

**Лекція №1. Політика та контроль освітнього компонента. Реактивність: основні поняття. Патерни OBSERVER та ITERATOR. Маніфест реактивності.**

**Лекції та лабораторні заняття з дисципліни «Реативне програмування»:**

**к.т.н., доц. Полупан Юлія Вікторівна**

Лекція №1:

<https://meet.google.com/zqe-nhep-yny>

[https://drive.google.com/drive/folders/111bSN39wDRPEJdiSkTyg2FFH7HJ9xbKR?usp=drive\\_link](https://drive.google.com/drive/folders/111bSN39wDRPEJdiSkTyg2FFH7HJ9xbKR?usp=drive_link)

Всі лабораторні роботи:

[https://drive.google.com/drive/folders/1FHyfAjPC7YYn5vu\\_SzJvEDLSnyfMkZIM?usp=sharing](https://drive.google.com/drive/folders/1FHyfAjPC7YYn5vu_SzJvEDLSnyfMkZIM?usp=sharing)

**Політика та контроль освітнього компонента**

**1. Політика навчальної дисципліни (освітнього компонента)**

- **Форма організації освітнього процесу, види навчальних занять і оцінювання результатів навчання** регламентуються Положенням про організацію освітнього процесу в Національному технічному університеті України «Київському політехнічному інституті імені Ігоря Сікорського». <https://kpi/regulations>
- **Політика виставлення оцінок:** кожна оцінка виставляється відповідно до розроблених викладачем та заздалегідь оголошених студентам критеріїв, а також мотивується в індивідуальному порядку на вимогу студента; у випадку не виконання студентом усіх передбачених навчальним планом видів контролю (лабораторних робіт, контрольних робіт) залік студент не отримує.
- **Відвідування є обов'язковим** (за винятком випадків, коли існує поважна причина, наприклад, хвороба чи дозвіл працівників деканату). Якщо студент не може бути присутнім на заняттях, він все одно несе відповідальність за виконання завдань, що проводились на заняттях та винесені в домашнє завдання. Відпрацювання пропущеного лабораторного заняття здійснюється шляхом самостійного виконання завдання і його захисту відповідно до графіку консультацій викладача.
- **Політика академічної поведінки та доброчесності:** конфліктні ситуації мають відкрито обговорюватись в академічних групах з викладачем, необхідно бути взаємно толерантним, поважати думку іншого. Плагіат та інші форми нечесної роботи неприпустимі. Всі види завдань студент має виконувати самостійно із використанням рекомендованої літератури й отриманих знань та навичок. Цитування в письмових

роботах допускається тільки із відповідним посиланням на авторський текст. Недопустимі підказки і списування у ході захисту лабораторних робіт, на контрольних роботах, на іспиті.

- **Норми академічної етики:** дисциплінованість; дотримання субординації; чесність; відповідальність; робота в аудиторії з відключеними мобільними телефонами. Повага один до одного дає можливість ефективніше досягати поставлених командних результатів. При виконанні лабораторних робіт студент може користуватися ноутбуками. Заборонено використовувати свій ноутбук чи телефон для аудіо- чи відеозапису без офіційного дозволу викладача.
- **Дотримання академічної доброчесності студентів й викладачів** регламентується кодексом честі Національного технічного університету України «Київський політехнічний інститут», положення про організацію освітнього процесу в КПІ ім. Ігоря Сікорського <https://kpi.ua/code>

## 2. Види контролю та рейтингова система оцінювання результатів навчання (PCO)

1. Рейтинг студента з кредитного модуля розраховується зі 100 балів, з них 85 балів складає шкала  $L$  та 15 балів шкала  $C$ . Шкала  $L$  протягом семестру складається з балів, що студент отримує за виконання та захист 8-ми лабораторних робіт. Як оцінюється кожна лабораторна робота дивиться таблицю 1. Шкала  $C$  складається з балів, які студент отримує за одну контрольну роботу за лекційним матеріалом протягом семестру.

### 2. Критерії нарахування балів

2.1. Виконання та захист лабораторних робіт (максимальна кількість балів для  $L_{1-7}^{\max}=10$ , для  $L_8^{\max}=15$ ).

1) представлення звіту та додатку на хостінгу Firebase для  $L_{1-7}=4$  бали, для  $L_8=6$  балів.

2) захист роботи складається із захисту звіту по роботі та із захисту додатку на хостінгу:

а) при захисті звіту виставляються бали, згідно таблиці 1, де:

\* Повні відповіді на всі питання по звіту: не менше 90% потрібної інформації (повні, безпомилкові відповіді).

\*\* Часткові відповіді на питання по звіту: достатньо повна відповідь, не менше 75% потрібної інформації або незначні неточності.

\*\* Задовільні відповіді на питання по звіту: неповна відповідь, не менше 60% потрібної інформації та деякі помилки (завдання виконане з певними недоліками).

\*\*\* Незадовільні відповіді на всі питання по звіту (або звіту, додатку, контрольної немає): відповідь не відповідає умовам до «задовільні відповіді».

б) при захисті додатку на хостінгу бали виставляються також, згідно таблиці 1, де:

\* Повні відповіді на всі питання по додатку: не менше 90% потрібної інформації (повні, безпомилкові відповіді).

\*\* Часткові відповіді на питання по додатку: достатньо повна відповідь, не менше 75% потрібної інформації або незначні неточності.

\*\* Задовільні відповіді на питання по додатку: неповна відповідь, не менше 60% потрібної інформації та деякі помилки (завдання виконане з певними недоліками).

\*\*\* Незадовільні відповіді на всі питання по додатку (або звіту, додатку, контрольної немає): відповідь не відповідає умовам до «задовільні відповіді».

2.2. За участь у виконанні завдань із удосконалення дидактичних матеріалів з дисципліни: від 1 до 5 заохочуваних балів.

Кількість заохочуваних балів не може перевищувати 8,5 (10% від шкали  $L$ ).

3. Умовою отримання заліку є набір не менше ніж 60 балів по всім видам контролю  $\sum_{i=1}^8 L_i + C \geq 60$ .

Якщо певний вид контролю  $L_i$  або  $C$  не здається вчасно, то бали за цей вид контролю нараховуються від  $L_i$  або  $C$  наступним чином:

- 1) якщо запізнення на два тижні - 2% від відповідного  $L_i$  або  $C$ ;
- 2) якщо запізнення від двох до чотирьох тижнів - 3% від відповідного  $L_i$  або  $C$ ;
- 3) якщо запізнення від чотирьох до шести тижнів - 4% від відповідного  $L_i$  або  $C$ ;
- 4) якщо запізнення від шести до восьми тижнів - 5% від відповідного  $L_i$  або  $C$ ;
- 5) якщо запізнення від восьми до десяти тижнів - 6% від відповідного  $L_i$  або  $C$ ;
- 6) якщо запізнення від десяти до дванадцяти тижнів - 7% від відповідного  $L_i$  або  $C$ ;
- 7) якщо запізнення від дванадцяти до чотирнадцяти тижнів - 8% від відповідного  $L_i$  або  $C$ ;
- 8) якщо запізнення від чотирнадцяти до шістнадцяти тижнів - 9% від відповідного  $L_i$  або  $C$ ;
- 9) якщо запізнення від шістнадцяти до вісімнадцяти тижнів - 10% від відповідного  $L_i$  або  $C$ ;
- 10) якщо запізнення більше ніж вісімнадцять тижнів – відповідний вид контролю не зараховується;
- 11) якщо передвчасна здача на два тижні + 2% від відповідного  $L_i$  або  $C$ ;
- 12) якщо передвчасна здача від двох до чотирьох тижнів + 3% від відповідного  $L_i$  або  $C$ ;
- 13) якщо передвчасна здача від чотирьох до шести тижнів + 4% від відповідного  $L_i$  або  $C$ ;
- 14) якщо передвчасна здача від шести до восьми тижнів + 5% від відповідного  $L_i$  або  $C$ ;
- 15) якщо передвчасна здача від восьми до десяти тижнів + 6% від відповідного  $L_i$  або  $C$ ;
- 16) якщо передвчасна здача від десяти до дванадцяти тижнів + 7% від відповідного  $L_i$  або  $C$ ;

17) якщо передвчасна здача від дванадцяти до чотирнадцяти тижнів + 8% від відповідного  $L_i$  або  $C$ ;

18) якщо передвчасна від чотирнадцяти до шістнадцяти тижнів + 9% від відповідного  $L_i$  або  $C$ ;

19) якщо передвчасна від шістнадцяти до вісімнадцяти тижнів + 10% від відповідного  $L_i$  або  $C$ ;

4. Сума всіх балів  $L_i+C$  переводиться до залікової оцінки згідно з таблицею 2:

Таблиця 2.

Бали ( $L_i+C$ ): лабораторні роботи + контрольна робота	Оцінка
95...100	Відмінно
85...94	Дуже добре
75...84	Добре
65...74	Задовільно
60...64	Достатньо
Менше 60	Незадовільно
Є не зараховані лабораторні роботи або відсутня контрольна робота	Не допущено

Таблиця 1. Розподіл рейтингових балів при умові вчасного виконання відповідного виду контролю

№	Вид контролю	Наявність звіту (бали) $R_1$	Наявність на платформі FireBase (бали) $R_2$		Захист додатку на FireBase (відповідь на питання), бали $R_3$				Захист звіту (відповідь на питання) $R_4$				Контрольна робота $R_5$				Мін. кількість балів для зарахування	Макс. кількість балів
					Повні відповіді на всі питання (*)	Часткові відповіді на питання (**)	Задовільні відповіді на питання (***)	Незадовільні відповіді на всі питання (****) або додатку немає	*	**	***	**** або звіту немає	*	**	***	**** або контрольної немає		
1	Лаб. робота №1 ( $L_1$ )	1	Laba1_1: 1,5	Laba1_2: 1,5	3	2	1	0	3	2	1	0					$L_1^{\min} : 6$	$L_1^{\max} : 10$
2	Лаб. робота №2 ( $L_2$ )	1	Laba2_1: 1,5	Laba2_5: 1,5	3	2	1	0	3	2	1	0					$L_2^{\min} : 6$	$L_2^{\max} : 10$
3	Лаб. робота №3 ( $L_3$ )	2	Laba3: 2		3	2	1	0	3	2	1	0					$L_3^{\min} : 6$	$L_3^{\max} : 10$
4	Лаб. робота №4 ( $L_4$ )	2	Laba4: 2		3	2	1	0	3	2	1	0					$L_4^{\min} : 6$	$L_4^{\max} : 10$
5	Лаб. робота №5 ( $L_5$ )	1	Laba5_1: 1,5	Laba5_2: 1,5	3	2	1	0	3	2	1	0					$L_5^{\min} : 6$	$L_5^{\max} : 10$
6	Лаб. робота №6 ( $L_6$ )	1	Laba6_1: 1,5	Laba6_4: 1,5	3	2	1	0	3	2	1	0					$L_6^{\min} : 6$	$L_6^{\max} : 10$
7	Лаб. робота №7 ( $L_7$ )	2	Laba7: 2		3	2	1	0	3	2	1	0					$L_7^{\min} : 6$	$L_7^{\max} : 10$
8	Лаб. робота №8 ( $L_8$ )	3	Laba8: 3		4	2	1	0	5	3	1	0					$L_8^{\min} : 8$	$L_8^{\max} : 15$
9	Контрольна робота за лекційним матеріалом (C)												15	13	10	0	$C^{\min} : 10$	$C^{\max} : 15$
	Загалом:																60	100
	Загалом на оцінку «відмінно»	13	21		25				26				15					100
	Загалом на оцінку «задовільно»	13	21				8				8				10		60	

## Реактивність: основні поняття.

У реактивному програмуванні дані розглядаються як потоки, а події у системах віконного інтерфейсу можуть розглядатися як потоки, різноманітні елементи яких мають оброблятися однаково. Реактивна модель програмування надає засоби для збору подій з різних джерел у потік, фільтрації потоків, різних перетворень над потоками, виконання тих чи інших дій над елементами потоків і т. д. Ця модель програмування містить у своїй основі засоби асинхронної обробки та керування розкладом асинхронних дій.

Реальним реактивним програмам внутрішньо притаманний асинхронний принцип роботи. Перш ніж переходити до асинхронного порядку виконання та управління розкладом роботи асинхронних потоків, треба спершу пояснити низку теоретичних принципів та відповідних мовних конструкцій.

Коротка відповідь на питання, чим займається реактивне програмування, звучить так: конкурентністю та паралелізмом. А якщо вдається до неформальної термінології, то реактивне програмування усуває «пекло зворотних викликів», які є неминучим результатом імперативного вирішення реактивних і асинхронних задач.

Реактивне програмування корисне у таких ситуаціях:

1) Обробка подій, ініційованих користувачем, наприклад: переміщення миші, клацання мишею, введення з клавіатури, зміна сигналів GPS в наслідок переміщення користувача разом зі своїм пристроєм, сигнали від вбудованого гіроскопа, події торкання пальцем і т.д.

2) Обробка будь-яких подій введення-виведення від диска чи мережі, що характеризуються наявністю затримки. У цих випадках введення-виведення за своєю природою асинхронне (відправлено запит, минає час, потім отримано - або не отримано - відповідь, що запускає подальшу обробку).

3) Обробка подій або даних, що надходять до додатку від виробника, якого він не може контролювати (системні події сервера, вищезгадані події користувача, сигнали від обладнання, події аналогових датчиків і т.д.)

Таким чином, сучасні програми повинні бути чутливі і мати можливість обробляти дані з різних джерел одночасно. Існуючі на даний момент техніки в сучасному JavaScript ні як не приведуть нас до бажаного. Вони не дозволяють масштабувати наш код тому, що цей код експоненційно ускладнюється, адже часто до нього додається паралелізм.

У реактивному програмуванні основною парадигмою є - більш простим та природним способом працювати з асинхронним кодом. При цьому, потоки подій, які ми будемо називати Observables, є чудовим способом обробки подій. При допомозі Observable можна створювати об'єкти, які ініціюють асинхронні потоки, а "реактивне мислення" та RxJS значною мірою покращують існуючі техніки.

## Коротко

Реактивність – це спосіб автоматично оновлювати систему залежно від зміни [потоків даних](#).

Потік даних – будь-яка послідовність подій із будь-якого джерела, впорядкована у часі. Наприклад, у кейлогері - програмі, яка записує натискання клавіш на клавіатурі, потоком даних будуть сигнали натискання клавіш.

Реактивне програмування – парадигма у програмуванні, у якій програма більше зосереджена на керуванні потоками даних, таким чином описуючи взаємозв'язки між ними.

## Що таке реактивність?

**Реактивність з точки зору користувача – це можливість програмного засобу миттєво реагувати на його дії.** Наприклад, почали вводити текст, він одразу ж з'являється у полі. Натиснули на кнопку, форма відразу змінилася, з'явилися ладери і т.д.

Для розробника цей термін має трохи інше значення. Коли йдеться про реактивність, фокус зміщується на дані. **Для розробника реактивність – це робота с асинхронними потоками даних.** Наприклад напишемо код, щоб скласти два числа:

```
let a= 3
let b= 4
const sum=a+b
console.log(sum)
// 7
```

У змінній sum тепер зберігається сума чисел, і ми знаємо, що вона ніколи не зміниться. Якщо змінити значення змінних a і b після додавання, це вже не вплине на результат:

```
let a= 3
let b= 4
const sum=a+b
console.log(sum)
// Змінюємо значення доданків
a= 10
b= 8
console.log(sum)
// Досі 7
```

Щоб отримати нову суму, потрібно наново скласти ці цифри. Здається, що все логічно, але що якщо ці числа вводить користувач, і вони можуть змінюватися? У такому разі було б зручніше, якби програма могла самостійно оновлювати значення.

```
// це псевдокод, цей код не працює!
```

```
let a$= 3
```

```
let b$= 4
```

```
const sum$=a$+b$
```

```
console.log(sum$)
```

```
// 7
```

```
a$= 2
```

```
b$= 9
```

```
console.log(sum$)
```

```
// 11
```

```
a$= -4
```

```
console.log(sum$)
```

```
// 5
```

Приклад вище несправжній, він не працюватиме таким чином, але дає можливість зрозуміти, яку проблему може вирішувати реактивність.

У фронтенді ми можемо пов'язати будь-які зміни в інтерфейсі зі змінами даних усередині програми. **Реактивність з погляду фреймворків – це оновлення інтерфейсу з урахуванням змін стану.**

У реактивності дані часто представляються як потік – послідовність подій, впорядкованих за часом.



Потік даних можна почати обробляти в будь-який час, і всі нові події можуть бути оброблені. Таким чином, реактивність – це зручний спосіб синхронізувати дані, наприклад з інтерфейсом, який бачить користувач. Реактивність допомагає зосередитися на даних, як вони пов'язані між собою, що набагато ближче до бізнес-логіки.



## Лекція №1. Реактивність: основні поняття. Патерни OBSERVER та ITERATOR.

### Маніфест реактивності

#### Типи реактивності

Основна ідея реактивності будується на [патерні «Спостерігач» \(Observer\)](#). Це [поведінковий патерн проектування](#), який створює механізм підписки, що дозволяє іншим стежити і реагувати на події, що відбуваються в джерелі. А от коли підписники дізнаються про оновлення, залежить від типу реактивності. Вона буває двох типів: push та pull.

#### Push-реактивність

Коли у реактивній системі методом push відбувається зміна, вона самостійно проштовхує (від англ. push – «штовхати») цю зміну всім підписникам. При такому типі реактивності всі підписники будуть отримувати актуальні зміни одразу, як вони відбулися.

Якщо ми підписуємось на подію (наприклад, [клік](#)), то браузер одразу повідомить всіх підписників, коли вона трапиться.

```
e:\Project(RxJS)\Project2\  
document.addEventListener('click', () => {  
  console.log('Я зреагував на клік!')  
})
```

Метод `EventTarget.addEventListener()` реєструє певний обробник події, викликаний на `EventTarget`.

`EventTarget` може бути `Element`, `Document`, `Window`, або будь-яким іншим об'єктом, що підтримує події.

У реальному світі можна підписатися на повідомлення в додатку. Коли відбудеться оновлення, ми отримаємо на телефон push-повідомлення.

В екосистемі JavaScript найпопулярніший спосіб використовувати push-реактивність – це використовувати бібліотеку [RxJs](#).

```
e:\Project(RxJS)\Project2\  
// створюємо джерело  
const clicks$=fromEvent(document, 'click')  
clicks$.subscribe(event=> {  
  console.log('Я зреагував на клік реактивно!')  
})
```

Функція `fromEvent` дозволяє згенерувати `Observable` з події за переданим джерелом подій, наприклад посиланням, кнопкою або об'єктом типу `EventEmitter`.

#### Методи Observable

Ось деякі методи:

- of(arg)

Перетворює будь-яке передане йому значення або значень, розділених комою, в об'єкт Observable.

- from(iterable)

Перетворює значення, що ітеруються, в об'єкт Observable. Наприклад:

```
of([1,2,3]).subscribe( x => {
```

```
  console.log(x);
```

```
});
```

```
from([1,2,3]).subscribe( x => {
```

```
  console.log(x);
```

```
})
```

У наведеному вище прикладі, у першому випадку в потік зайде весь масив, коли в другому варіанті будуть послідовно згенеровані елементи масиву. За допомогою функції from можна також згенерувати Observable з промісу.

- fromPromise(promise)

Перетворює Promise на об'єкт Observable

- fromEvent(element, eventName)

Створює об'єкт Observable, додає спостерігача елементу та слухає події, наприклад, DOM-events.

Програмування потоків інакше обробляє виникнення події, Observable реалізує "ліниву поведінку", тобто виклик або підписка буде ізольованою операцією: кожен виклик функції викликає окремий процес, а кожна Observable-підписка запускає свій окремий процес.

Недоліком push-реактивності можуть бути обчислення, що повторюються, так як при зміні даних всі підписники можуть заново проробляти свою роботу з цими даними. Це може бути проблемою при частих оновленнях, коли система може бути просто завалена великим потоком даних, а всі підписники без кінця робитимуть обчислення.

Припустимо, що є додаток-дашборд, на якому виводиться актуальна інформація із графіками та змінами. Сторінка може складатися з різних блоків, і кожному потрібна інформація. Коли дані надходять безперервно, вся сторінка постійно оновлюватиметься в різних місцях. Одні обчислення можуть бути складнішими і довшими за інші, через що в черзі будуть накопичуватися нові дані, які потрібно порахувати.

У JavaScript-кодi, де використовуються бібліотеки для реактивності, часто можна зустріти позначення змінних через знак долара \$, як у прикладі вище. Таким чином, змінній дають

спеціальний префікс, показуючи, що в ній знаходиться потік (stream) і на нього можна підписатися. Другим варіантом може бути додавання на початку символу \$: \$clicks=rxjs.Observable.fromEvent(...).

```
e:\Project(RxJS)\Project2\  
var stream$=rxjs.Observable.create(function(observer){  
  console.log('stream$ was created!');  
  observer.next('One');  
  setTimeout(function(){  
    observer.next('After 5 seconds!');  
  },5000);  
  setTimeout(function(){  
    observer.next('After 3 seconds!');  
  },3000);  
  setTimeout(function(){  
    observer.next('After 2 seconds!');  
  },2000);  
  setTimeout(function(){  
    observer.complete();  
  },2500);  
  observer.next("Two");  
});  
stream$.subscribe(  
  function(data){  
    console.log('Subscribe:',data);  
  },  
  function(error){  
    console.log('Error:',error);  
  },  
  function(){  
    console.log('Completed!');  
  }  
);
```

// це псевдокод, цей код не працює!

```
let a$= 3
```

```
let b$= 4
```

```
const sum$=a$+b$
```

```
console.log(sum$)
```

```
// 7
```

```
a$= 2
```

```
b$= 9
```

```
console.log(sum$)
```

```
// 11
```

```
a$= -4
```

```
console.log(sum$)
```

```
// 5
```

e:\Project(RxJS)\Project2\

// Цей код працює реактивно при допомозі RxJS

```
const a1 = from([2, 55,25])
```

```
.pipe(
```

```
  zipWith(interval(1200), x => x));
```

```
|
```

```
const b1 = from([4, 20])
```

```
.pipe(
```

```
  zipWith(interval(1500), x => x));
```

```
|
```

```
a1
```

```
.pipe(
```

```
  combineLatestWith(b1,(x, y) => x + y))
```

```
.subscribe(val => console.log("c1=" + val));
```

## Pull реактивність

Реактивність методом pull працює протилежно push-реактивності. Обчислення, викликані зміною даних у джерелі, тут відкладаються до того часу, поки не будуть потрібні. За такого типу реактивності підписники витягнуть (від англ. «pull» – тягнути) нові дані, лише коли оновиться вся система.

Прикладом реального життя може бути патерн pull-to-refresh з мобільних додатків. Наприклад, у Twitter користувач може потягнути контент із верхнього краю трохи вниз, а потім відпустити. Така дія змусить стрічку твітів оновитись, і користувач зможе побачити актуальні дані. При push-підході стрічка оновлювалася б щоразу самостійно, щойно з'являється новий твіт.

Pull-механіку рідко можна зустріти в бібліотеках, тому часто вона лише частково присутня в деяких місцях, наприклад в [MobX](#).

Недоліком pull-підходу є проблеми з продуктивністю — щоразу оновлювати всю систему, щоб повідомити підписників нових даних, може бути витратним.

Таким чином, ідеї реактивності в результаті призвели до появи нової парадигми, що базується на [асинхронному](#) управлінні потоками даних. Хоча раніше вже було сказано, що підписки на події у браузері теж є реактивною моделлю, реактивне програмування зводить ці ідеї в абсолют. Це означає, що потоки даних можна створювати з чого завгодно і як завгодно ними управляти: схрещувати, трансформувати, фільтрувати і т. д. Наприклад, у додатку, написаному з

використанням підходу [MVC](#), за допомогою реактивного програмування можна створити автоматичне відображення змін із Model у View, і навпаки, з View до Model.

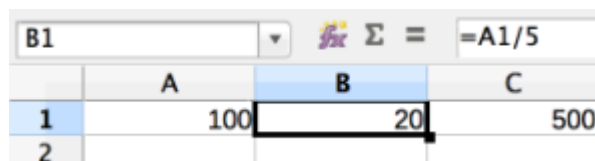
Реактивне програмування допомагає абстрагуватися від опису дій у коді безпосередньо та зосередитися на взаємозв'язку даних. Воно створено, щоб спростити створення програм із великою кількістю зв'язків.

У сучасному фронтенді можна зустріти складні клієнтські програми, де зміна значень у полях веде до цілого ряду оновлень інтерфейсу. Реактивне програмування може полегшити створення таких систем. Існує навіть цілий [маніфест реактивних систем](#), який описує, як повинна поводитися реактивна система.

Саме собою реактивне програмування рідко зустрічається у чистому вигляді. Часто воно поєднане з іншими парадигмами. Так виникли такі змішання, як імперативне реактивне програмування, об'єктно-орієнтоване реактивне програмування та функціональне реактивне програмування.

### Таблиці типу Excel реактивні

Почнемо із розгляду найбільш типового прикладу реактивної системи: таблиці. Ми всі використовували їх, але ми рідко зупиняємося і думаємо, наскільки вони дуже інтуїтивні. Допустимо, у нас є значення в клітинці A1 таблиці. Потім ми можемо посилатися на нього в інших осередках електронної таблиці, і щоразу, коли ми змінюємо A1, кожен осередок, що залежить від A1, автоматично оновлює своє значення.



	A	B	C
1	100	20	500
2			

Така поведінка здається нам природною для таблиць. Нам не потрібно було повідомляти комп'ютер про оновлення клітинок, що залежать від A1 або про те, як це зробити. Ці осередки самі реагували на зміну. Ми ж просто декларуємо поведінку, і більше не переймаємося тим, як комп'ютер обчислює результати.

Це те, чого прагне реактивне програмування. Наше завдання оголосити відносини між учасниками, а програма обчислить та підставить нове значення.

### Події миші також можуть бути потоком

Щоб зрозуміти, як події можуть бути потоками, давайте повернемося до нашої мікропрограми, де ми використовували клацання миші як нескінченну послідовність подій, що генеруються в режимі реального часу при натисканні користувачем. Ерік Мейєр, винахідник RxJS, запропонував у статті таку думку: «Твоя миша - база даних».

У реактивному програмуванні ми дивимося на клацання миші як на безперервний потік подій, які можемо запитувати і маніпулювати. Роздуми про потоки замість ізольованих значень відкриває зовсім новий спосіб програмування, у якому ми можемо маніпулювати цілими послідовностями значень, які ще навіть не створені.

Це відрізняється від того, що було раніше, де зберігаються дані, такі як база даних або масив і код чекає, поки ці дані будуть доступні, перш ніж їх використати. Якщо дані ще не доступні (наприклад, запит мережі), то чекаємо на них і використовуємо їх лише тоді, коли вони стають доступними.



Але можна думати про потокову послідовність як про масив даних, в якому елементи розділені за часом, а не за пам'яттю:



**Побачити свою програму як послідовність даних – ось ключ до розуміння програмування на RxJS!** Це потребує практики. Насправді більшість даних, які ми використовуємо в будь-якому додатку, можуть бути виражені у вигляді послідовності.

### Запит послідовності

Давайте реалізуємо просту версію цього потоку за допомогою традиційних обробників подій JavaScript. Щоб записати координати X та Y для миші, ми могли б написати приблизно так:

```
e:\Project(RxJS)\Project2\  
document.body.addEventListener('mousemove', (e) => {  
  console.log(e.clientX, e.clientY)  
})
```

Цей код виводитиме в консоль X- та Y-координати кожного переміщення миші. Вивод виглядає так:

252 183

211 232

153 323

...

Схоже на послідовність, чи не так? Проблема, звичайно ж, у тому, що маніпулювати подіями не так просто, як маніпулювати масивами. Наприклад, якщо ми хочемо змінити попередній код, щоб він реєстрував лише перші 10 кліків у правій частині екрана, ми напишемо щось таке:

```
e:\Project(RxJS)\Project2\  
let clicks = 0  
document.addEventListener('click', function registerClicks (e) {  
  if (e.clientX > window.innerWidth / 2) {  
    clicks += 1  
    console.log(e.clientX, e.clientY)  
    if (clicks === 10) {  
      document.removeEventListener('click', registerClicks)  
    }  
  }  
})
```

Якщо ми будемо рахувати кількість кліків, то нам потрібно десь зберігати стан. У цьому випадку ми запровадили глобальну змінну, тобто зовнішній стан. Також нам потрібно перевірити дві різні умови та використовувати вкладені умовні блоки. І коли ми робимо 10 клік, нам потрібно зняти обробник, щоб він у пам'яті не висів.

### **Побічні ефекти (side effects) та зовнішній стан (external state)**

Якщо виконання коду починає впливати на зовнішню середу ми називаємо це побічним ефектом. Зміна змінних, зовнішніх по відношенню до нашої функції, виведення в консоль або оновлення значень в базі даних, все це є прикладами побічних ефектів.

Приклад: Зміна значення змінної, яка існує всередині нашої функції, безпечна. Але якщо ця змінна виходить за межі нашої функції, інші функції можуть змінити її значення. Це означає, що наша функція більше не контролюється і не може припускати, що зовнішня змінна містить очікуване значення. Нам потрібно буде стежити за змінною та додавати якісь перевірки, щоб переконатися, що її значення відповідає очікуваному. І в цей момент ми будемо змушені додавати

код, який взагалі не має відношення до нашої програми, що робить його більш складним і схильним до помилок.

Незважаючи на те, що для створення будь-якої програми неминучі побічні ефекти, ми повинні прагнути до того, щоб у нашому коді їх було якнайменше. Це особливо важливо в реактивних програмах, де ми маємо багато рухомих частин, які змінюються з часом.

Нам вдалося реалізувати задумане, але в результаті вийшов досить складний код для такого простого завдання. Цей код складний для розширення і не є очевидним для розробника, який дивиться на нього вперше. Що ще важливіше, ми зробили його схильним до помилок, тому що ми використовуємо зовнішній стан.

По суті, все, що нам потрібно в цій ситуації – це запросити базу даних кліків. Якби ми мали справу з реляційною базою даних, ми використовували б декларативну мову SQL і записали б наступне:

```
SELECT x, y FROM clicks LIMIT 10
```

А що, якби ми ці події кліка мишкою обробляли як потоки даних, які можна запросити і перетворити? Врешті-решт цей потік нічим не відрізняється від бази даних, яка генерує значення в реальному часі. Все, що нам потрібне, це тип даних, який абстрагує цю концепцію для нас.

Давайте поглянемо як саме можна це зробити за допомогою RxJS і його типу даних Observable:

```
e:\Project(RxJS)\Project2\  
let clicks = 0  
document.addEventListener('click', function  
registerClicks (e) {  
  if (e.clientX > window.innerWidth / 2) {  
    clicks += 1  
    console.log(e.clientX, e.clientY)  
    if (clicks === 10) {  
      document.removeEventListener('click',  
registerClicks)  
    }  
  }  
})
```

```
e:\Project(RxJS)\Project2\  
fromEvent(document, 'click')  
.pipe(  
  filter(c => c.clientX > window.innerWidth / 2),  
  take(10)  
)  
.subscribe(  
  (c) => console.log(c.clientX, c.clientY)  
)
```



Цей код повністю замінює попередній приклад, а читається він так:

*Створи об'єкт Observable, який стежитиме за DOM-подіями кліка миші по document і відфільтруй тільки кліки які у правій частині екрана, потім виведи координати в консоль лише перших 10 кліків у міру їх появи.*

При цьому, код легко читається, навіть якщо ви не знайомі з ним. Крім того, немає необхідності створювати зовнішні змінні для збереження стану, що робить код самодостатнім та ускладнює внесення помилок. Тепер також не потрібно стежити за тим, щоб не було витoku пам'яті (прибирати за собою), тобто знімати обробники подій, коли вони стають непотрібними, і тому немає мови про витік пам'яті.

Разом, Observable надає нам послідовність чи потік подій, яким ми можемо маніпулювати як хочемо, замість однієї ізольованої події щоразу. Робота з послідовностями дає дуже багато переваг. Можна легко об'єднувати, трансформувати або фільтрувати Observable. Таким чином, можна перетворити наші події на якусь матеріальну структуру даних, яка така ж проста у використанні, як і масив, але набагато гнучкіша.

## **Observers та Iterators**

І так, щоб зрозуміти, що таке Observable, нам потрібно подивитися на його фундаментальну складову, а саме патерни проектування Observer та Iterator.

### **PATTERN OBSERVER**

Досить очевидно, що Observables має прямий зв'язок із патерном Observer. Цей патерн визначає залежність типу «один до багатьох» між об'єктами таким чином, що при зміні стану одного об'єкта всі, хто стежить за ним, сповіщаються про цю зміну.

#### **Назва та класифікація патерну**

Спостерігач – патерн поведінки об'єктів (поведінковий патерн).

#### **Призначення**

Визначає залежність типу "один до багатьох" між об'єктами таким чином, що при зміні стану одного об'єкта всі, що залежать від нього, сповіщаються про це і автоматично оновлюються.

#### **Інші назви**

Dependents (підлегли), Publisher-Subscriber (видавець – підписник).

#### **Мотивація**

Одним із типових побічних ефектів розбиття системи на взаємодіючі класи є необхідність узгодження стану взаємозалежних об'єктів. Проте узгодженість має досягатися при допомозі жорсткої зв'язаності класів, оскільки це знижує можливості їх повторного використання.

Паттерн спостерігач визначає, як встановлюються такі відносини. Ключовими об'єктами у ньому є суб'єкт та спостерігач. У суб'єкта може бути скільки завгодно залежних від нього

спостерігачів. Усі спостерігачі повідомляються про зміни у стані суб'єкта. Отримавши повідомлення, спостерігач опитує суб'єкта, щоб синхронізувати свій стан.

Така взаємодія часто ще називається відношенням видавець — підписник. Суб'єкт видає або публікує повідомлення та розсилає їх, навіть не маючи інформації про те, які об'єкти є підписниками. На отримання повідомлень може підписатися необмежена кількість підписників (спостерігачів).

### **Застосовність**

Основні умови для застосування патерну спостерігач:

- абстракція має два аспекти, один з яких залежить від іншого. Інкапсуляції цих аспектів у різні об'єкти дозволяють змінювати та повторно використовувати їх незалежно;
- при модифікації одного об'єкта потрібно змінити інші, і ви не знаєте скільки саме об'єктів потрібно змінити;
- один об'єкт повинен сповіщати інших, не роблячи припущень про об'єкти, що сповіщуються. Іншими словами, об'єкти не повинні бути тісно пов'язані між собою.

### **Учасники**

Subject — суб'єкт: має інформацію про своїх спостерігачів. За суб'єктом може «стежити» будь-яка кількість спостерігачів; надає інтерфейс для приєднання та видалення спостерігачів;

Observer - спостерігач: визначає інтерфейс оновлення для об'єктів, які повинні повідомлятися про зміну суб'єкта;

ConcreteSubject - конкретний суб'єкт: зберігає стан, що представляє інтерес для конкретного спостерігача ConcreteObserver; надсилає інформацію своїм спостерігачам, коли відбувається зміна;

ConcreteObserver - конкретний спостерігач: зберігає посилання на об'єкт класу ConcreteSubject; зберігає дані, які мають бути узгоджені з даними суб'єкта; реалізує інтерфейс оновлення, визначений у класі Observer, щоб підтримувати узгодженість із суб'єктом.

### **Відносини**

- об'єкт ConcreteSubject повідомляє своїх спостерігачів про будь-яку зміну, яка могла б призвести до неузгодженості станів спостерігача та суб'єкта;
- після отримання від конкретного суб'єкта повідомлення про зміну об'єкт ConcreteObserver може запросити у суб'єкта додаткову інформацію, яку використовує для того, щоб опинитися у стані, узгодженому зі станом суб'єкта.

### **Результати**

Паттерн спостерігач дозволяє змінювати суб'єкти та спостерігачі незалежно один від одного. Суб'єкти дозволяється повторно використовувати без участі спостерігачів і навпаки. Це дозволяє додавати нових спостерігачів без модифікації суб'єкта чи інших спостерігачів.

Основні переваги та недоліки патерну спостерігач:

- абстрактна пов'язаність суб'єкта та спостерігача. Суб'єкт має інформацію лише ту, що у нього є ряд спостерігачів, кожен із яких підпорядковується простому інтерфейсу абстрактного класу Observer. Суб'єкту невідомі конкретні класи спостерігачів. Таким чином, зв'язки між суб'єктами та спостерігачами носять абстрактний характер і зведені до мінімуму.

Оскільки суб'єкт і спостерігач не тісно пов'язані, вони можуть перебувати на різних рівнях абстракції системи. Суб'єкт нижчого рівня може повідомляти спостерігачів на верхніх рівнях, не порушуючи ієрархії системи. Якби суб'єкт і спостерігач являли собою єдине ціле, то утворюваний об'єкт, або перетинав би межі рівнів (порушуючи принцип їх формування), або повинен був перебувати на якомусь одному рівні (порушуючи абстракцію рівня);

- Підтримка широкомовних комунікацій. На відміну від звичайного запиту, для повідомлення, що надсилається суб'єктом, не потрібно задавати певного одержувача. Повідомлення автоматично надходить всім об'єктам, що підписалися на нього. Суб'єкта не цікавить скільки існує таких об'єктів; від нього потрібно лише повідомити своїх спостерігачів. Таким чином, ми можемо у будь-який час додавати та видаляти спостерігачів. Спостерігач сам вирішує, обробити отримане повідомлення чи ігнорувати його;

- Несподівані оновлення. Оскільки спостерігачі не мають інформації один про одного, їм невідомо і про те, у що обходиться зміна суб'єкта. Безневинна на перший погляд операція над суб'єктом може викликати цілу низку оновлень спостерігачів та об'єктів, що залежать від них. Більше того, нечітко визначені чи погано підтримувані критерії залежності можуть стати причиною непередбачених оновлень, відстежити які дуже складно.

Проблема посилюється ще й тим, що простий протокол оновлення не містить жодної інформації про те, що саме змінилося в суб'єкті. Без додаткового протоколу, який дозволяє отримати інформацію про зміни, спостерігачі будуть змушені зробити складну роботу для непрямого отримання такої інформації.

## **ПАТТЕРН ITERATOR (ІТЕРАТОР)**

### **Назва та класифікація патерну**

Ітератор - паттерн поведінки об'єктів (поведінковий паттерн).

### **Призначення**

Надає спосіб послідовного звернення до всіх елементів складового об'єкта без розкриття його внутрішнього змісту.

## Інші назви

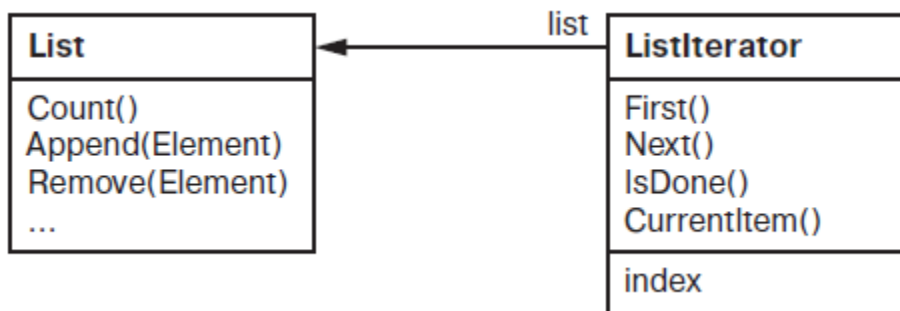
Cursor (курсор).

## Мотивація

Складовий об'єкт, скажімо, список повинен надавати спосіб доступу до своїх елементів, не розкриваючи їх внутрішню структуру. Більше того, іноді потрібно обходити список по-різному в залежності від завдання, що розв'язується. Але навряд чи необхідно засмічувати інтерфейс класу List операціями для різних варіантів обходу, навіть якщо їх можна передбачити заздалегідь. Крім того, іноді буває потрібно, щоб в той самий момент діяло декілька активних обходів списку.

Все це дозволяє зробити патерн Ітератор. Основна його ідея полягає в тому, щоб за звернення до елементів та спосіб обходу відповідав не сам список, а окремий об'єкт-ітератор. У класі Iterator визначено інтерфейс доступу до елементів списку. Об'єкт цього класу відстежує поточний елемент, тобто він має у своєму розпорядженні інформацію, які елементи вже відвідувалися.

Наприклад, для класу List міг би існувати клас ListIterator; відносини між цими класами могли б виглядати так:



Перш ніж створювати екземпляр класу ListIterator, необхідно мати список для обходу. З об'єктом ListIterator можна відвідати всі елементи списку. Операція CurrentItem повертає поточний елемент списку, First ініціалізує поточний елемент першим елементом списку, Next робить поточним наступний елемент, а IsDone перевіряє, чи обхід за останній елемент не вийшов, якщо вийшов — то обхід завершується.

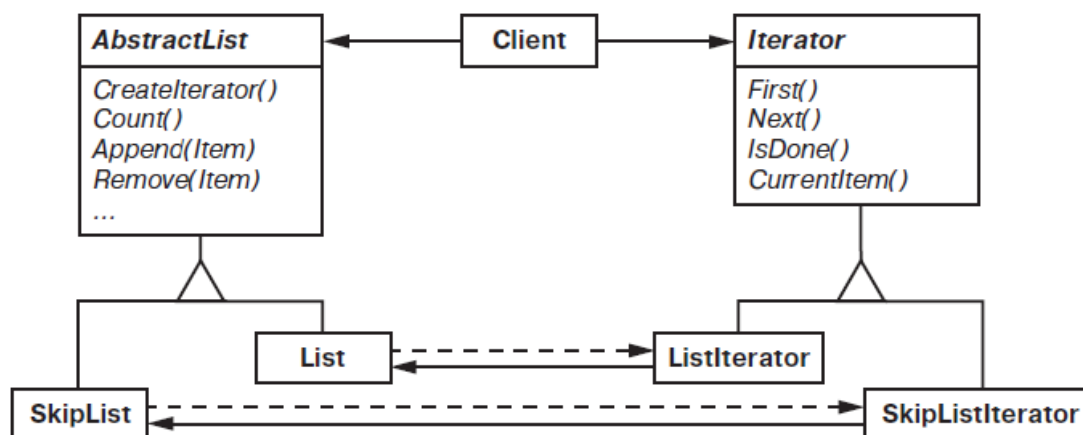
Відділення механізму обходу від об'єкта List дозволяє визначати ітератори, що реалізують різні стратегії обходу, не перераховуючи їх в інтерфейсі класу List. Наприклад, ітератор FilteringListIterator міг би надавати доступ лише до тих елементів, які відповідають критеріям фільтрації.

Слід зазначити, що між ітератором та списком існує тісний зв'язок. Клієнт повинен знати, що він обходить саме список, а не якусь іншу агреговану структуру. Тому клієнт прив'язаний до конкретного способу агрегування. Було б краще, якби клас агрегату можна було змінювати без

зміни коду клієнта. Для цього можна узагальнити концепцію ітератора та розглянути поліморфну ітерацію.

Допустимо, у вас є ще клас `SkipList`, що реалізує список з перепустками (skiplist) - імовірнісну структуру даних. Потрібно мати можливість писати код, здатний працювати з об'єктами як класу `List`, і класу `SkipList`.

Визначимо клас `AbstractList`, у якому оголошено загальний інтерфейс для маніпулювання списками. Ще нам знадобиться абстрактний клас `Iterator`, який визначає загальний інтерфейс ітерації. Потім ми змогли б визначити конкретні підкласи класу `Iterator` для різних реалізацій списку. Внаслідок цього механізм ітерації перестав залежати від конкретних агрегованих класів.



Як створюється ітератор? Оскільки необхідно писати код, який залежить від конкретних підкласів `List`, то не можна просто створити екземпляр конкретного класу. Натомість відповідальність за створення відповідних об'єктів-списків буде покладено на самі об'єкти-списки; ось чому буде потрібна операція `CreateIterator`, за допомогою якої клієнти зможуть вимагати об'єкт-ітератор.

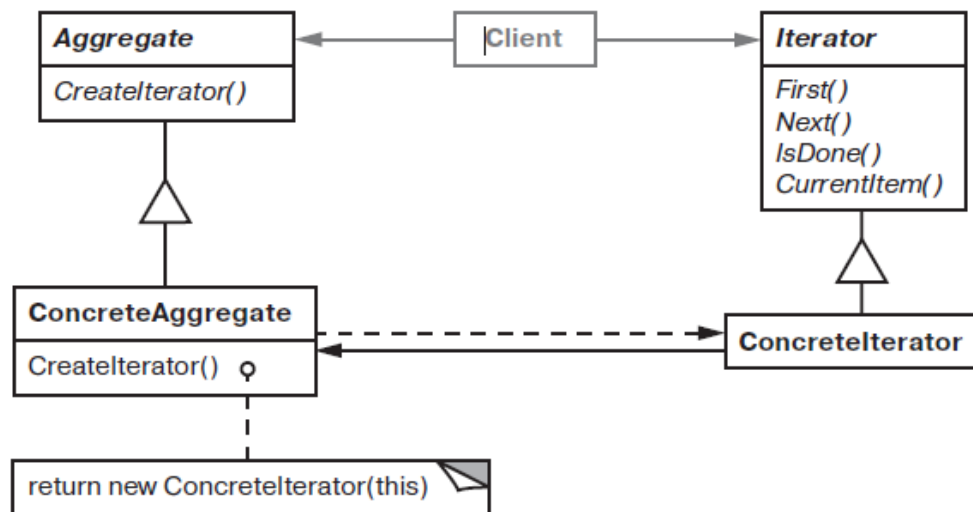
`CreateIterator` - це приклад використання патерну фабричний метод. У цьому випадку він служить для того, щоб клієнт міг запросити у об'єкта-списку відповідний ітератор. Застосування фабричного методу призводить до двох ієрархій класів — однієї для списків, іншої для ітераторів. Фабричний метод `CreateIterator` пов'язує ці дві ієрархії.

### Застосовність

Основні умови для застосування патерну:

- звернення до вмісту агрегованих об'єктів без розкриття їх внутрішнього подання;
- підтримка кількох активних обходів одного і того ж агрегованого об'єкта;
- надання одноманітного інтерфейсу для обходу різних агрегованих структур (тобто підтримки поліморфної ітерації).

### Структура



### Учасники

Iterator - ітератор: визначає інтерфейс для доступу та обходу елементів;

ConcreteIterator – конкретний ітератор: реалізує інтерфейс класу Iterator; стежить за поточною позицією при обході агрегату;

Aggregate – агрегат: визначає інтерфейс для створення об'єкта-ітератора;

ConcreteAggregate - конкретний агрегат: реалізує інтерфейс створення ітератора та повертає екземпляр відповідного класу ConcreteIterator.

### Відносини

ConcreteIterator відстежує поточний об'єкт в агрегаті і може обчислити той, що іде за ним.

### Результати

Основні переваги та недоліки патерну ітератор:

- Підтримка різних способів обходу агрегату. Складні агрегати можна обходити по-різному. Наприклад, для генерації коду та семантичних перевірок потрібно обходити дерева синтаксичного аналізу. Генератор коду може обходити дерево у внутрішньому чи прямому порядку. Ітератори полегшують зміну алгоритму обходу — досить просто замінити один екземпляр ітератора іншим. Для підтримки нових видів обходу можна визначити підкласи класу Iterator;
- Спрощення інтерфейсу класу Aggregate. Наявність інтерфейсу для обходу в класі Iterator робить зайвим дублювання цього інтерфейсу в Aggregate. Тим самим інтерфейс агрегату спрощується;
- Можливість наявності кількох активних обходів для даного агрегату. Ітератор стежить за інкапсульованим у ньому самому станом обходу, тому одночасно можуть існувати кілька обходів агрегату.

І так, у реалізацію патерна Observable входять два об'єкти. Це Видавець (Publisher) та Підписник (Subscriber). Видавець - це об'єкт за яким хочуть спостерігати підписники, власне він і

зберігатиме в собі список цих підписників, які підписалися на нього. Щойно Видавець змінює свій стан, він одразу сповіщає всіх своїх Підписників шляхом виклику їхнього методу оновлення. (У більшості пояснень патерна Observer цей об'єкт називається Subject, але щоб уникнути плутанини з власним типом Subject RxJS, ми називаємо його Видавець).

Наступний код реалізує спрощену версію патерну:

```
e:\Project(Javascript)\Project0\  
function Publisher () {  
  this.subscribers = []  
}  
// Додаємо новий метод для всіх об'єктів нащадків:  
Publisher.prototype.add = function (subscriber) {  
  this.subscribers.push(subscriber)  
}  
  
Publisher.prototype.remove = function (subscriber) {  
  const index = this.subscribers.indexOf(subscriber)  
  this.subscribers.splice(index, 1)  
}  
  
Publisher.prototype.notify = function (message) {  
  this.subscribers.forEach((subscriber) => {  
    subscriber.update(message)  
  })  
}
```

Об'єкт Publisher зберігає динамічний список Subscribers як subscribers, який є масивом. Також об'єкт Publisher містить метод add за допомогою якого будуть додаватися підписники. Власне, ці підписники і будуть сповіщені при виклику методу notify.

У наступному коді ми створюємо два об'єкти, які підписуються на оповіщення екземпляра об'єкта Publisher:

```
// будь-який об'єкт, що має метод update, буде працювати  
const subscriber1 = {  
  update(message) {  
    console.log('Subscriber 1 received:', message)
```

```

    }
  }

  const subscriber2 = {
    update(message) {
      console.log('Subscriber 2 received:', message)
    }
  }

  const notifier = new Publisher
  notifier.add(subscriber1)
  notifier.add(subscriber2)

  notifier.notify('Hello there!')

```

Після запуску програми ми побачимо в консолі:

Subscriber 1 received: Hello there!

Subscriber 2 received: Hello there!

Об'єкти `subscriber1` і `subscriber2` будуть сповіщені про зміну стану об'єкта `notifier` щоразу як викликатиметься метод `notify`. Зауважте, об'єктам `subscriber1` і `subscriber2`, не потрібно нічого робити!

## Паттерн Rx та об'єкт Observable

Незважаючи на те, що патерни `Observer` та `Iterator` сильні власними силами, їх комбінація додає ще більше потужності. Цю комбінацію іноді ще називають як патерн `Rx`, названим на честь бібліотек `Reactive Extensions`.

`Observable` є основним елементом `Rx`. `Observable` транслює свої значення по порядку подібно до ітератора, але замість того, щоб споживачі запитували наступне значення, `Observable` посилає значення споживачам у міру їх появи. Він має аналогічну роль `Publisher`'а в патерні `Observer`: транслює значення та передає їх своїм підписникам.

Простіше кажучи, `Observable` - це послідовність, значення якої стають доступними з часом. Споживачі `Observables` є еквівалентом підписників у патерні `Observer`. Коли підписник підписується на `Observable`, він отримує значення у послідовності, коли вони стають доступними,



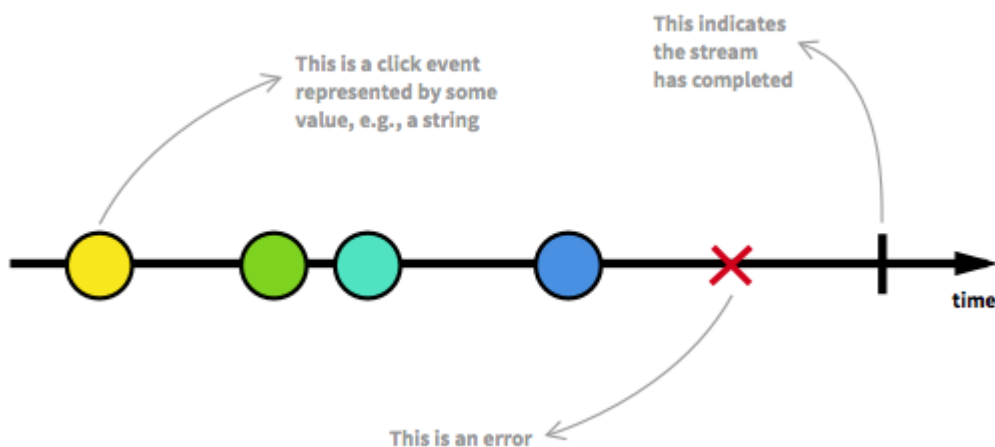
без необхідності вимагати їх. Можна сказати, що немає особливої різниці із традиційним патерном Observer. Але насправді є дві істотні відмінності:

- Observable не запускає потік значень, доки на них не буде підписано хоча б один підписник.
- Подібно до ітераторів, Observable може сигналізувати, коли послідовність завершена.

Використовуючи Observables, ми можемо об'явити, як реагувати на послідовність значень, які вони транслюють, замість реагувати на окремі значення. Ми можемо ефективно копіювати, перетворювати та запрошувати послідовність, і ці операції будуть застосовуватися до всіх значень послідовності.

### Потоки

Потік даних не що інше, як подія, що повторюється у часі. Для початку, можна подати у вигляді потоку послідовність кліків користувача:



Події, що ініціюються в потоці, можна розділити на три типи: значення, помилка і завершення потоку. У прикладі завершенням буде закриття користувачем сторінки. Потік може отримувати нескінченну кількість подій і в той же час, будь-яка одинична подія може бути представлена у вигляді потоку (той самий клік).

Потік вище зображено як діаграма. RxJS дозволяє використовувати такі діаграми для написання тестів.

```
--a---b--c---d---X---|-->
```

a, b, c, d події значення

X помилка

| завершення потоку

---> шкала часу

Крім кліків, веб-додаток, як правило, має вміти оперувати безліччю як синхронних, так і асинхронних подій, кожні з яких, у свою чергу, можуть бути одноразовими та багаторазовими. До асинхронних подій можна віднести:

- UI події, будь-яка взаємодія користувача з інтерфейсом
- Запити на сервер
- Події пристрою, пуш повідомлення, системні нотифікації
- Веб хукі

Асинхронна подія - подія, яка відбудеться в якийсь невизначений момент у майбутньому.

Повторення будь-якої з перерахованих вище асинхронних подій досить просто уявити у форматі потоку даних за проміжок часу. Складніше мислити реактивно щодо синхронних, послідовних подій, якими є, наприклад, масив

```
[1,2,3,4,5].forEach(item => {  
  item;// подія-значення  
});
```

Перебравши масив із 5 елементів, ми визначили синхронний потік, в якому були ініційовані 5 подій значень та одне завершення потоку. Процес аналогічний сесії користувача, який 5 разів натиснув на кнопку і закрив сторінку. Таким же чином у ролі послідовного потоку може виступати не тільки масив, а й будь-який ітерабельний об'єкт: Set, Map, String і т.д.

Якщо подивитися ще ширше, можна перекласти цю ж концепцію на реальне життя та уявити у вигляді потоку такі синхронні події як сон, читання книги, поїздка в автобусі та асинхронні – момент закипання чайника, замовлення їжі додому. Уявити у вигляді потоку можна все, що завгодно. І це підводить нас до головної ідеї реактивного програмування:

### **Чим обробляти потоки?**

Зі сказаного вище можна зробити висновок, що будь-який додаток - інкубатор всіляких потоків. Давайте поверхово розглянемо засоби їхньої обробки. Далі йтиметься про асинхронні події.

- EventTarget інтерфейс обробки різних UI подій у JavaScript. Клік користувача, наприклад, можна зловити так:

```
document.getElementById('button').addEventListener('click', event => {  
  // обробка
```

```
});
```

- EventEmitter використовується при побудові асинхронно-подійної архітектури в Node.js. Метод `eventEmitter.on()` використовується для реєстрації слухачів, а метод `eventEmitter.emit()` використовується для запуску події.

```
const emitter = new EventEmitter();
emitter.on('event', data => {
  // обробка
});
emitter.emit('event', Data); // Пуш події
```

- Функції зворотного виклику як варіант обробки одноразової асинхронної події, наприклад, читання файлу

```
const cb = function(err, data) {
  if (err) {
    // подія-помилка
  }
  // подія-значення
}
fs.readFile('data.txt', cb);
```

- Promise так само використовуються для одноразових асинхронних обчислень

```
const request = new Promise((resolve, reject) => {
  // імітація запиту до сервера. Функція-виконавець
  window.setTimeout(() => { resolve(true); }, 3000);
});
request
  .then(response => { ... }) // подія-значення
  .catch(err => { ... }) // подія-помилка
```

- Генератори- функції з можливістю призупинити виконання на деякий час, після чого відновити знову. Це дозволяє писати асинхронний код у синхронному стилі

```
function request(url) {
```

```

    ajaxCall(url, response => { it.next(response); });
  }

  function *main() {
    const result = yield request('http://some.url');
    const data = JSON.parse(result); // событие-значение
  }

  const it = main();
  it.next();

```

- Web sockets і Web workers
- Async функції було запропоновано як частину стандарту ES2016.
- Async generators спочатку були задумані для реалізації обробки багаторазових асинхронних подій.

Було б зручно мати єдиний інтерфейс взаємодії з будь-яким засобом обробки подій. І такий інтерфейс надає бібліотека RxJS (Reactive Extensions for Javascript).

З її допомогою можна комбінувати між собою потоки різних типів даних, перетворювати їх, скасовувати, приводити до певного типу і багато іншого. Observable - сутність, в RxJS використовується як потік даних.

### Маніфест реактивності

Для проектування додатків необхідний чіткий і зрозумілий підхід, а всі його окремі аспекти вже сформульовані: необхідно, щоб система була чутливою, стійкою, гнучкою і заснованою на обміні повідомленнями. Такі системи називатимемо Реактивними.

Програми, написані відповідно до принципів Реактивних Систем, відрізняються підвищеною гнучкістю, масштабованістю та слабкою пов'язаністю. Завдяки цьому їх простіше розробляти та модифікувати. Вони більш стійкі до відмов та коректно обробляють виняткові ситуації, уникаючи катастрофічних наслідків. Реактивні системи характеризуються високою чуйністю, забезпечуючи користувачам ефективний та інтерактивний зворотний зв'язок.

### Реактивні системи:

**Чутливі:** Система відповідає своєчасно, якщо це взагалі можливе. Чутливість є наріжним каменем зручного та корисного застосування, але, крім цього, вона дозволяє швидко виявляти проблеми та ефективно їх усувати. Чутливі системи орієнтовані на забезпечення швидкого і узгодженого часу відгуку, встановлюючи надійні верхні кордону, щоб забезпечити стабільну якість обслуговування. Така передбачувана поведінка, у свою чергу, спрощує обробку помилок,

підвищує впевненість кінцевого користувача у працездатності та сприяє подальшій взаємодії із системою.

**Стійкі:** Система залишається доступною навіть у випадку відмов. Це стосується не тільки високодоступних, критично важливих додатків — без стійкості будь-яка система при збої втрачає чутливість. Стійкість досягається за рахунок реплікації, стримування, ізоляції і делегування. Ефект від відмов утримується всередині компонентів, ізолюючи їх один від одного, що дозволяє їм виходити з ладу і відновлюватися, не порушуючи роботу системи в цілому. Відновлення кожного компонента делегується іншому (зовнішньому) модулю, а висока доступність забезпечується за допомогою реплікації там, де це потрібно. Клієнт компонента не відповідає за обробку його збоїв.

**Гнучкі:** Система залишається чутливою під різними навантаженнями. Реактивні Системи здатні реагувати на коливання швидкості вхідних потоків, збільшуючи або зменшуючи кількість виділених на їх обслуговування ресурсів. Для цього архітектура не повинна допускати наявності вузьких централізованих місць або конкуренції за ресурси, що дозволяє сегментувати або реплікувати компоненти, розподіляючи між ними вхідні дані. Реактивні Системи підтримують передбачувальні та Реактивні алгоритми масштабування, дозволяючи робити вимірювання продуктивності в режимі реального часу. Гнучкість досягається застосуванням економічно ефективних апаратних та програмних платформ.

**Засновані на обміні повідомленнями:** Реактивні системи використовують асинхронний обмін повідомленнями, щоб встановити межі між компонентами та забезпечити слабку зв'язаність, ізоляцію та прозорість розміщення. Ці межі також дозволяють перетворювати та передавати інформацію про збій у вигляді повідомлень. Відкритий обмін повідомленнями робить можливими регулювання навантаження, гнучкість та керування потоком, для чого в системі створюються та відстежуються черги повідомлень і у разі потреби використовується зворотний тиск. Прозорість розміщення при взаємодії на основі повідомлень дозволяє застосовувати до механізму обробки помилок ті самі обмеження та семантику як в межах одного комп'ютера, так і в масштабах цілого кластера. Завдяки неблокуючій взаємодії приймаюча сторона споживає ресурси тільки при активній роботі, що дозволяє знизити накладні витрати.

Великі системи складаються з підсистем, що мають такіж самі властивості і, отже, залежать від їх реактивних характеристик. Це означає, що принципи Реактивних Систем застосовуються на всіх рівнях, що дозволяє компонувати їх між собою. Архітектура, заснована на цих принципах, реалізована в наймасштабніших додатках у світі, які щодня обслуговують мільярди користувачів.

## Зміст

<b>Частина 1: Створення Angular-додатків “HelloApp” і «Shopping list»</b>	<b>2</b>
1.1. Опис основних структурних блоків Angular-додатку «HelloApp»: модулі, компоненти, шаблони.....	2
1.2. Опис основних структурних блоків Angular-додатку «Shopping list»: модулі, компоненти, шаблони.....	5
1.3. Опис файлу package.json. Призначення, основні параметри.....	9
1.4. Опис файлу tsconfig.json. Призначення, основні параметри.....	9
1.5. Опис файлу angular.json. Призначення, основні параметри.....	9
<b>Частина 2: Прив'язка даних.....</b>	<b>9</b>
2.1 Інтерполяція в Angular: огляд приклади використання .....	9
2.2. Прив'язка властивостей елементів HTML: огляд приклади використання.....	9
2.3. Прив'язка до атрибуту: огляд приклади використання.....	11
2.4. Прив'язка до події: огляд приклади використання.....	12
2.5. Двостороння прив'язка: огляд приклади використання.....	13
2.6. Прив'язка до класів CSS: огляд приклади використання.....	

## Основна частина з нової сторінки

### Список використаних джерел

1. Lamis Chebbi "Reactive Patterns with RxJS for Angular", Published by Packt Publishing Ltd, 2022.
2. Sergi Mansilla "Reactive Programming with RxJS". The Pragmatic Programmers, 2015.
3. Дворецький М.Л., Дворецька С.В. Розробка односторінкових вебзастосунків та адаптивних інтерфейсів. Навчальний посібник – Миколаїв, Чорноморський університет імені Петра Могили, 2020 –69 с.
4. Vampakos A., Deeleman P. Learning Angular. Packt Publishing, 2020
5. Chivukula S.R., Iskandar A. Web Development with Angular and Bootstrap, PacktPublishing, 2019.
6. Freeman A. Pro Angular 9: Build Powerful and Dynamic Web Apps, Apress, 2020
7. Doguhan Uluca Angular для Enterprise-Ready Web Applications. -PacktPublishing, 2020
8. Valerio De Sanctis ASP.NET Core 5 and Angular: Full-stack web development with .NET 5 and Angular 11.

9. Oswald Campesato Angular and Machine Learning. - Mercury Learning and Information, 2020
10. Adam Freeman, Pro Angular. Build Powerful and Dynamic Web Apps». Fifth Edition. Apress.
11. Adam Freeman, "Pro Angular". Second Edition. Apress.

### Корисні посилання

- 1) Офіційний сайт Angular -<https://angular.io>
- 2) Сучасний підручник JavaScript. – [Електронний ресурс]. Режим доступу: <https://uk.javascript.info>
- 3) The State of JavaScript. – [Електронний ресурс]. – Режим доступу: <https://2019.stateofjs.com/>
- 4) 2020 Developer Survey-<https://insights.stackoverflow.com/survey/2020>
- 5) Guide of RxJS <https://rxjs.dev/guide/overview>
- 6) TypeScript Documentation <https://www.typescriptlang.org/docs/>
- 7) RxJS guide: <https://rxjs-dev.firebaseapp.com/guide/overview>

## Розгортання Angular-додатку на платформі Firebase

Firebase — це платформа, яка містить великий набір інструментів, що широко використовуються в мобільних і веб-додатках. Це і сервер, і база даних, і хостинг, і аутентифікація в одній платформі. Так, Firebase Realtime Database надає розробникам API, який синхронізує дані додатки між клієнтами і зберігає їх в хмарному сховищі.

- 1) Для розгортання проекту Ви спочатку повинні отримати папку «dist» з готовими файлами для розгортання. Для цього Виконайте команду зі сценаріїв npm: build. В папці проекту Ви повинні отримати папку «dist».
- 2) Перейдіть за посиланням: <https://firebase.google.com/>
- 3) Зареєструйтесь на платформі Firebase
- 4) Зайдіть на <https://firebase.google.com> під своїм аккаунтом
- 5) Перейдіть в консоль (див. рис.)



- 6) Створіть проект з ім'ям ПрізвищеГрупаНомерЛабораторної, наприклад, KovalenkoIP01Lab1.
- 7) На наступному кроці відключить Google Analytics for this project.

×

Create a project (Step 2 of 2)

Google Analytics enables:

×

A/B testing ?

×

Crash-free users ?

×

User segmentation & targeting across Firebase products ?

×

Event-based Cloud Functions triggers ?

×

Free unlimited reporting ?

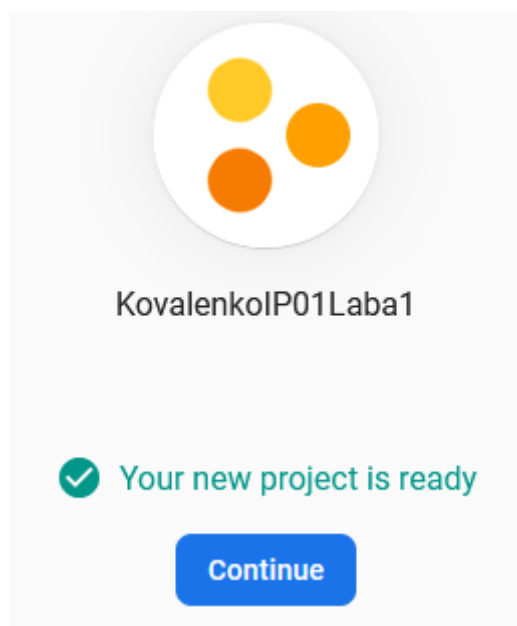
Enable Google Analytics for this project  
Recommended

Previous

Create project

Та активуйте кнопку «Create project».

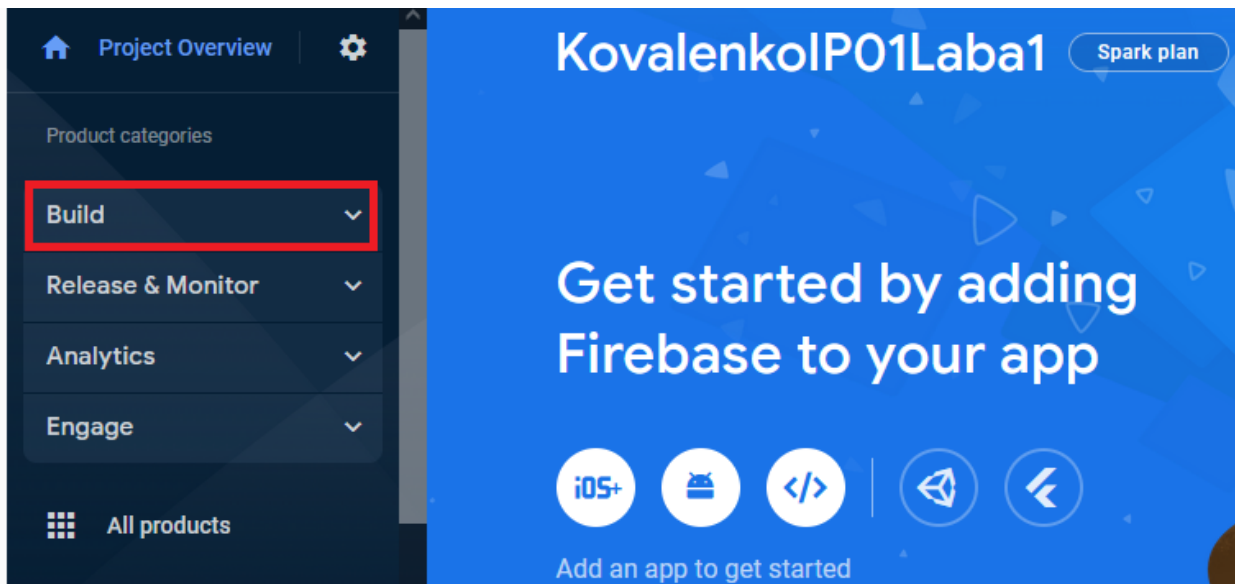
8) Ви отримаєте повідомлення, що Ваш новий проект створено:



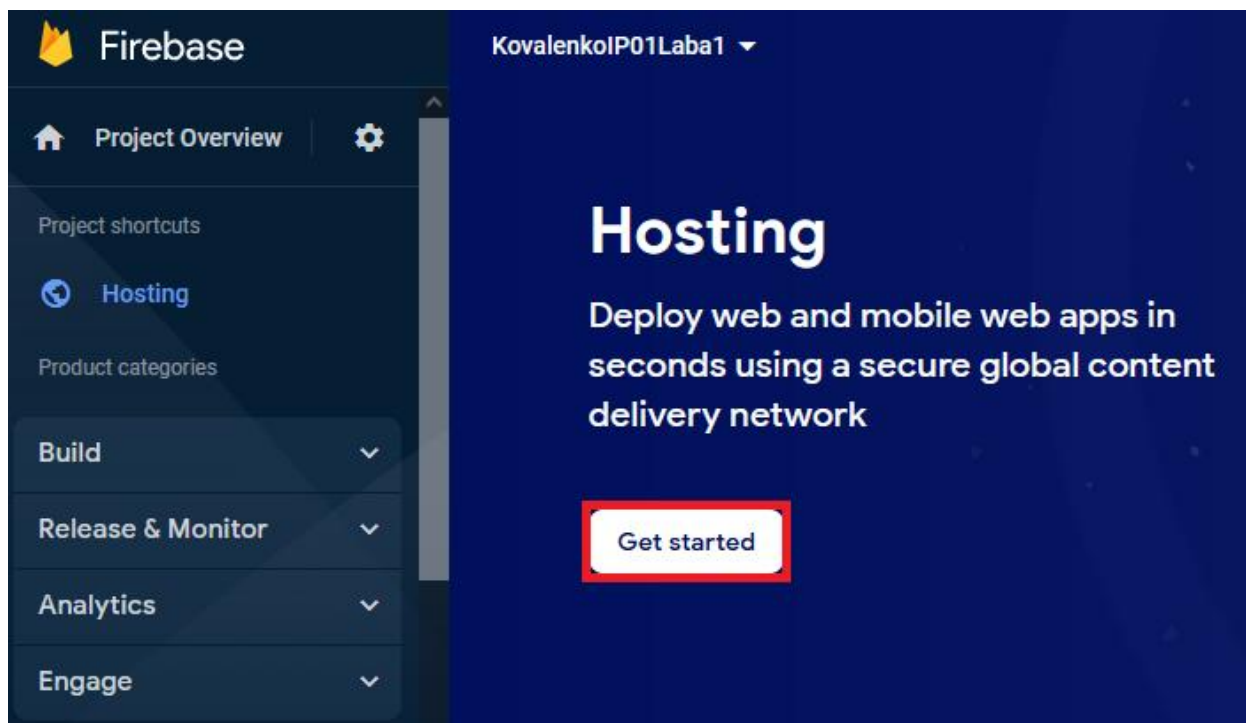
Активуйте кнопку «Continue».

9) На наступному кроці розкрийте меню “Build”





Та оберіть «Hosting». Активізуйте кнопку «Get started»:



1. В папці Вашого Angular-проекту встановіть інтерфейс командного рядка при допомозі команди: **npm install -g firebase-tools**. Ця команда встановить інтерфейс командного рядка для входу та розгортання додатку в Firebase.

```
PS E:\Angular_App\ToDo> npm install -g firebase-tools
...
+ firebase-tools@12.5.2
added 21 packages from 9 contributors, removed 17 packages and updated 56 packages in 414.599s
PS E:\Angular_App\ToDo> 
```

- 10) Для подальшого розгортання Angular-проекту необхідно увійти в firebase при допомозі команди: **firebase login** (бажано, щоб Ви уже увійшли в свій аккаунт на FireBase)

```
PS E:\Angular_App\ToDo> firebase login
Already logged in as . . .
```

- 11) Далі виконайте команду: **firebase init** (див. рис.). Кнопкою «пробіл» оберіть Hosting.

```
PS E:\Angular_App\ToDo> firebase init

##### 
##      ## ##      ## ##      ##      ## ##      ##      ##
##### 
##      ## ##      ## ##      ##      ## ##      ##      ##
##      ##### 
##      ## ##      ## ##      ##      ## ##      ##      ##
##      ##### 

You're about to initialize a Firebase project in this directory:

E:\Angular_App\ToDo

? Are you ready to proceed? Yes
? Which Firebase features do you want to set up for this directory? Press Space to select features,
then Enter to confirm your choices. (Press <space> to select, <a> to toggle all, <i> to invert
selection, and <enter> to proceed)
provision default instance
( ) Firestore: Configure security rules and indexes files for Firestore
( ) Functions: Configure a Cloud Functions directory and its files
>( ) Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys
( ) Hosting: Set up GitHub Action deploys
( ) Storage: Configure a security rules file for Cloud Storage
( ) Emulators: Set up local emulators for Firebase products
(Move up and down to reveal more choices)
```

- 12) Далі вкажіть в якому проекті на firebase будете розгортати Angular-додаток ToDo. Оберіть існуючий проект kovalenkoIP01Laba1 (див.рис.)

```
? Please select an option: Use an existing project
? Select a default Firebase project for this directory: (Use arrow keys)
> kovalenkoip01laba1 (KovalenkoIP01Laba1)
  sportsstorelab4 (SportsStoreLab4)
  sportstorelab3 (SportStoreLab3)
```

- 13) Далі вкажіть яка папка в Вашому проекті є публічною. В даному випадку це папка dist/ToDo (див. рис.)

```
Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be uploaded with firebase deploy. If you
have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? dist/ToDo
```

- 14) Далі необхідно відповісти на питання: «Необхідно конфігурувати даний проект як Single page application»? Так. Angular-додаток це SPA.

```
? What do you want to use as your public directory? dist/ToDo
? Configure as a single-page app (rewrite all urls to /index.html)? (y/N) y
```

15) Далі відповідаєте на питання наступним чином:

```
? What do you want to use as your public directory? dist/ToDo
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
? Set up automatic builds and deploys with GitHub? No
? File dist/ToDo/index.html already exists. Overwrite? (y/N) n
```

16) Після чого ви отримаєте повідомлення, що ініціалізація Firebase пройшла успішно!

```
i Skipping write of dist/myToDo/index.html
i Writing configuration info to firebase.json...
i Writing project information to .firebaserc...
+ Firebase initialization complete!
```

17) Останній крок при розгортанні: виконайте команду **firebase deploy**. Потрібно буде почекати.

```
PS E:\Angular_App\ToDo> firebase deploy

=== Deploying to 'kovalenkoip01laba1'...

i deploying hosting
i hosting[kovalenkoip01laba1]: beginning deploy...
i hosting[kovalenkoip01laba1]: found 7 files in dist/myToDo
+ hosting[kovalenkoip01laba1]: file upload complete
i hosting[kovalenkoip01laba1]: finalizing version...
+ hosting[kovalenkoip01laba1]: version finalized
i hosting[kovalenkoip01laba1]: releasing new version...
+ hosting[kovalenkoip01laba1]: release complete

+ Deploy complete!

Project Console: https://console.firebase.google.com/project/kovalenkoip01laba1/overview
Hosting URL: https://kovalenkoip01laba1.web.app
PS E:\Angular_App\ToDo>
```

18) Далі, Ваш проект можна буде переглянути за наступними посиланнями:

<https://kovalenkoip01laba1.web.app/>

<https://kovalenkoip01laba1.firebaseio.com/>

19) Для того щоб успішно захистити лабораторні роботи необхідно надати викладачу відповідні посилання.

20) При роботі з node.js може знадобитись можливість встановлювати нові версії node.js або змінювати активну версію node.js. Для цього може бути у пригоді пакетний менеджер NVM (node version manager).

## Пакетний менеджер NVM (<https://github.com/nvm-sh/nvm>)

nvm (Node Version Manager) – це досить простий скрипт, який дозволяє встановлювати, перемикати та видаляти версії Node.js на льоту. Простіше кажучи, nvm дає можливість тримати на одній машині будь-яку кількість версій Node.js. У разі встановлення нової версії для неї створюється окрема директорія, наприклад, 16.0.0 або 14.2.2. При перемиканні версій скрипт замінює шлях до Node.js на PATH.

### Встановлення Nvm:

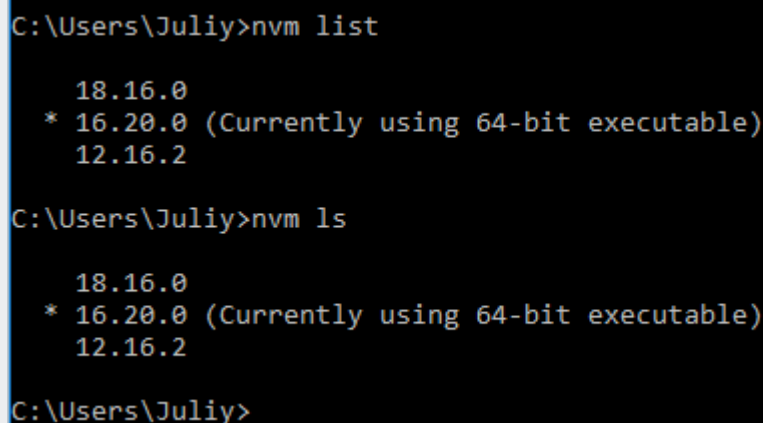
- Під Windows nvm встановлюється з інсталяційного файлу nvm-setup.exe або nvm-setup.zip так само, як і будь яка інша програма (<https://github.com/coreybutler/nvm-windows/releases>).

- Під Linux і macOS nvm можна встановити за допомогою команд curl або wget (<https://github.com/nvm-sh/nvm#installing-and-updating>):

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash
$ wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash
```

### Перемикання версій

Подивитись на список вже встановлених версій Node.js на вашій машині, можна, виконавши в консолі команду nvm list:



```
C:\Users\Juliya>nvm list

 18.16.0
* 16.20.0 (Currently using 64-bit executable)
 12.16.2

C:\Users\Juliya>nvm ls

 18.16.0
* 16.20.0 (Currently using 64-bit executable)
 12.16.2

C:\Users\Juliya>
```

Встановлення останньої версії Node.js у 16-й гілці виконується вказівкою команди install:

```
C:\Users\Juliy>nvm install 16
Downloading node.js version 16.20.1 (64-bit)...
Extracting node and npm...
Complete
npm v8.19.4 installed successfully.

Installation complete. If you want to use this version, type
nvm use 16.20.1
```

Також можна вказати точний номер версії, наприклад 16.20.1.

Тепер подивимося які версії Node.js встановлені на машині:

```
C:\Users\Juliy>nvm ls

 18.16.0
 16.20.1
* 16.20.0 (Currently using 64-bit executable)
 12.16.2
```

Символом «\*» позначається поточна робоча версія node.js.

Якщо ми виконаємо команду `node -v`, ми побачимо також поточну робочу версію node.js:

```
C:\Users\Juliy>node -v
v16.20.0
```

Щоб почати використовувати Node.js версії 18.6.0, потрібно прописати в консолі `nvm use 18.6.0`. Після цього для роботи стане доступна саме ця версія Node.js і версія npm, що поставляється разом з нею:

```
C:\Users\Juliy>nvm use 18.16.0
Now using node v18.16.0 (64-bit)

C:\Users\Juliy>nvm ls

* 18.16.0 (Currently using 64-bit executable)
 16.20.1
 16.20.0
 12.16.2
```