
# Automation Testing

# Angular Components

# A Day in the Life of an Automation Test Engineer

Samuel now understands how to set up an Angular development environment .

He has now decided to build an Angular project. He needs to understand Components and Core directives and build them.

To achieve the above, he will learn a few concepts in this lesson that can help him to find a solution for the scenario.

# Learning Objectives

By the end of this lesson, you will be able to:

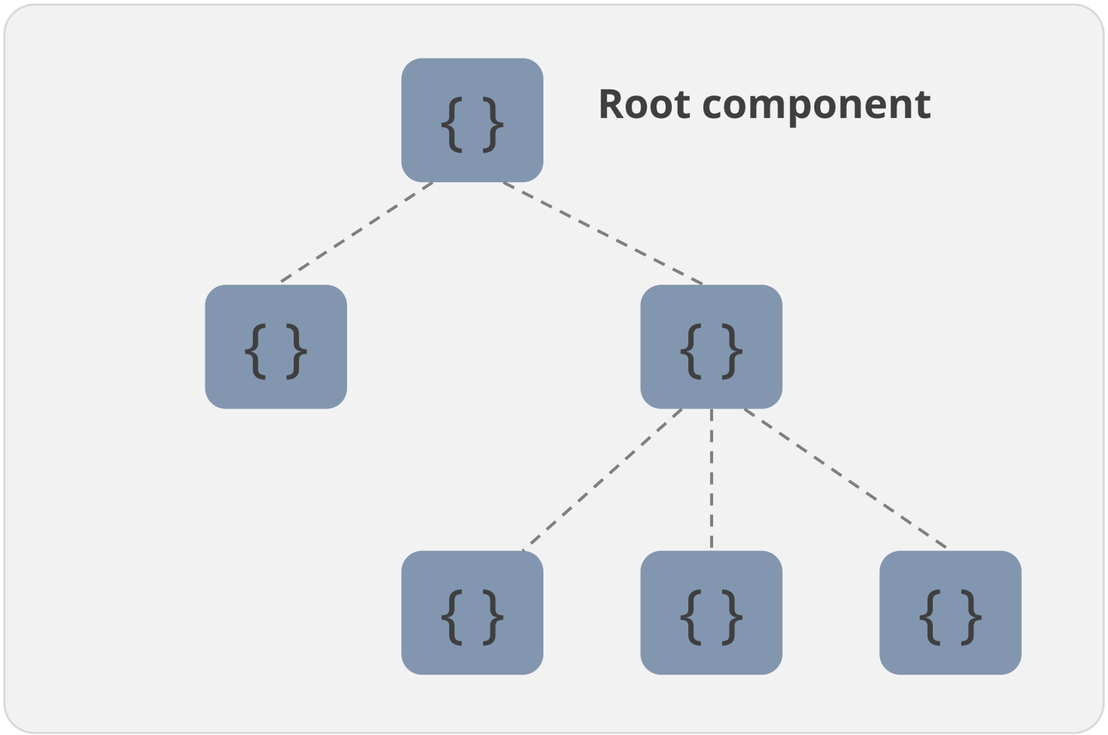
- Understand Angular Components
- List the important features of Angular Components
- Describe Angular Directives

# Components

# Components

{ }

Components in an Angular application encapsulate the template, data, and behavior of view.

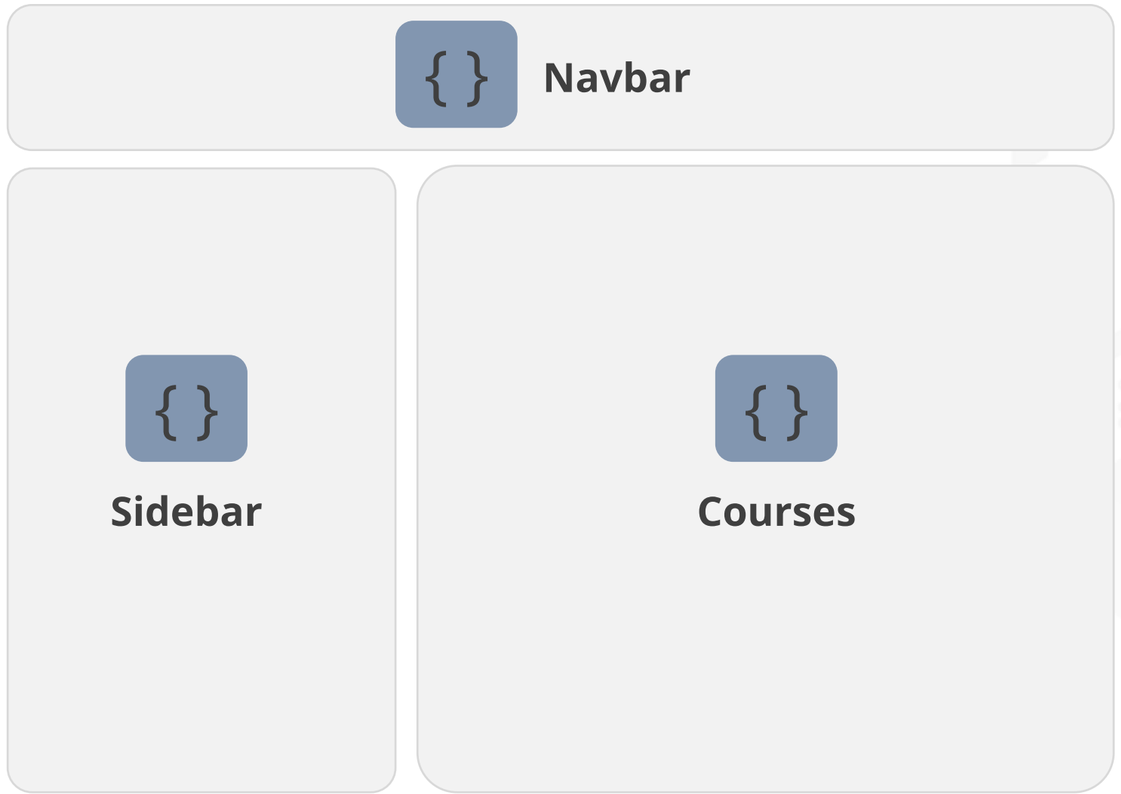



Every app has at least one component called the root component.

# Components

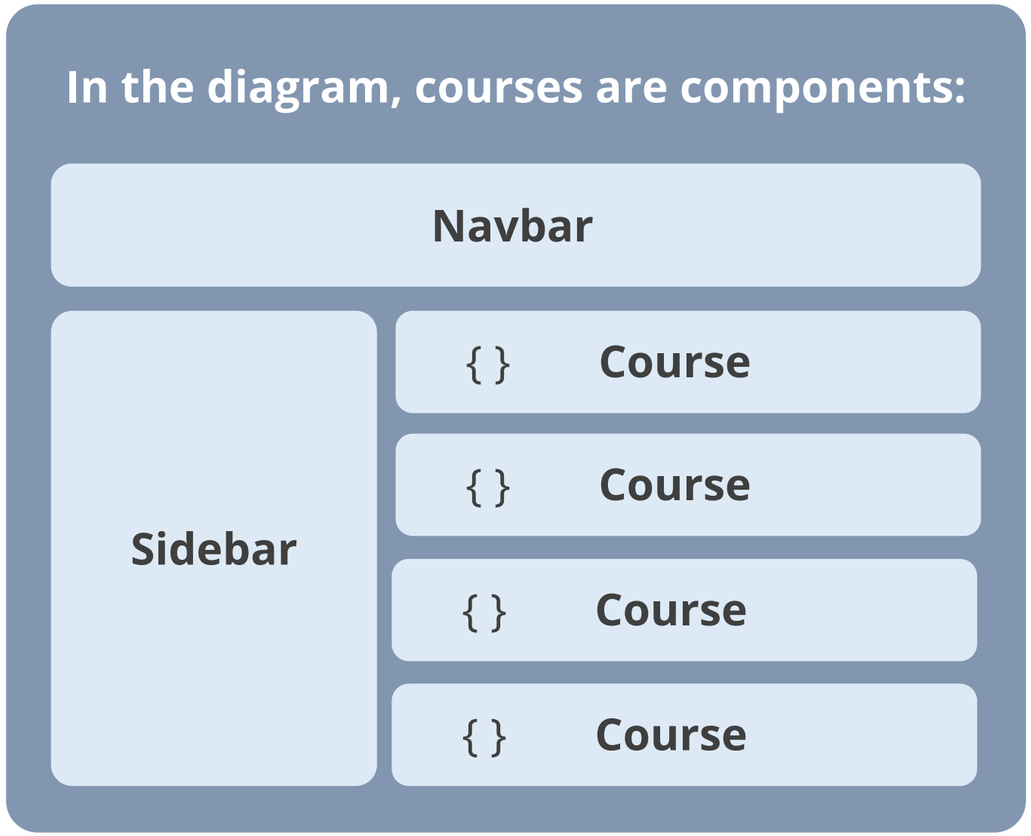
However, in real world, an application encapsulates many components.

**Here is an example:**



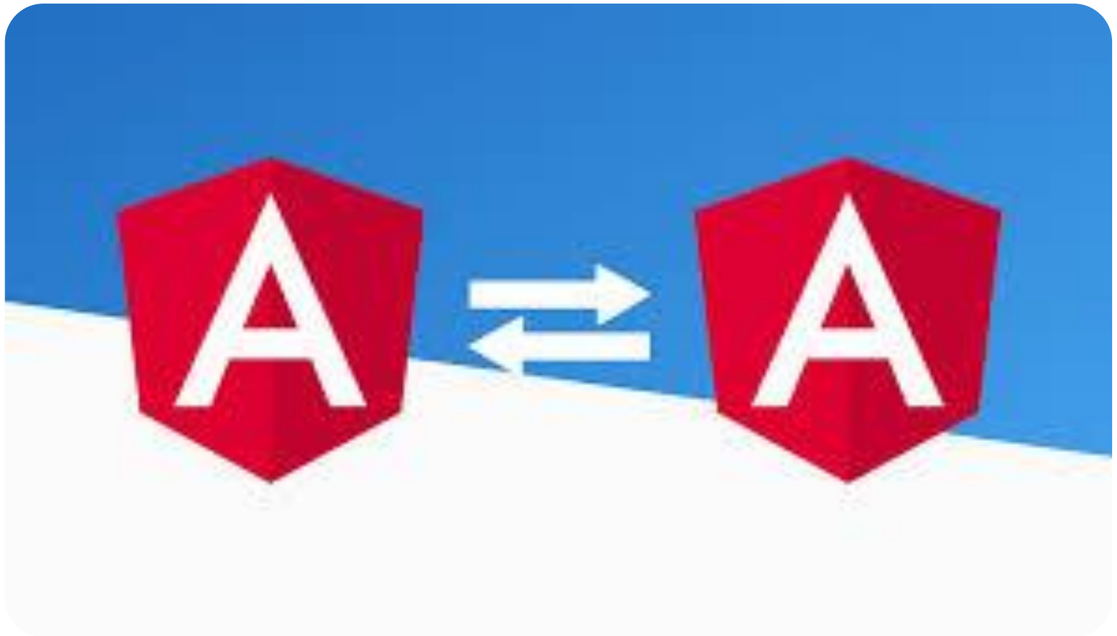
# Nested Components

Each component has a template for the view and data or logic behind the view. Components can also contain other components called nested components.



# Component-Level Interactions

Component-level interactions can be done between two components at a similar branch of hierarchy or a parent-child level interaction.



# Component-Level Interactions

The list of data shared among Angular components are:

- Parent to Child: via Input
- Child to Parent: via Output() and EventEmitter
- Child to Parent: via ViewChild
- Unrelated Components: via a Service

# @Input and @Output

@Input and @Output are mechanisms to receive and send data from one component to another, respectively.

```
@Component({
Selector: 'App-items'
…..
})

export class addTask{
 @Input() item_name
@Output() onNameChanger = new
EventEmitter()
}
```

**Example:**

- The **@Input** decorator indicates that a data will be received from a component. The received data will be stored in **item_name**.
- The **@Output** decorator indicates that a data will be sent to another component via **onNameChanger** property.

# Create and Navigate between Components in Angular

**Problem Statement:**

You are required to create and navigate between components in Angular.

ASSISTED PRACTICE

# Assisted Practice: Guidelines

Steps to create and navigate between components in Angular are:

1. Create and navigate between components in Angular

# Component Interaction to Pass Data

**Problem Statement:**

You are required to work with component interaction to pass data.

ASSISTED PRACTICE

# Assisted Practice: Guidelines

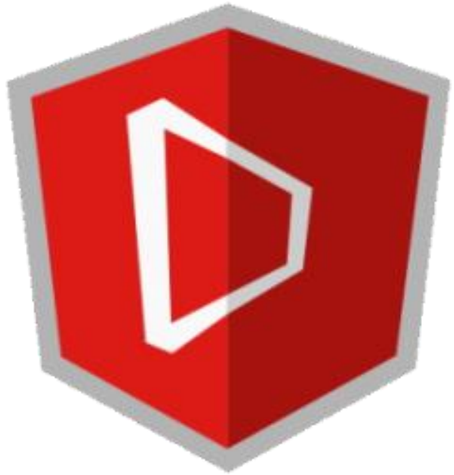
Steps to work with component interaction to pass data are:

1. Work with component interaction to pass data

# Directives

# What Are Directives?

The reusable methods and features of Angular are separated using directives.



**Note:**

Unlike components, directives do not have HTML templates.

They add behavior to an existing DOM element.

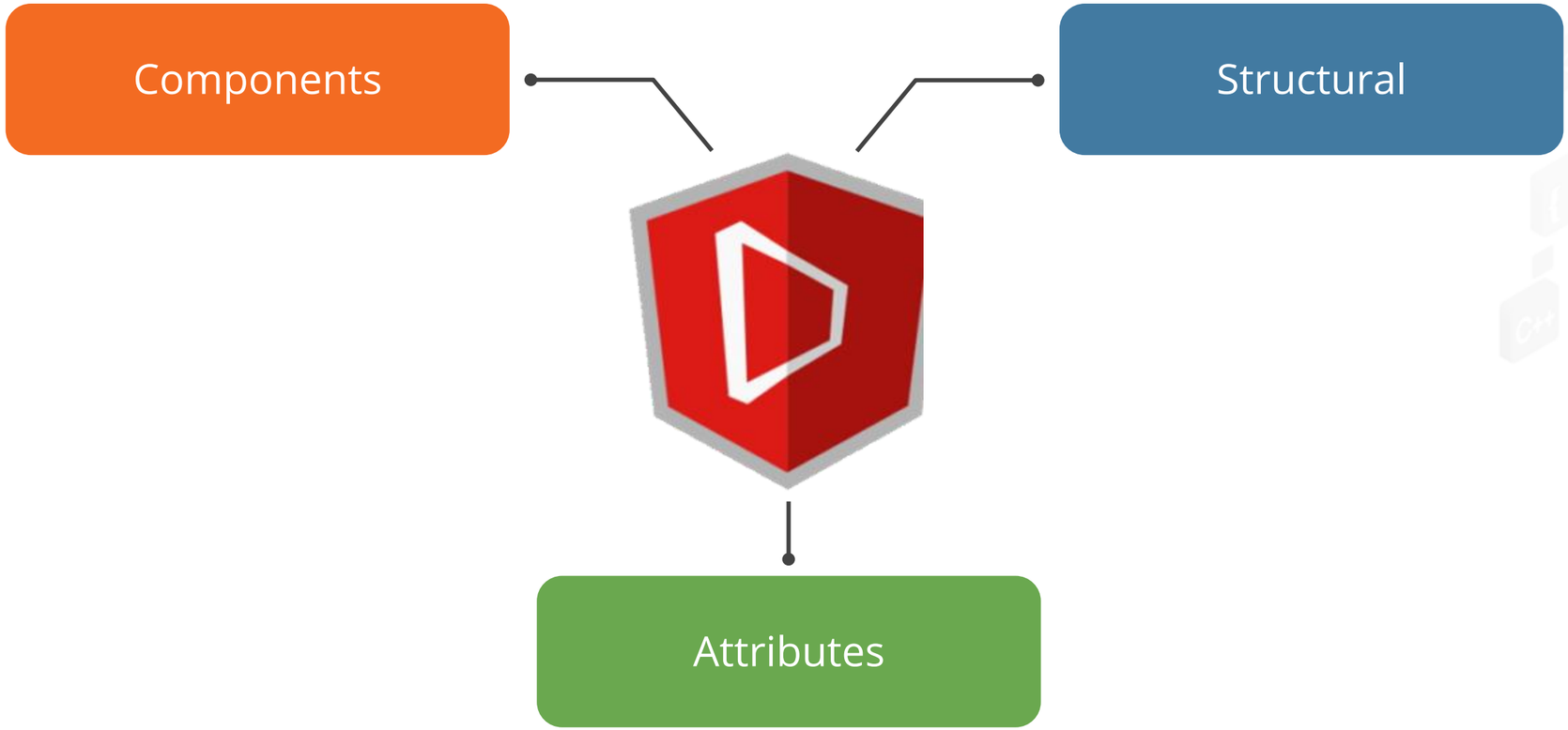
# Directives: Example

The example below is used to display the added behavior to an existing DOM element where the textbox size is enlarged on a mouseover event.

```
<input type="text" autoGrow />
```

# Directives Types

There are three types of directives:



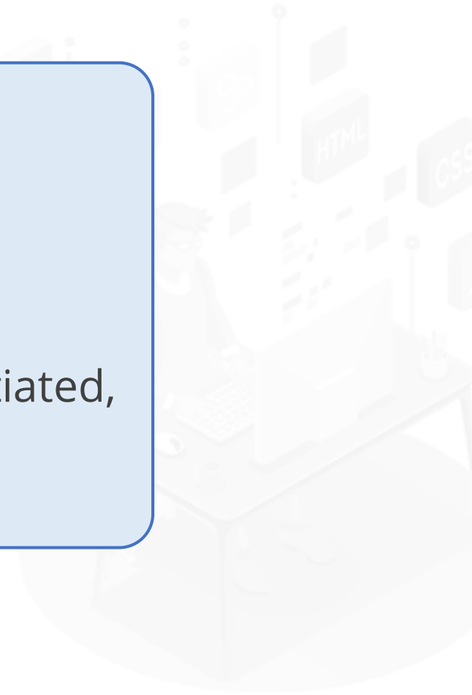
# Directives Types

Components

Structural Directives

Attributes Directives

- The component directive includes directives with the template.
- It contains information about how components are processed, instantiated, and used during runtime.



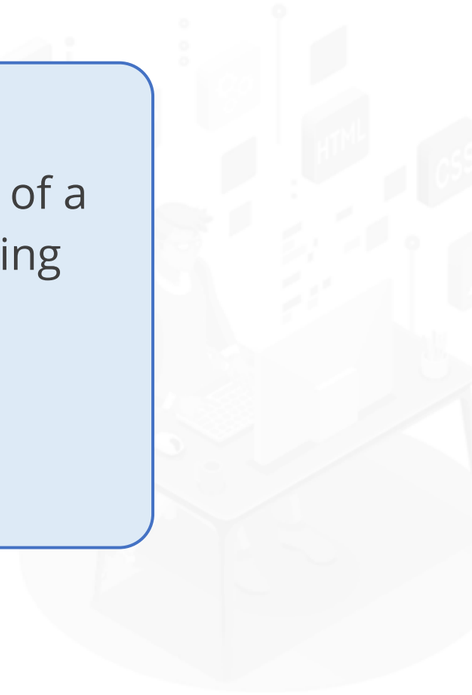
# Directives Types

Components

Structural Directives

Attributes Directives

- This directive changes the behavior of a component or an element by affecting how the template is rendered.
- It manipulates the DOM elements.



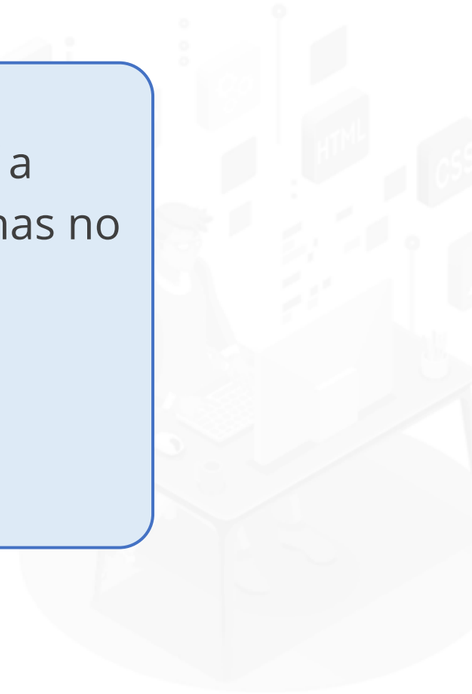
# Directives Types

Components

Structural Directives

Attributes Directives

- This directive alters the behavior of a calling component or element but has no impact on the template.
- It gives us the option of creating a directive.



# Component Directive: Example

This is how components look:

```
app.component.ts
      import {Component} from '@angular/core';
 @Component({
            selector: 'my-app',
            template: `
                  <h1>Angular demo Boilerplate</h1>
                  <p>Hello World batch good afternoon friends!!</p>
                  <rb-mycomp></rb-mycomp>

            `,
 })
      export class AppComponent{

      }
```

# Attribute Directive: Example

This is how the attribute looks:

```
import { Directive, ElementRef, Input } from '@angular/core';

@Directive({ selector: '[ColorChange]' })

export class ColorChangeDirective {
constructor(er: ElementRef) {
 er.nativeElement.style.backgroundColor = ‘green';
}
}
```

# Structural Directives

Structural directives handle how a component or element renders through the use of the template tag.

Angular has a few **built-in** structural directives:

# ngFor Directive

The core directive ngFor allows users to build data presentation lists and tables in HTML templates. With ngFor, users can print data to the screen as a data table by generating HTML.

**Example:**

```
<div *ngFor="let item of usernames">
<div *appMydirective="item">
{{item.name}}
</div>
</div>
```

# Angular ngIf Directive

- ngIf is the simplest structural directive and the easiest to understand.
- It takes a Boolean expression and makes an entire chunk of the DOM appear or disappear.
- The ngIf directive doesn't hide elements through CSS. It adds and removes them physically from the DOM.

**Example:**

```
<app-if-example *ngIf="exists">
```

# Angular ngClass Directive


- The ngClass directive modifies the class attribute of a component or an associated element.
- It evaluates the expression and modifies the element's class attribute.
- It is an Angular attribute directive that lets users to add or delete a CSS class from an HTML element.

# Angular ngStyle Directive

The ngStyle directive specifies the HTML element's style attribute. Its attribute's value must be an object or an expression that returns an object.

[ngStyle]
in Angular



CSS attributes and values are organized into key value pairs in the object.

# Angular ngSwitch Directive

The Angular ngSwitch is a set of cooperating directives: ngSwitch, ngSwitchCase, and ngSwitchDefault. It is comprised of two directives: an attribute directive and a structural directive.

**Example:**

```
<div [ngSwitch]="tab">

<app-tab-content *ngSwitchCase="1">content 1</app-
tab-content>

<app-tab-content *ngSwitchCase="2"> content 2</app-
tab-content>

</div>
```

# Using ngFor and nglf

**Problem Statement:**

You are asked to use ngFor and nglf directive.

ASSISTED PRACTICE

# Assisted Practice: Guidelines

Steps to use ngFor and nglf directive are:

1. Use ngFor and nglf directive

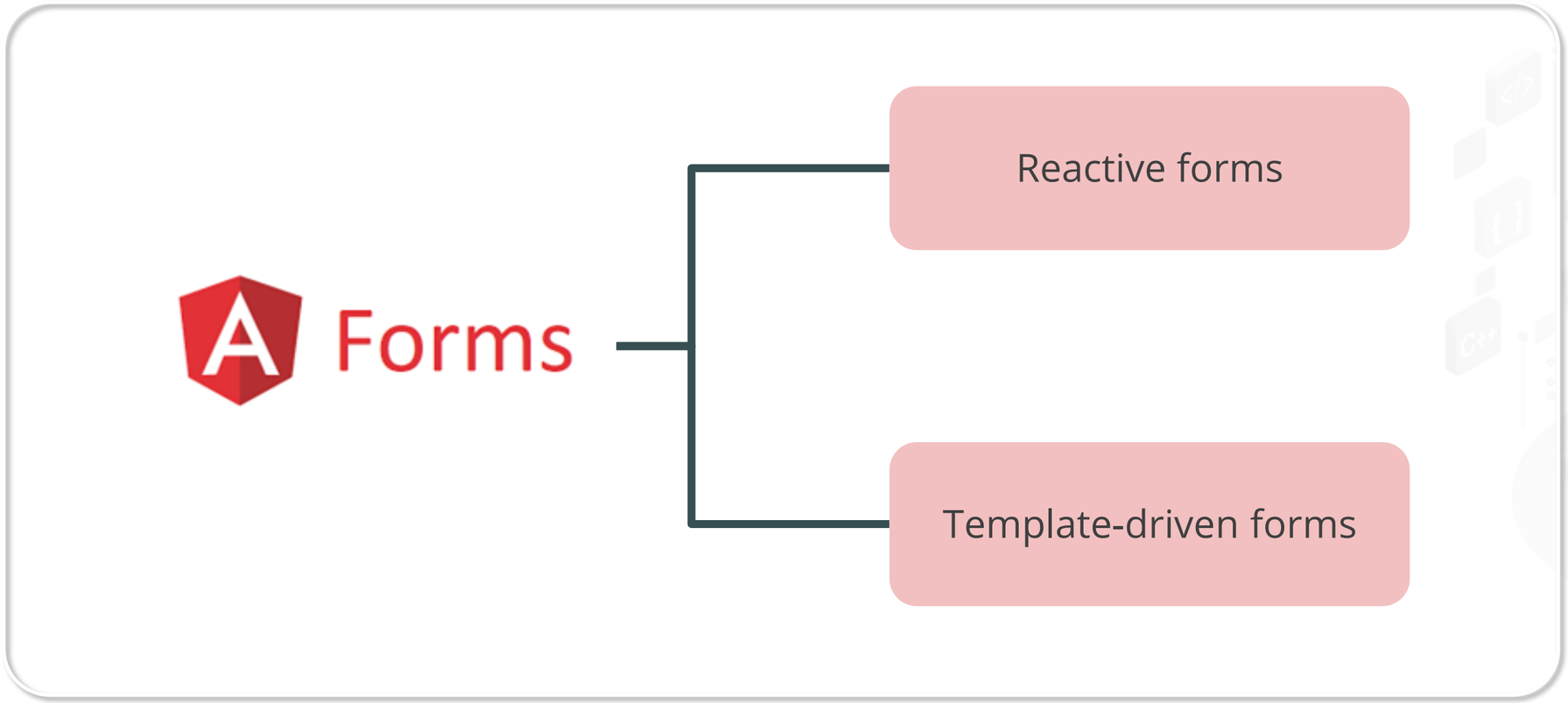
# Forms

# Forms in Angular

Applications in Angular use forms to allow users to log in, update profiles, enter sensitive information, and do a variety of other data-entry tasks.

# Types of Forms

There are two types of forms in Angular:



# Types of Forms

## Reactive forms

- Reactive forms use a model-based technique to deal with form inputs that change over time.
- Form inputs and values are presented as streams of input values that can be retrieved synchronously in reactive forms, which are designed on observable streams.

# Types of Forms

## Template-driven forms

- Template-driven forms use two-way data binding to update the component's data model as the template changes, and vice versa.
- Small or simple forms can benefit from template-driven forms.

# Reactive Forms vs. Template-Driven Forms

These are the differences between reactive forms and template-driven forms:

| Criteria | Reactive forms | Template-driven forms |
|---|---|---|
| Data model | Structured | Unstructured |
| Data flow | Synchronous | Asynchronous |
| Form validation | Functions | Directives |

# Reactive Forms vs. Template-Driven Forms

These are the differences between reactive forms and template-driven forms:

| Criteria | Reactive forms | Template-driven forms |
|---|---|---|
| Setup (form model) | Reactive forms are more explicit. They are created in component class. | Template-driven forms are less explicit. They are created by directives. |
| Mutability | Immutable | Mutable |
| Scalability | Low-level API access | Abstraction on top of APIs |

# Creating Reactive Form with FormControl

**Problem Statement:**

You have been asked to create reactive form with Formcontrol.

ASSISTED PRACTICE

# Assisted Practice: Guidelines

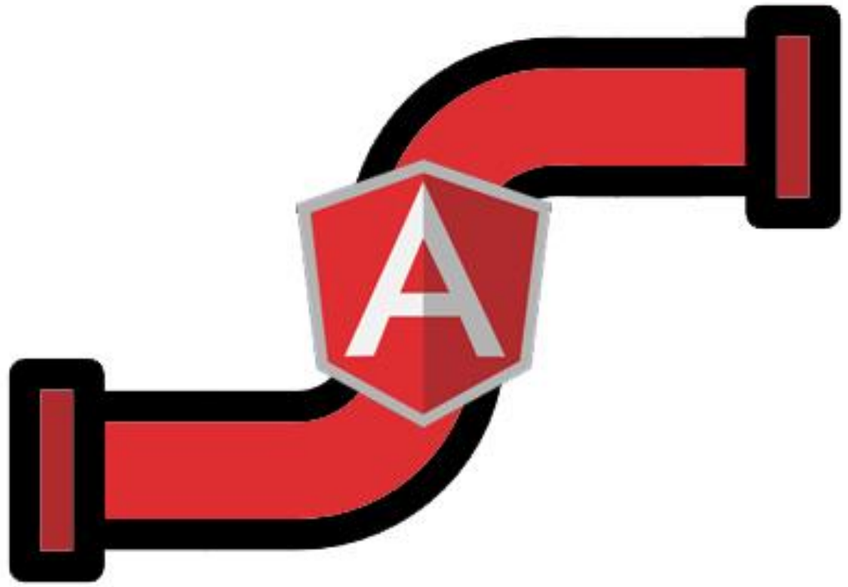
Steps to create reactive form with Formcontrol are:

1. Create reactive form with Formcontrol

# Pipes

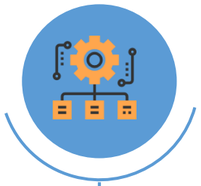
# Pipes in Angular

Pipes are a quick and easy way to change values in an Angular template. A pipe accepts one or more values and then returns one.



# Uses of Pipes

Pipes do not provide any additional features, however:



Pipes make the code structured and clear.

They are helpful because users can use them anywhere in their program and declare them once.

They are a good way to deal with functions and logics in templates.

They take in data as input and transforms it into the desired output.

# Built-in Pipes

# Built-in Pipes

Built-in pipes can be used to format values such as date, number, decimal, percentage, currency, uppercase, and lowercase. These pipes are defined in @angular or common package.

# Types of Built-in Pipes

Some built-in pipes in Angular are:

| Pipe Name | Description |
|---|---|
| Currency | Formats the currencies |
| UpperCase | Converts a text or string to uppercase |
| LowerCase | Converts a text or string to uppercase |
| Decimal | Transforms the decimal numbers |
| Json | Converts an object into a JSON string |

# Key Takeaways

- Pipes are the way to write display-value transformations that users can declare in HTML.
- The reusable methods and features of Angular are separated using directives.
- There are two types of forms in Angular: reactive forms and template-driven forms.
- Built-in pipes can be used to format values such as date, number, decimal, percentage, currency, uppercase, and lowercase.