

## Лабораторна робота №3: Pipes. Створення та робота з pipes.

**Мета:** Навчитися створювати та використовувати pipes у Angular.

**Завдання:** Створити чотири Angular-додатки під назвою Pipes1, Pipes2, Pipes3 та Blog.

I) Для Angular-додатку Pipes1 виконати вправу 1 (разом зі самостійним завданням);

II) Для Angular-додатку Pipes2 виконати вправу 2;

III) Для Angular-додатку Pipes3 виконати вправу 3.

IV) Створити проект “Blog”, в який додати два компоненти post та post-form. Компонент post-form – для створення нового поста. Компонент post – для відображення існуючих постів. Створити pipe для фільтрації постів.

V) Зробити звіт по роботі. Звіт повинен включати: титульний лист, зміст, основна частина, список використаних джерел.

VI) Angular-додатки Pipes1 та Blog розгорнути на платформі Firebase у проектах з ім'ям «ПрізвищеГрупаLaba3-1» та «ПрізвищеГрупаLaba3-4», наприклад «KovalenkoIP01Laba3-1» та «KovalenkoIP01Laba3-4».

### I) Angular-додаток Pipes1

Для Angular-додатку Pipes1 виконати вправу 1 (разом із самостійним завданням).

#### Вправа 1: Робота з pipes

Pipes представляють спеціальні інструменти, які дозволяють формувати значення, що відображаються [32, 33, 34, 35]. Наприклад, нам треба вивести певну дату:

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `<div>Без форматування: {{myDate}}</div>
    <div>З форматуванням: {{myDate | date}}</div>`
})
export class AppComponent {
  myDate = new Date(1961, 3, 12);
}
```

Тут створюється дата, яка двічі виводиться у шаблоні (див. рис. 5.1). У другому випадку до дати застосовується форматування за допомогою класу DatePipe.

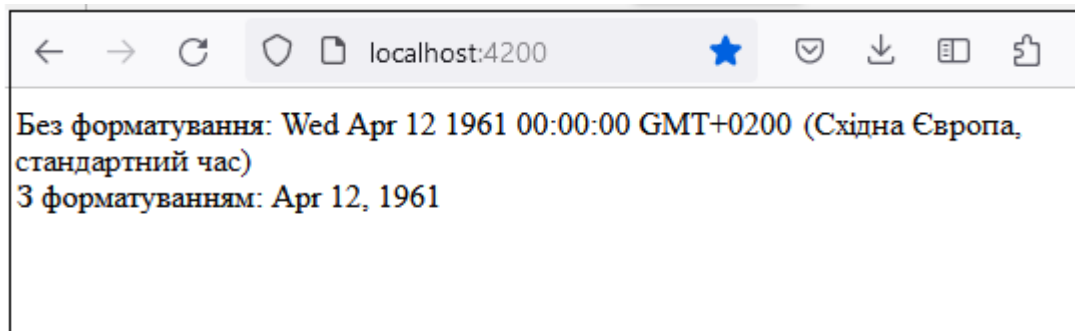


Рис. 5.1.

#### Вбудовані pipes

У Angular є ряд вбудованих pipes [32]. Основні з них:

- CurrencyPipe: форматує валюту
- PercentPipe: форматує відсотки

- UpperCasePipe: переводить рядок у верхній регістр
- LowerCasePipe: переводить рядок у нижній регістр
- DatePipe: форматує дату
- DecimalPipe: задає формат числа
- SlicePipe: обрізає рядок

При застосуванні класів суфікс Pipe відкидається (за винятком DecimalPipe - для застосування використовується назва "number"):

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <div>Без форматування: {{myDate}}</div>
    <div>3 форматуванням: {{myDate | date}}</div>
    <div>{{welcome | uppercase}}</div>
    <div>{{welcome | lowercase}}</div>
    <div>{{persentage | percent}}</div>
    <div>{{persentage | currency}}</div>`
})
export class AppComponent {
  myDate = new Date(1961, 3, 12);
  welcome: string = "Hello World!";
  persentage: number = 0.14;
}
```

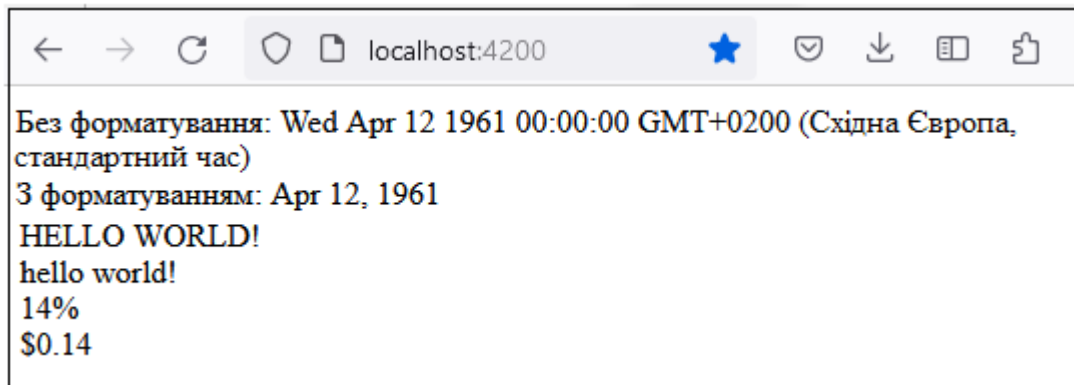


Рис. 5.2.

### Параметри в pipes

Pipes можуть одержувати параметри. Наприклад, пайп SlicePipe, який обрізає рядок, може отримувати як параметр початковий і кінцевий індекси підрядка, який треба вирізати:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <div>Без форматування: {{myDate}}</div>
    <div>3 форматуванням: {{myDate | date}}</div>
    <div>{{welcome | uppercase}}</div>
    <div>{{welcome | lowercase}}</div>
    <div>{{persentage | percent}}</div>
    <div>{{persentage | currency}}</div>`
})
export class AppComponent {
  myDate = new Date(1961, 3, 12);
  welcome: string = "Hello World!";
  persentage: number = 0.14;
}
```

```

    <div>{{welcome | slice:3}}</div>
    <div>{{welcome | slice:6:11}}</div>`
  })
  export class AppComponent {
    myDate = new Date(1961, 3, 12);
    welcome: string = "Hello World!";
    persentage: number = 0.14;
  }

```

Всі параметри в пайп передаються через двокрапку. У даному випадку slice:6:11 вирізає підрядок, починаючи з 6 до 11 індексу. При цьому якщо початок вирізу рядка обов'язково передавати, то кінцевий індекс необов'язковий. В цьому випадку як кінцевий індекс виступає кінець рядка.

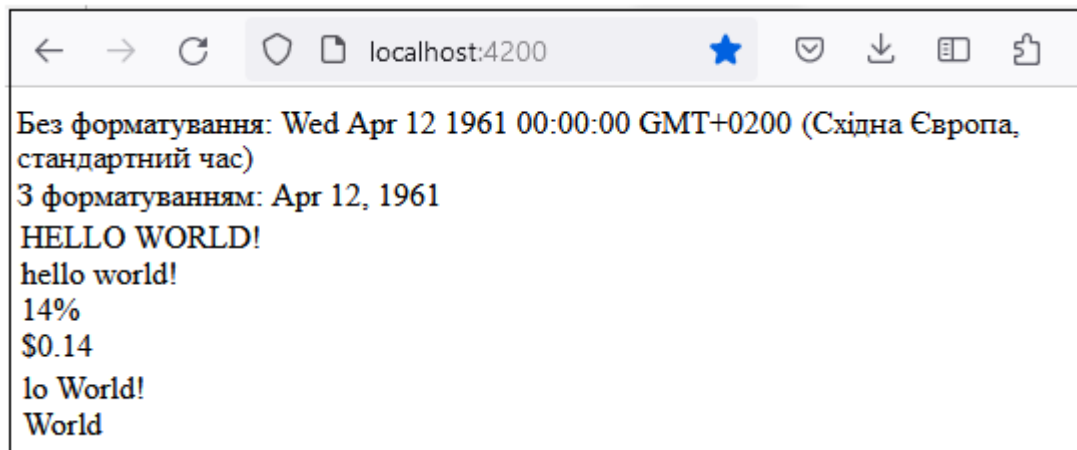


Рис. 5.3.

### Форматування дат

DatePipe як параметр може приймати шаблон дати:

```

import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <div>Без форматування: {{myDate}}</div>
    <div>3 форматуванням: {{myDate | date}}</div>
    <div>{{welcome | uppercase}}</div>
    <div>{{welcome | lowercase}}</div>
    <div>{{persentage | percent}}</div>
    <div>{{persentage | currency}}</div>
    <div>{{welcome | slice:3}}</div>
    <div>{{welcome | slice:6:11}}</div>
    <div>{{myNewDate | date:"dd/MM/yyyy"}}</div>
  `
})
export class AppComponent {
  myDate = new Date(1961, 3, 12);
  welcome: string = "Hello World!";
  persentage: number = 0.14;
  myNewDate = Date.now();
}

```

### Форматування чисел

DecimalPipe як параметр приймає формат числа у вигляді шаблону:

```
{{ value | number [ : digitsInfo [ : locale ] ] }}
```

- value: саме значення, що виводиться
- digitsInfo: рядок у форматі "minIntegerDigits.minFractionDigits-maxFractionDigits", де
  - minIntegerDigits - мінімальна кількість цифр у цілій частині
  - minFractionDigits - мінімальна кількість цифр у дробовій частині
  - maxFractionDigits - максимальна кількість цифр у дробовій частині
- locale: код застосовуваної культури

```
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'my-app',
  template: `
    <div>Без форматування: {{myDate}}</div>
    <div>3 форматуванням: {{myDate | date}}</div>
    <div>{{welcome | uppercase}}</div>
    <div>{{welcome | lowercase}}</div>
    <div>{{percentage | percent}}</div>
    <div>{{percentage | currency}}</div>
    <div>{{welcome | slice:3}}</div>
    <div>{{welcome | slice:6:11}}</div>
    <div>{{myNewDate | date:"dd/MM/yyyy"}}</div>

    <div>{{pi | number:'2.1-2'}}</div>
    <div>{{pi | number:'3.5-5'}}</div>`
  })
export class AppComponent {
  myDate = new Date(1961, 3, 12);
  welcome: string = "Hello World!";
  percentage: number = 0.14;
  myNewDate = Date.now();
  pi: number = 3.1415;
}
```

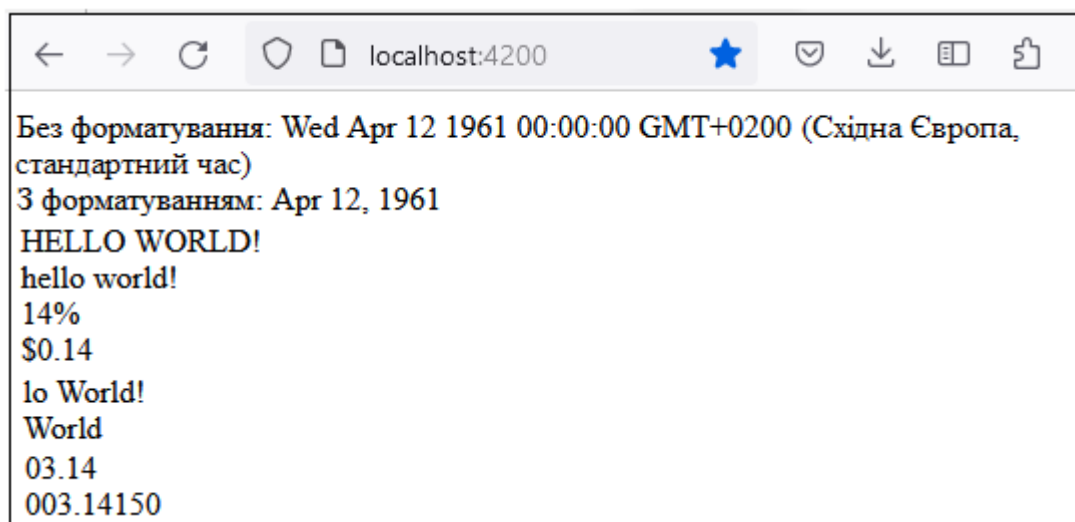


Рис. 5.4.

## Форматування валюти

CurrencyPipe може приймати низку параметрів:

```
{{ value | currency[currencyCode[:display[:digitsInfo[:locale]]]] }}
```

- **value:** сума, що виводиться
- **currencyCode:** код валюти згідно зі специфікацією ISO 4217. Якщо не вказано, то за замовчуванням застосовується USD
- **display:** вказує, як відображати символ валюти. Може приймати такі значення:
  - **code:** відображає код валюти (наприклад, USD)
  - **symbol** (значення за промовчанням): відображає символ валюти (наприклад, \$)
  - **symbol-narrow:** деякі країни використовують як символ валюти кілька символів, наприклад, канадський долар - CA\$, цей параметр дозволяє отримати власне символ валюти - \$
  - **string:** відображає довільний рядок
  - **digitsInfo:** формат числа, який застосовується в DecimalPipe
  - **locale:** код використовуваної локалі

```
import { Component } from '@angular/core';
```

```
@Component({  
  selector: 'my-app',  
  template: `
```

```
    <div>Без форматування: {{myDate}}</div>  
    <div>3 форматування: {{myDate | date}}</div>  
    <div>{{welcome | uppercase}}</div>  
    <div>{{welcome | lowercase}}</div>  
    <div>{{percentage | percent}}</div>  
    <div>{{percentage | currency}}</div>  
    <div>{{welcome | slice:3}}</div>  
    <div>{{welcome | slice:6:11}}</div>  
    <div>{{myNewDate | date:"dd/MM/yyyy"}}</div>  
    <div>{{pi | number:'2.1-2'}}</div>  
    <div>{{pi | number:'3.5-5'}}</div>
```

```
    <div>{{money | currency:'UA':'code'}}</div>  
    <div>{{money | currency:'UA':'symbol-narrow'}}</div>  
    <div>{{money | currency:'UA':'symbol':'1.1-1'}}</div>  
    <div>{{money | currency:'UA':'symbol-narrow':'1.1-1'}}</div>  
    <div>{{money | currency:'UA':'тільки сьогодні по ціні'}}</div>
```

```
  })  
  export class AppComponent {  
    myDate = new Date(1961, 3, 12);  
    welcome: string = "Hello World!";  
    percentage: number = 0.14;  
    myNewDate = Date.now();  
    pi: number = 3.1415;  
  
    money: number = 23.45;  
  }
```

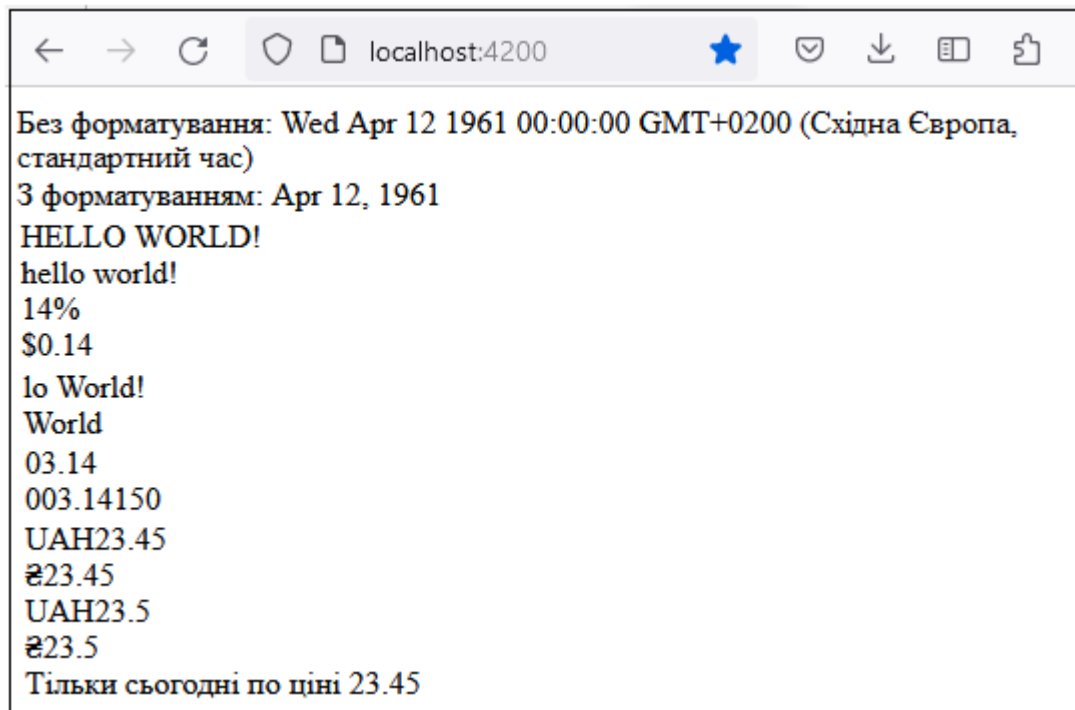


Рис. 5.5.

### Ланцюжки pipes

Цілком можливо, що ми захочемо застосувати відразу кілька pipes до одного значення, тоді ми можемо скласти ланцюжки виразів, розділені вертикальною рисою:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <div>Без форматування: {{myDate}}</div>
    <div>3 форматуванням: {{myDate | date}}</div>
      <div>{{welcome | uppercase}}</div>
      <div>{{welcome | lowercase}}</div>
      <div>{{percentage | percent}}</div>
      <div>{{percentage | currency}}</div>
      <div>{{welcome | slice:3}}</div>
      <div>{{welcome | slice:6:11}}</div>
      <div>{{myNewDate | date:"dd/MM/yyyy"}}</div>
      <div>{{pi | number:'2.1-2'}}</div>
      <div>{{pi | number:'3.5-5'}}</div>
      <div>{{money | currency:'UA':'code'}}</div>
      <div>{{money | currency:' UA':'symbol-narrow'}}</div>
      <div>{{money | currency:' UA':'symbol':'1.1-1'}}</div>
      <div>{{money | currency:' UA':'symbol-narrow':'1.1-1'}}</div>
      <div>{{money | currency:' UA':'тільки сьогодні по ціні'}}</div>
      <div>{{message | slice:6:11 | uppercase}}</div>`
  })
export class AppComponent {
  myDate = new Date(1961, 3, 12);
  welcome: string = "Hello World!";
  percentage: number = 0.14;
  myNewDate = Date.now();
  pi: number = 3.1415;
```

```

    money: number = 23.45;

    message = "Hello World!";
}

```

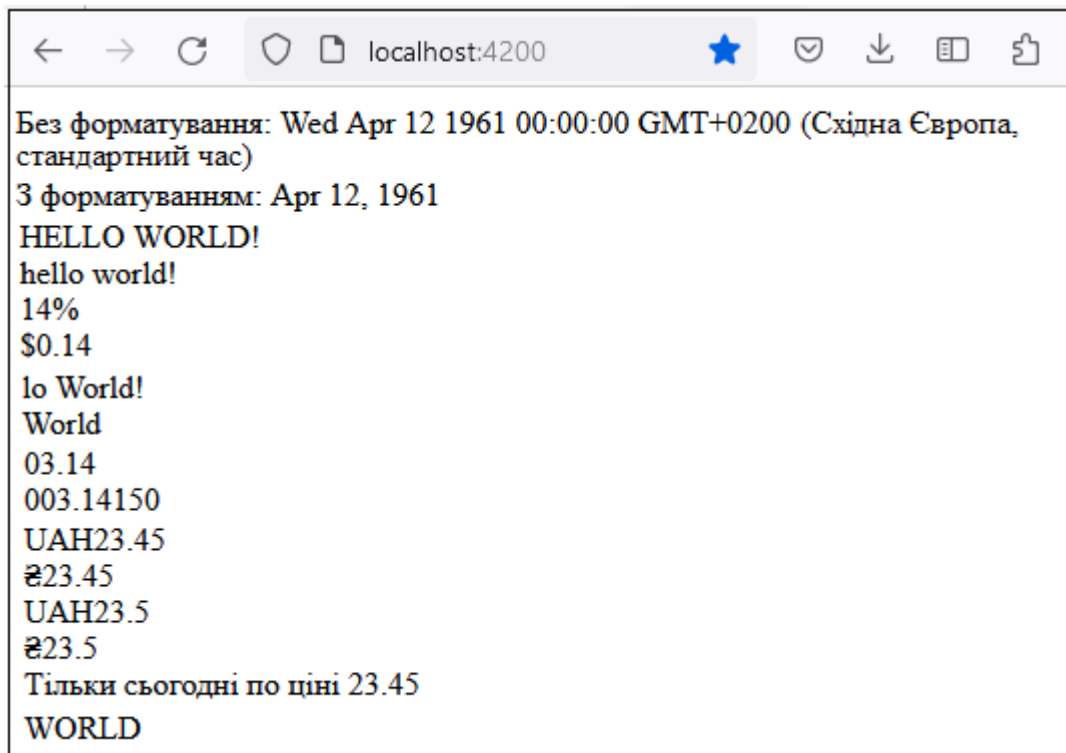


Рис. 5.6.

### Створення своїх pipes

Якщо нам знадобиться деяка передобробка при виведенні даних, додаткове форматування, то ми можемо для цієї мети написати свої власні pipes.

Класи pipes мають реалізувати інтерфейс PipeTransform

```

interface PipeTransform {
  transform(value: any, ...args: any[]): any
}

```

Метод transform має перетворити вхідне значення. Цей метод як параметр приймає значення, до якого застосовується pipe, а також опціональний набір параметрів. А на виході повертається відформатоване значення. Оскільки перший параметр представляє тип any, а другий параметр - масив типу any[], то ми можемо передавати дані будь-яких типів. Також можемо повертати об'єкт будь-якого типу.

Розглянемо найпростіший приклад. Припустимо, нам треба виводити число, в якому роздільником між цілою та дробовою частиною є кома, а не точка. Для цього ми можемо написати маленький pipe. Для цього додамо до проекту до папки src/app новий файл format.pipe.ts (див. рис. 5.7):

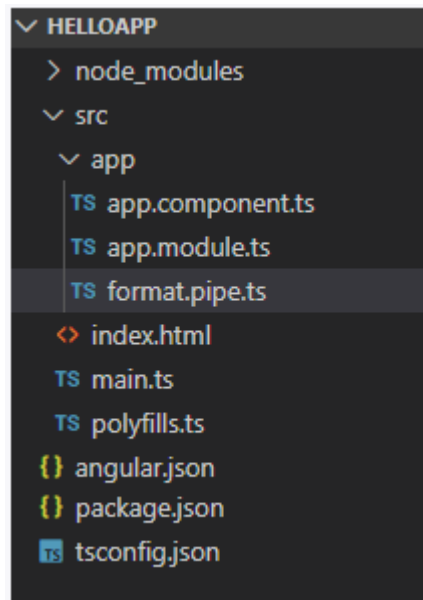


Рис. 5.7.

Визначимо у цьому файлі наступний код:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'format'
})
export class FormatPipe implements PipeTransform {
  transform(value: number, args?: any): string {

    return value.toString().replace(".", ",");
  }
}
```

До кастомного pipe повинен застосовуватися декоратор Pipe. Цей декоратор визначає метадані, зокрема, назву pipe, за якою він використовуватиметься:

```
@Pipe({
  name: 'format'
})
```

Застосуємо FormatPipe у коді компонента:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <div>Без форматування: {{myDate}}</div>
    <div>3 форматуванням: {{myDate | date}}</div>
    <div>{{welcome | uppercase}}</div>
    <div>{{welcome | lowercase}}</div>
    <div>{{persentage | percent}}</div>
    <div>{{persentage | currency}}</div>
    <div>{{welcome | slice:3}}</div>
    <div>{{welcome | slice:6:11}}</div>
```



```

<div>{{pi | number:'2.1-2'}}</div>
<div>{{pi | number:'3.5-5'}}</div>
<div>{{money | currency:'UAH': 'code'}}</div>
<div>{{money | currency:'UAH': 'symbol-narrow'}}</div>
<div>{{money | currency:'UAH': 'symbol': '1.1-1'}}</div>
<div>{{money | currency:'UAH': 'symbol-narrow': '1.1-1'}}</div>
<div>{{money | currency:'UAH': 'Тільки сьогодні по ціні '}}</div>
<div>{{message | slice:6:11 | uppercase}}</div>
<div>Число до форматування: {{x}}<br>Число після форматування: {{x |
format}}</div>

```

```

})
export class AppComponent {
  myDate = new Date(1961, 3, 12);
  welcome: string = "Hello World!";
  percentage: number = 0.14;
  pi: number = 3.1415;
  money: number = 23.45;
  message = "Hello World!";
  x: number = 15.45;
}

```

Але щоб задіяти FormatPipe, його треба додати до головного модуля AppModule:

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { FormatPipe } from './format.pipe';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent, FormatPipe ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }

```

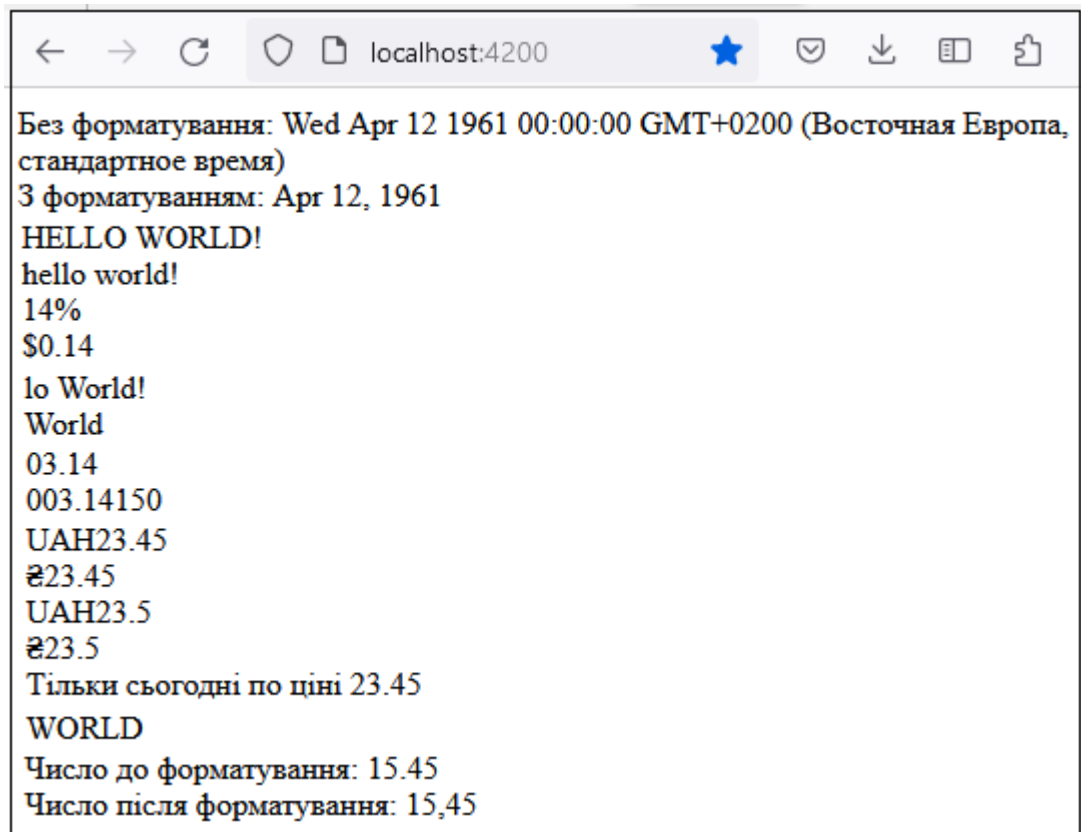


Рис. 5.8.

### Передача параметрів

Додамо ще один `pipe`, який прийматиме параметри. Нехай це буде клас, який з масиву рядків створюватиме рядок, приймаючи початковий та кінцевий індекси для вибірки даних із масиву. Для цього додамо до проекту новий файл `join.pipe.ts`, в якому визначимо наступний вміст:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'join'
})
export class JoinPipe implements PipeTransform {
  transform(array: any, start?: any, end?: any): any {
    let result = array;
    if(start!==undefined){
      if(end!==undefined){
        result = array.slice(start, end);
      }
      else{
        result = array.slice(start, result.length);
      }
    }
    return result.join(", ");
  }
}
```

У метод `transform` класу `JoinPipe` першим параметром передається масив, другий необов'язковий параметр `start` є початковим індексом, з якого проводиться вибірка, а третій параметр `end` - кінцевий індекс.

За допомогою методу slice() отримуємо потрібну частину масиву, а за допомогою методу join() з'єднуємо масив у рядок.

Застосуємо JoinPipe:

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `
    <div>Без форматування: {{myDate}}</div>
    <div>3 форматуванням: {{myDate | date}}</div>
    <div>{{welcome | uppercase}}</div>
    <div>{{welcome | lowercase}}</div>
    <div>{{percentage | percent}}</div>
    <div>{{percentage | currency}}</div>
    <div>{{welcome | slice:3}}</div>
    <div>{{welcome | slice:6:11}}</div>
    <div>{{pi | number:'2.1-2'}}</div>
    <div>{{pi | number:'3.5-5'}}</div>
    <div>{{money | currency:'UAH':'code'}}</div>
    <div>{{money | currency:'UAH':'symbol-narrow'}}</div>
    <div>{{money | currency:'UAH':'symbol':'1.1-1'}}</div>
    <div>{{money | currency:'UAH':'symbol-narrow':'1.1-1'}}</div>
    <div>{{money | currency:'UAH':'Тільки сьогодні по ціні'}}</div>
    <div>{{message | slice:6:11 | uppercase}}</div>
    <div>Число до форматування: {{x}}<br>Число після форматування: {{x |
format}}</div>
  <hr/>

    <div>{{users | join}}</div>
    <div>{{users | join:1}}</div>
    <div>{{users | join:1:3}}</div>`
})
export class AppComponent {
  myDate = new Date(1961, 3, 12);
  welcome: string = "Hello World!";
  percentage: number = 0.14;
  pi: number = 3.1415;
  money: number = 23.45;
  message = "Hello World!";
  x: number = 15.45;
  users = ["Tom", "Alice", "Sam", "Kate", "Bob"];
}
```

Знову ж таки підключимо JoinPipe в модулі програми:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { FormatPipe } from './format.pipe';
import { JoinPipe } from './join.pipe';
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent, FormatPipe, JoinPipe ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Результат роботи:

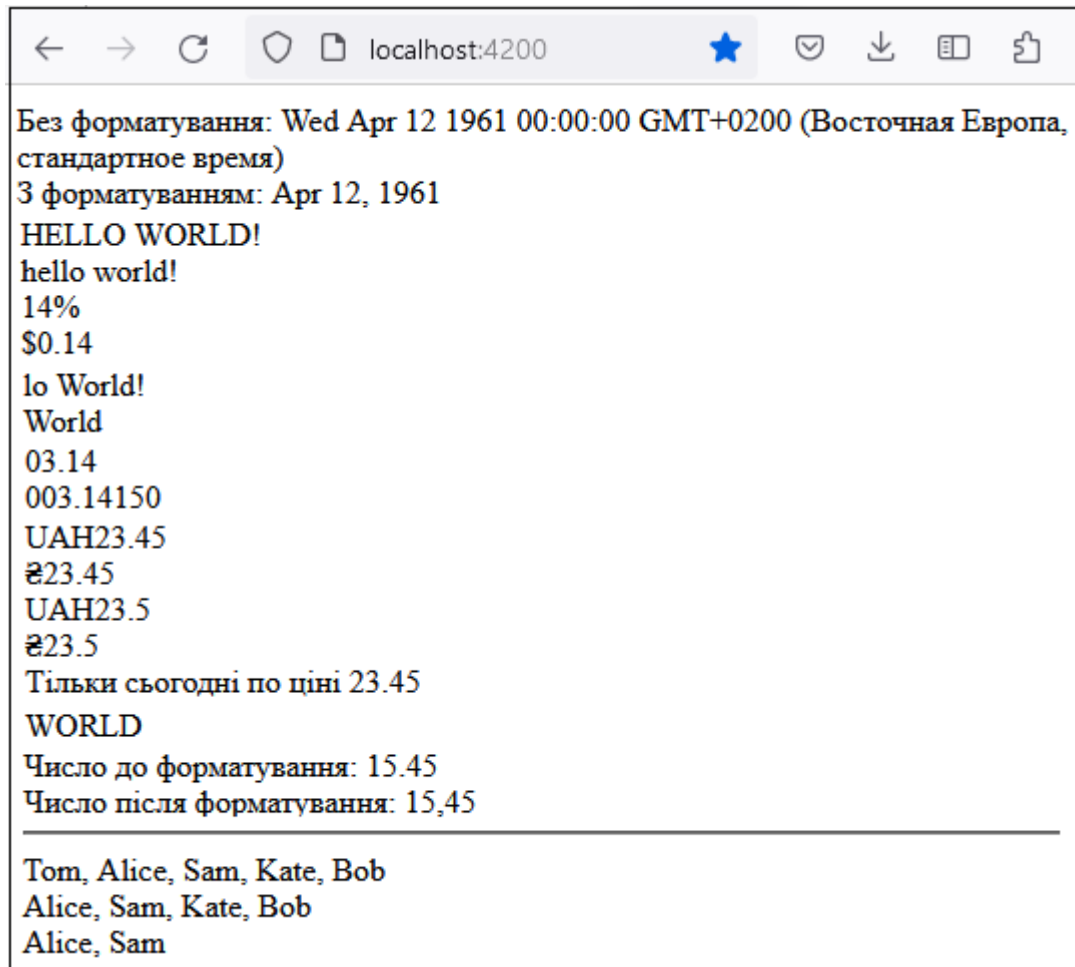


Рис. 5.9.

### Завдання для самостійного виконання

Розробити pipe, який приймає в якості аргументу число і повертає число після отримання квадратного кореня.

## II) Angular-додаток Pipes2

Для Angular-додатку Pipes2 виконати вправу 2.

### Вправа 2: Pure та Impure Pipes

Pipes бувають двох типів: pure (що не допускають змін) та impure (допускають зміни) [36]. Відмінність між цими двома типами полягає у реагуванні на зміну значень, що передаються в pipe.

За замовчуванням усі pipes є типом "pure". Такі об'єкти відстежують зміни у значеннях примітивних типів (String, Number, Boolean, Symbol). У інших об'єктах - типів Date, Array, Function, Object зміни відстежуються, коли змінюється посилання, але не значення за посиланням. Тобто, якщо в масив додали елемент, масив змінився, але посилання змінної, яка представляє даний масив, не змінилася. Тому подібну зміну pure pipes не відстежуватиме.

Impure pipes відстежують усі зміни. Можливо, постає питання, навіщо тоді потрібні pure pipes? Справа в тому, що відстеження змін позначається на продуктивності, тому pure pipes можуть показувати кращу продуктивність. До того ж не завжди необхідно відслідковувати зміни у складних об'єктах, іноді це не потрібно.

Тепер подивимося на прикладі. У вправі 1 було створено клас FormatPipe:

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'format'
})
export class FormatPipe implements PipeTransform {
  transform(value: number, args?: any): string {
    return value.toString().replace(".", ",");
  }
}
```

За замовчуванням це pipe pipe. А це означає, що він може відстежувати зміну значення, яке йому передається, оскільки воно є типом number.

У компоненті ми могли динамічно змінювати значення, для якого виконується форматування:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<input [(ngModel)]="num" name="fact">
    <div>Результат: {{num | format}}</div>`
})
export class AppComponent {
  num: number = 15.45;
}
```

У файлі app.module.ts підключимо FormsModule, щоб використовувати двосторонню прив'язку:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { FormatPipe } from './format.pipe';
import { JoinPipe } from './join.pipe';
@NgModule({
  imports: [ BrowserModule, FormsModule ],
  declarations: [ AppComponent, FormatPipe, JoinPipe ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Тут жодних проблем із введенням би не виникло - змінюємо число в текстовому полі, і відразу змінюється форматований результат (див. рис. 5.10):

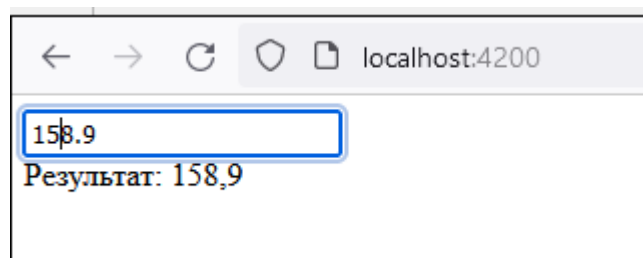


Рис. 5.10.

Але в минулій темі був також створений інший pipe:

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'join'
})
export class JoinPipe implements PipeTransform {
  transform(array: any, start?: any, end?: any): any {
    return array.join(", ");
  }
}
```

Цей pipe виконує операції над масивом. Відповідно, якщо в компоненті динамічно додавати нові елементи в масив, до якого застосовується JoinPipe, то ми не побачимо змін. Так як JoinPipe не відстежуватиме зміни над масивом.

Тепер зробимо його impure pipe. Для цього додамо до декоратора Pipe параметр pure: false:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'join',
  pure: false
})
export class JoinPipe implements PipeTransform {
  transform(array: any, start?: any, end?: any): any {
    return array.join(", ");
  }
}
```

За замовчуванням параметр pure дорівнює true.

Тепер ми можемо додавати в компонент нові елементи в цей масив:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <input [(ngModel)]="num" name="fact">
    <div>Результат: {{num | format}}</div>
    <hr/>
    <input #user name="user" class="form-control">
    <button class="btn" (click)="users.push(user.value)">Add</button>
    <p>{{users | join}}</p>`
})
export class AppComponent {
  num: number = 15.45;
  users = ["Tom", "Alice", "Sam", "Kate", "Bob"];
}
```

І до всіх доданих елементів також застосовуватиметься JoinPipe:

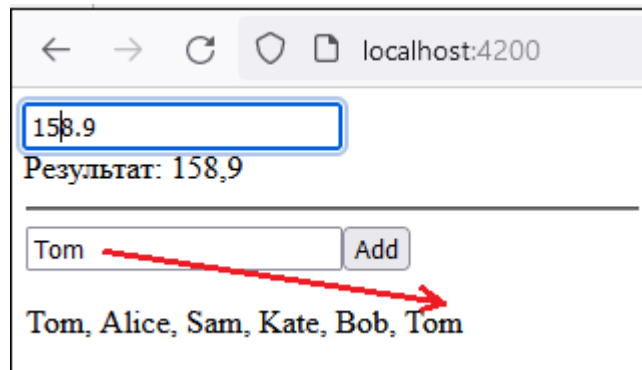


Рис. 5.11.

Коли додається новий елемент, клас JoinPipe знову починає обробляти масив. Тому pipe застосовується до всіх елементів.

### AsyncPipe

Одним із вбудованих класів, який на відміну від інших pipes вже за замовчуванням є тип impure. AsyncPipe дозволяє отримати результат асинхронної операції.

AsyncPipe відстежує об'єкти Observable та Promise та повертає отримане з цих об'єктів значення. При отриманні значення AsyncPipe сигналізує компонент про те, що треба перевірити зміни. Якщо компонент знищується, AsyncPipe автоматично відписується від об'єктів Observable і Promise, що унеможливорює можливі витoki пам'яті.

Використовуємо AsyncPipe:

```
import { Component } from '@angular/core';
import { Observable, interval } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'my-app',
  template: `
    <input [(ngModel)]="num" name="fact">
    <div>Результат: {{num | format}}</div>
    <hr/>
    <input #user name="user" class="form-control">
    <button class="btn" (click)="users.push(user.value)">Add</button>
    <p>{{users | join}}</p>
    <p>Модель: {{ phone| async }}</p>
    <button (click)="showPhones()">Посмотреть модели</button>
  `
})
export class AppComponent {
  num: number = 15.45;
  users = ["Tom", "Alice", "Sam", "Kate", "Bob"];
  phones = ["iPhone 7", "LG G 5", "Honor 9", "Idol S4", "Nexus 6P"];
  phone: Observable<string>|undefined;
  constructor() { this.showPhones(); }
  showPhones() {
    this.phone = interval(500).pipe(map((i:number)=> this.phones[i]));
  }
}
```

Тут з періодичністю 500 мілісекунд у шаблон компонента передається черговий елемент з масиву phones (див. рис. 5.12).

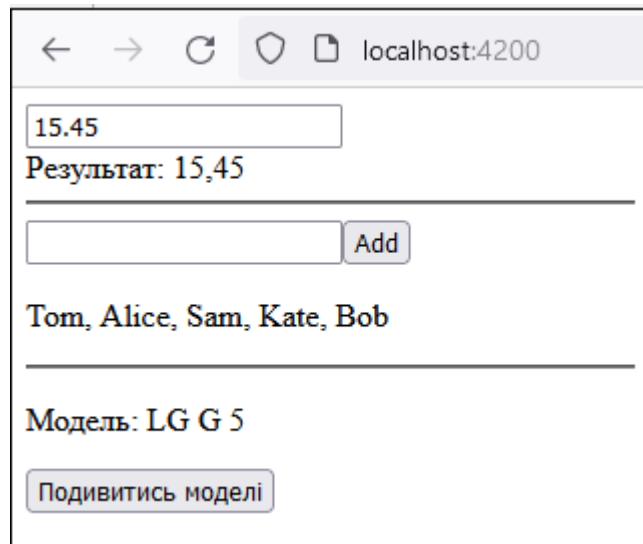


Рис. 5.12.

Компонент не повинен підписуватись на асинхронне отримання даних, обробляти їх, а при знищенні відписуватись від отримання даних. Всю цю роботу робить AsyncPipe.

### III) Angular-додаток Pipes3

Для Angular-додатку Pipes3 виконати вправу 3.

#### Вправа 3: Використання pipes для отримання даних з серверу

Оскільки AsyncPipe дозволяє легко витягувати дані з результату асинхронних операцій, його дуже зручно застосовувати, наприклад, при завантаженні даних з мережі. Наприклад визначимо наступний проєкт (див. рис. 5.13):

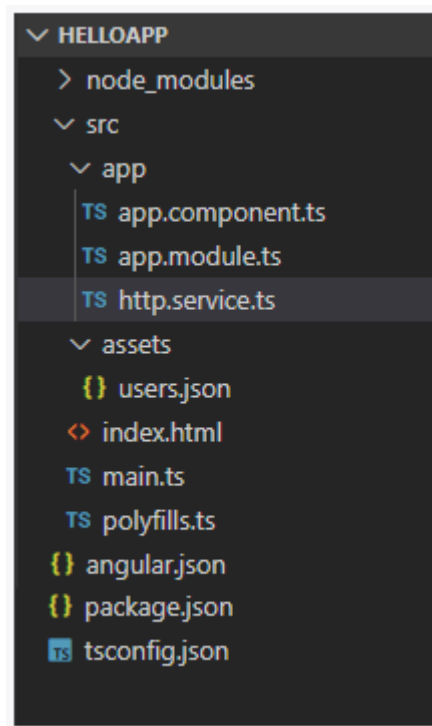


Рис. 5.13.

У файлі http.service.ts визначимо сервіс, який отримує дані із сервера:

```
import {Injectable} from '@angular/core';
```



```
import {HttpClient} from '@angular/common/http';
@Injectable()
export class HttpService{
  constructor(private http: HttpClient){ }
  getUsers(){
    return this.http.get('assets/users.json');
  }
}
```

Для зберігання даних у папці src/assets визначимо файл users.json:

```
[{
  "name": "Bob",
  "age": 28
},{
  "name": "Tom",
  "age": 45
},{
  "name": "Alice",
  "age": 32
}]
```

У файлі app.component.ts використовує сервіс:

```
import { Component, OnInit} from '@angular/core';
import { HttpService} from './http.service';
import {Observable} from 'rxjs';

@Component({
  selector: 'my-app',
  template: `<ul>
    <li *ngFor="let user of users | async">
      <p>Ім'я користувача: {{user.name}}</p>
      <p>Вік користувача: {{user.age}}</p>
    </li>
  </ul>`,
  providers: [HttpService]
})
export class AppComponent implements OnInit {
  users: Observable<Object>|undefined;
  constructor(private httpService: HttpService){}
  ngOnInit(){
    this.users = this.httpService.getUsers();
  }
}
```

Знову ж таки завантаження даних запускається в методі ngOnInit(). У шаблоні компонента до отриманих даних застосовується AsyncPipe:

```
<li *ngFor="let user of users | async">
```

І коли дані будуть отримані, вони відразу будуть відображені на веб-сторінці:

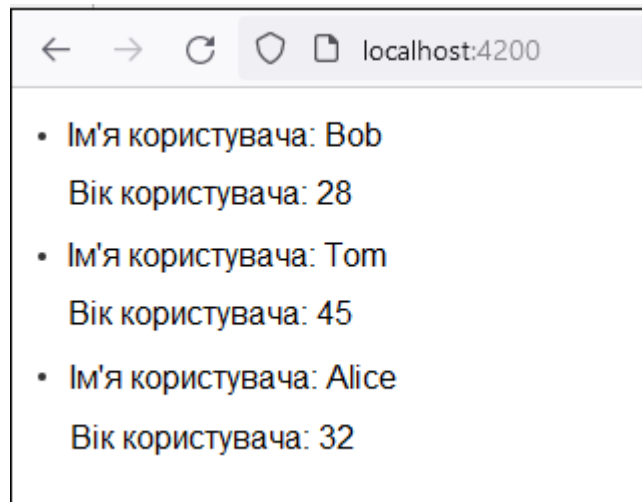


Рис. 5.14.

Щоб завантаження даних з мережі спрацювало, треба додати в AppModule модуль HttpClientModule:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  imports: [ BrowserModule, HttpClientModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

## IV) Створення проекту “Blog”

### Створення проекту та каркасу додатку

- 1) Створіть проект “Blog” при допомозі команди:

```
ng new Blog
```

- 2) В файлі app.component.html створіть наступний вміст:

```
<div class="container">
  <h1>Angular Components</h1>
</div>
```

- 3) В файлі src/styles.css створіть наступний зміст:

```
@import url('https://fonts.googleapis.com/css?family=Roboto');
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}
body {
  font-family: 'Roboto', sans-serif;
  font-size: 1rem;
  line-height: 1.6;
```

```

        background-color: #fff;
        color: #333;
    }
    .container {
        max-width: 1000px;
        margin: 0 auto;
        padding-top: 1rem;
    }
    a {
        text-decoration: none;
    }
    a:hover {
        color: #666;
    }
    ul {
        list-style: none;
    }
    img {
        width: 100%;
    }
    .btn {
        display: inline-block;
        background: #333333;
        color: #fff;
        padding: 0.4rem 1.3rem;
        font-size: 1rem;
        border: none;
        cursor: pointer;
        margin-right: 0.5rem;
        transition: opacity 0.2s ease-in;
        outline: none;
    }
    .btn:hover {
        opacity: 0.8;
    }
    .form-control {
        display: block;
        margin-top: 0.3rem;
    }
    .card {
        padding: 1rem;
        border: #ccc 1px dotted;
        margin: 0.7rem 0;
    }
    input,
    select,
    textarea {
        display: block;
        width: 100%;
        padding: 0.4rem;
        font-size: 1.2rem;
        border: 1px solid #ccc;
        margin: 1.2rem 0;
    }
    hr {
        margin: .5rem 0;
    }

```

- 4) При допомозі вкладки «СЦЕНАРІЙ NPM» в Visual Studio Code запустіть проект на виконання. Ви отримаєте наступний результат (див. рис. 5.15):

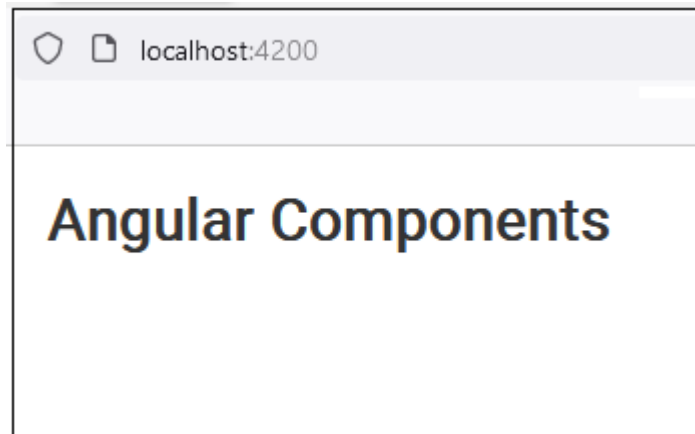


Рис. 5.15.

- 5) При допомозі Angular CLI створіть компонент `post-form` для створення нового поста. Для цього в терміналі в папці проекту введіть наступну команду:

```
Ng g c post-form --skip-tests
```

- 6) При допомозі Angular CLI створіть компонент `post` для відображення існуючих постів. Для цього в терміналі в папці проекту введіть наступну команду:

```
Ng g c post --skip-tests
```

Після цього відповідні класи створених компонентів будуть автоматично додані до модуля `app.module.ts`. Автоматично створені компоненти будуть мати селектори `app-post-form` та `app-post` відповідно.

- 7) В шаблоні компонента `app.component.html` виведіть два компонента у наступному виді:

```
<div class="container">
  <h1>Angular Components</h1>
  <app-post-form></app-post-form>
  <hr/>
  <app-post></app-post>
</div>
```

- 8) В шаблоні, який призначений для створення нового посту блога, створіть два поля для введення даних посту “Title”, “Text” та кнопку `<button>` для додавання нового посту. Наприклад так:

```
<div>
  <input type="text" class="form-control" placeholder="Title...">
  <input type="text" class="form-control" placeholder="Text...">
  <button class="btn">Додати пост</button>
</div>
```

- 9) В шаблоні, який призначений для відображення існуючих постів, виведіть існуючий (статичний) пост, використовуючи із файлу styles.css клас "card". Наприклад, так:

```
<div class="card">
  <h2>Post title</h2>
  <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Dicta, atque.</p>
</div>
```

- 10) В шаблоні app.component.html виведіть дані компонентів post-form та post через їхні селектори app-post-form та app-post відповідно:

```
<div class="container">
  <app-post-form></app-post-form>
  <hr/>
  <app-post></app-post>
  <app-post></app-post>
</div>
```

Додайте у файл Styles.css відступи padding-left, padding-right та color:brown для класу "container". Результат повинен бути наступний (див. рис. 5.16):

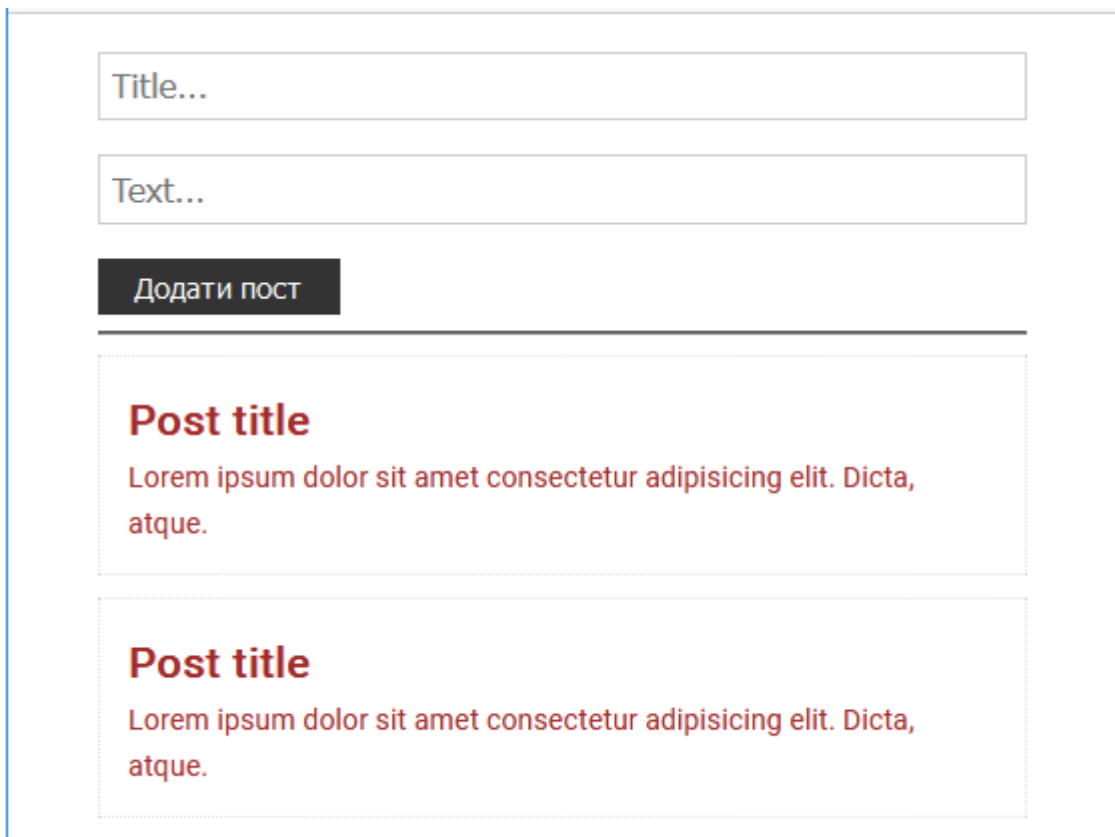


Рис. 5.16. Результат роботи додатку.

### Передача параметрів у компонент. Створення постів.

- 11) В компоненті app.component.ts створимо інтерфейс для визначення типів майбутніх об'єктів проекту і на його основі створимо масив постів.

```
import { Component } from '@angular/core';
export interface Post {
  title:string;
```

```

    text:string;
    id?:number;
  }
  @Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
  })
  export class AppComponent {
    title = 'BlogComponents';
    posts: Post[]=[{title:'Вивчаю компоненти', text:'Створюю проект "Блог"', id:1},
      {title:'Вивчаю директиви', text:'Все ще створюю "Блог"', id:2}]
  }

```

- 12) В шаблоні app.component.html при допомозі структурної директиви ngFor у селекторі компонента, призначеного для виведення постів, виведемо створені статичні пости наступним чином:

```

<div class="container">
<app-post-form></app-post-form>
<hr/>
<app-post
  *ngFor="let p of posts"
  [myPost]="p"
></app-post>
</div>

```

При чому, через змінну myPost в шаблоні app.component.html компонента app.component.ts ми будемо передавати дані в компонент post.component.ts.

- 13) В компоненті, який відповідає за відображення постів post.component.ts введемо нову змінну, при допомозі якої будемо приймати дані. Назвемо її myPost.

```

import { Component, Input, OnInit } from '@angular/core';
import { Post } from './app.component';
@Component({
  selector: 'app-post',
  templateUrl: './post.component.html',
  styleUrls: ['./post.component.css']
})
export class PostComponent implements OnInit {
  @Input() myPost!:Post;
  constructor() { }
  ngOnInit(): void {
  }
}

```

TypeScript видає помилки, якщо не ініціалізувати всі властивості класів під час побудови. Якщо неможливо ініціалізувати дані безпосередньо, але Ви впевнені, що властивість буде призначено під час виконання, можна використовувати оператор затвердження певного присвоєння ! (Оператор затвердження ненульового значення (non-null assertion operator)), щоб попросити TypeScript ігнорувати цю властивість.

Тобто TypeScript надає спеціальний синтаксис для видалення null і undefined із типу без необхідності виконання явної перевірки. Вказівка ! після виразу означає, що цей вираз не може бути нульовим, тобто мати значення null або undefined.

При чому, якщо ми захочемо передавати дані в шаблоні `app.component.html` не через змінну `myPost`, а наприклад через змінну `toPost`, так:

```
<div class="container">
  <app-post-form></app-post-form>
  <hr/>
  <app-post
    *ngFor="let p of posts"
    [toPost]="p">
  </app-post>
</div>
```

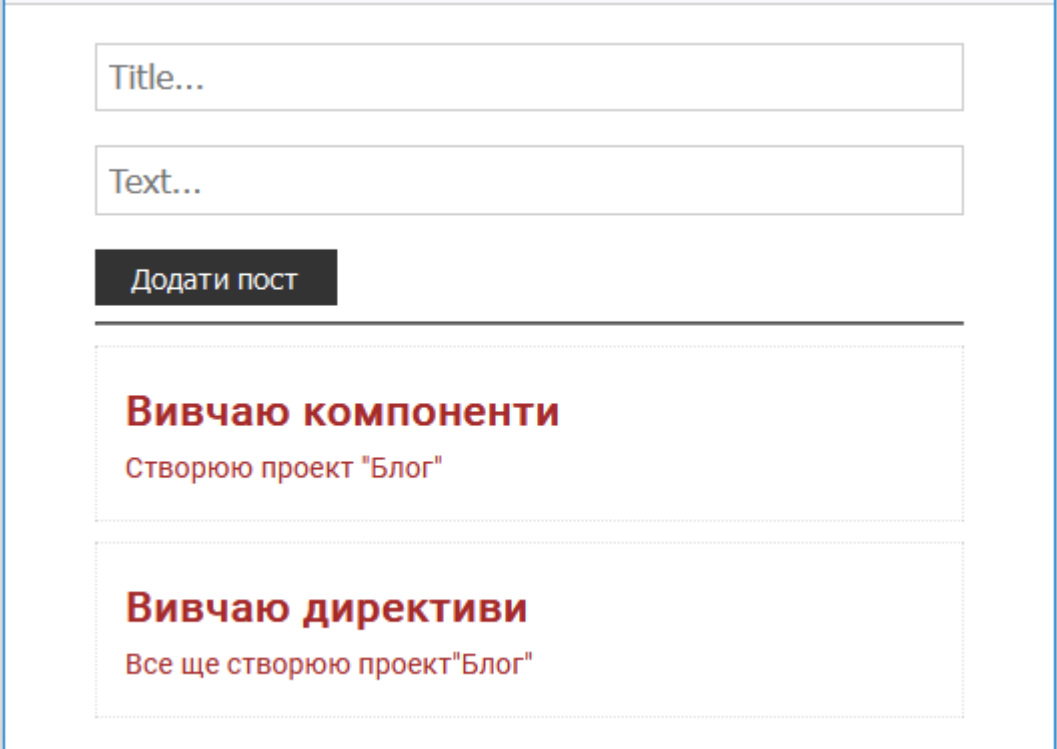
а в компоненті `post.component.ts` ми хочемо залишити змінну `myPost`, то в декораторі ми повинні записати так:

**`Input('toPost') myPost!:Post;`**

- 14) В шаблоні `post.component.html` через змінну `myPost` ми приймаємо дані та виводимо їх на сторінку:

```
<div class="card">
  <h2>{{myPost.title}}</h2>
  <p>{{myPost.text}}</p>
</div>
```

Отримаємо наступний результат (див. рис. 5.17):



The screenshot shows a web application interface. At the top, there is a form with two input fields: 'Title...' and 'Text...'. Below these fields is a dark button labeled 'Додати пост' (Add post). A horizontal line separates the form from the list of posts below. The list contains two items, each in a dashed box. The first item has the title 'Вивчаю компоненти' (Learning components) and the text 'Створюю проект "Блог"' (Creating a project 'Blog'). The second item has the title 'Вивчаю директиви' (Learning directives) and the text 'Все ще створюю проект "Блог"' (Still creating the project 'Blog').

Рис. 5.17. Результат роботи додатку

- 15) При чому, якщо ми захочемо передавати дані в шаблоні `app.component.html` не через змінну `myPost`, а наприклад через змінну `toPost`, так:

```
<div class="container">
```

```

<app-post-form></app-post-form>
<hr/>
<app-post
  *ngFor="let p of posts"
  [toPost]="p"
></app-post>
</div>

```

а в компоненті `post.component.ts` ми хочемо залишити змінну `myPost`, то в декораторі ми повинні записати так:

```
Input('toPost') myPost!:Post;
```

а в шаблоні `post.component.html` ми нічого не змінюємо, і продовжуємо працювати зі змінною `myPost`.

**16)** На наступному кроці реалізуємо додавання нового поста через компонент `post-form` з очисткою полів цієї форми після додавання та відображення нового поста через компонент `post`. Спочатку переконайтесь, що в модулі `app.module.ts` у вас імпортований модуль `FormsModule` для реалізації двосторонньої прив'язки (two-way binding).

```

import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { PostFormComponent } from './post-form/post-form.component';
import { PostComponent } from './post/post.component';

```

```

@NgModule({
  declarations: [
    AppComponent,
    PostFormComponent,
    PostComponent
  ],
  imports: [BrowserModule, AppRoutingModule, FormsModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

**17)** В шаблоні `post-form.component.html` введемо директиву `[(ngModel)]` таким чином:

```

<div>
  <input
    type="text"
    class="form-control"
    placeholder="Title..."
    [(ngModel)]="title">
  <input
    type="text"
    class="form-control"
    placeholder="Text..."
    [(ngModel)]="text">

```



```
<button class="btn">Додати пост</button>
</div>
```

що буде означати, що зміна значення в цьому шаблоні (в полі title та text) призводить до миттєвої зміни значення в компоненті post-form.component.ts цього шаблону і навпаки. Зміна значення в компоненті post-form.component.ts призводить до миттєвої зміни значення в шаблоні post-form.component.html. Тобто, як тільки буде введено будь-яке значення в текстове поле, то воно одразу ж буде передаватися в компонент.

18) Додамо також до кнопки «Додати пост» обробник кліку з назвою addPost:

```
<button class="btn" (click)="addPost()">Додати пост</button>
```

19) В компоненті post-form.component.ts внесемо наступні зміни, щоб можна було оброблювати клік по кнопці «Додати пост»:

```
import { Component, OnInit } from '@angular/core';
import { Post } from '../app.component';
@Component({
  selector: 'app-post-form',
  templateUrl: './post-form.component.html',
  styleUrls: ['./post-form.component.css']
})
export class PostFormComponent implements OnInit {
  title="";
  text="";

  constructor() {}
  ngOnInit(): void {}

  addPost(){
    if (this.title.trim()&&this.text.trim()){
      const post: Post={
        title:this.title,
        text:this.text
      }
      console.log('New post',post);
      this.title=this.text="";           // очищення полів
    }
  }
}
```

Метод trim() тут використовується для видалення пробілів з рядка title та text і якщо в ці змінні були передані не пусті значення, то ми заповнюємо змінну post.

Після запуску проекту, внесення в поля title та text значень «Новий титл» і «Новий текст» та активізації кнопки «Додати пост» в консолі ми отримаємо наступний результат (див. рис. 5.18):

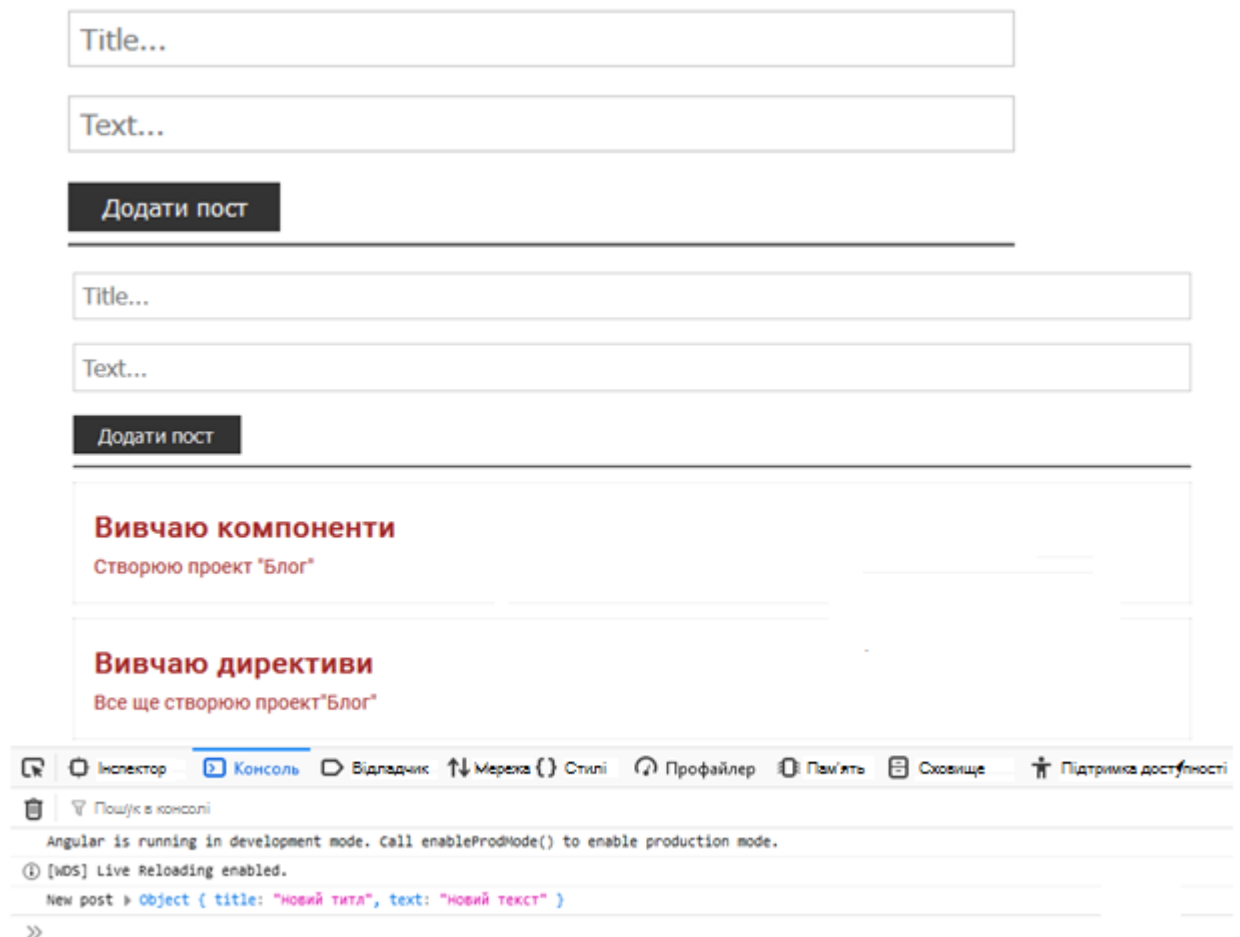


Рис. 5.18.

20) На наступному кроці необхідно передати дані, отримані в змінній `post` дочірнього компонента `post-form.component.ts` в батьківський компонент `app.component.ts` в змінну `posts`.

Якщо нам необхідно з одного компонента щось відправляти назовні, то ми повинні використовувати декоратор `Output`.

Завдання декораторів `Input` та `Output` полягає в обміні даними між компонентами. Вони є механізмом отримання/відправки даних від одного компонента до іншого. `Input` використовується для отримання даних, у той час як `Output` для їх надсилення. `Output` відправляє дані, виставляючи їх в якості виробників подій, зазвичай як об'єкти класу `EventEmitter`.

Тому в дочірньому компоненті `post-form.component.ts` із якого треба забрати дані і відправити в `app.component.ts` введемо нову змінну з декоратором `Output()`.

```
import { Component, EventEmitter, OnInit, Output } from '@angular/core';
import { Post } from '../app.component';

@Component({
  selector: 'app-post-form',
  templateUrl: './post-form.component.html',
  styleUrls: ['./post-form.component.css']
})
export class PostFormComponent implements OnInit {
  @Output() onAdd:EventEmitter<Post> = new EventEmitter<Post>()
  title="";
  text="";
}
```

```

constructor() { }
ngOnInit(): void { }
addPost(){
    if (this.title.trim() && this.text.trim()){
        const post: Post = {
            title: this.title,
            text: this.text
        }
        this.onAdd.emit(post);
        console.log('New post', post);
        this.title = this.text = "";
    }
}
}

```

Метод `eventEmitter.emit()` дозволяє передавати довільний набір аргументів функції слухача. При виклику звичайної функції слухача стандартне ключове слово `this` навмисно встановлюється для посилання на екземпляр `EventEmitter`, до якого прикріплений слухач.

- 21) Тепер в шаблоні `app.component.html` ми повинні прийняти дані в селекторі `<app-post-form>` наступним чином:

```

<app-post-form
  (onAdd)="updatePosts($event)"
></app-post-form>

```

- 22) А в компоненті необхідно створити новий метод для прийняття даних і оновлення масиву `posts`:

```

updatePosts(post: Post){
  this.posts.unshift(post);
}

```

Метод `unshift()` додає один або більше елементів на початок масиву і повертає нову довжину масиву. Індекси всіх елементів, що спочатку присутні в масиві, збільшуються на одиницю (якщо методу було передано лише один аргумент) або на число, що дорівнює кількості переданих аргументів. Метод `unshift()` змінює вихідний масив, а чи не створює його модифіковану копію.

В результаті будемо мати можливість додавати пости, як на рисунку нижче. В консолі розробника, якщо до метода `updatePosts(post: Post)` додати виведення у консоль:

```

console.log('Post', post);

```

можна також побачити дані нових постів (див. рис. 5.19).

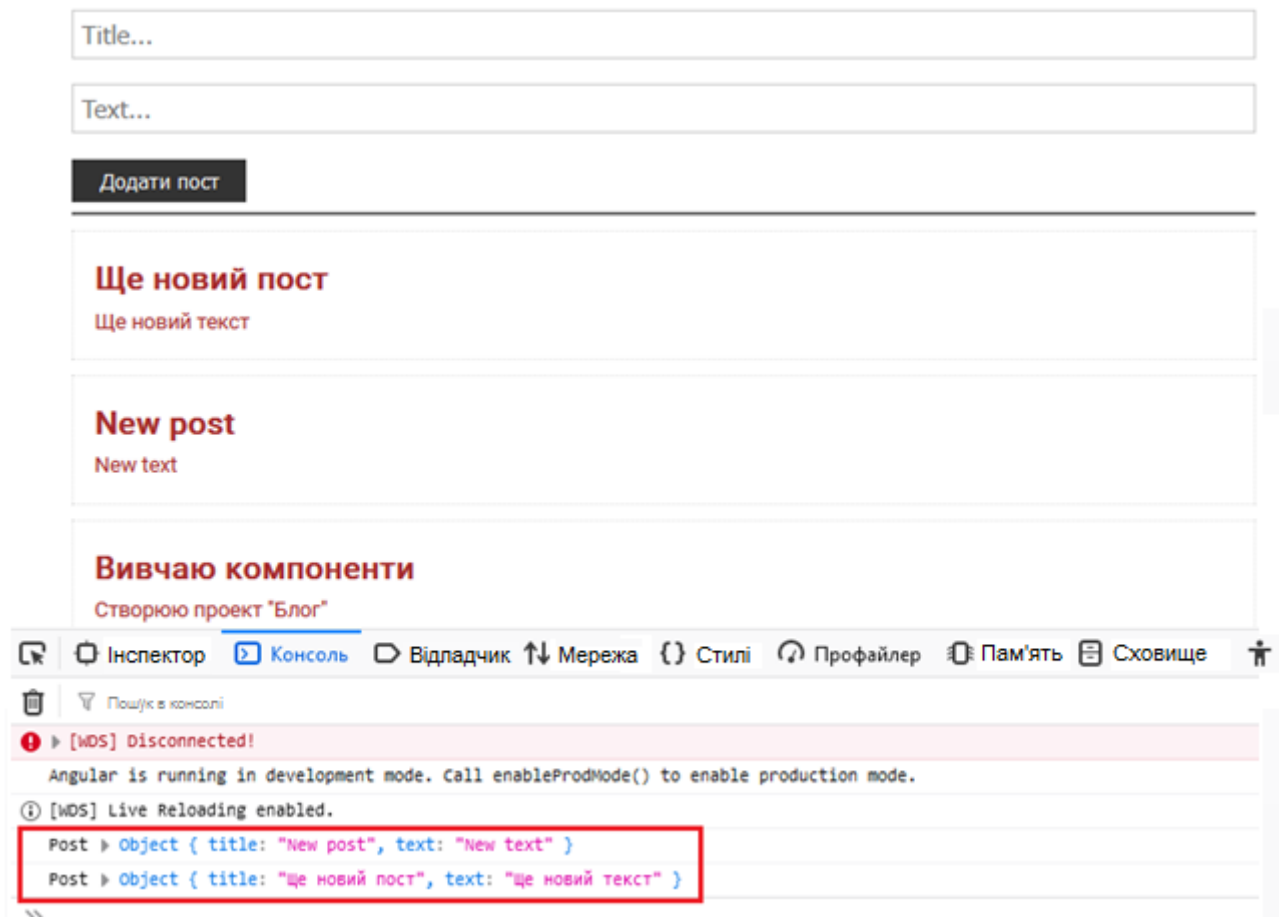


Рис. 5.19. Виведення даних у консоль.

- 23) На наступному кроці реалізуємо логіку перевірки довжини поста, а саме довжину `Post.text`, і якщо довжина менше 20 символів, то виведемо нижче поста «Пост короткий», інакше «Пост довгий».

Для цього в шаблоні `app.component.html` змінюємо вміст тега `<app-post>` на наступне:

```
<app-post
  *ngFor="let p of posts"
  [myPost]="p">
  <small *ngIf="p.text.length>20; else short">Пост довгий</small>
  <ng-template #short>
    <small>Пост короткий</small>
  </ng-template>
</app-post>
```

Angular директива `ng-template` 'відрисовує' Angular шаблон: це означає, що вміст цього тега міститиме частину шаблону, яка потім може бути використана разом з іншими шаблонами для формування остаточного шаблону компонента. Директива `ng-template` використовується спільно з `ngIf`, `ngFor` та `ngSwitch` директивами.

- 24) Далі, щоб отобразити рядок «Пост короткий» та «Пост довгий» необхідно в шаблоні `post.component.html` використати елемент `<ng-content></ng-content>`, який ще називають проекцією контенту, таким чином:

```
<div class="card">
  <h2>{{myPost.title}}</h2>
```

```

    <p>{{myPost.text}}</p>
    <ng-content></ng-content>
  </div>

```

`<ng-content>` дає можливість імпортувати HTML контент ззовні компонента та вставити його в шаблон іншого компонента в певне місце. В результаті будемо мати (див. рис. 5.20):

Рис. 5.20. Результат роботи додатку.

- 25) Останнім кроком у нас буде додавання можливості для видалення будь-якого поста і демонстрація роботи метода `ngOnDestroy()` (<https://angular.io/guide/lifecycle-hooks>). Спочатку додамо в компонент, який відповідає за відображення постів `post.component.ts` метод `ngOnDestroy()` та імплементуємо відповідний інтерфейс `OnDestroy`.

```

import { Component, Input, OnInit, OnDestroy } from '@angular/core';
import { Post } from '../app.component';

```

```

@Component({
  selector: 'app-post',
  templateUrl: './post.component.html',
  styleUrls: ['./post.component.css']
})
export class PostComponent implements OnInit, OnDestroy {

```

```

  @Input() myPost!: Post;
  constructor() { }

```

```

ngOnInit(): void { }
ngOnDestroy(){
  console.log('метод ngOnDestroy');
}
}

```

26) Далі реалізуємо логіку видалення поста. Спочатку створимо кнопку в шаблоні post.component.html для видалення поста:

```

<div class="card">
  <h2>{{myPost.title}}</h2>
  <p>{{myPost.text}}</p>
  <button class="btn" (click)="removePost()">&times;</button>
  <ng-content></ng-content>
</div>

```

Отримаємо (рис. 5.21):

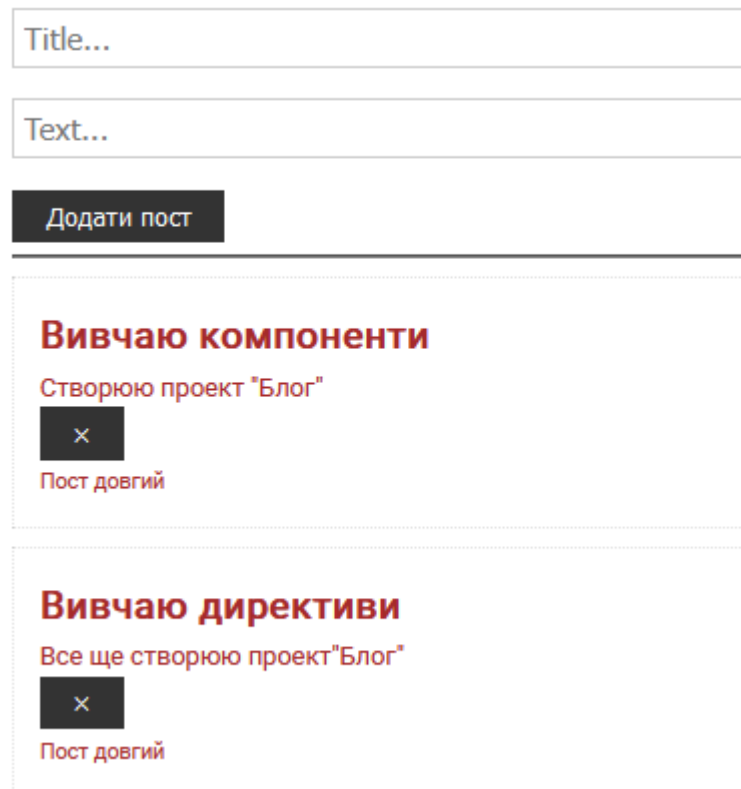


Рис. 5.21.

27) Тепер додаємо логіку до компонента post.component.ts. Для прослуховування події вносимо параметр onRemove через декоратор Output, в який будемо передавати id поста:

```

import { Component, Input, OnInit, OnDestroy, Output, EventEmitter } from
 '@angular/core';
import { Post } from '../app.component';

@Component({
  selector: 'app-post',
  templateUrl: './post.component.html',
  styleUrls: ['./post.component.css']
})

```

```

    })
    export class PostComponent implements OnInit, OnDestroy {

        @Input() myPost!: Post;
        @Output() onRemove=new EventEmitter<number>()
        constructor() {}
        removePost(){
            this.onRemove.emit(this.myPost.id)
        }
        ngOnInit(): void {
        }
        ngOnDestroy(){
            console.log('метод ngOnDestroy');
        }
    }
}

```

28) В app.component.html ми можемо прослухати подію onRemove у поста та видалити пост при допомозі метода deletePost, який нам ще треба створити в цьому компоненті:

```

<app-post
  *ngFor="let p of posts"
  [myPost]="p"
  (onRemove)="deletePost($event)"
>

```

29) В app.component.ts створимо метод deletePost():

```

deletePost(id:number){
    this.posts=this.posts.filter(p=>p.id!==id)
}

```

В результаті, після видалення поста ми побачимо, що визивається метод ngOnDestroy, який часто використовується для того, щоб відписуватись від різних слухачів (див. рис. 5.22).

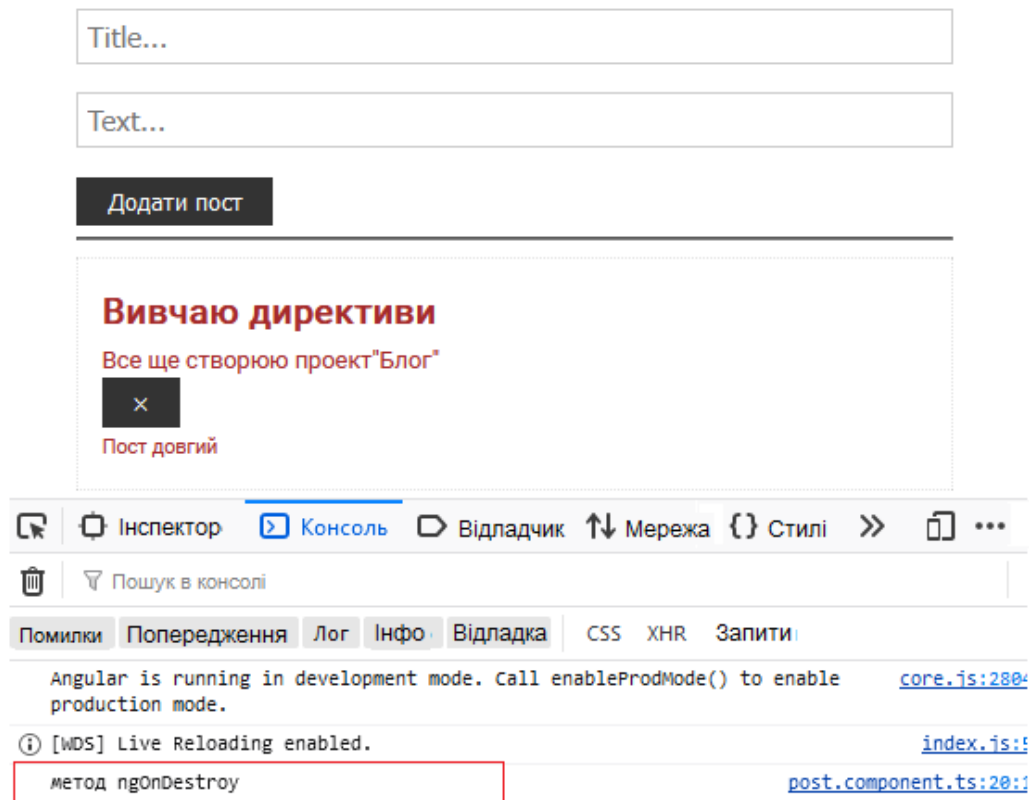


Рис. 5.22. Результат роботи методу ngOnDestroy в консолі

### Завдання для самостійного виконання

Зараз, якщо додати декілька постів при допомозі кнопки «Додати пост», то при видаленні одного з них можна побачити, що будуть видалені всі додані пости. Для видалення конкретного посту, самостійно створіть додаткову логіку в веб-додатку для видалення конкретного посту.

### Створення pipes для фільтрації постів у додатку

30) У додаток Blog додайте поле:

```
<input type="text" [(ngModel)]="search" placeholder="Search...">
```

для фільтрації постів по полю Title. У поле Search користувач вводить дані поста (Title) і в компоненті виводяться тільки ті пости, які включають шуканий рядок без врахування регістра (див. рис. 5.23-5.24).



Title...

Text...

Додати пост

Search...

Знайомлюсь з pipes
Додаю pipes у додаток

×

Пост довгий

Вивчаю компоненти
Створюю проект "Блог"

×

Пост довгий

Вивчаю директиви
Все ще створюю проект"Блог"

×

Пост довгий

Рис. 5.23. Додавання поля Search для пошуку постів.

Title...

Text...

Додати пост

ЗН

Знайомлюсь з pipes
Додаю pipes у додаток

×

Пост довгий

Рис. 5.24.

При чому, якщо в полі Search введено рядок для пошуку і користувач паралельно додає новий пост, title якого включає наявний рядок для пошуку, то новий рядок повинен бути відображений в шаблоні. Наприклад при фільтрації постів по полю title (шукаємо вміст «pipes»), додаємо новий пост з title «Pipes для фільтрації», то новий пост повинен одразу відобразитися в шаблоні (див. рис. 5.25-5.26):

Ріпес для фільтрації

Розроблюємо ріпес для фільтрації постів

Додати пост

pipes

**Вивчаю ріпес**  
Розроблюю ріпес ріпес  
Пост короткий

Рис. 5.25.

Title...

Text...

Додати пост

pipes

**Ріпес для фільтрації**  
Розроблюємо ріпес для фільтрації постів  
Пост довгий

**Вивчаю ріпес**  
Розроблюю ріпес ріпес  
Пост короткий

Рис. 5.26.

Метод transform з двома параметрами можна оформити наступним чином:

```
transform(posts: Post[], search:string=""): Post[] {
    if (!search.trim()){
        return posts;
    }
    return posts.filter(post=>
        {return post.title.toLowerCase().includes(search.toLowerCase())})
}
```

- 31) На сторінці при допомозі ріпес виведемо поточні час та дату. При додаванні нового посту вказувати час та дату його створення (рис. 5.27).

Дата: 17:09:31 28:07:2023

Title...

Text...

Додати пост

---

Search...

Дата: 17:09:18 28:07:2023

Новий пост з часом

Дадаємо час до посту

×

Пост короткий

Дата: 17:08:12 28:07:2023

Новий пост

Тестую новий пост

×

Пост короткий

Дата: 17:03:55 28:07:2023

Вивчаю компоненти

Створюю проект "Блог"

×

Пост довгий

Рис. 5.27.

В шаблоні компоненту `post-form.component.html` можна вивести час наступним чином:

```
<p style="text-align:right">Дата: {{myDate$ | async | date:'HH:mm:ss dd:MM:yyyy'}}</p>
```

Задавши при цьому параметр `myDate$` наступним чином:

```
myDate$:Observable<Date>=new Observable(obs=>
  {setInterval(()==>{
    obs.next(new Date())
  },1000)})
```

В шаблоні компоненту `post.component.ts` можна вивести час наступним чином:

```
<p style="text-align:right">Дата: {{myPost.date | date:'HH:mm:ss dd:MM:yyyy'}}</p>
```

Задавши при цьому параметр `myPost.date` наступним чином:

```
date_post!:Date
ngOnInit():void {
```

```

        this.myDate$.subscribe(date=>{this.date_post=date })
    }
    addPost(){
        if (this.title.trim()&&this.text.trim()){
            const post: Post={
                title:this.title,
                text:this.text,
                date:this.date_post
            }
            this.onAdd.emit(post);
            this.title=this.text="";
        }
    }
}

```

## **V) Завдання для звіту по роботі**

Основна частина звіту повинна мати не менше 15 сторінок тексту з рисунками (шрифт Times New Roman, 14, полуторний інтервал). Титульний аркуш приводиться у додатку 1. Основна частина звіту повинна містити наступні розділи:

- a) pipes: призначення та використання;
- b) ланцюжки pipes;
- c) створення своїх pipes;
- d) передача параметрів у pipes;
- e) pure та Impure Pipes;
- f) asyncPipe.
- g) детальний огляд компоненту post додатку Blog;
- h) детальний огляд компоненту post-form додатку Blog.

## **VI) Розгортання Angular-додатку «Pipes1» та «Blog» на платформі Firebase**

Згідно завдання, необхідно розгорнути Angular-додатки «Pipes1» та «Blog» на платформі Firebase у проектах з назвою «ПрізвищеГрупаLaba3-1» та «ПрізвищеГрупаLaba3-4» відповідно, наприклад, KovalenkoIP01Laba3-1 та KovalenkoIP01Laba3-4. Детальні матеріали з розгортання Angular-додатків на хостингу платформи Firebase наведено у лабораторній роботі №1.

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ  
СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки  
Кафедра інформатики та програмної інженерії

Звіт по лабораторній роботі №\_\_\_\_\_

---

назва лабораторної роботи

з дисципліни: «Реактивне програмування»

Студент: \_\_\_\_\_

Група: \_\_\_\_\_

Дата захисту роботи: \_\_\_\_\_

Викладач: доц. Полупан Юлія Вікторівна \_\_\_\_\_

Захищено з оцінкою: \_\_\_\_\_