# Core Angular Foundations

## Angular Basics

### Components

* **Definition / Concept**
  + - A Component in Angular is the fundamental building block of the UI.
    - It controls a portion of the screen called a view.
    - Each component is a TypeScript class decorated with @Component, which defines:
      * Selector (HTML tag used to render the component)
      * Template (HTML view)
      * Styles (CSS specific to the component)
* Think of components as reusable UI + logic bundles.
* **How it Works / Key Idea**
  + - Angular apps are component trees (root component → child components).
    - A component is created by Angular when referenced in HTML using its selector.
    - Data flows between components using:
      * @Input() → to receive data from parent.
      * @Output() → to send events back to parent.
    - Change detection ensures the view updates when data changes.
    - Components are modular and can be reused, nested, and tested independently.

### Templates

* **Definition / Concept**
  + - A Template in Angular is the HTML view of a component.
    - It defines the structure, layout, and how the component’s data (from the TypeScript class) is rendered in the browser.
    - Templates are enriched HTML — they can include Angular’s template syntax like interpolation ({{ }}), directives (\*ngIf, \*ngFor), property & event bindings, and pipes.
* **How it Works / Key Idea**
  + - Each Angular component has a template (inline using template property, or external via templateUrl).
    - The Component Class (TS) provides the data and logic.
    - The Template (HTML) presents that data using Angular’s binding features.
    - Angular’s change detection keeps the DOM in sync with component state automatically.
    - Key mechanisms:
      * Interpolation: {{ propertyName }} → Inserts component data into the DOM.
      * Property Binding: [property]="expression" → Binds DOM property to component data.
      * Event Binding: (event)="handler()" → Responds to user actions.
      * Structural Directives: \*ngIf, \*ngFor → Dynamically add/remove DOM elements.

### Data Binding

* **Definition / Concept**
  + - Data Binding in Angular is the mechanism that establishes a connection between the component’s data (TypeScript/logic layer) and the view (HTML/template layer). It allows data to flow from component to template, from template to component, or both ways, keeping the UI and logic in sync without manual DOM manipulation.
* **How it Works / Key Idea**
  + - Angular uses bindings (special syntax in the template) to connect the component class and the DOM.
    - Behind the scenes, Angular’s change detection mechanism checks for updates in component properties or DOM events and reflects those changes automatically.
    - Types of data binding:
      * **Interpolation ({{ }})** → one-way binding from component to view.
      * **Property Binding ([ ])** → binds DOM element/property/attribute to a component property.
      * **Event Binding (( ))** → sends DOM events (click, input, etc.) back to the component
      * **Two-way Binding ([( )] or banana-in-a-box)** → combination of property + event binding, keeps component and view synchronized (commonly used with ngModel).

### Directives

* **Definition / Concept**
  + - Directives are special instructions in Angular that let you manipulate the DOM or extend HTML by adding custom behavior to elements.
    - They are classes with the @Directive decorator.
    - Types of Directives:
      * **Component Directives** → Technically a directive with a template (@Component).
      * **Structural Directives** → Change DOM structure (e.g., \*ngIf, \*ngFor, \*ngSwitch).
      * **Attribute Directives** → Change the appearance/behavior of elements (e.g., ngClass, ngStyle, custom highlight directive).
* **How it Works / Key Idea**
  + - Angular parses templates and looks for directives attached to elements.
    - Structural directives manipulate view containers (add/remove DOM elements).
    - Attribute directives listen to events or modify properties/styles of DOM elements.
    - Custom directives are created using @Directive({ selector: '[myDirective]' }).

### Pipes

* **Definition / Concept**
  + - Pipes in Angular are a way to transform data in templates before displaying it to the user.
    - They are simple functions that take in data as input and return a transformed version of that data.
    - Example: Formatting dates, converting text to uppercase, filtering lists, or creating custom transformations.
* **How it Works / Key Idea**
  + - Pipes are applied in the template expressions using the | (pipe) symbol.
    - Angular provides built-in pipes like DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, etc.
    - Developers can also create custom pipes for application-specific transformations.
    - Pure vs Impure Pipes:
      * **Pure pipes:** Transform data only when the input changes (default, more efficient).
      * **Impure pipes:** Recalculate on every change detection cycle, even if the data hasn’t changed (used for dynamic transformations like filtering arrays).

## Modules & Bootstrapping

### NgModule

* **Definition / Concept**
  + - An NgModule is a decorator function (@NgModule) that takes a metadata object describing how to compile a component’s template and how to create an injector at runtime.
    - It is the basic building block of an Angular application, used to organize related pieces of functionality (components, directives, pipes, and services) into cohesive blocks.
    - Every Angular app has at least one module, the root module (usually AppModule).
* **How it Works / Key Idea**
  + - @NgModule groups Angular entities into containers for better structure and reusability.
    - It has important metadata properties:
      * **declarations** → components, directives, and pipes that belong to this module.
      * **imports** → other Angular modules needed for this module (e.g., BrowserModule, FormsModule).
      * **exports** → components, directives, pipes made available for use in other modules.
      * **providers** → services/injectables available to the module’s injector.
      * **bootstrap** → root component(s) to start the application (only in the root module).
    - By organizing features into separate NgModules (feature modules, shared modules, etc.), Angular enables modularity, lazy loading, and maintainability.

### Imports/Exports

* **Definition / Concept**
  + - In Angular, imports and exports are part of the @NgModule metadata configuration.
    - Imports: Bring in other modules so their exported classes, components, directives, and pipes can be used in the current module.
    - Exports: Make components, directives, and pipes from the current module available for use in other modules that import it.
    - Essentially, imports = what you bring in, exports = what you share out.
* **How it Works / Key Idea**
  + - Each Angular module (@NgModule) can declare components, directives, and pipes. But those declarations are private by default.
    - If another module needs them:
      * The declaring module must export them.
      * The consuming module must import the declaring module.
    - This system promotes **modularity**, **code reusability**, and **clean separation of concerns**.

## Lifecycle Hooks

* **Definition / Concept**
  + Lifecycle hooks in Angular are special methods that allow developers to tap into key moments in a component or directive’s life — from its creation, change detection, rendering, view/content initialization, updates, and finally destruction.
  + Each hook corresponds to a specific event in the Angular lifecycle, enabling developers to run custom logic (e.g., initializing data, subscribing to services, cleaning up resources) at the right stage.
  + The most common lifecycle hooks include:
    - **ngOnChanges()** → Runs when input properties change.
    - **ngOnInit()** → Runs once, after the component is initialized.
    - **ngDoCheck()** → Runs on every change detection cycle.
    - **ngAfterContentInit() / ngAfterContentChecked()** → Run after external content is projected into the component.
    - **ngAfterViewInit() / ngAfterViewChecked()** → Run after the component’s view and child views are initialized/checked.
    - **ngOnDestroy()** → Runs right before the component is destroyed.
* **How it Works / Key Idea**
  + Component Initialization → Angular creates the component, sets input properties, then calls hooks in order.
  + Change Detection → Angular continuously checks for changes (via its change detection mechanism) and triggers hooks (ngOnChanges, ngDoCheck, ngAfterViewChecked, etc.).
  + Content & View Lifecycle → When Angular projects external content (<ng-content>) or renders child components, it triggers AfterContent... and AfterView... hooks.
  + Destruction → When the component is removed, Angular calls ngOnDestroy() so resources (subscriptions, timers, etc.) can be cleaned up.
* Key Idea:
  + Lifecycle hooks = well-defined entry points for adding custom behavior at specific points in a component’s life. They ensure proper setup, monitoring, and teardown without interfering with Angular’s internal flow.

## Dependency Injection & Services

### DI hierarchy

### Providers

## TypeScript Essentials

### Interfaces

### Generics

### Classes

## Mock Interview Questions

### Components

* + 1. What is a Component in Angular?
    2. How do Components work in Angular?

### Templates

* + 1. What is a template in Angular?
    2. How do you define a template in Angular?

### Data Binding

* 1. What is Data Binding in Angular?
  2. How many types of Data Binding are there in Angular?
  3. What is the difference between Interpolation and Property Binding?
  4. How does Angular achieve Two-way Data Binding?
  5. Why is Event Binding important in Angular?
  6. Is Angular’s Data Binding one-way or two-way by default?
  7. Can we achieve two-way binding without ngModel?
  8. What are the advantages of Data Binding in Angular compared to vanilla JavaScript or jQuery?

### Directives

* + 1. What are directives in Angular?
    2. What are the different types of directives?
    3. What is the difference between structural and attribute directives?
    4. How to create and give an example of a custom directive?
    5. Why is \*ngIf written with a star (\*)?
    6. Can we apply multiple directives on the same element?

### Pipes

* + 1. What are pipes in Angular?
    2. Can you give some examples of built-in pipes in Angular?
    3. What is the difference between pure and impure pipes?
    4. How do you create a custom pipe?
    5. When should you use a pipe instead of a method call in a template?

### NgModule

* + 1. What is an NgModule in Angular?
    2. What are the key properties of the @NgModule decorator?
    3. How many types of modules are there in Angular?
    4. Can a component be declared in more than one NgModule?

### Imports/Exports

* + 1. What is the purpose of imports in Angular?
    2. What is the difference between imports and exports in NgModule?
    3. How are imports different from exports in Angular?
    4. If a module declares a component but doesn’t export it, can another module use it?
    5. Can services be exported in Angular modules?
    6. What happens if you import the same module multiple times?

### Lifecycle hooks

* + 1. What are lifecycle hooks in Angular?
    2. What is the difference between ngOnInit() and constructor()?
    3. When does ngOnChanges() run, and what does it receive as input?
    4. Why do we use ngOnDestroy()?
    5. Can you explain the order of some key lifecycle hooks?

### DI hierarchy

### Providers

### Interfaces

### Generics

### Classes