებისთვის, არ უნდა გამოვიყენოთ **.complete()** მეთოდი, რადგან იგი დაასრულებს ობზერვებლს.

**5. RxJs Subjects and the Store Pattern**

37. AsyncSubject and ReplaySubject - Learn the Differences

**AsyncSubject()**-ის შემთხვევაში, მომხმარებელი დაინახავს მხოლოდ ბოლოს დაემიტებულ მნიშნველობას, ანუ სანამ **.complete()** გაეშვება:

1. const subject = new AsyncSubject();

**ReplaySubject()**-ის შემთხვევაში კი პირველად დასუბსქრაიბებისას დანექსთებულ მნიშვნელობებს, მეორე დასუბსქრაიბებისას წამოიღებს სათითაოდ ყველას კიდევ ერთხელ, ანუ რეფლეი.:

1. const subject = new ReplaySubject();

**5. RxJs Subjects and the Store Pattern**

42. Forcing the Completion of Long Running Observables - First and Take Operators

იმისათვის, რომ კომპლიტი დავფორსოთ, ანუ დავაძალოთ ობზერვებლს, შეგვიძლია გამოვიყენთ ოპერატორი**.first()**, რომელიც მხოლოდ პირველ ჯერზე მოტანილ მნიშვნელობას აიღებს და დააკომპლიტებს ობზერვებლს.

1. .pipe(
2. first()
3. )
4. .subscribe()

ხოლო თუ გვსურს პირველის გარდა სხვა, მაგალითად მეორე ან მესამე მნიშვნელობა, უნდა გმაოვიყენოთ: **.take(3)** - იგი პირველ სამ მნიშნველობას აიღებს.

**5. RxJs Subjects and the Store Pattern**

43. The withLatestFrom RxJs Operator - Detailed Explanation

**withLatestFrom()**ოპერატორი გვეხმარება იმაში, რომ ჯერ დავასუბსქრაიბოთ ერთ ობზერვებლი, ამოვიღოთ იქიდან ბოლოს მიღებული მნიშვნელობა და გადავაწოდოთ საბსქრიბშენ ჩეინის მომდევნო ოპერატორს ან სუბსქრაიბს თავად. არგუმენტად იღებს ობზერვებლს, ხოლო რომელ ოპერატორსაც გადააწოდებს, მის არგუმენტში უკვე ორი მნიშვნელობა უნდა შევიდეს, რადგან იგი უკვე ორი ობზერვებლის მნიშვნელობას შეინახავს:

1. .pipe(
2. withLatestFrom(course$)
3. )
4. .subscribe(([lessons, course])=> {
5. console.log(lessons, course)
6. })