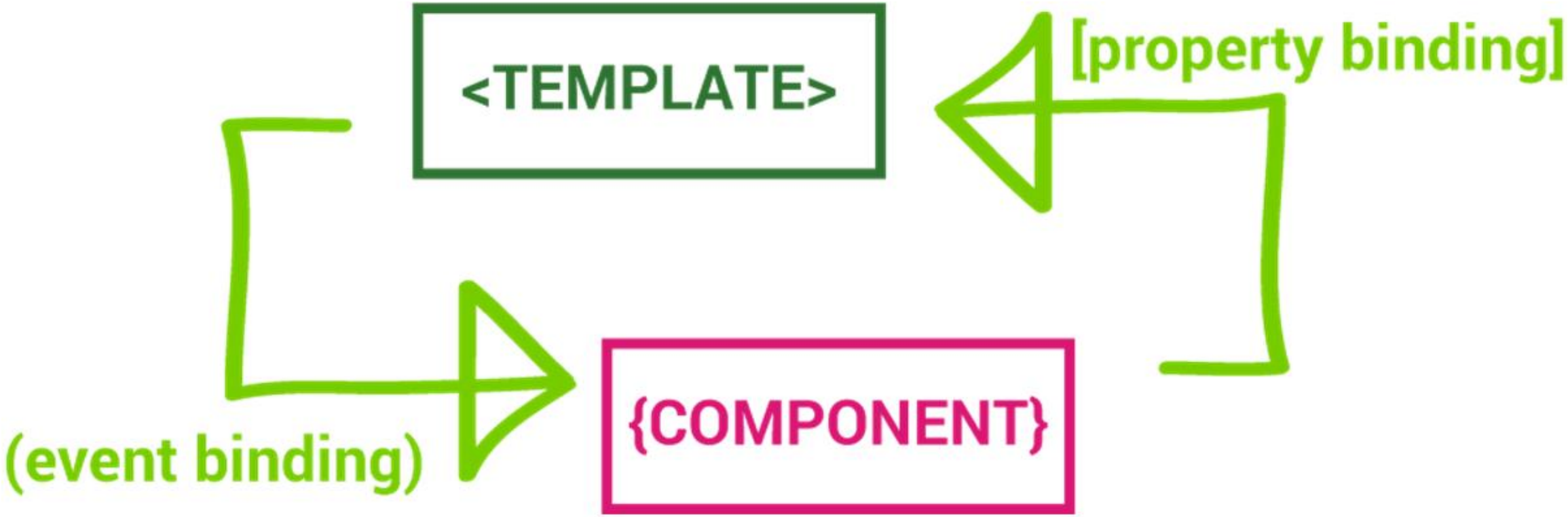
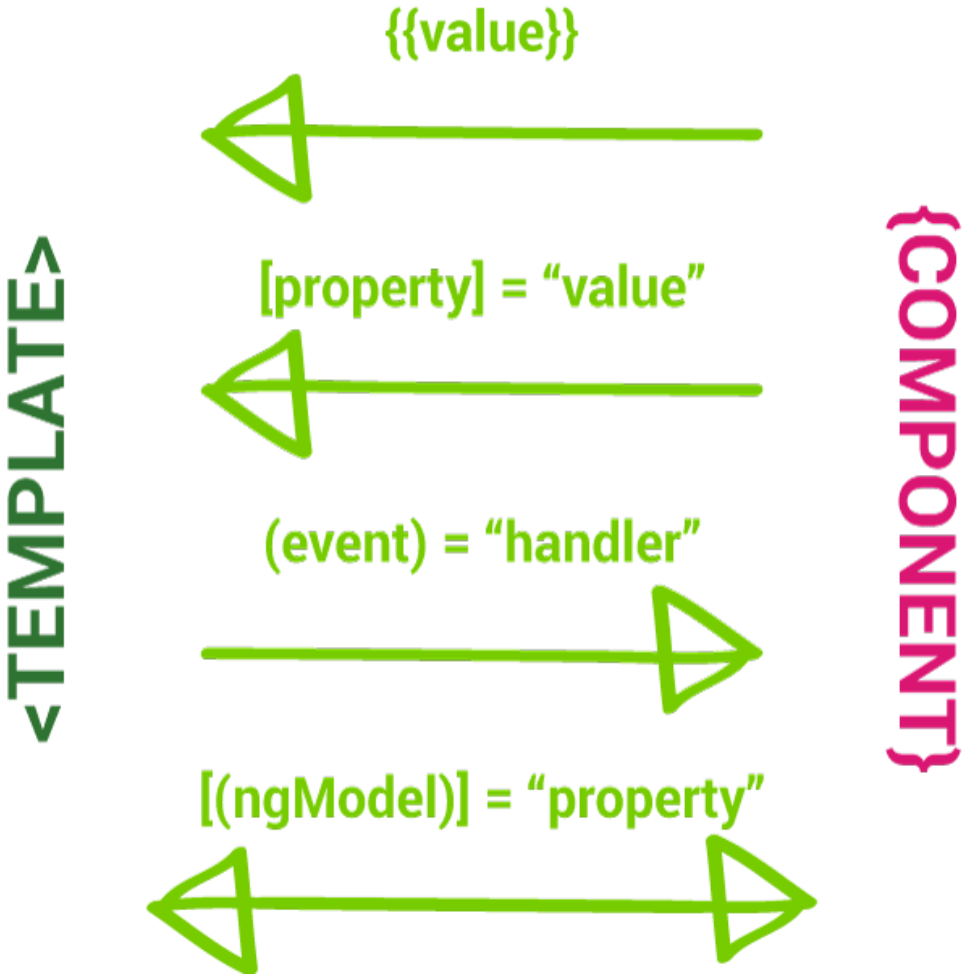
ANGULAR DATA BINDING


<TEMPLATE>
[property binding]
(event binding)
{COMPONENT}

DATA BINDING

- Enables data to flow from the component to template and vice-versa
- Includes interpolation, property binding, event binding, and two-way binding (property binding and event binding combined)
- The binding syntax has expanded but the result is a much smaller framework footprint



INTERPOLATION

- **Interpolation (One-way binding from component to view)**:
- Syntax: {{ expression }}
- Variable Declaration in ts file

```
export class AppComponent {
  title = 'Angular 15 Data Binding Example';
}
```

- Use in HTML file

```
<h1>{{ title }}</h1>
```

PROPERTY BINDING

- **Property Binding (One-way binding from component to view)**:
- This binds a component property to an HTML element property.
- Syntax: [property]="expression“

```
export class AppComponent {
  imageUrl = 'https://example.com/image.jpg';
}

<img [src]="imageUrl" />
```

EVENT BINDING

- **Event Binding (One-way binding from view to component)**:
- This binds an event in the template (like click) to a method in the component.
- Syntax: (event)="method()“

```
export class AppComponent {
  handleClick() {
    alert('Button clicked!');
  }
}

<button (click)="handleClick()">Click Me</button>
```

TWO WAY BINDING

- This binds both the data and the event together.
- The most common use case is with form inputs.
- **Syntax: [(ngModel)]="property“**
- Make sure you import **FormsModule** for two-way binding using **ngModel**.

```
export class AppComponent {
  username = '';
}

<input [(ngModel)]="username" />
<p>{{ username }}</p>
```

CLASS BINDING

Class binding is used to add or remove classes from an HTML element dynamically.

We can implement it in 2 ways:

- Using the class attribute for single class binding.
- Using ngClass for multiple class bindings.

SINGLE CLASS BINDING:

```
<button [class.active]="isActive">Click Me</button>

export class AppComponent {
  isActive = true;
}

<button [ngClass]="{ 'active': isActive, 'disabled': isDisabled }">Click Me</button>
```

(Multi Class Binding)

STYLE BINDING

```
<button [style.background-color]="isActive ? 'green' : 'red'">Click Me</button>

export class AppComponent {
  isActive = true;
}

<button [ngStyle]="{ 'background-color': isActive ? 'green' : 'red', 'font-size':
fontSize }">Click Me</button>

export class AppComponent {
  isActive = true;
  fontSize = '20px';  // Button will have 20px font size
}
```

DIRECTIVES

In Angular, **directives** are used to extend HTML by adding custom behavior to elements in your Angular applications.

3 types:

- Component Directive
- Structural Directives
- Attribute Directives

COMPONENT DIRECTIVES

- Components themselves are a type of directive.
- They are the most common directives in Angular.
- Components include a template (view) and are built with a decorator @Component.

```
@Component({
  selector: 'app-my-component',
  template: `<h1>Hello, World!</h1>`
})
export class MyComponent {}
```

- **Here, <app-my-component></app-my-component> in the HTML would invoke this component, which is a directive.**

STRUCTURAL DIRECTIVES

- Structural directives are responsible for altering the DOM structure by adding, removing, or manipulating elements.
- They are prefixed with an asterisk (*).
- 3Types:
  1. *ngIf (conditionally adds/removes an element)
  2. *ngFor (iterates over a collection of items)
  3. *ngSwitch (conditionally adds/removes elements based on multiple conditions)

*NGIF

```
<p *ngIf="isVisible">This text is visible if isVisible is true.</p>

export class AppComponent {
  isVisible = true;
}
```

*NGFOR

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>

export class AppComponent {
  items = ['Item 1', 'Item 2', 'Item 3'];
}
```

*NGSWITCH

```
<div [ngSwitch]="color">
  <p *ngSwitchCase="'red'">Red Color</p>
  <p *ngSwitchCase="'blue'">Blue Color</p>
  <p *ngSwitchDefault>Unknown Color</p>
</div>

export class AppComponent {
  color = 'red';
}
```

ATTRIBUTE DIRECTIVES

- Attribute directives are used to modify the behavior or appearance of an element.
- They do not change the structure of the DOM, but they can change the look or functionality of an element.
- Two Types:
  1. ngClass(dynamically adds/removes classes)
  2. ngStyle (dynamically adds/removes styles)

ACTIVITY

- **Employee Management System**
- Build a simple Employee Management System.
- **Objectives:**
  - Implement **two-way data binding** for capturing employee information through a form.
  - Use **property binding** to control the form's elements.
  - Implement **event binding** to handle form submissions and button clicks.
  - Use **interpolation** to display the captured employee data on the same page.

STABLE NGOPTIMIZEDIMAGE IMAGE DIRECTIVE

- The NgOptimizedImage directive is a new feature in Angular that was introduced to improve image optimization in Angular applications.
- **Automatic Optimization**: It automatically handles common image performance tasks like lazy loading, setting srcset for responsive images, and ensuring images have proper dimensions.
- **Lazy Loading**: Ensures that images are only loaded when they are visible on the screen, helping reduce initial load times.
- **Responsive Images**: It helps set up different image sizes for different screen widths, improving the experience across devices.
- **Prevents Common Mistakes**: Automatically detects and throws errors for common image-related mistakes, like missing alt attributes, incorrect width/height, or unoptimized image formats.

HOW TO USE IT?

- Let's apply the feature:

```
import { provideImgixOptimizer } from '@angular/platform-browser';

@NgModule({
  declarations: [AppComponent],
  providers: [provideImgixOptimizer()],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

LET'S USE IT

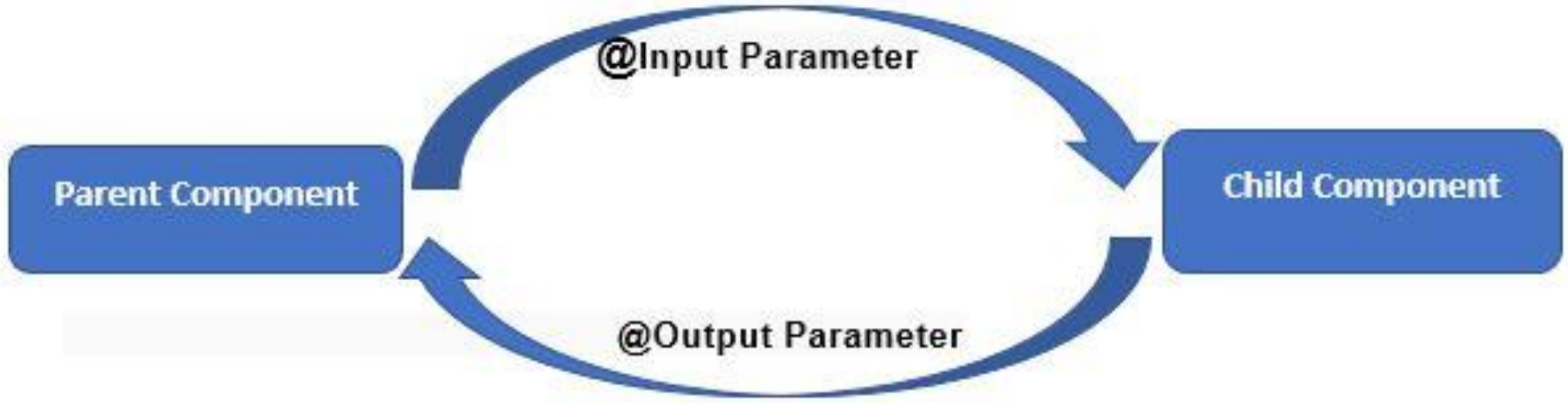
```
<img ngSrc="https://example.com/image.jpg" width="600" height="400" alt="Example Image" />
```

Automatic Lazy Loading: By default, the directive automatically lazy loads images, meaning the loading="lazy" attribute is applied.

Responsive Images:

```
<img
  ngSrc="https://example.com/image.jpg"
  ngSrcset="https://example.com/image-600w.jpg 600w, https://example.com/image-1200w.jpg 1200w"
  width="600" height="400"
  alt="Responsive Image" />
```

INTER-COMPONENT COMMUNICATION



TEMPLATE VARIABLES

- A **template variable** is a reference to a DOM element, Angular component, or directive within the template.
- It's declared using the # symbol followed by the variable name.

```
<input #inputElement type="text" />
<button (click)="logValue(inputElement.value)">Log Value</button>

export class AppComponent {
  logValue(value: string) {
    console.log(value);  // Logs the input's value
  }
}
```

VIEWCHILD

- The **@ViewChild** decorator is used to get a reference to a DOM element, directive, or child component within the same view.
- It is primarily used to interact with elements after the view has been initialized.

```
<input #inputElement type="text" />
```

ACCESS VIA VIEWCHILD

```
import { Component, ViewChild, ElementRef, AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<input #inputElement type="text" />`
})
export class AppComponent implements AfterViewInit {
  @ViewChild('inputElement') inputEl!: ElementRef;

  ngAfterViewInit() {
    // Access the input element and set focus
    this.inputEl.nativeElement.focus();
  }
}
```

VIEW CHILD TO ACCESS CHILD

```
import { Component, ViewChild } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-root',
  template: `<app-child-component></app-child-component>`
})
export class AppComponent {
  @ViewChild(ChildComponent) child!: ChildComponent;

  ngAfterViewInit() {
    console.log(this.child.someMethod());  // Call a method from the child component
  }
}
```

LIFECYCLE HOOKS

- **Lifecycle Hooks** are special methods that provide insight into key moments in the lifecycle of a component or directive.
- Angular invokes these hooks at specific points during the creation, update, and destruction of components.

NGONCHANGES

- It is called whenever data-bound input properties (@Input()) change.
- It runs Before ngOnInit() and whenever any input properties of the component change.
- Use to trigger something when there is some changes in component's input properties.

```
export class AppComponent {
  title = 'myapp';
  data = "test"
}
```

```
<input type="text" #pdata (keyup)="0" />
<app-child [data]="pdata.value"></app-child>
```

CHILD COMPONENT

```
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `<p>{{data}}</p>`
})
export class ChildComponent implements OnChanges {
  @Input() data!: string;

  ngOnChanges(changes: SimpleChanges) {
    console.log('Input changed:', changes);
  }
}
```

NGONINIT

- It is called once after the component's initial data-bound properties are set.
- It runs once, after the component is initialized and the input properties are available.
- Used to initialize data or set up logic that depends on the input properties.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'myapp';

  ngOnInit() {
    console.log('Component Initialized');
  }
}
```

NGDOCHECK

- It is used to detect and act upon changes that Angular can't or doesn't detect on its own.
- It runs after ngOnChanges() and during every change detection cycle.
- used to perform custom change detection or actions in response to changes that Angular doesn't detect automatically.

```
export class AppComponent {
  title = 'myapp';

  ngOnInit() {
    console.log('Component Initialized');
  }
  ngDoCheck() {
    console.log('Custom change detection logic');
  }
}
```

SIMILAR TO THAT

- **ngAfterContentInit** (after content is projected)
- **ngAfterContentChecked** (every time content is checked)
- **ngAfterViewInit** (after the view is initialized)
- **ngAfterViewChecked** (every time the view is checked)
- **ngOnDestroy** (before the component is destroyed)

STABLE STANDALONE COMPONENTS API

- The **Stable Standalone Components API** is a significant feature introduced in Angular 15.
- Which is providing a new way to create components, directives, and pipes without requiring them to be part of an NgModule.
- This feature simplifies Angular's modular system, making components more self-contained and reducing boilerplate code.

```
ng generate component child --standalone
```

```
imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule,
  ReactiveFormsModule,
  ChildComponent
],
```

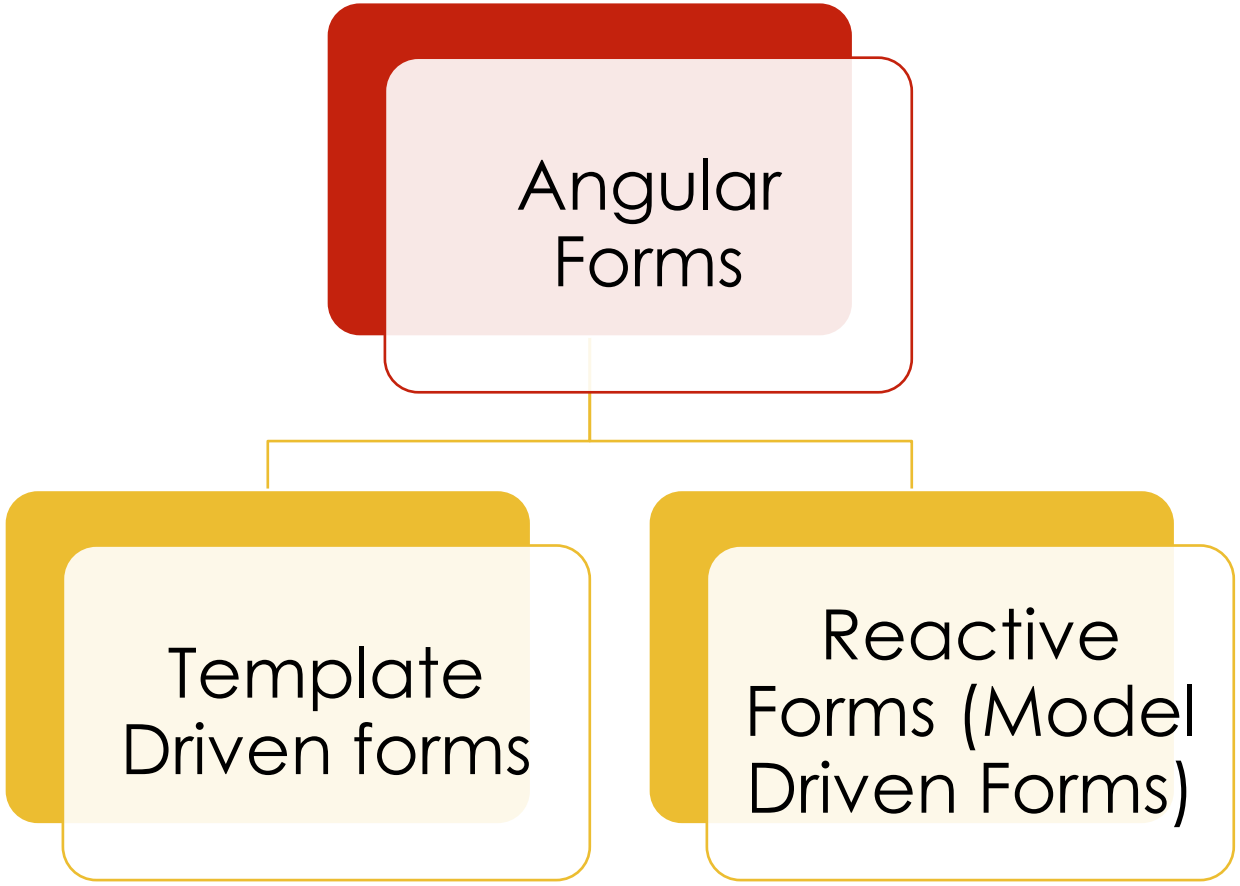
MODULE 4

- Template Driven Forms
- Reactive forms
- Form Validations
- Custom Synchronous form validations
- Custom Asynchronous form validations

ANGULAR FORMS

- A large category of frontend applications are very form-intensive, especially in the case of enterprise development.
- Many of these applications are basically just huge forms, spanning multiple tabs and dialogs and with non-trivial validation business logic.
- The Angular framework provides us a couple of alternative strategies for handling forms

ANGULAR FORM TYPE



TEMPLATE DRIVEN FORMS

- Template-driven forms use **two-way data binding** to update the data model in the component as changes are made in the template and vice versa.
- You can build almost any kind of form with an Angular template
  - **login forms**
  - **contact forms**
  - **any business form.**
- You can lay out the controls creatively and bind them to the data in your object model.
- You can specify validation rules and display validation errors, conditionally allow input from specific controls, trigger built-in visual feedback, and much more.

ADVANTAGES

- Handling the forms is as simple as reactive forms.
- Template-driven forms are easy to use.
- They are similar to AngularJs.
- Two-way data binding (using [(NgModel)] syntax).
- Minimal component code.
- Template-driven form is also more than enough to build a large range of forms.

HOW TO BUILD A TEMPLATE DRIVEN FORM

- Use **FormsModule** to work with template Driven Forms.
- Directive used:

1. **NgModel:** allowing you to respond to user input with input validation and error handling.

2. **Ng Form:** use with <form> tag.

3. **NgModelGroup:** Create & bind FormGroup instance to a DOM element

TEMPLATE DRIVEN FORM

```
export class RegisterComponent {

  user = {
    name: '',
    email: '',
    password: ''
  };

  onSubmit(form: any) {
    if (form.valid) {
      console.log('User:', this.user);
    }
  }
}
```

```
<h2>Registration Form</h2>
<form #registerForm="ngForm" (ngSubmit)="onSubmit(registerForm)">
    <div>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" [(ngModel)]="user.name" required #name="ngModel">
        <!-- Name field validation -->
        <div *ngIf="name.invalid && name.touched">
            <div *ngIf="name.errors && name.errors['required']">Name is required</div>
        </div>
    </div>

    <div>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" [(ngModel)]="user.email" required email #email="ngModel">
        <!-- Email field validation -->
        <div *ngIf="email.invalid && email.touched">
            <div *ngIf="email.errors && email.errors['required']">Email is required</div>
            <div *ngIf="email.errors && email.errors['email']">Invalid email format</div>
        </div>
    </div>
```

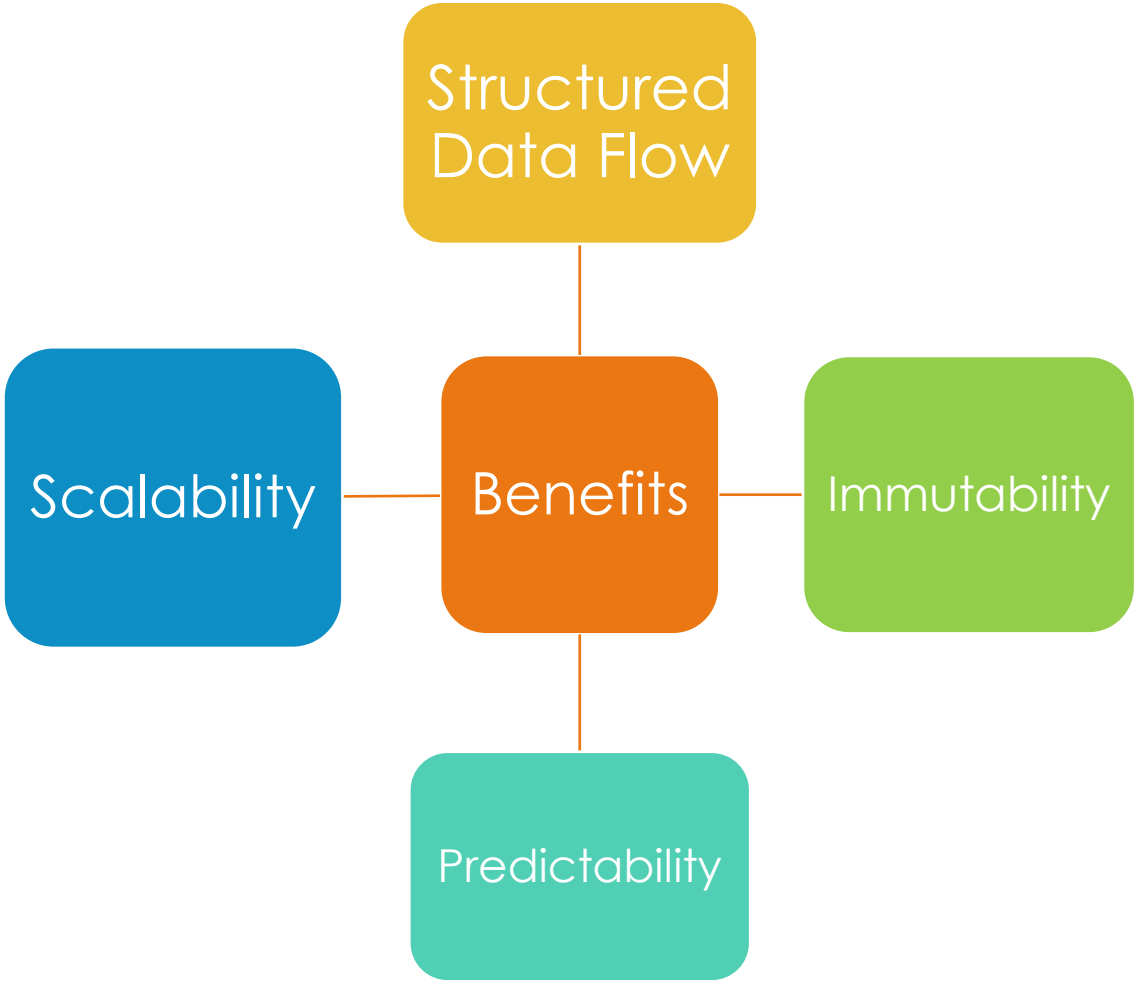
```
  <div>
      <label for="password">Password:</label>
      <input type="password" id="password" name="password" [(ngModel)]="user.password" required minlength="6"
          #password="ngModel">
      <!-- Password field validation -->
      <div *ngIf="password.invalid && password.touched">
          <div *ngIf="password.errors && password.errors['required']">Password is required</div>
          <div *ngIf="password.errors && password.errors['minlength']">
              Password must be at least 6 characters long
          </div>
      </div>
  </div>

  <button type="submit" [disabled]="registerForm.invalid">Register</button>
/form>
```

REACTIVE FORM

- Reactive forms uses a model-driven approach to handling form inputs.
- Each change to the form state returns a new state, which maintains the integrity of the model between changes.
- Reactive forms are built around observable streams, where form inputs and values are provided as streams of input values, which can be accessed synchronously.

BENEFITS OF USING REACTIVE FORMS



REACTIVE FORM

```
export class RegisterComponent {
  registerForm:FormGroup | any;
  submitted:boolean=false;
  //dependency injection
  constructor(private builder:FormBuilder){}
  ngOnInit(){
    this.registerForm= this.builder.group({
      firstname:['',Validators.required],
      lastname:['',Validators.required],
      email:['',[Validators.required,Validators.email]],
      gender: ['', [Validators.required]],
      password:['',[Validators.required,Validators.minLength(8)]]
    })
  }
```

REACTIVE FORM

```
  get form(){
    return this.registerForm.controls;
  }
  onSubmit(){
    this.submitted=true;
    if(!this.registerForm.valid)
      return;
    alert("Form Submitted for Approval")
    console.log(this.registerForm.value);

  }
}
```

```
<div class="container"></div>
<h2>Registration Form</h2>
<div class="row">
    <div class="col-md-6 offset-3">
        <form [formGroup]="registerForm" (ngSubmit)="onSubmit()" novalidate>
            <div class="form-group">
                <label>FirstName</label>
                <input type="text" formControlName="firstname" class="form-control"
                    [ngClass]="{'is-invalid':submitted && form['firstname'].errors}">

                <div *ngIf="submitted && form['firstname'].errors" class="invalid-feedback">
                    FirstName cannot be Empty
                </div>
            </div>
            <div class="form-group">
                <label>LastName</label>
                <input type="text" formControlName="lastname" class="form-control"
                    [ngClass]="{'is-invalid':submitted && form['lastname'].errors}">

                <div *ngIf="submitted && form['lastname'].errors" class="invalid-feedback">
                    LastName cannot be Empty
                </div>
            </div>
```

```
<div class="form-group">
    <label>Email</label>
    <input type="text" formControlName="email" class="form-control"
        [ngClass]="{'is-invalid':submitted && form['email'].errors}">

    <div *ngIf="submitted && form['email'].errors" class="invalid-feedback">
        <div *ngIf="form['email'].errors['required']">
            Email cannot be Empty
        </div>
        <div *ngIf="form['email'].errors['email']">
            Invalid EmailID
        </div>
    </div>
</div>
```

```
<div class="d-block my-3">
    <label>gender</label>
    <div class="custom-control custom-radio">
        <input id="male" type="radio" class="custom-control-input" value="male" name="gender"
            formControlName="gender">
        <label class="custom-control-label" for="male">Male</label>
    </div>
    <div class="custom-control custom-radio">
        <input id="female" type="radio" class="custom-control-input" value="female" name="gender"
            formControlName="gender">
        <label class="custom-control-label" for="female">Female</label>
    </div>
    <div class="invalid-feedback-polyfill" *ngIf="submitted && form['gender'].errors?.['required']">
        <p>Please select either value</p>
    </div>
</div>
```

```
</div>
<div class="form-group">
    <label>Password</label>
    <input type="password" formControlName="password" class="form-control"
        [ngClass]="{'is-invalid':submitted && form['password'].errors}">

    <div *ngIf="submitted && form['password'].errors" class="invalid-feedback">
        <div *ngIf="form['password'].errors['required']">
            Password cannot be Empty
        </div>
        <div *ngIf="form['password'].errors['minlength']">
            Password must be atleast 8 character long
        </div>
    </div>
</div>
```

CREATE CUSTOM VALIDATION

```
import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';

// Custom Validator Function: Username should not contain 'admin'
export function forbiddenNameValidator(forbiddenName: string): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const forbidden = control.value.includes(forbiddenName);
    return forbidden ? { forbiddenName: { value: control.value } } : null;
  };
}
```

CREATE CUSTOM VALIDATION

```
export class RegisterComponent {
  submitted:boolean=false;
  registrationForm: FormGroup;

  constructor(private fb: FormBuilder) {
    this.registrationForm = this.fb.group({
      username: ['', [Validators.required, forbiddenNameValidator('admin')]],
      password: ['', [Validators.required, Validators.minLength(6)]]
    });
  }

  onSubmit() {
    console.log(this.registrationForm.value);
  }

}
```

CREATE CUSTOM VALIDATION

```
<form [formGroup]="registrationForm" (ngSubmit)="onSubmit()">
    <div>
      <label for="username">Username</label>
      <input id="username" formControlName="username">
      <div *ngIf="registrationForm.get('username')?.hasError('forbiddenName')">
        Username cannot contain 'admin'.
      </div>
    </div>

    <div>
      <label for="password">Password</label>
      <input id="password" type="password" formControlName="password">
    </div>

    <button type="submit" [disabled]="registrationForm.invalid">Register</button>
  </form>
```