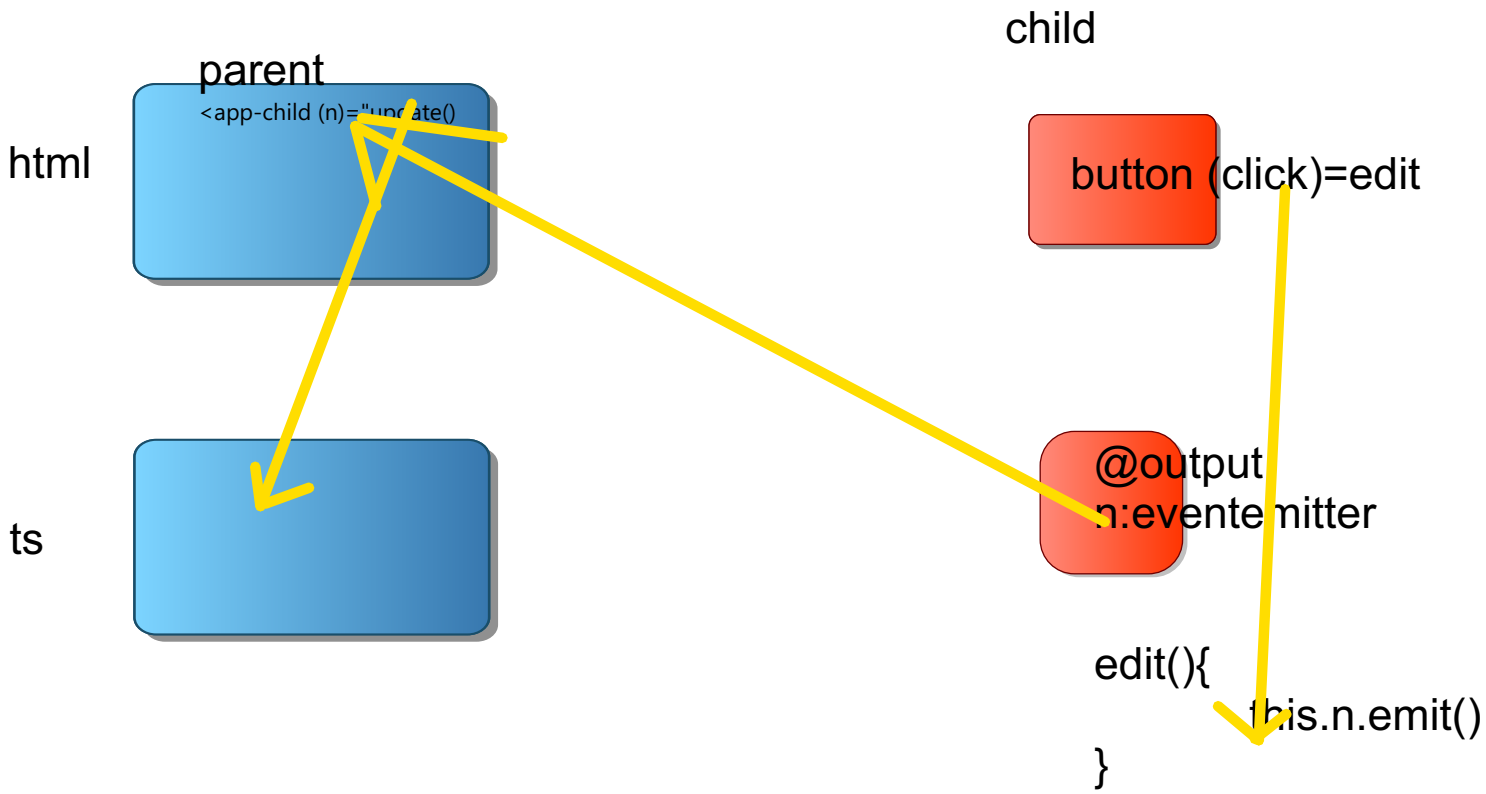
pipes
component interaction
app
parent componet
child componet


child
parent
<app-child (n)="update()
html
button (click)=edit
ts
@output
n:eventemitter
edit(){
this.n.emit()
}

Agenda

- Recap last lecture points
- Pipes
- Decorators-Based Sharing Data Between Components
- Intro About Signals
- Signal-Based Sharing Data Between Components

Pipes

Pipes

Pipes are simple functions you can use in template expressions to accept an input value and return a transformed value.

Angular provides built-in pipes for typical data transformations like :

- DatePipe: Formats a date value according to locale rules.
  - `{{ today | date : 'MMMM YYYY' }}`
- UpperCasePipe: Transforms text to all upper case.
  - `{{ name | uppercase }}`
- LowerCasePipe: Transforms text to all lower case.

For More Info : https://angular.dev/guide/templates/pipes

Custom Pipes

- To generate custom pipe you need to run :
  - **`ng generate pipe pipeName`**

For example you can generate custom pipe that transform file size to MB :

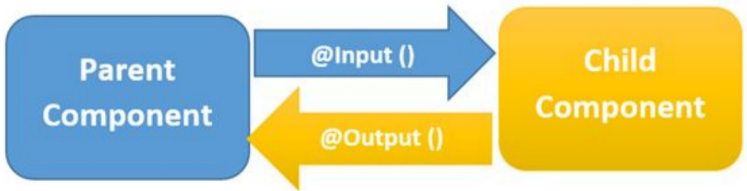
```
transform(value: any, ...args: any[]): unknown {
   return (value / (1024 * 1024)).toFixed(2) + 'MB';
 }
```

Sharing Data between Components

Sharing data between components

Parent to child :

This is probably the most common and straightforward method of sharing data. It works by using the @Input() decorator to allow data to be passed via the template.



Sharing data between components

Child to parent

Using **output and event emitter** is a way to share data is to emit data from the child, which can be listed to by the parent. This approach is ideal when you want to share data changes that occur on things like button clicks, form entires, and other user events.

Example :

- `<app-child (messageEvent)="receiveMessage($event)"></app-child> - parent`
- `receiveMessage($event) {this.message = $event}- parent.ts`
- `@Output() messageEvent = new EventEmitter<string>();- Child.ts`
- `sendMessage() {this.messageEvent.emit(this.message)} - Child.ts`
- `<button (click)="sendMessage()">Send Message</button> - Child.html`

Sharing data between components

Unrelated Components

- When passing data between components that lack a direct connection, such as siblings, grandchildren, etc, you should you a shared service.
- And you can also create a service to set and get values across unrelated components.

Will be covered in details later with services

Signals [Extra]

What are signals?

Signals are a new reactivity model introduced in Angular 16+ that provides a fine-grained way to manage state and change detection. They replace or complement traditional methods like RxJS and @Input() by offering automatic dependency tracking and synchronous updates.

Fine-grained reactivity means that only the specific parts of the UI or state that depend on a reactive value get updated when that value changes. Unlike traditional approaches that trigger updates for entire components or even the whole application, fine-grained reactivity ensures efficient updates with minimal re-renders.

How signals work

Traditional Change Detection in Angular

- Angular typically uses a zone-based change detection mechanism (via Zone.js). In this model:
  - Any event, HTTP request, or state change triggers global change detection.
  - The entire component tree is checked, even if only a small part of the UI needs updating.

How signals work

Signals-Based Change Detection

- Signals use fine-grained dependency tracking instead of global change detection.
  - Only components directly using a signal are updated.
  - No need for ngOnChanges()
  - Eliminates unnecessary UI re-renders.

Example

```
1  import { Component, signal } from '@angular/core';
2
3  @Component({
4    selector: 'app-child',
5    template: `<p>Data: {{ data() }}</p>`,
6  })
7  export class ChildComponent {
8    data = signal('Initial Value');
9
10   updateData(newValue: string) {
11     this.data.set(newValue);
12   }
13 }
```

Update Signal Value

- **Using .set(newValue) (Replace the value)**

```
this.data.set(newValue);
```

- **Using .update(fn) (Modify based on current value)**

```
this.data.update((prev) => prev + ' Updated');
```

- **Using .mutate(fn) (Modify objects/arrays in place)**

```
this.data.mutate((d) => { d.age += 1; });
```

Sharing Data using Signal-based functions

From Parent to Child

```
recipe = input<Recipe>();
```

Using this in html will call the recipe() function

From Child to Parent

```
outputFunction = output<number>();
```

No Need to use EventEmitter with the output function.

Conclusion

Key Differences

Feature	`@Input()` (Traditional)	`input()` (Signal-Based)
Reactivity	Not reactive (requires change detection)	Fully reactive (auto-tracked)
Change Handling	Uses `ngOnChanges` for updates	Automatically updates with signals
Performance	Triggers component re-renders	More efficient, updates only dependent parts
Usage	Decorator-based	Function-based
Introduced In	Older Angular versions	Angular 17+

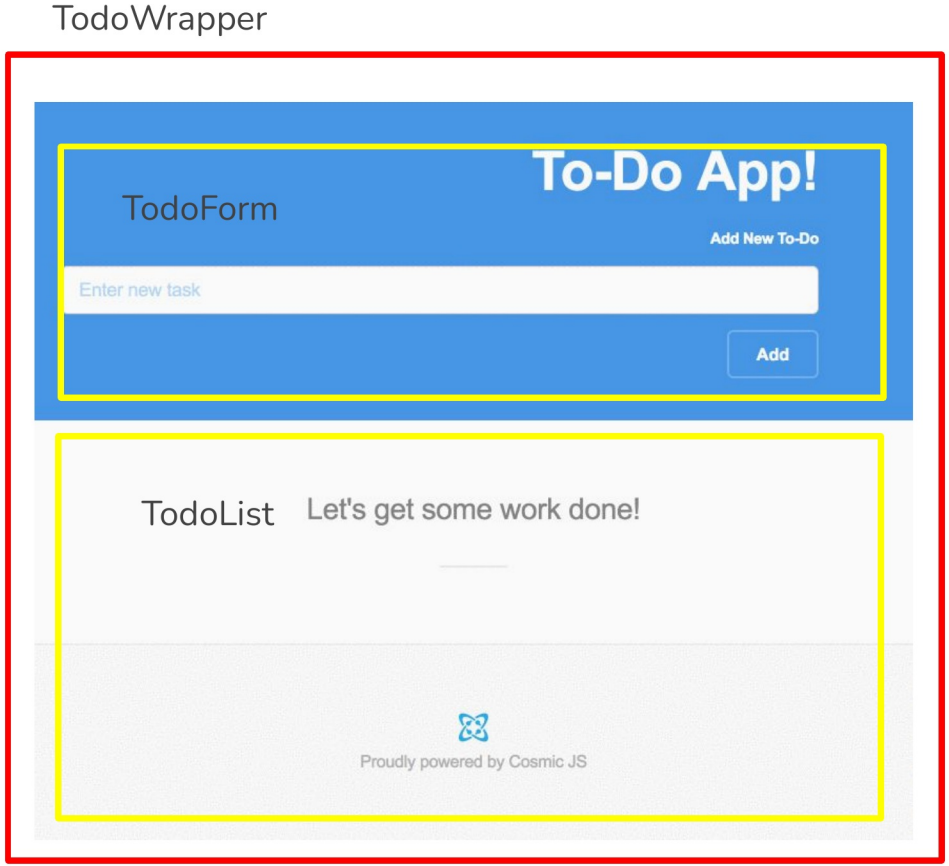
Thank you

Lab

To-do App

Refactor the previous task to share data between components

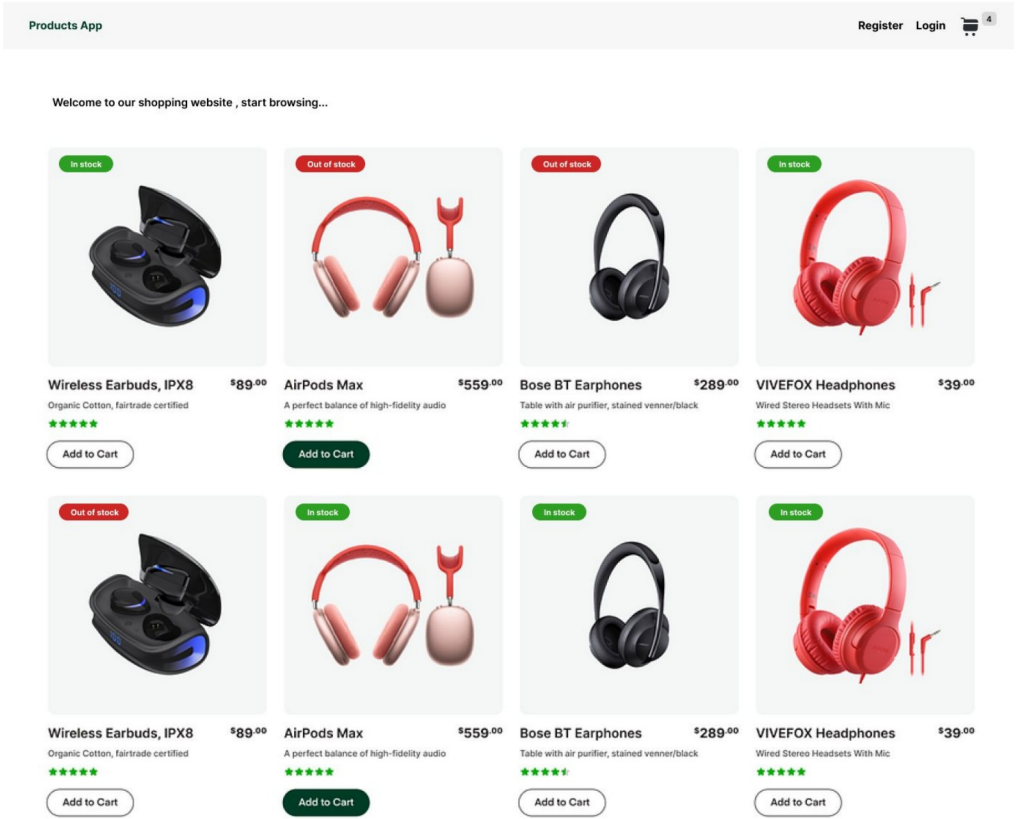
- **Red** is the parent component for the To-Do (Array of todos)
- **Yellow** borders are child components, one for the input with add button [which will emit the input value to the parent component]
- The other **yellow** border is for the other child component of the items list.
- User can add new task
- User can delete Task
- User can mark as completed, and when marked as completed, it will be marked with a linethrough. [Bonus]



Products

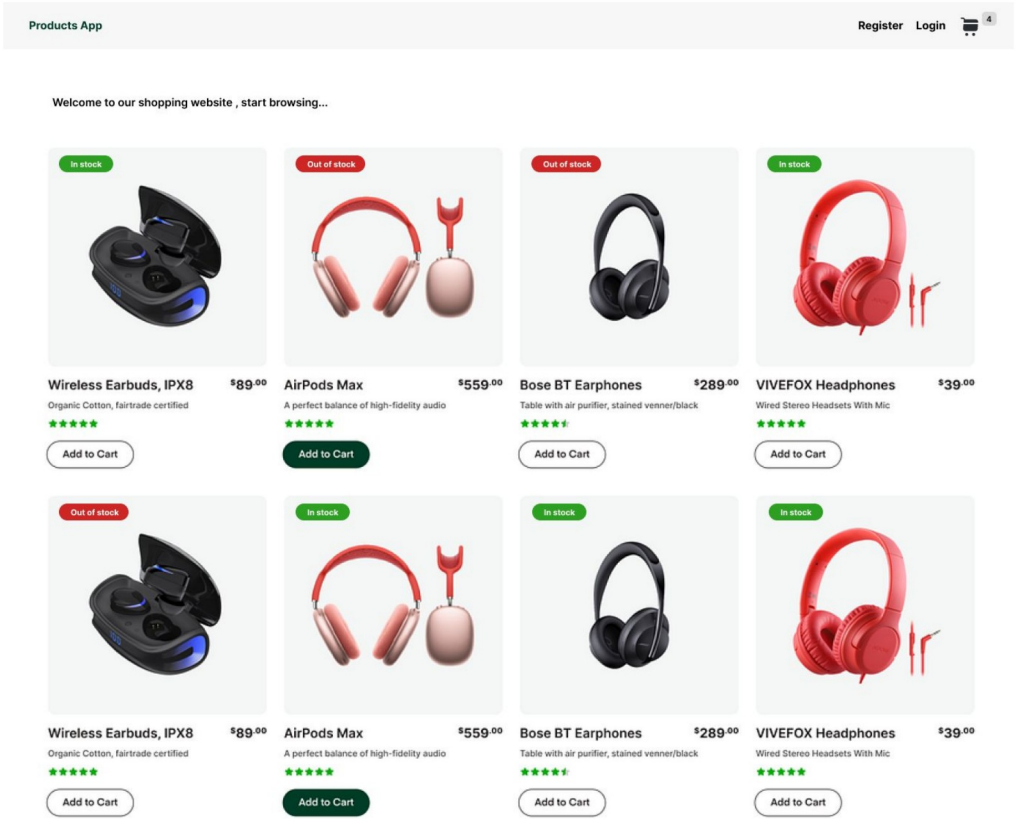
Create a new project

- Render Products from the provided products array [attached]
- Each product card is separate component



Products

- if stock= 0 will return out of stock else will return in stock
- If stock= 0 => out of stock text will be red
- If stock> 0 => In stock text will be green



Products

Each product card item should have:

- Product images
- Product name
- Product price: **20 EGP** (Use pipes to render Currency)
- Rate (Bonus if used stars to show rate)
- “Add to cart” button
- Use interface to define the product type