# Angular

Qinshift Academy

# Introduction to Angular

Qinshift Academy

# What is Angular?

Angular is a development platform, built on TypeScript.

As a platform, Angular includes:

- A component-based framework for building scalable web applications
- A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more
- A suite of developer tools to help you develop, build, test, and update your code

With Angular, you're taking advantage of a platform that can scale from single-developer projects to enterprise-level applications. Best of all, the Angular ecosystem consists of a diverse group of over 1.7 million developers, library authors, and content creators.

# Components

Components are the fundamental building block for creating applications in Angular. By leveraging component architecture, Angular aims to provide structure for organizing your project into manageable, well organized parts with clear responsibilities so that your code is maintainable and scalable.

An Angular component can be identified by the component suffix (e.g., my-custom-name.component.ts and has the following:

- A decorator to define configuration options for things like:
    - A selector that defines what the tag name is when referring a component in a template
    - An HTML template that controls what is rendered to the browser
- A TypeScript class that defines the behavior of the component. Examples include handling user input, managing state, defining methods, etc.

# Simplified example of a TodoListItem component.

```
// 📄 todo-list-item.component.ts
@Component({
  standalone: true,
  selector: 'todo-list-item',
  template: ` <li>(TODO) Read cup of coffee introduction</li> `,
  styles: ['li { color: papayawhip; }'],
})
export class TodoListItem {
  /* Component behavior is defined in here */
}
```

# State

When defining data that you want the component to manage, this can be accomplished by declaring it by defining class fields.

In the example of a todo-list-item.component.ts, there are two properties we want to track: taskTitle and isComplete. Using the class field syntax, they can be defined as follows:

```
// 📄 todo-list-item.component.ts
@Component({ ... })
export class TodoList {
  taskTitle = '';
  isComplete = false;
}
```

# Methods

You can define functions for a component by declaring methods within the component class.

```
// 📄 todo-list-item.component.ts
@Component({ ... })
export class TodoList {
  taskTitle = '';
  isComplete = false;

  updateTitle(newTitle: string) {
    this.taskTitle = newTitle;
  }

  completeTask() {
    this.isComplete = true;
  }
}
```

# Templates

Every component has an HTML template that defines what that component renders to the DOM.

HTML templates can be defined as an inline template within the TypeScript class, or in separate files with the templateUrl property. To learn more, check out the docs on defining component templates.

Within this document, the examples will use inline templates for more concise code snippets.

# Rendering Dynamic Data

When you need to display dynamic content in your template, Angular uses the double curly brace syntax in order to distinguish between static and dynamic content.

```
@Component({
  template: ` <p>Title: {{ taskTitle }}</p> `,
})
export class TodoListItem {
  taskTitle = 'Read cup of coffee';
}
```

This is how it renders to the page.

```
<p>Title: Read cup of coffee</p>
```

This syntax declares an interpolation between the dynamic data property inside of the HTML. As a result, whenever the data changes, Angular will automatically update the DOM reflecting the new value of the property.

# Dynamic Properties and Attributes

When you need to dynamically set the value of attributes in an HTML element, the target property is wrapped in square brackets. This binds the attribute with the desired dynamic data by informing Angular that the declared value should be interpreted as a JavaScript-like statement (with some Angular enhancements) instead of a plain string.

<button [disabled]="hasPendingChanges"></button>

In this example, the disabled property is tied to the hasPendingChanges variable that Angular would expect to find within the component’s state.

# Event Handling

You can bind event listeners by specifying the event name in parenthesis and invoking a method on the right-hand-side of the equals sign:

<button (click)="saveChanges()">Save Changes</button>

If you need to pass the event object to your event listener, Angular provides an implicit $event variable that can be used inside the function call:

<button (click)="saveChanges($event)">Save Changes</button>

# Styles

When you need to style a component, there are two optional properties that you can configure inside of the @Component decorator.

Similar to component templates, you can manage a component's styles in the same file as the TypeScript class, or in separate files with the styleUrls properties.

Components can optionally include a list of CSS styles that apply to that component's DOM:

```
@Component({
  selector: 'profile-pic',
  template: `<img src="profile-photo.jpg" alt="Your profile photo" />`,
  styles: [
    `img {
       border-radius: 50%;
     }`,
  ],
})
export class ProfilePic {
  /* Your code goes here */
}
```

By default, a component's style will only apply to elements in that component's template in order to limit the side effects.

# Directives

When building applications, developers often need to extend on the behavior of an HTML element or Angular directives/components. Examples of this include: displaying content based on a certain condition, rendering a list of items based on application data, changing the styles on an element based on user interaction, etc.

To solve this problem, Angular uses the concept of directives, which allow you to add new behaviors to an element in a declarative and reusable way.

# Conditional rendering

One of the most common scenarios that developers encounter is the desire to show or hide content in templates based on a condition.

Similar to JavaScript's if control block, Angular provides a built-in ngIf directive to control whether an element will render if the expression returns a truthy value.

```
<section class="admin-controls" *ngIf="hasAdminPrivileges">
  The content you are looking for is here.
</section>
```

If hasAdminPrivileges is true, the application will display the content to the user, otherwise, the element is removed from the DOM entirely.

# Rendering a list

Another common scenario is to render a list of items based on dynamic data.

Similar to JavaScript’s for loop, Angular provides another built-in directive called ngFor, The following code will render one <li> element for each item in taskList.

```
<ul class="ingredient-list">
  <li *ngFor="let task of taskList">{{ task }}</li>
</ul>
```

# Custom directives

While built-in directives help to solve common problems that developers encounter, there are situations where developers require custom behavior that's specific to their application. In these cases, Angular provides a way for you to create custom directives.

Custom Angular directives can be identified by the directive suffix (e.g., my-custom-name.directive.ts).

Similar to defining a component, directives are comprised of the following:

- A TypeScript decorator to define configuration options for things like:
  - A selector that defines the tag name is when the component is called
- A TypeScript class that defines the extended behavior the directive will add to the respective HTML element.

For example, here's a custom directive for highlighting an element:

```
@Directive({
  selector: '[appHighlight]',
})
export class HighlightDirective {
  private el = inject(ElementRef);
  constructor() {
    this.el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

To apply this to an element, the directive is called by adding it as an attribute.

```
<p appHighlight>Look at me!</p>
```

# Services

When you need to share logic between components, Angular allows you to create a “service” which allows you to inject code into components while managing it from a single source of truth.

Angular services can be identified by the service suffix (e.g., my-custom-name.service.ts).

Similar to defining a component, services are comprised of the following:

A TypeScript decorator to define configuration options for things like:

providedIn - This allows you to define what parts of the application can access the service. For example, ‘root’ will allow a service to be accessed anywhere within the application.

A TypeScript class that defines the desired code that will be accessible when the service is injected

# Example of a Calculator service

```
import {Injectable} from '@angular/core';
@Injectable({
  providedIn: 'root',
})
class CalculatorService {
  add(x: number, y: number) {
    return x + y;
  }
}
```

If we wanted to call the service in a Receipt component for example, here's what it might look like:

```
import { Component } from '@angular/core';
import { CalculatorService } from './calculator.service';
@Component({
  selector: 'app-receipt',
  template: `<h1>The total is {{ totalCost }}</h1>`,
})
export class Receipt {
  private calculatorService = inject(CalculatorService);
  totalCost = this.calculatorService.add(50, 25);
}
```

In this example, the CalculatorService is being used by calling the Angular function inject and passing in the service to it.

# Organization

Standalone components are a new organizational pattern that were introduced in Angular v15 and is the recommended place to start. In contrast to NgModules, it allows developers to organize code and manage dependencies through components rather than feature modules.

For example, in the traditional NgModule pattern, you would need to create a TodoModule and manage all of its dependencies through this module.

```
import {NgModule} from '@angular/core';
import {FormsModule} from '@angular/forms';
import {TodoList} from '../todo/todo-list.component';

@NgModule({
  declarations: [TodoList],
  imports: [FormsModule],
  exports: [TodoList, FormsModule],
})
export class TodoModule {}
```

# Standalone components

```
import {FormsModule} from '@angular/forms';
import {TodoList} from '../todo/todo-list.component';

@Component({
  standalone: true,
  selector: 'todo-app',
  imports: [FormsModule, TodoList],
  template: ` ... <todo-list [tasks]="taskList"></todo-list> `,
})
export class PhotoGalleryComponent {
  // component logic
}
```

While most of this should be familiar (from the Components section), two things that are unique to this new pattern are the standalone flag and the imports key.

- standalone - When provided the value true, this tells Angular that the component does not need to be declared in an NgModule
- imports - Allows developers to declare what dependencies will be used in the component

In other words, rather than having to define a specific context in which code should be organized, developers are able to specify the dependencies directly within the component context itself.

# Questions?


## Trainer Name

Trainer

trainer@mail.com

## Assistant Name

Assistant

assistant@mail.com


Qinshift Academy