Contents

Components 3

- Selector 3
- Providers 3
- Pipes 3
- Directives 3
  - Components 3
  - Structural 3
  - Attribute directives 3

Template 3

Directive 3

Binding 4

- Event binding 4
- Property binding 4
- Two-way data binding 4

Service 4

NgModules 4

- Declarations 4
- ~~providers~~ 4

Dependency injection (DI) 5

Template expressions 5

Template statements 5

@Input() 5

@Output() 5

Safe navigation operator (?) 5

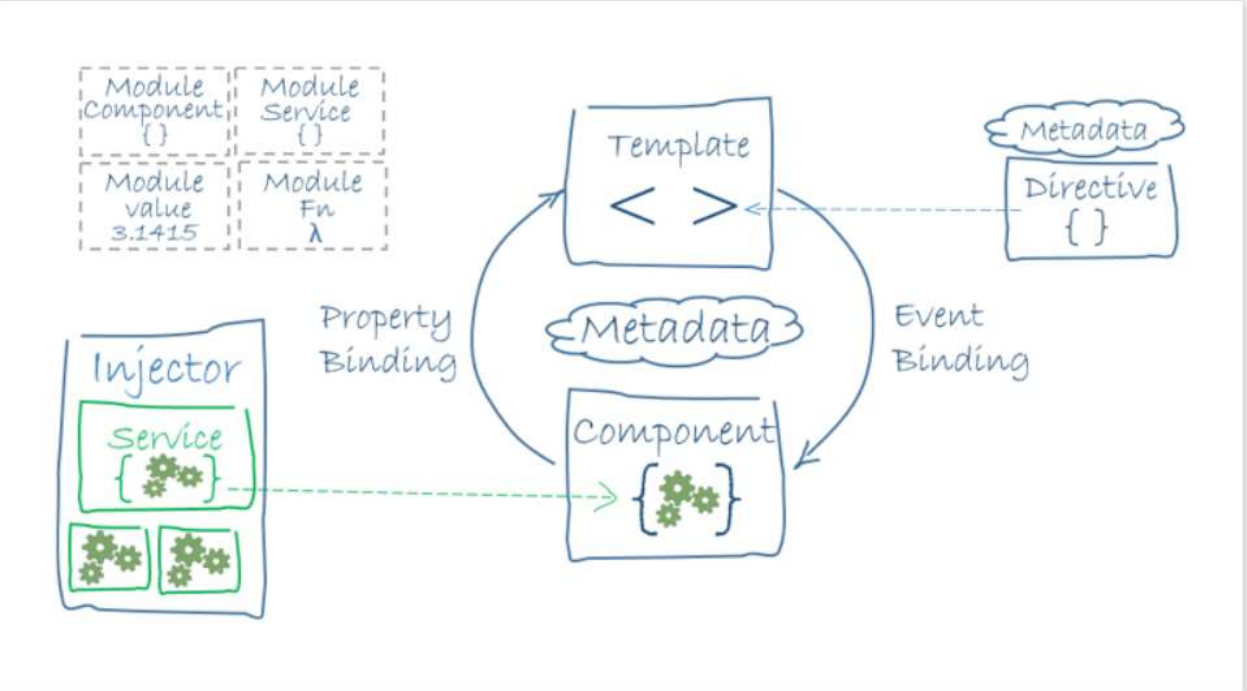
Lifecycle sequence 5

- OnInit() 5
- OnChanges() 5
- AfterView 5
- AfterContent 6

Component Interaction 6

ComponentFactoryResolver 6

Forms 6
Reactive forms 6
Template-driven forms 6
Data flow in forms 7
Reactive Forms 7
Form Validation 8
Template-driven validation 8
Reactive form validation 8
Built-in validators 8
Custom validators 8
Observables 8
Subject.next() 8
Multicasting 8
RxJS 9
Operators 9
Observables in Angular 9
Observables VS promises 9
Entry Components 9
Feature Modules 9
forRoot() 10
forChild() 10
Hierarchical injectors 10
Routing 10
Route guards 10
The Ahead-of-Time (AOT) compiler 11
QA 12
What would you not put shared module? 12
Difference between TypeScript and JavaScript: 12
Zone 12



Components

Component defines a class that contains application data and logic, and is associated with an HTML template that defines a view.

- Selector: tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML.
- Providers: A provider is an instruction that tells an injector how to obtain or create a dependency. An array of providers for services that the component requires.
- Pipes: to transform data before it is displayed
- Directives: to apply app logic to what gets displayed.
    - Components: directives with a template.
    - Structural: change the DOM layout by adding and removing DOM elements (NgFor and NgIf.)
    - Attribute directives: change the appearance or behavior of an element, component, or another directive (NgStyle)

Template

A template combines HTML with Angular markup that can modify HTML elements before they are displayed.

Directive

Directives provide program logic

Binding

Binding markup connects our application data and the DOM. Two types of data binding:

- Event binding lets your app respond to user input in the target environment by updating your application data
- Property binding lets you interpolate values that are computed from your application data into the HTML
- Two-way data binding (used mainly in template-driven forms) combines property and event binding in a single notation.
    - Sets a specific element property.
    - Listens for an element change event.

Type	Syntax	Category
Interpolation Property Attribute Class Style	{{expression}} [target]="expression" bind-target="expression"	One-way from data source to view target
Event	(target)="statement" on-target="statement"	One-way from view target to data source
Two-way	[(target)]="expression" bindon-target="expression"	Two-way

Service

For data or logic that isn't associated with a specific view, and that we want to share across components

- Fetching data from the server, validating user input.

NgModules

- are containers for a cohesive block of code dedicated to an application domain
- can contain components, service providers, and other code files
    - Declarations: Make the components, directives, and pipes from the current module available to others in the current module
    - ~~providers: The same instance of a service is available to all components in that NgModule.~~

Dependency injection (DI)

- DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need
- An injector creates dependencies, and maintains a container of dependency instances that it reuses if possible

Template expressions

A template expression produces a value and appears within the double curly braces, {{ }}. Angular executes the expression and assigns it to a property of a binding target.

- Simplicity
- Quick execution
- No visible side effects

Template statements

A template statement responds to an event raised by a binding target such as an element, component, or directive. A template statement has a side effect.

@Input()

@Input() decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component.

@Output()

Use the @Output() decorator in the child component or directive to allow data to flow from the child out to the parent.

Safe navigation operator (?)

{{item?.name}} If item is null, the view still renders but the displayed value is blank

Lifecycle sequence

OnInit()

Use ngOnInit() for two main reasons:

- To perform complex initializations shortly after construction
- To set up the component after Angular sets the input properties.

OnChanges()

Angular calls its ngOnChanges() method whenever it detects changes to input properties of the component

AfterView

The AfterView sample explores the AfterViewInit() and AfterViewChecked() hooks that Angular calls after it creates a component's child views

AfterContent

The AfterContent sample explores the AfterContentInit() and AfterContentChecked() hooks that Angular calls after Angular projects external content into the component

Component Interaction

- Pass data from parent to child with input binding
- Parent listens for child event
    - The child component exposes an EventEmitter property with which it emits events when something happens. The parent binds to that event property and reacts to those events
- Intercept input property changes with a setter
    - Use an input property setter to intercept and act upon a value from the parent.
- Intercept input property changes with ngOnChanges()
- Parent calls an @ViewChild()
- Parent and children communicate via a service

ComponentFactoryResolver

use ComponentFactoryResolver to add components dynamically.

A simple registry that maps Components to generated ComponentFactory classes that can be used to create instances of components

Forms

Reactive forms: are more robust. They're more scalable, reusable, and testable.

- The reactive form directive (in this case, FormControlDirective) then links the existing FormControl instance to a specific form element in the view using a value accessor (ControlValueAccessor instance).
- Define custom validators as functions that receive a control to validate.

Template-driven forms: are useful for adding a simple form to an app.

- The template-driven form directive NgModel is responsible for creating and managing the FormControl instance for a given form element. It's less explicit, but you no longer have direct control over the form model
- Are tied to template directives, and must provide custom validator directives that wrap validation functions.

	REACTIVE	TEMPLATE-DRIVEN
Setup (form model)	More explicit, created in component class	Less explicit, created by directives
Data model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Form validation	Functions	Directives
Mutability	Immutable	Mutable
Scalability	Low-level API access	Abstraction on top of APIs

Data flow in forms

In reactive forms each form

Element in the view is directly linked to a form model. Updates from the view to the model and from the model to the view are synchronous (The FormControl instance emits the new value through the valueChanges observable. Any subscribers to the valueChanges observable receive the new value.)
https://angular.io/guide/forms-overview#data-flow-in-reactive-forms

Data flow in template-driven forms

Each form element is linked to a directive that manages the form model internally.

Reactive Forms

There are two ways to update the model value:

- Use the setValue() method to set a new value for an individual control. It replaces the entire value for the control.
- Use the patchValue() method to replace any properties defined in the object that have changed in the form model.

```
this.user = this.userService.loadUser().pipe(
  tap(user => this.form.patchValue(user))
);
```

https://angular.io/guide/reactive-forms#dynamic-controls-using-form-arrays

Form Validation

Template-driven validation

To add validation to a template-driven form, you add the same validation attributes as you would with native HTML form validation. Angular uses directives to match these attributes with validator functions in the framework

Reactive form validation

There are two types of validator functions:

- **Sync validators**: functions that take a control instance and immediately return either a set of validation errors or null. You can pass these in as the second argument when you instantiate a FormControl.
- **Async validators**: functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or null. You can pass these in as the third argument when you instantiate a FormControl.
    - https://alligator.io/angular/async-validators/
- *Note: for performance reasons, Angular only runs async validators if all sync validators pass. Each must complete before errors are set*

Built-in validators

We can choose to write your own validator functions, or you can use some of Angular's built-in validators.

Custom validators

Since the built-in validators won't always match the exact use case of your application, sometimes you'll want to create a custom validator.

https://angular.io/guide/form-validation#adding-to-reactive-forms

https://angular.io/guide/form-validation#control-status-css-classes

Observables

Observables provide support for passing messages between publishers and subscribers in your application. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe. An observable can deliver multiple values of any type—literals, messages, or events, depending on the context.

Subject.next()

The subject next method is used to send messages to an observable which are then sent to all angular components that are subscribers (a.k.a. observers) of that observable.

Multicasting

A list of multiple subscribers in a single execution. With a multicasting observable, you don't register multiple listeners on the document, but instead re-use the first listener and send values out to each subscriber

RxJS

It Is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code

RxJS provides an implementation of the Observable type

Converting existing code for async operations into observables

- Iterating through the values in a stream
- Mapping values to different types
- Filtering streams
- Composing multiple streams

Operators

Operators are functions that build on the observables foundation to enable sophisticated manipulation of collections

Observables in Angular

Angular makes use of observables as an interface to handle a variety of common asynchronous operations:

- You can define custom events that send observable output data from a child to a parent component.
- The HTTP module uses observables to handle AJAX requests and responses.
- The Router and Forms modules use observables to listen for and respond to user-input events.

Observables VS promises

- Observables are declarative; computation does not start until subscription. Promises execute immediately on creation
- Observables provide many values. Promises provide one.
- Observables differentiate between chaining and subscription. Promises only have .then() clauses.
- Observables subscribe() is responsible for handling errors. Promises push errors to the child promises.

Entry Components

An entry component is any component that Angular loads imperatively.

There are two main kinds of entry components:

- The bootstrapped root component.
- A component we specify in a route definition.

Feature Modules

Feature modules are NgModules for the purpose of organizing code relevant for a specific feature

forRoot()

Creates and configures a module with all the router providers and directives

forChild()

Creates a module with all the router directives and a provider registering routes.

Hierarchical injectors

There are two injector hierarchies in Angular:

- ModuleInjector hierarchy—configure a ModuleInjector in this hierarchy using an @NgModule() or @Injectable() annotation.
- ElementInjector hierarchy—created implicitly at each DOM element. An ElementInjector is empty by default unless you configure it in the providers property on @Directive() or @Component().

Routing

The Angular router enables you to show different components and data to the user based on where the user is in the application. The router enables navigation from one view to the next as users perform application tasks

- Enter a URL in the address bar, and the browser navigates to a corresponding page.
- Click links on the page, and the browser navigates to a new page.
- Click the browser's back and forward buttons, and the browser navigates backward and forward through the history of pages you've seen.

Route guards

Guards to the route configuration to handle these scenarios

- Perhaps the user is not authorized to navigate to the target component.
- Maybe the user must login (authenticate) first.
- Maybe you should fetch some data before you display the target component.
- You might want to save pending changes before leaving a component.
- You might ask the user if it's OK to discard pending changes rather than save them.

A guard's return value controls the router's behavior:

- If it returns true, the navigation process continues.
- If it returns false, the navigation process stops and the user stays put.
- If it returns a UrlTree, the current navigation cancels and a new navigation is initiated to the UrlTree returned.

The router supports multiple guard interfaces:

- CanActivate to mediate navigation to a route. (requiring authentication)
- CanActivateChild to mediate navigation to a child route. (it runs before any child route is activated.)

- CanDeactivate to mediate navigation away from the current route.
- Resolve to perform route data retrieval before route activation.
- CanLoad to mediate navigation to a feature module loaded asynchronously.

The Ahead-of-Time (AOT) compiler

The Angular Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads and runs that code. Compiling your application during the build process provides a faster rendering in the browser.

QA

What would you not put shared module?

I would not put services in a shared module which may be imported by a lazy loaded module. When a lazy loaded module imports a module which provide a service, angular will create another instance of this service which may result in unexpected behaviors.

Difference between TypeScript and JavaScript:

- TypesScript is known as Object oriented programming language whereas JavaScript is a scripting language.
- TypeScript has a feature known as Static typing but JavaScript does not have this feature.
- TypeScript gives support for modules whereas JavaScript does not support modules.
- TypeScript has Interface but JavaScript does not have Interface.
- TypeScript support optional parameter function but JavaScript does not support optional parameter function.

Zone

NgZone is a wrapper around Zone.js which The most common use of this service is to optimize performance when starting a work consisting of one or more asynchronous tasks that don't require UI updates or error handling to be handled by Angular