* **Tell me about your self:** 
  + **“Sure! I’m Rahul Parikh, an application developer with over 6 years of experience, including strong front-end development and 3 years of specialized work with Microsoft Power Platform and Dynamics 365 CRM. I've led the design and development of responsive web portals, primarily using HTML, CSS, JavaScript, jQuery, and I’ve also been involved in Angular-based projects.”**
  + **“In my current role at DynaTech, I’ve served as the lead developer on several high-impact client portals like SAICA’s Education and Exam platforms. I worked end-to-end on real-time validations, Excel parsing, and data workflows using Power Apps Portals, while building reusable UI components and driving performance improvements.”**
  + **“I’ve also created internal tools like Timesheet and Performance Review apps to automate business processes using Power Automate. Earlier, at Impexperts, I contributed to custom CRM/ERP systems and helped enhance UI/UX across key business tools.”**
  + **“I’m now looking to deepen my skills in Angular and modern front-end frameworks — especially in enterprise-grade applications — and this Angular developer role is a great fit for both my current strengths and my career goals.”**
* **Angular Project skills script:**
  + **“Currently, I’m working on a project that’s built in Angular — it’s a web application designed to streamline internal processes like performance tracking and timesheet management. I’m mainly focused on building reusable UI components and handling client-side logic using Angular.”**
  + **“In this project, I’ve worked with core Angular concepts like component communication using @Input and @Output, custom event handling with EventEmitter, and form control using ngModel. I’ve also started exploring newer Angular features like Signals and the modern @for syntax to improve reactivity and performance in the UI.”**
  + **“While working on this, I’ve been actively researching best practices — for example, how Signals differ from Zone.js, and how to use input() and output() functions in standalone components. This has helped me write cleaner, more reactive code that performs better and is easier to maintain.”**
  + **“This project has been a great opportunity for me to both apply my frontend fundamentals and explore Angular’s modern features, which aligns well with the role I’m applying for.”**

**In Component-services-template-more project file**

* **Angular CLI**: It is the **command line interface tool** which allows you to **scaffold, develop, test, deploy and maintain angular application** directly from a command shell.
* Module Base component and Standalone component: In older versions of angular only module base components but in newer version of angular both the components types are existed.
* In Component-services-template-more:
  + Data binding
    - One way binding
    - State management

**Zone JS**: Zone.js notifies angular about user events, expired timers etc. – Used for state management. This process works automatically when any event calls. Don’t need to call manually.

Zone.js is a library Angular uses to monkey-patch async operations (like setTimeout, Promise, HTTP calls, etc.) and automatically trigger **change detection**.

**How it works:**

* It intercepts async events in the browser.
* When an event happens (e.g. button click, HTTP response), Zone.js tells Angular to run **change detection** across the entire component tree.
* This ensures the UI is always updated, but can be **inefficient** because it checks everything, even if nothing changed.

**Pros:**

* Zero config; it "just works".
* Handles all async operations automatically.

**Cons:**

* Performance overhead in large apps.
* Lack of fine-grained control.
* Harder to reason about what triggers re-renders.

**Signals**: A signal is an object that stores a value (any type of value, including nested objects). This is also used for state management.

Signals are Angular’s modern **reactivity primitive** introduced to give fine-grained, efficient reactivity (like in SolidJS or React signals).

**How it works:**

* A **Signal** is a reactive value. You can **read** from it, and when it changes, only the parts of the UI that depend on it update.
* Think of it like BehaviorSubject without RxJS boilerplate, and way more optimized.
* You don’t need Zones — Angular knows *exactly* what to re-render and when.

**Pros:**

* Super efficient — updates only the affected DOM.
* Better developer control and predictability.
* Works without Zone.js (Zone-less Angular is now possible).

**Cons:**

* Still new, with a learning curve.
* Might not yet cover every use case that devs are used to with traditional Angular.
  + **Computed**: It is a function that meant to be use with signals. Why we use this – angular automatically analyse where you are reading some signal value inside of that function you pass to computed. So set up a subscription to that signal.
  + **One Important point**: If you want to set map method or showing multiple data in any component which different from core component then you have to import that data to that component as well.
  + **@Input() Decorator**: @Input() is a decorator that makes a property **publicly bindable** from a **parent component**.
    - This is the process flow of parent to child data transfer using @Input decorator.  
      app.component.ts (has users[])

⬇ passes data

app.component.html (binds avatar and name to <app-user>)

⬇ sends values

user.component.ts (receives via @Input())

* **input() function**: A **function** from Angular’s new **Signals API** to define **reactive inputs** — meant to **replace** @Input() when using Signals.
  + Example: name = input<string>();
* **@Output():**
  + @Output() is a decorator in Angular used to create custom events that a child component can emit, and the parent can listen to.
  + It works with an EventEmitter, which is like a mini broadcaster.
  + **Why Use It?**
    - When a child component wants to notify the parent about something:
    - A button click
    - A selected item
    - A form submission
    - Any interactionor state change
* **$event:** 
  + $event is a special Angular keyword that represents the data emitted or passed during an event — whether it’s a DOM event (like a button click) or a custom event from a component (@Output() + EventEmitter).
    - Think of $event as:
      * "Whatever data is coming with this event, put it here."
* **Output() function:** 
  + Uses the **output() function** instead of the @Output() decorator.
  + It simplifies the code and fits the **signal-first** style.
  + Still uses EventEmitter under the hood.
* **When to Use Which?**

| **If you're using...** | **Use...** |
| --- | --- |
| Classic Angular | @Output() |
| Angular Signals / Standalone Components | output() |
| Migration to Zone-less Angular | output() |

* **@for loop** in app.component.html file:
  + **Syntax:** @for(user of users; track user.id)
* **Track in for loop:** 
  + track is the **modern equivalent of trackBy** in \*ngFor.
  + It tells Angular **how to uniquely identify** each item in the loop.
  + This is **critical for performance** — it helps Angular avoid destroying and recreating DOM elements unnecessarily.
* **@if and @else** in app.component.html file:
  + **Syntax:**

**@if (selectedUser) {**

**<app-tasks [userName]="selectedUser.name" />**

**} @else {**

**<p id="fallback">Select a user to see there task</p>**

**}**

* **\*ngfor, \*ngif and else:** This syntax is used in previous versions of angular (Angular 2 to Angular 16) after that versions latest syntax is used.
  + **Older syntaxs:**

**<li \*ngFor="let user of users">**

**<app-user [user]="user" (select)="onSelectUser($event)" />**

**</li>**

**<app-tasks \*ngIf="selectedUser; else fallback" [userName]="selectedUser!.name" />**

**<ng-template #fallback>**

**<p id="fallback">Select a user to see there task</p>**

**</ng-template>**

* **2 way Binding: This is in new-task.component.html file.**
  + **[(ngModel)]:** 
    - ngModel is a directive that enables two-way data binding in Angular forms. It connects a form input (like a textbox, checkbox, etc.) to a property in your component — so when the user types or selects something, the component automatically updates, and vice versa.

<input [(ngModel)]="username" />

**This is called banana in a box syntax:**

() = event binding (from input → component)

[] = property binding (from component → input)

Combined: [(...)] = two-way binding

* for using ngModel need to import **FormsModule.**