Лекція 5. Основні структурні блоки Angular-додатку. Створення дочірнього компонента. Передача даних між компонентами. Життєвий цикл компоненту. Директиви, пайпи.

Директорія app

Вихідні файли папки src/app. Файли, що згенеруються CLI автоматично в нашому додатку myToDo:

1. app.module.ts: визначає файли, які використовує додаток. Цей файл діє як центральний вузол для інших файлів у вашому додатку.

app.module.ts

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

Тут імпортуються всі модулі та компоненти програми.

Декоратор @NgModule() створює кореневий модуль, якому передається об'єкт конфігурації із властивостями:

- imports - використовувані другорядні модулі Angular;
- declarations - всі компоненти програми;
- bootstrap – основний компонент, що відповідає за завантаження.
- providers – потрібен для перерахування залежностей в модулі або в компоненті.

Назва може бути будь-якою, але прийнято використовувати AppModule.

2. app.component.ts: Визначає клас, який містить логіку основної сторінки програми.

app.component.ts

```
@Component({
```

```
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'app';
}
```

За оголошення компонента відповідає декоратор @Component() з @angular/core. Об'єкт, що приймається:

- selector – назва селектору;
- template (templateUrl) - HTML-розмітка у вигляді рядка (шлях до HTML-файлу);
- styles - масив шляхів до CSS-файлів, що містять стилі для створюваного компонента.

3. app.component.html: Містить HTML для AppComponent. Цей файл також називається шаблоном. Шаблон визначає представлення або те, що ви бачите у браузері.

4. app.component.css: Містить стилі для AppComponent. Цей файл використовується, коли вам потрібно стилізувати певний компонент, а не весь додаток.

Як ми бачимо, компонент Angular складається з трьох основних частин: шаблон, стилі та клас. Наприклад, app.component.ts, app.component.html і app.component.css разом складають AppComponent. Ця структура поділяє логіку, представлення та стилі, щоб програма була більш масштабованою та зручною в обслуговуванні.

Angular CLI також генерує файл для тестування компонента app.component.spec.ts.

Щоразу, коли ви генеруєте компонент, CLI створює ці чотири файли в каталозі із вказаним вами ім'ям.

Структура Angular програми

Angular побудований на TypeScript. Так як TypeScript - це надмножина JavaScript, то будь-який правильний JavaScript-код працюватиме в TypeScript. TypeScript пропонує типізацію та більш лаконічний синтаксис, ніж простий JavaScript, який дає вам інструмент для створення зручнішого у супроводі коду та мінімізації помилок.

Компоненти – це будівельні блоки Angular-додатків. Компоненти включають TypeScript-клас, який має декоратор @Component(), HTML-шаблон та стилі.

Сам фреймворк складається з декількох бібліотек (або модулів), кожна з яких містить певний функціонал, а кожен модуль складається з сукупності класів та їх властивостей і методів.

Кожен клас має своє функціональне призначення.

Не всі бібліотеки є обов'язковими для використання в Ангуляр-додатку, частина підключається при необхідності, наприклад, FormsModule або HttpClientModule.

Модуль¶

Кожен модуль має власний набір структурних елементів:

- component- відповідає за частину web-сторінки і включає HTML-шаблон, CSS-стилі та логіку поведінки;
- service- постачальник даних для component;
- directive- перетворює певну частину DOM заданим чином.

Все перераховане вище збирається в кореневий модуль, який загально прийнято називати AppModule.

Кореневий модуль може бути лише один, але він може використовувати функціонал інших модулів, оголошених в об'єкті декоратора @NgModule() як imports.

@NgModule() - це декоратор, який приймає об'єкт, що описує модуль.

Перелік властивостей об'єкта:

- declarations- компоненти (Component), директиви (Directive) та фільтри (Pipe) кореневого модуля;
- exports- компоненти, сервіси, директиви та фільтри, доступні для використання розробникам, які використовуватимуть ваш модуль у своїх розробках;
- imports- інші модулі, що використовуються у кореневому модулі;
- providers- сервіси (Service) програми;
- bootstrap- ім'я головного компонента програми (як правило, називається AppComponent).

Таким чином:

•Модуль Angular є контейнером для групи пов'язаних компонентів, сервісів, директив, пайпів тощо.

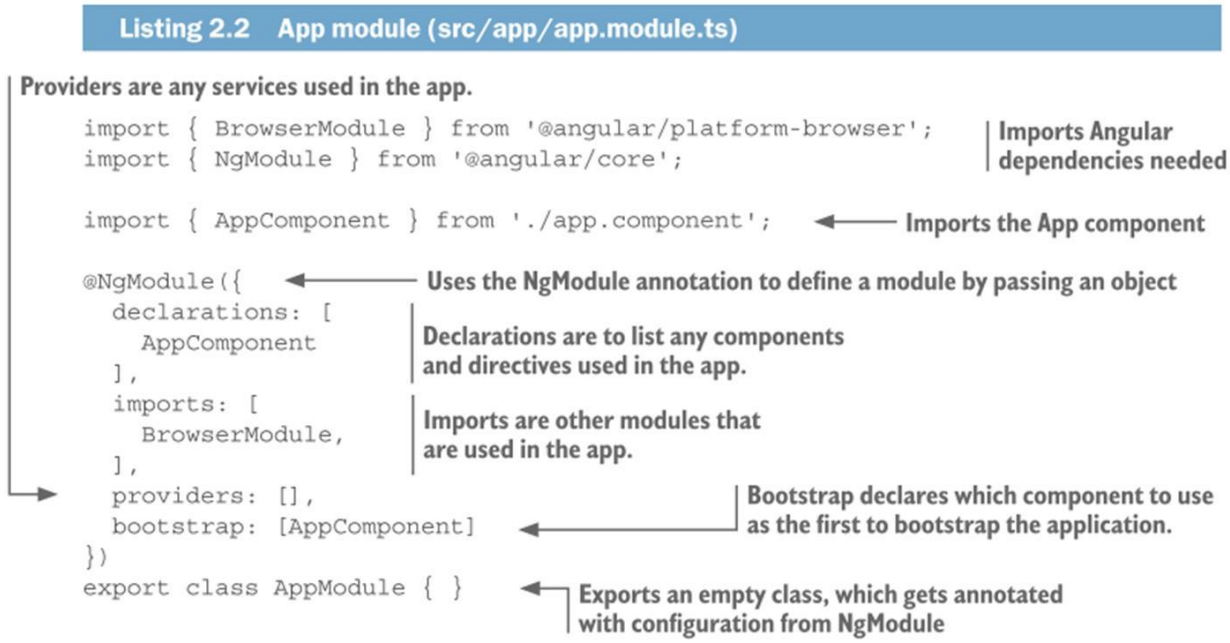
•Модуль можна вважати бібліотекою компонентів та сервісів, що реалізує певну функціональність, наприклад, модуль відправлення товару або модуль формування рахунків.

•Всі елементи невеликого додатку можуть знаходитися в одному модулі (кореневому), а більші додатки мають більше одного модуля.

•Всі додатки повинні мати хоча б один кореневий модуль, який ініціалізується під час запуску програми.

•З точки зору синтаксису модуль – це клас, анотований декоратором NgModule, який може містити інші ресурси.

App module



Компонент¶

Компонент - це частина інтерфейсу додатку з власною логікою. Вся видима частина Angular App реалізується за допомогою компонентів, тому можна почути, що архітектура Angular компонентна.

Раніше згадувалося, що за створення компоненту відповідає декоратор @Component(). Основні властивості об'єкта, який приймає декоратор:

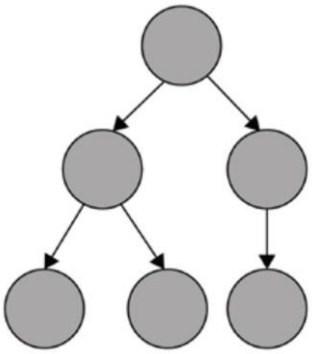
- selector- Назву компонента;
- template (або templateUrl) - HTML-розмітка у вигляді рядка (або шлях до HTML-файлу);
- providers- список сервісів, які постачають дані для компонента;
- styles- масив шляхів до CSS-файлів, що містять стилі для створюваного компонента.

Всі компоненти разом і є Angular App.

Створення дочірнього компонента

•Компоненти є основними будівельними блоками програми Angular. Вони контролюють різні частини веб-сторінки, які називаються представленнями (views), наприклад, список продуктів чи реєстраційна форма.

•Додаток Angular складається з дерева компонентів, які можуть взаємодіяти між собою:



•Однією з найчастіше використовуваних команд Angular CLI є команда generate, яку ми використовуємо для створення певних Angular артефактів.

ng generate <type> <name>

•Щоб створити компонент, перейдіть до кореневої папки проекту Angular CLI та запустіть у командному рядку наступне:

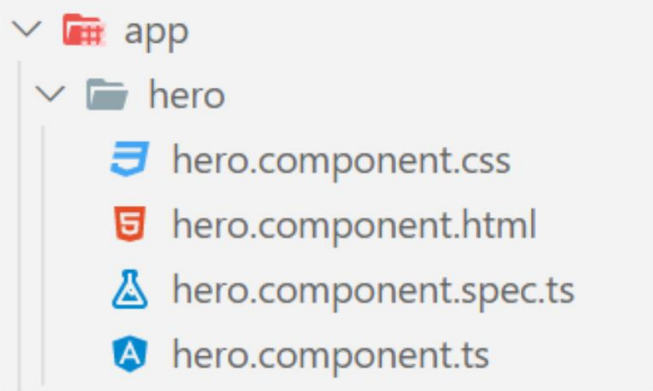
```
ng generate component hero
```

•Якщо ви використовуєте VSCode, розгляньте можливість використання іншого вікна терміналу для запуску команд Angular CLI. Виберіть Terminal | New Terminal з головного меню, щоб відкрити його.

•Створення компонента Angular - це двоетапний процес. Він включає створення необхідних файлів компонента та його реєстрацію за допомогою Angular module.

Файли нового компонента

•Коли ми запускаємо команду ng generate component hero, AngularCLI створює папку hero всередині папки app та генерує такі файли:



Реєстрація в модулі (Файл app.module.ts)

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

Взаємодія з іншими компонентами

•Компоненти Angular мають publicAPI, що дозволяє їм взаємодіяти з іншими компонентами.

•Цей API охоплює input properties, які ми використовуємо для передачі компоненту даних.

•Він також надає output properties, до яких ми можемо прив'язати прослуховувачів подій (event listeners), тим самим одержуючи своєчасну інформацію про зміни стану компонента.

Давайте розглянемо на простих прикладах, як Angular вирішує проблеми введення даних у компонент та отримання даних із компонентів.

Передача даних за допомогою прив'язки вводу

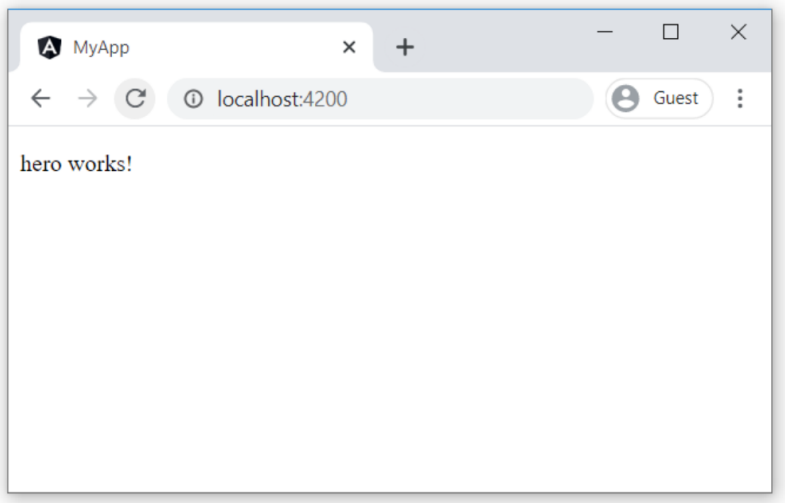
Раніше ми створили новий компонент hero, тепер розглянемо як він працює.

1. Перейдіть до папки app, яка знаходиться всередині папки src.

2. Відкрийте шаблон основного компонента програми, app.component.html.

3. Замініть вміст шаблону на селектор компонента hero, <app-hero></app-hero>.

4. Якщо ми запустимо додаток, ми побачимо на екрані шаблон нашого нового компонента:



• Шаблони, що відображають лише статичну інформацію, рідко зустрічаються у програмі Angular.

•Давайте зробимо наш компонент hero більш інтерактивним, показуючи ім'я фактичного героя, який працює.

5. Ім'я буде динамічно передаватися з AppComponent. Спочатку ми визначаємо властивість у класі компонента hero за допомогою декоратора @Input, а потім імені властивості:

```
@Input() hero!: string;
```

•Декоратор @Input - це спеціалізований декоратор TypeScript, створений командою Angular, який використовується, коли ми хочемо передати дані з компонента в інший компонент.

6. Спочатку нам потрібно імпортувати декоратор з пакету @angular/core, щоб використовувати його:

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

7. Після того, як ми визначили вхідну властивість hero, ми використовуємо прив'язку властивості name до шаблону компонента hero:

```
<p>{{hero}} hero works!</p>
```

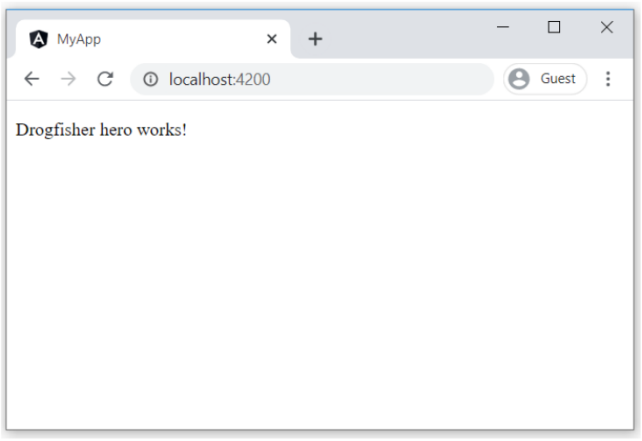
8. Більшу частину роботи ми вже виконали; тепер нам потрібно передати значення властивості hero з AppComponent.

```
<app-hero [hero]="hero"></app-hero>
```

9. Змінна hero, яку ми використовуємо у вхідному прив'язуванні, відповідає властивості в класі AppComponent:

```
export class AppComponent{
  title = 'my-app';
  hero = 'Drogfisher';
}
```

Після збереження змін ми побачимо:



Прослуховування подій за допомогою output binding

•Ми дізналися, що введення прив'язки використовується, коли ми хочемо передавати дані між компонентами.

•Цей метод застосовується у сценаріях, коли ми маємо два компоненти, один виконує роль батьківського компонента, а інший -дочірнього.

•Що робити, якщо ми хочемо спілкуватися навпаки, від дочірнього компонента до батьківського?

•Як ми повідомляємо батьківський компонент про конкретні дії, які відбуваються з дочірнім компонентом?

•Розглянемо сценарій, коли шаблон компонента героя містить елемент кнопки Like, який при натисканні повинен повідомляти батьківський компонент AppComponent про дії користувача.

1. Спочатку ми визначаємо вихідну властивість у класі компонента hero при допомозі декоратора @Output():

```
@Output() liked = new EventEmitter();
```

•Властивість **liked**, - це EventEmitter, позначений декоратором @Output. Декоратор @Output - це спеціалізований декоратор TypeScript, який використовується, коли ми хочемо, наприклад, запускати події від компонента до іншого компонента.

Часто у складних додатках стандартних подій буває недостатньо і розробникам доводиться реалізовувати власні. За цей функціонал відповідає клас EventEmitter.

2. Спочатку нам потрібно імпортувати їх обох із пакета @angular/core, щоб використовувати їх:

```
import { Output, EventEmitter} from '@angular/core';
```

3.Наша кнопка повинна викликати метод emit властивості **liked**, щоб запустити EventEmitter:

```
<button (click)="liked.emit()">Like</button>
```

4. Нам потрібно зробити прив’язку в AppComponent, щоб ці два компоненти могли взаємодіяти між собою. Ми використовуємо прив'язку подій, щоб прив'язати метод onLike від AppComponent до вихідної властивості **liked** компонента **hero**. Цей підхід називається вихідним прив'язуванням:

```
<app-hero [name]="hero" (liked)="onLike()“ ></app-hero>
```

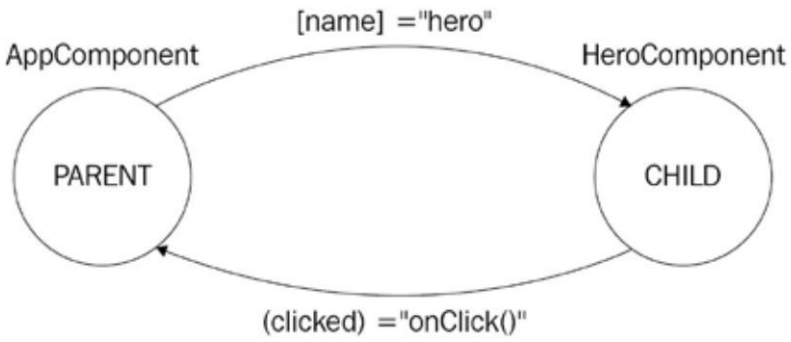
5. Коли користувач натискає кнопку Like у компоненті **hero**, AppComponent викликає метод onLike:

```
onLike() {
```

```
  window.alert(`I like ${this.hero}`);
}
```

•Після збереження змін ми побачимо (при натисканні кнопки Like):

•Тут ви можете побачити огляд механізму зв'язку компонентів, який ми вже обговорювали:



•Клас EventEmitter також може бути використаний для передачі довільних даних через метод emit.

Життєвий цикл компоненту

За час життєвого циклу з компонентом Angular трапляються різні події.

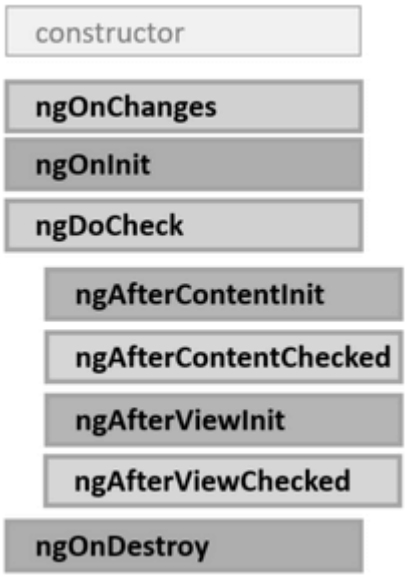
•Після створення елемента механізм визначення змін починає за ним спостерігати.

•Компонент ініціалізується, додається в модель DOM і промальовується, щоб користувач міг побачити його.

•Потім стан елемента (значення його властивостей) може змінитися, що викличе перерисовку інтерфейсу; і нарешті, компонент знищується.

•Події життєвого циклу – це зачепи, прив’язки (hooks), які дозволяють нам підглядати за певними етапами життєвого циклу компонента та застосовувати власну логіку коли це потрібно.

Після створення компонента фреймворк Angular викликає у цього компонента ряд методів, які представляють різні етапи життєвого циклу:



На рисунку показані прив'язки життєвого циклу (функції зворотного виклику), де ви додаєте призначений для користувача код, якщо потрібно.

- **ngOnChanges**: викликається до методу ngOnInit() при початковій установці властивостей, які пов'язані механізмом прив'язки, а також при будь-якій їхній переустановці або зміні їхніх значень. Даний метод як параметр приймає об'єкт класу SimpleChanges, який містить попередні та поточні значення властивості.
- **ngOnInit**: викликається один раз після встановлення властивостей компонента, що беруть участь у прив'язці. Виконує ініціалізацію компонента. Метод ngOnInit() застосовується для комплексної ініціалізації компонента. Тут можна виконувати завантаження даних із сервера або інших джерел даних.

ngOnInit() не аналогічний конструктору. Конструктор також може виконувати деяку ініціалізацію об'єкта, водночас щось складне у конструкторі робити не рекомендується. Конструктор має бути по можливості простим та виконувати саму базову ініціалізацію. Щось складніше, наприклад, завантаження даних із сервера, яке може зайняти тривалий час, краще робити в методі ngOnInit.

- **ngDoCheck**: викликається при кожній перевірці змін властивостей компонента відразу після методів ngOnChanges та ngOnInit.
- **ngAfterContentInit**: викликається один раз після методу ngDoCheck() після вставки вмісту у представлення компонента.
- **ngAfterContentChecked**: викликається фреймворком Angular при перевірці змін вмісту, який додається до представлення компонента. Викликається після методу ngAfterContentInit() та після кожного наступного виклику методу ngDoCheck().

- **ngAfterViewInit**: викликається фреймворком Angular після ініціалізації представлення компонента, а також представлень дочірніх компонентів Викликається лише один раз відразу після першого виклику методу ngAfterContentChecked().
- **ngAfterViewChecked**: викликається фреймворком Angular після перевірки змін у представленні компонента, а також перевірки представлень дочірніх компонентів. Викликається після першого виклику методу ngAfterViewInit() та після кожного наступного виклику ngAfterContentChecked().
- **ngOnDestroy**: викликається перед тим, як фреймворк Angular видалить компонент. Метод ngOnDestroy() викликається перед видаленням компонента. І в цьому методі можна звільняти ті ресурси, які не видаляються автоматично збирачем сміття. Тут також можна видаляти підписку на якісь події елементів DOM, зупиняти таймери тощо.

Кожен такий метод визначено в окремому інтерфейсі, який називається по імені методу без префікса "ng". Наприклад, метод ngOnInit визначено в інтерфейсі OnInit. Тому, якщо ми хочемо відстежувати якісь етапи життєвого циклу компонента, то клас компонента має застосовувати відповідні інтерфейси:

Реалізація всіх методів

Визначимо наступний дочірній компонент:

e:\Angular_App\myToDo\

```
import { Component,
Input,
OnInit,
DoCheck,
OnChanges,
AfterContentInit,
AfterContentChecked,
AfterViewChecked,
AfterViewInit} від '@angular/core';

@Component({
selector: 'child-comp',
template: `<p>Привіт {{hero2}}</p>`
})
export class ChildComponent implements OnInit,
DoCheck,
OnChanges,
AfterContentInit,
AfterContentChecked,
AfterViewChecked,
AfterViewInit {
@Input() hero2: string = "";
count:number = 1;
```

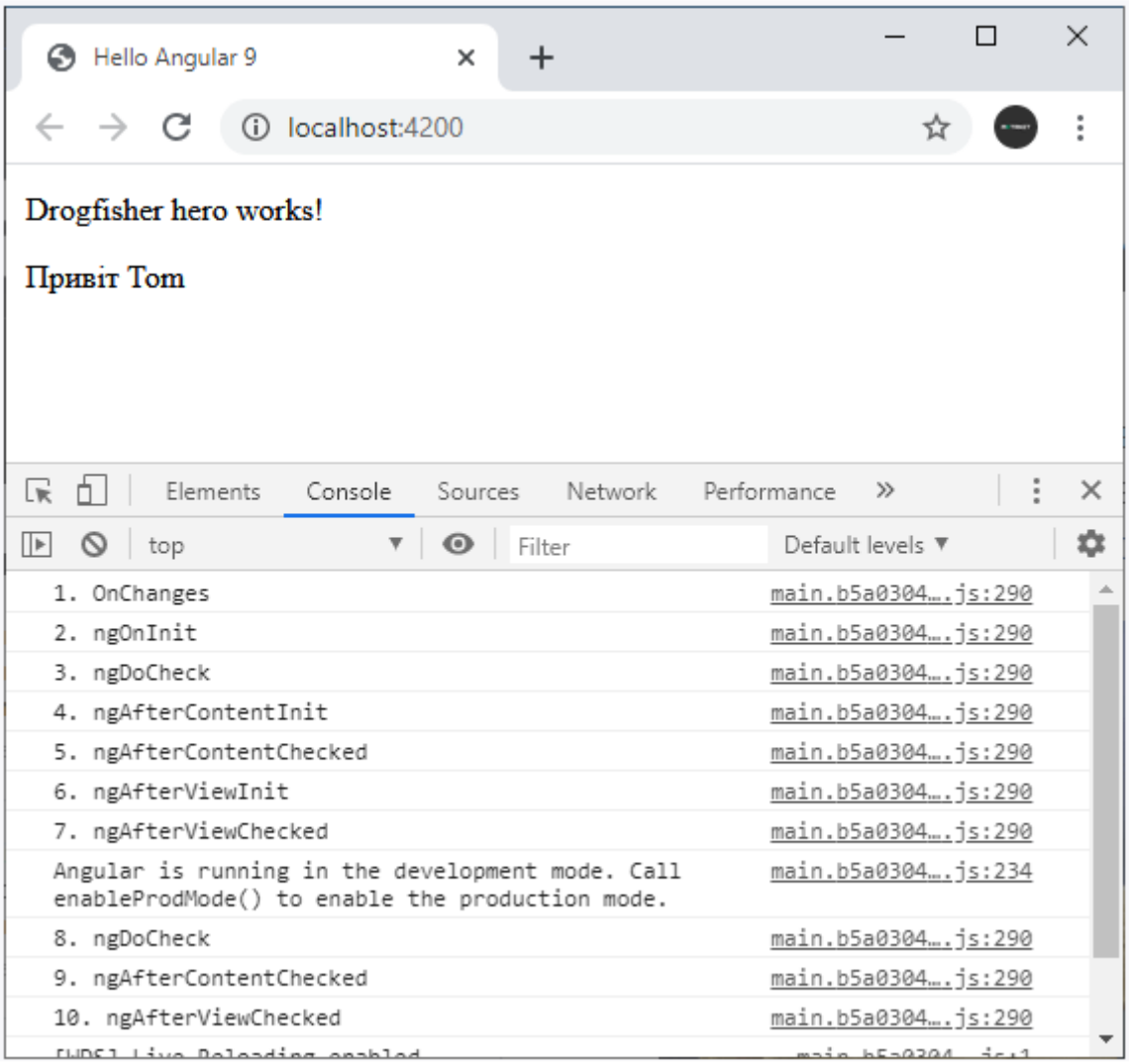
```
ngOnInit() {

this.log(`ngOnInit`);
}
ngOnChanges() {

this.log(`OnChanges`);
}
ngDoCheck() {

this.log(`ngDoCheck`);
}
ngAfterViewInit() {

this.log(`ngAfterViewInit`);
}
ngAfterViewChecked() {

this.log(`ngAfterViewChecked`);
}
ngAfterContentInit() {

this.log(`ngAfterContentInit`);
}
ngAfterContentChecked() {

this.log(`ngAfterContentChecked`);
}

private log(msg: string) {
console.log(this.count + "." + msg);
this.count++;
}
}
```

І використовуємо цей компонент у головному компоненті:

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

@Component({
selector: 'my-app',
template: `
<app-hero [hero]="hero"></app-hero>
<child-comp [hero2]="hero2"></child-comp>
`
})
```

```
export class AppComponent{
title= ‘myToDo’;
hero = ‘Drogfisher’;
hero2 = ‘Tom’;
}
```

І при зверненні до програми ми отримаємо наступний ланцюжок викликів:



Сервіс

Сервіси необхідні для надання даних компонентам. Це можуть бути не тільки запити до сервера, а й функції, що перетворюють вихідні дані згідно заданого алгоритму. Вони дозволяють архітектурі Angular програми бути більш гнучкою та масштабованою.

Завдання сервісу має бути вузьким і строго визначеним.

Не буде вважатися помилкою, якщо ви реалізуєте функціонал у компонентах, але вважається гарною практикою всі звернення до сервера та функції, що повертають дані, виносити до сервісів.

Директива

Призначення директив - перетворення DOM заданим чином, наділення елемента поведінкою.

За своєю реалізацією директиви практично ідентичні компонентам, компонент - це директива з HTML-шаблоном, але з концептуальної точки зору вони різні.

Є два види директив:

- *структурні*- додають, видаляють або замінюють елементи DOM;
- *атрибутивні*- задають елементу іншу поведінку.

Вони створюються за допомогою декоратора @Directive() із конфігураційним об'єктом.

У Angular є безліч вбудованих директив (ngfor, ngIf), але часто їх недостатньо для великих додатків, тому доводиться реалізовувати свої.

Перетворення елементів за допомогою директив

Фреймворк Angular включає в себе набір готових структурних директив, які ми можемо почати використовувати відразу в наших програмах:

•ngIf додає чи видаляє частину дерева DOM на основі виразу.

•ngFor переглядає список елементів і прив'язує кожний елемент до шаблону.

•ngSwitch перемикається між шаблонами в межах певного набору і відображає кожен із них залежно від умови.

Відображення даних умовно

•Директива ngIf додає чи видаляє елемент HTML у DOM з урахуванням оцінки виразу.

•Якщо вираз має значення true, елемент вставляється у DOM. В іншому випадку елемент видаляється з DOM.

•Ми могли б удосконалити наш компонент hero, використовуючи цю директиву:

```
<p *ngIf="hero==='Max'">{{hero}} hero works!</p>
```

•Коли властивість hero класу компонента має значення Max, елемент абзацу відображається на екрані. В іншому випадку він повністю видаляється.

Перебір даних (Iterating through data)

•Директива ngFor дозволяє нам перебирати колекцію елементів і рендерити шаблон для кожного з них.

•Ми можемо розглядати ngFor як цикл for для шаблонів HTML.

•Припустимо, що ми маємо масив об'єктів Heroes, які хочемо відобразити. Ми можемо зробити це за допомогою ngFor:

```
<ul>
  <li *ngFor="let hero of Heroes">
    {{hero.name}}
  </li>
</ul>
```

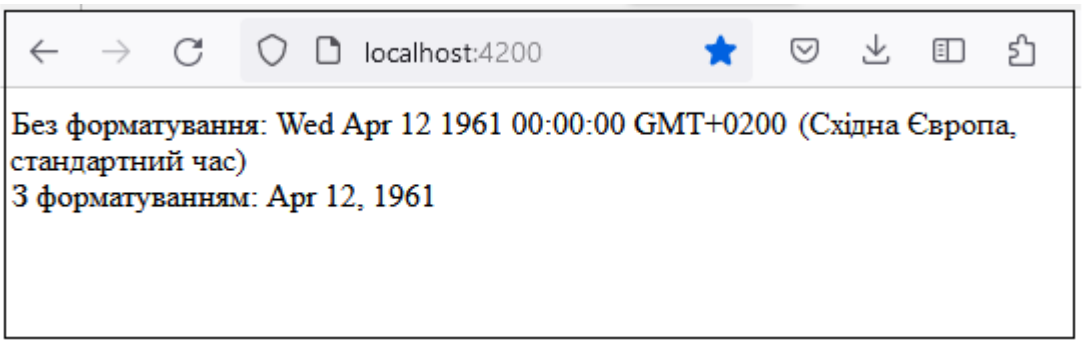
Компонентна модель Angular пропонує надійну інкапсуляцію та інтуїтивно зрозумілу структуру застосування. Компоненти також спрощують модульне тестування вашої програми та можуть покращити загальну читаність коду.

Робота з pipes в Angular

Pipes представляють спеціальні інструменти, які дозволяють форматувати значення, що відображаються. 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. У другому випадку до дати застосовується форматування за допомогою класу DatePipe.



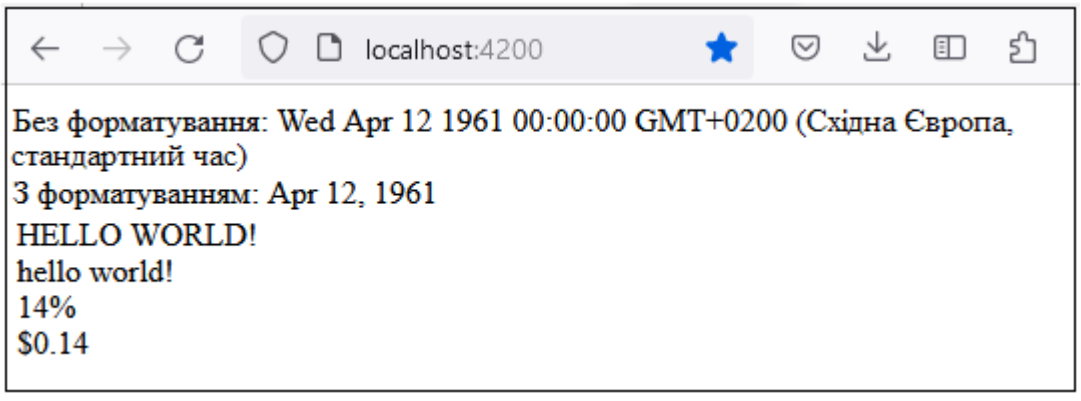
Вбудовані pipes

У Angular є ряд вбудованих pipes. 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>З 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;
}
```

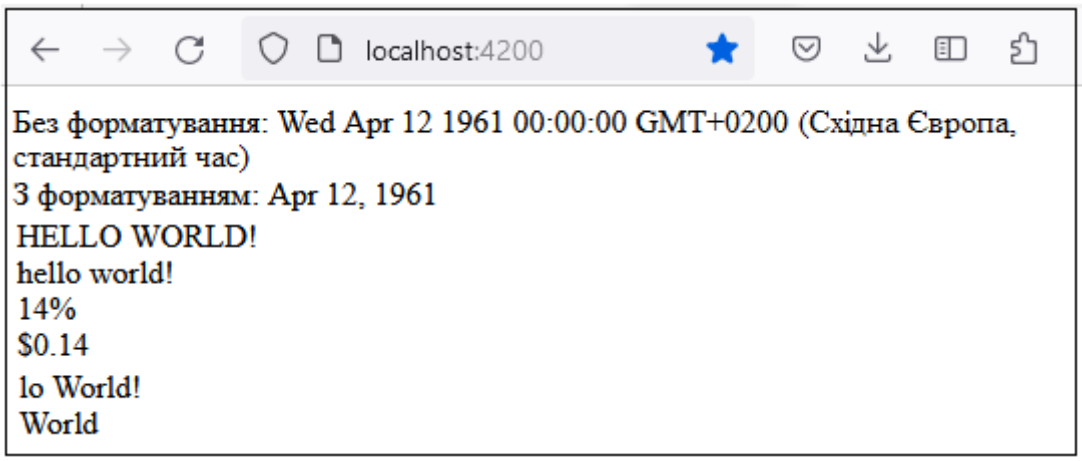


Параметри в pipes

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>З 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>`
})
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.



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

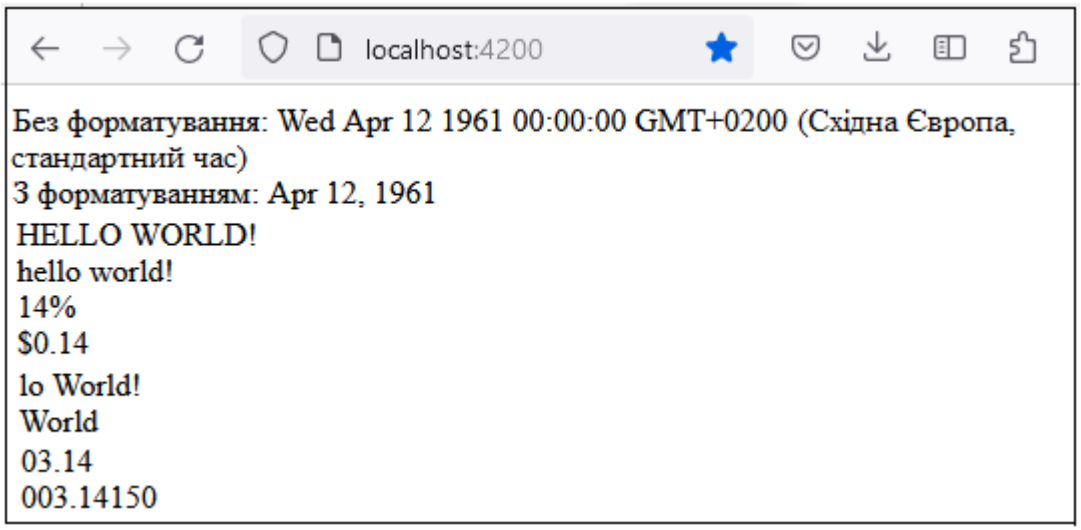
DecimalPipe як параметр приймає формат числа у вигляді шаблону:

```
{{ 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>{{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>
```

```
    <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!";
    persentage: number = 0.14;
    myNewDate = Date.now();
    pi: number = 3.1415;
}
```



Форматування валюти

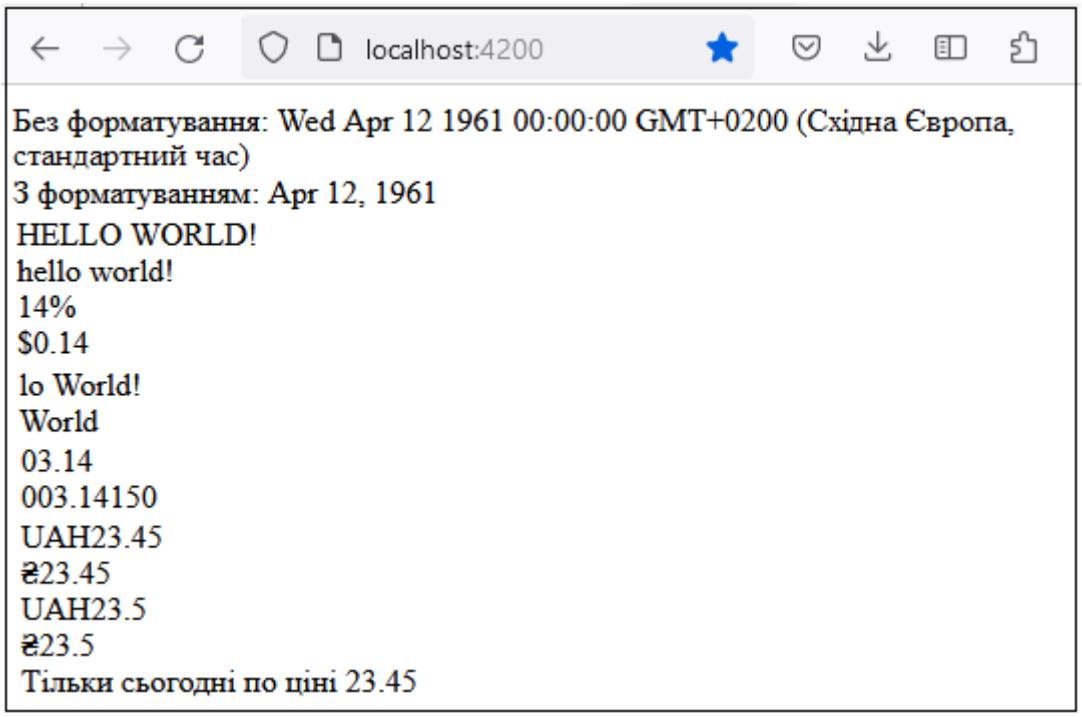
CurrencyPipe може приймати низку параметрів:

```
{{ 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>З 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>
            <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':'тока седня по цене '}}</div>`
})
export class AppComponent {
   myDate = new Date(1961, 3, 12);
   welcome: string = "Hello World!";
   persentage: number = 0.14;
   myNewDate = Date.now();
   pi: number = 3.1415;

   money: number = 23.45;
}
```

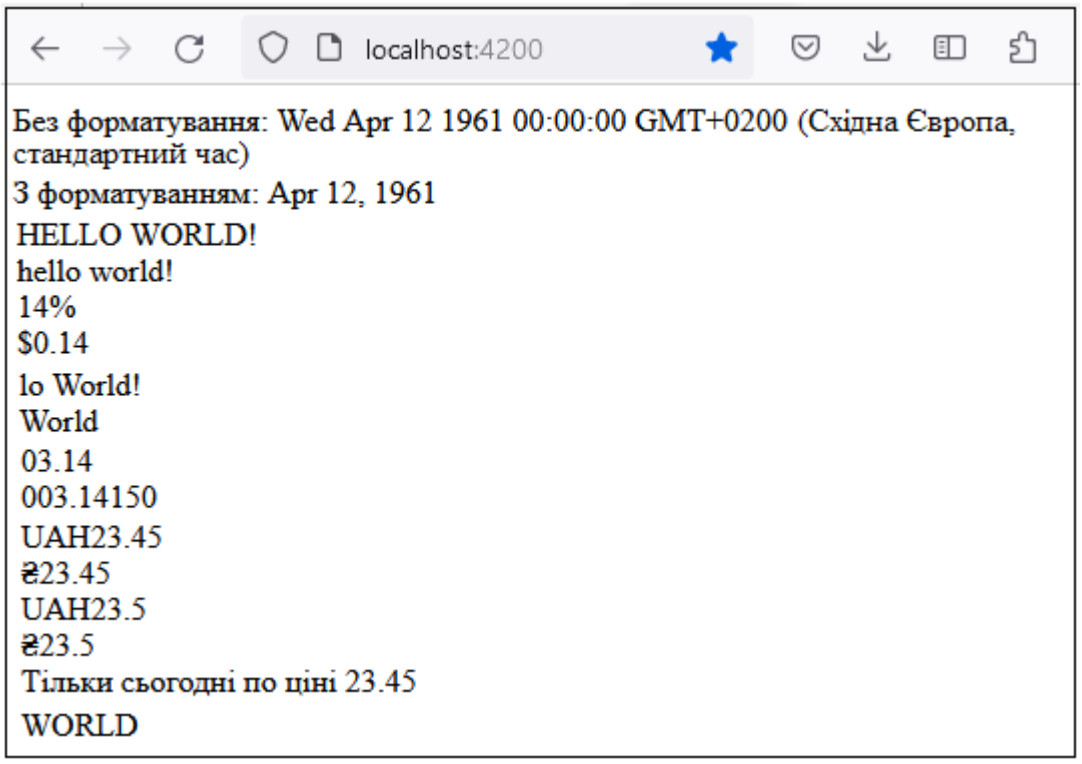


Ланцюжки pipes

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

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

@Component({
   selector: 'my-app',
   template: `
            <div>Без форматування: {{myDate}}</div>
            <div>З 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>
            <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':'тока седня по цене '}}</div>
   <div>{{message | slice:6:11 | uppercase}}</div>`
})
```

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

  message = "Hello World!";
}
```



Пайпи та оператори RxJS

Pipe - це метод класу Observable, доданий в RxJS у версії 5.5. Завдяки йому ми можемо будувати ланцюжки операторів для послідовної обробки значень, отриманих у потоці. Pipe є односпрямованим каналом, який пов'язує між собою оператори. Самі оператори є звичайними функціями, описаними в RxJS, які обробляють значення потоку. Наприклад, вони можуть перетворити значення і передати його далі в потік, або можуть виконувати роль фільтрів і не пропускати будь-які значення, якщо вони не відповідають заданій умові.

Подивимося на оператори у справі. Помножимо кожне значення потоку на 2 за допомогою оператора map:

```
of(1,2,3).pipe(
  map(value => value * 2)
```

```
).subscribe({
  next: console.log
});
```

Ось так виглядатиме потік до використання оператора map:

Після оператора map:

Давайте скористаємося оператором filter. Цей оператор працює так само, як функція filter у класі Array. Першим аргументом метод приймає функцію, в якій описано будь-яку умову. Якщо значення потоку задовольняє умові, воно пропускається далі:

```
of(1, 2, 3).pipe(
  // пропускаємо тільки непарні значення
  filter(value => value % 2 !== 0),
  map(value = value * 2)
).subscribe({
  next: console.log
});
```

Ось так буде виглядати вся схема нашого потока:

Після filter:

Після map:

Примітка: pipe!== Subscribe. Метод pipe декларує поведінку потоку, але не підписується. Допоки ви не викличете метод subscribe, ваш потік не почне працювати.

Створення своїх pipes в Angular

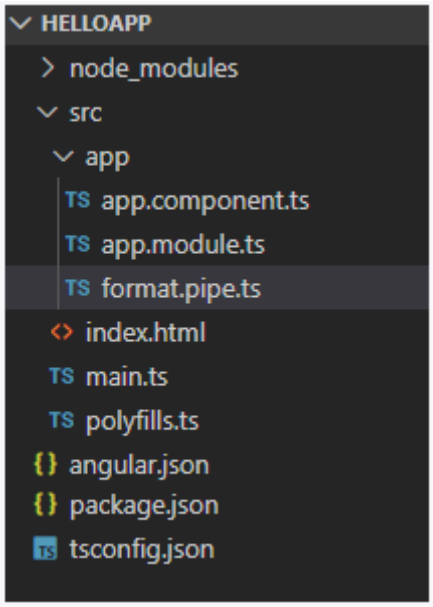
Якщо нам знадобиться деяка передобробка при виведенні даних, додаткове форматування, то ми можемо для цієї мети написати свої власні pipes.

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:



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'
})
```

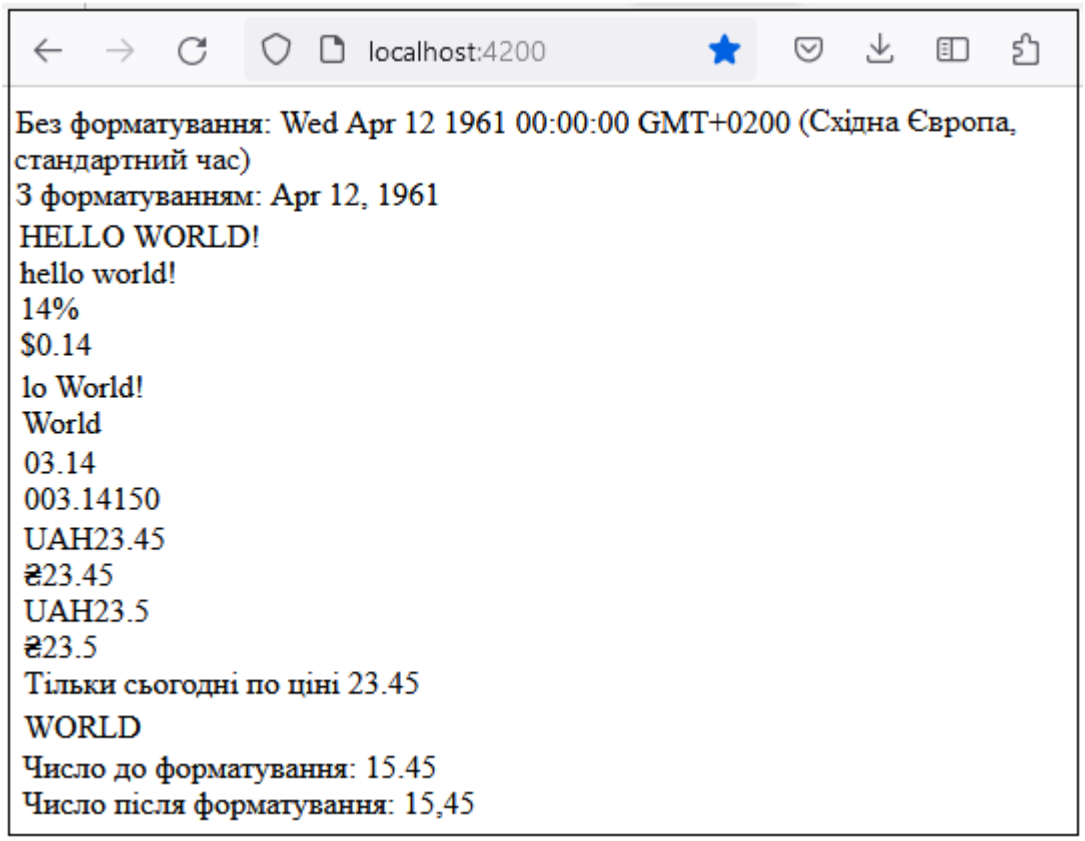
Застосуємо FormatPipe у коді компонента:

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

@Component({
   selector: 'my-app',
   template: `
            <div>Без форматування: {{myDate}}</div>
            <div>З 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!";
  persentage: 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 { }
```



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:

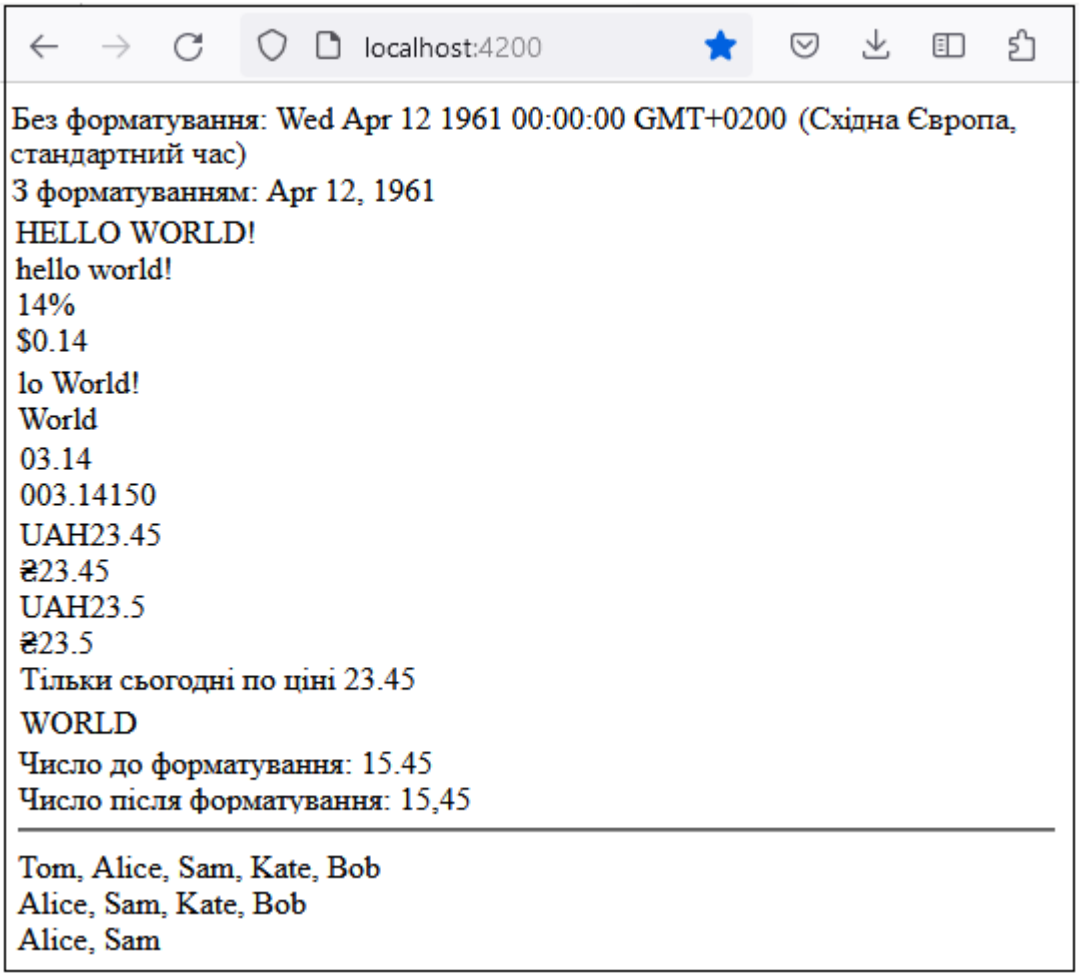
```
import { Component} from '@angular/core';
@Component({
   selector: 'my-app',
   template: `
          <div>Без форматування: {{myDate}}</div>
            <div>З форматуванням: {{myDate | date}}</div>
            <div>{{welcome | uppercase}}</div>
            <div>{{welcome | lowercase}}</div>
            <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>
                <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!";
      persentage: 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:


localhost:4200
Без форматування: Wed Apr 12 1961 00:00:00 GMT+0200 (Східна Європа, стандартний час)
З форматуванням: Apr 12, 1961
HELLO WORLD!
hello world!
14%
$0.14
lo World!
World
03.14
003.14150
UAH23.45
₴23.45
UAH23.5
₴23.5
Тільки сьогодні по ціні 23.45
WORLD
Число до форматування: 15.45
Число після форматування: 15,45
Tom, Alice, Sam, Kate, Bob
Alice, Sam, Kate, Bob
Alice, Sam