Angular part - 4

Angular provides two main types of forms:

Reactive Forms: These forms are controlled primarily through the component class. They are scalable, reusable, and ideal for complex scenarios with many form controls or logic-driven requirements.

Template-driven Forms: These forms are set up and managed directly within the HTML template. They work well for simpler forms where validation and state management are minimal.

Reactive forms good for ——————

Provide direct, explicit access to the underlying form's object model.

Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

Template forms good for ——-

Rely on directives in the template to create and manipulate the underlying object model. They are useful for adding a simple form to an app, such as an email list signup form. They're straightforward to add to an app, but they don't scale as well as reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, template-driven forms could be a good fit.

———————

If forms are a central part of your application, scalability is very important. Being able to reuse form models across components is critical.

Reactive forms are more scalable than template-driven forms. They provide direct access to the underlying form API, and use synchronous data flow between the view and the data model, which makes creating large-scale forms easier. Reactive forms require less setup for testing, and testing does not require deep understanding of change detection to properly test form updates and validation.

Template-driven forms focus on simple scenarios and are not as reusable. They abstract away the underlying form API, and use asynchronous data flow between the view and the data model. The abstraction of template-driven forms also affects testing. Tests are deeply reliant on manual change detection execution to run properly, and require more setup.

————————————

Both reactive and template-driven forms track value changes between the form input elements that users interact with and the form data in your component model. The two approaches share underlying building blocks, but differ in how you create and manage the common form-control instances.

FormControl: Tracks the value and validation status of a single form element.
FormGroup: Manages multiple FormControls as a group, ideal for grouping related controls.
FormArray: Manages an array of FormControls, useful for dynamic lists of inputs.
ControlValueAccessor: Creates a bridge between a form control and the DOM element.

Basic Reactive form

```
<h3>Favorite COlor: (Reactive FOrm)</h3>

<div>
  <input type="text" [formControl]="favoriteColorControl" />
</div>

<p>Favorite Color: {{ favoriteColorControl.value }}</p>
```

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-reactive-form',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './reactive-form.component.html',
  styleUrl: './reactive-form.component.css',
})
export class ReactiveFormComponent {
  favoriteColorControl = new FormControl('');
}
```

Basic template driven form

```
<h3>Favorite color: (Template Form)</h3>

<div>
  <input type="text" [(ngModel)]="favoriteColor" />
</div>

<p>FavoriteColor: {{ favoriteColor }}</p>
```

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-template-form',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './template-form.component.html',
  styleUrl: './template-form.component.css',
})
export class TemplateFormComponent {
  favoriteColor = '';
}
```

Reactive forms —----------

Reactive forms in Angular allow for a direct connection between the form elements in the view and the data model through instances of FormControl. This ensures immediate and synchronous data binding between the view and the model.

In Angular reactive forms, the interaction flows in two directions:

View to Model: User actions in the view update the data model.
Model to View: Programmatic changes in the data model reflect in the view.

Listen for value changes (View to model)

```
export class ReactiveFormComponent {
  favoriteColorControl = new FormControl('');

  constructor() {
    this.favoriteColorControl.valueChanges.subscribe((color) => {
      console.log('new Favorite color ' + color);
    });
  }
}
```

Model to view

```
<h3>Favorite COlor: (Reactive FOrm)</h3>

<div>
  <input type="text" [formControl]="favoriteColorControl" />
</div>

<p>Favorite Color: {{ favoriteColorControl.value }}</p>

<button (click)="changeColor()">Change Color</button>
```

```
export class ReactiveFormComponent {
  favoriteColorControl = new FormControl('');

  constructor() {
    this.favoriteColorControl.valueChanges.subscribe((color) => {
      console.log('new Favorite color ' + color);
    });
  }

  changeColor() {
    this.favoriteColorControl.setValue('Red');
  }
}
```

Template-Driven Forms —----------

View to model flow

```
In the view-to-model flow, data changes from the view (input element) propagate to the
model (component property).

User Types in the Input:

Let's say the user types "Blue" into an input box bound to favoriteColor in the
component.
The Input Emits an "input" Event:

Angular listens to the input's native "input" event.
ControlValueAccessor Triggers setValue() on FormControl:

A ControlValueAccessor (like DefaultValueAccessor for basic inputs) detects the change
and calls setValue() on the FormControl bound to favoriteColor.
FormControl Emits valueChanges Observable:

The FormControl instance emits the new value to any subscribers via the valueChanges
observable.
viewToModelUpdate() is Called:

The ControlValueAccessor calls viewToModelUpdate() on the NgModel directive, emitting
an ngModelChange event with the new value ("Blue").
Two-Way Binding Updates favoriteColor in Component:

Since the template uses Angular's two-way binding syntax [ (ngModel) ], the component's
favoriteColor property updates to "Blue".
```

Model to view flow

```
In the model-to-view flow, data changes in the component propagate to the view.

Update favoriteColor in Component:

We programmatically update favoriteColor to "Red" in the component.
Change Detection (CD) Begins:

Angular's change detection detects that favoriteColor has changed and triggers an
ngOnChanges event on the NgModel directive.
ngOnChanges() Sets FormControl Value Asynchronously:

The ngOnChanges() method of NgModel schedules an asynchronous task to update the
FormControl with the new value ("Red").
Second Change Detection Triggered:

Angular runs a second change detection to prevent
ExpressionChangedAfterItHasBeenChecked errors.
FormControl Emits valueChanges Observable:

After the asynchronous update, the FormControl instance emits the latest value ("Red")
through its valueChanges observable.
ControlValueAccessor Updates the Input Element:
```

```ts
export class TemplateFormComponent {
  favoriteColor = '';

  updateColor() {
    this.favoriteColor = 'red';
  }
}
```

```html
<h3>Favorite color: (Template Form)</h3>

<div>
  <input type="text" [(ngModel)]="favoriteColor" />
</div>

<p>FavoriteColor: {{ favoriteColor }}</p>

<button (click)="updateColor()">Update COlor</button>
```

Listen for value changes

```html
<h3>Favorite color: (Template Form)</h3>

<div>
  <input type="text" [(ngModel)]="favoriteColor" #colorInput="ngModel" />
</div>
                                    ngModel
                                    a-ngModel
<p>FavoriteColor: {{ favoriteColor }}</p>

<button (click)="updateColor()">Update COlor</button>
```

```ts
export class TemplateFormComponent {
  @ViewChild('colorInput') colorInputModel!: NgModel;
  favoriteColor = '';

  ngAfterViewInit() {
    this.colorInputModel.valueChanges?.subscribe((data) => {
      console.log(data);
    });
  }

  updateColor() {
    this.favoriteColor = 'red';
  }
}
```

Data Model Mutability —----------

In Reactive forms >>

Reactive forms in Angular keep the data model immutable, meaning that instead of modifying the existing data model directly, a new version of the data model is created each time there's a change. This lets Angular more efficiently detect and respond to changes.

How It Works: Each change generates a new FormControl instance value, allowing us to track these unique changes and integrate seamlessly with Angular's change detection.

Advantages: It's easier to manage change tracking and integrates well with observable operators.

In template driven forms >>

In contrast, template-driven forms use a mutable data model that directly binds to component properties. Angular's two-way data binding ([(ngModel)]) updates the data model immediately as changes occur in the template.

How It Works: The input field updates the bound component property directly, making the data model mutable.
Implications: Change detection is less efficient because Angular has to check for changes every time data binding triggers an update, as there's no observable to track unique changes.

Reactive Form Controls implementation —----------

Import reactiveFormsControlModule

```ts
@Component({
  selector: 'app-name-editor',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './name-editor.component.html',
  styleUrl: './name-editor.component.css',
})
export class NameEditorComponent {
  name = new FormControl('');

  updateName() {
    this.name.setValue('Nancy');
  }
```

```
Go to component
<label for="name">Name: </label>
<input type="text" id="name" [formControl]="name" />

<p>Value: {{ name.value }}</p>

<button (click)="updateName()">Update Name</button>
```

Forms typically contain several related controls. Reactive forms provide two ways of grouping multiple related controls into a single input form.

Form group
Defines a form with a fixed set of controls that you can manage together

Form array

Defines a dynamic form, where you can add and remove controls at run time. You can also nest form arrays to create more complex forms.

Just as a form control instance gives you control over a single input field, a form group instance tracks the form state of a group of form control instances (for example, a form). Each control in a form group instance is tracked by name when creating the form group.

Formgroups >>>

```ts
@Component({
  selector: 'app-profile-editor',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './profile-editor.component.html',
  styleUrl: './profile-editor.component.css',
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });

  onSubmit(event: Event) {
    event.preventDefault();
    console.log(this.profileForm.value);
  }
}
```

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit($event)">
  <div><label for="first-name">First Name: </label></div>
  <div><input type="text" id="first-name" formControlName="firstName" /></div>

  <div><label for="last-name">Last Name: </label></div>
  <div><input type="text" id="last-name" formControlName="lastName" /></div>

  <div>
    <button type="submit" [disabled]="!profileForm.valid">Submit</button>
  </div>
</form>
```

Nested form groups >>>

```ts
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl(''),
    }),
  });

  onSubmit(event: Event) {
    event.preventDefault();
    console.log(this.profileForm);
  }
}
```

```
Go to component
<form [formGroup]="profileForm" (ngSubmit)="onSubmit($event)">
  <div><label for="first-name">First Name: </label></div>
  <div><input type="text" id="first-name" formControlName="firstName" /></div>

  <div><label for="last-name">Last Name: </label></div>
  <div><input type="text" id="last-name" formControlName="lastName" /></div>

  <div formGroupName="address">
    <div><label for="street">Street: </label></div>
    <div><input type="text" id="street" formControlName="street" /></div>

    <div><label for="city">city: </label></div>
    <div><input type="text" id="city" formControlName="city" /></div>

    <div><label for="zip">zip: </label></div>
    <div><input type="text" id="zip" formControlName="zip" /></div>
  </div>

  <div>
    <button type="submit" [disabled]="!profileForm.valid">Submit</button>
  </div>
</form>
```

```ts
updateProfile() {
  this.profileForm.setValue({
    firstName: 'Nancy',
    lastName: 'Smith',
    address: {
      street: '123 Drew Street',
      city: 'San Francisco',
      state: 'CA',
      zip: '94105',
    },
  });
  console.log(this.profileForm.value);
}
```

FormBuilders and Dynamic forms >>>

```ts
export class ProfileEditorComponent {
  private formBuilder = inject(FormBuilder);

  profileForm = this.formBuilder.group({
    firstName: ['', Validators.required],
    lastName: [''],
    address: this.formBuilder.group({
      street: [''],
      city: [''],
      state: [''],
      zip: [''],
    }),
  });
```

```
<div><label for="first-name">First Name: </label></div>
<div><input type="text" id="first-name" formControlName="firstName" /></div>
@if (profileForm.get('firstName')?.invalid &&
profileForm.get('firstName')?.touched) {
<div>First Name is required</div>
}
```

showing error messages

```ts
onSubmit(event: Event) {
  event.preventDefault();
  if (this.profileForm.valid) {
    console.log(this.profileForm.value);
  } else {
    console.log('form is not valid');
  }
}
```

FormArray >>>

```ts
export class ProfileEditorComponent {
  profileForm = this.formBuilder.group({
    lastName: [''],
    address: this.formBuilder.group({
      street: [''],
      city: [''],
      state: [''],
      zip: [''],
    }),
    aliases: this.formBuilder.array([this.formBuilder.control('')]),
  });

  get aliases(): FormArray {
    return this.profileForm.get('aliases') as FormArray;
  }

  addAlias() {
    this.aliases.push(this.formBuilder.control(''));
  }

  get formValue() {
    return this.profileForm.value;
  }
```

```html
<div>
  <h3>Form value:</h3>
  <pre>{{ formValue | json }}</pre>
</div>
```

```html
<div formArrayName="aliases">
  <h2>Aliases</h2>
  <button (click)="addAlias()" type="button">+Add Alias</button>

  @for (alias of aliases.controls; track $index; let index = $index) {
  <div>
    <label>alias- {{ index }}</label>
  </div>
  <div>
    <input type="text" id="alias-{{ index }}" [formControlName]="index" />
  </div>
  }
</div>
```

Strictly Typed Forms —----------

Strictly typed forms in Angular ensure that the types of your form controls are known at compile-time. You, 5 hours ago • Uncommitted changes

This means that instead of relying on any or unknown types, Angular can infer and enforce the exact type of data that each form control holds. With this, TypeScript's powerful type-checking capabilities come into play, providing you with instant feedback during development and reducing the likelihood of errors.

```ts
interface User {
  name: FormControl<string>;
  age: FormControl<number>;
  email: FormControl<string>;
}

@Component({
  selector: 'app-user-login',
  standalone: true,
  imports: [],
  templateUrl: './user-login.component.html',
  styleUrl: './user-login.component.css',
})
export class UserLoginComponent {
  fb = inject(FormBuilder);

  userLoginForm = this.fb.group<User>({
    name: new FormControl(),
    age: new FormControl(),
    email: new FormControl(),
  });
}
```

```ts
ngOnInit() {
  this.userLoginForm.controls.age.valueChanges.subscribe((data) => {
    console.log(data);
  });
}
```

```ts
userLoginForm = new UntypedFormGroup({
  name: new FormControl(),
  age: new FormControl(),
  email: new FormControl(),
});
```

```html
Go to component
<form [formGroup]="userLoginForm" (ngSubmit)="onSubmit($event)">
  <div>
    <label for="name">Name</label>
    <input type="text" id="name" formControlName="name" />
  </div>
  <div>
    <label for="email">email</label>
    <input type="text" id="email" formControlName="email" />
  </div>
  <div>
    <label for="age">age</label>
    <input type="number" id="age" formControlName="age" />
  </div>

  <div>
    <button>Submit</button>
    <button (click)="updateValues()" type="button">Update Values</button>
  </div>
</form>
```

Nullable vs. Non-Nullable Controls >>>

Usually when the form get reset it’s values assigned null. But if we want to enforce not to set null values and set to it’s initial values

```
@Component({
  selector: 'app-user-login',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './user-login.component.html',
  styleUrl: './user-login.component.css',
})
export class UserLoginComponent {
  fb = inject(FormBuilder);

  emailControl = new FormControl('angular@gmail.com', { nonNullable: true });

  resetEmail() {
    console.log(this.emailControl.value);
    this.emailControl.reset();
    console.log(this.emailControl.value);
  }
}
```

When this get reset it sets it;s initial values

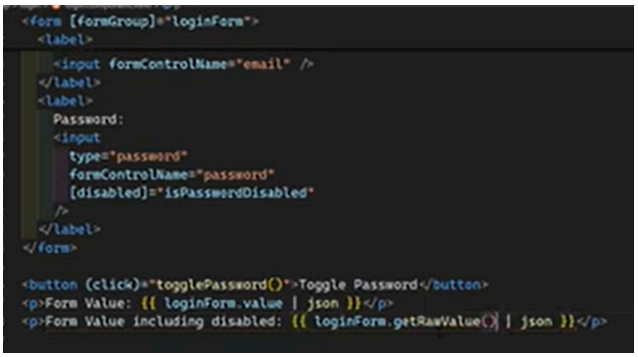
—

```
fb = inject(FormBuilder);

emailControl = new FormControl<string | null>(null);

setEmail() {
  this.emailControl.setValue('angular@gmail.com');
}

resetEmail() {
  console.log(this.emailControl.value);
  this.emailControl.reset();
  console.log(this.emailControl.value);
}
```

.

FormGroup, FormRecord >>>

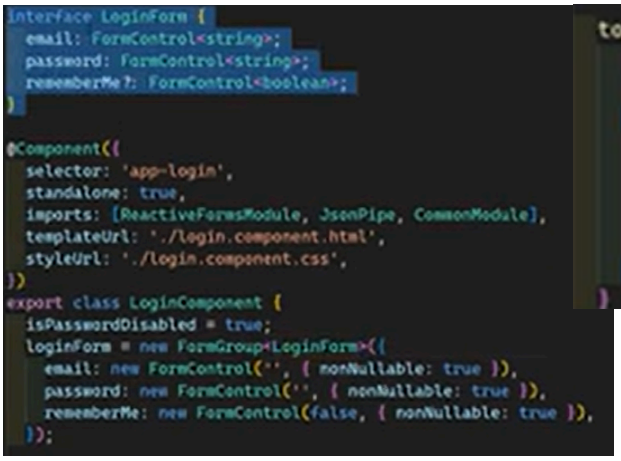
Disabled controls values not present in form.value object, it is present in raw object



```
<form [formGroup]="loginForm">
  <label>
    <input formControlName="email" />
  </label>
  <label>
    Password:
    <input
      type="password"
      formControlName="password"
      [disabled]="isPasswordDisabled"
    />
  </label>
</form>

<button (click)="togglePassword()">Toggle Password</button>
<p>Form Value: {{ loginForm.value | json }}</p>
<p>Form Value including disabled: {{ loginForm.getRawValue() | json }}</p>
```

```
export class LoginComponent {
  isPasswordDisabled = true;
  loginForm = new FormGroup({
    email: new FormControl('', { nonNullable: true }),
    password: new FormControl('', { nonNullable: true }),
    rememberMe: new FormControl(false, { nonNullable: true }),
  });

  togglePassword() {
    this.isPasswordDisabled = !this.isPasswordDisabled;
    if (this.isPasswordDisabled) {
      this.loginForm.controls.password.disable();
    } else {
      this.loginForm.controls.password.enable();
    }
  }
}
```

Dynamic include exclude



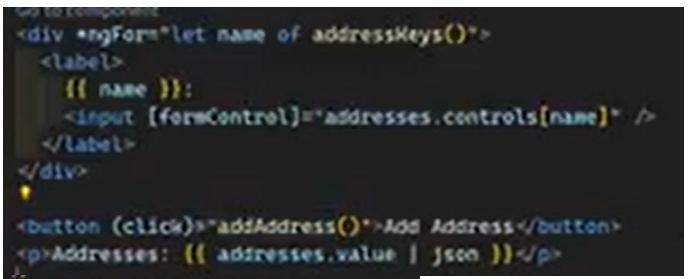
```
interface LoginForm {
  email: FormControl<string>;
  password: FormControl<string>;
  rememberMe?: FormControl<boolean>;
}

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ReactiveFormsModule, JsonPipe, CommonModule],
  templateUrl: './login.component.html',
  styleUrl: './login.component.css',
})
export class LoginComponent {
  isPasswordDisabled = true;
  loginForm = new FormGroup<LoginForm>({
    email: new FormControl('', { nonNullable: true }),
    password: new FormControl('', { nonNullable: true }),
    rememberMe: new FormControl(false, { nonNullable: true }),
  });
```

```
toggleRememberMe() {
  if (this.loginForm.controls.rememberMe) {
    this.loginForm.removeControl('rememberMe');
  } else {
    this.loginForm.addControl(
      'rememberMe',
      new FormControl(false, { nonNullable: true })
    );
  }
}
```

```
<label *ngIf="loginForm.controls.rememberMe">
  Remember Me:
  <input type="checkbox" formControlName="rememberMe" />
</label>
</form>
```

FormRecord Example –

```
export class LoginComponent {
  addresses = new FormRecord<FormControl<string | null>>({});

  addressKeys() {
    return Object.keys(this.addresses.controls);
  }

  addAddress() {
    const name = prompt('Enter name for the new Address');
    if (name) {
      this.addresses.addControl(name, new FormControl(null));
    }
  }
}
```



```
<div *ngFor="let name of addressKeys()">
  <label>
    {{ name }}:
    <input [formControl]="addresses.controls[name]" />
  </label>
</div>

<button (click)="addAddress()">Add Address</button>
<p>Addresses: {{ addresses.value | json }}</p>
```

```
export class LoginComponent {
  fb = inject(FormBuilder);

  loginForm = this.fb.nonNullable.group({
    email: '',
    password: '',
  });
}
```

Non nullable form builder >

Create Template Driven Forms using ngModel two-way data binding —----------

The control elements in the form are bound to data properties that have input validation. The input validation helps maintain data integrity and styling to improve the user experience.

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.

Angular supports two design approaches for interactive forms. Template-driven forms allow you to use form-specific directives in your Angular template. Reactive forms provide a model-driven approach to building forms.

Template-driven forms are a great choice for small or simple forms, while reactive forms are more scalable and suitable for complex forms. For a comparison of the two approaches,

You can build almost any kind of form with an Angular template —login forms, contact forms, and pretty much 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.

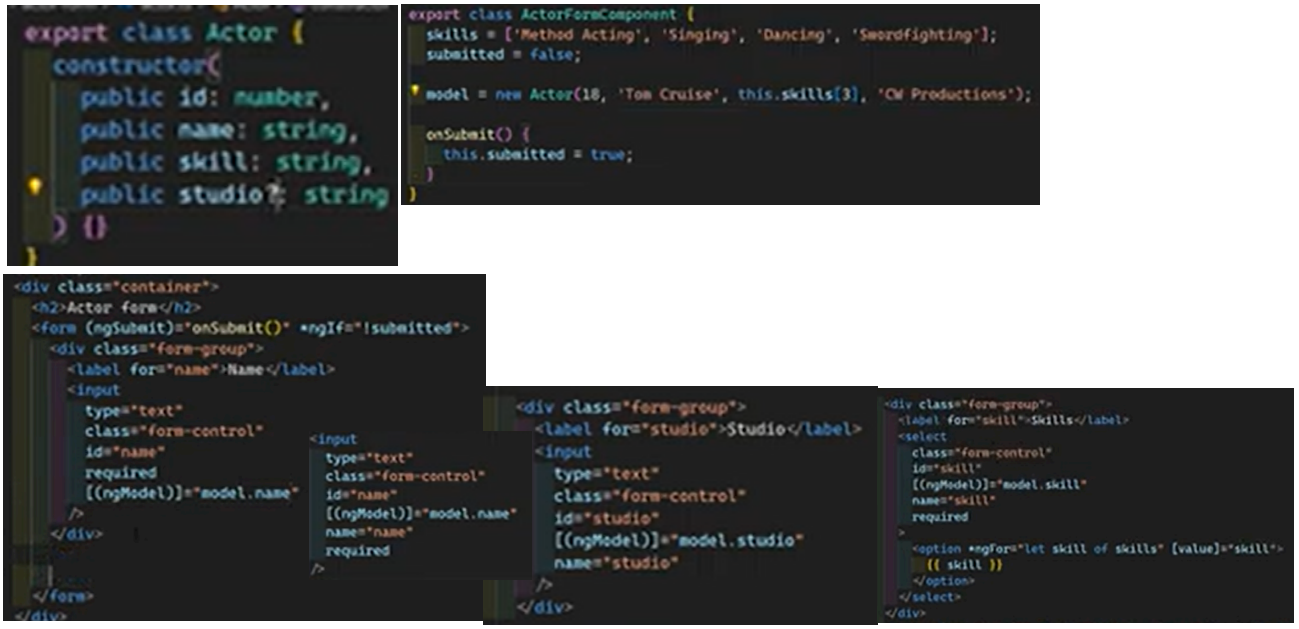
NgModel
Reconciles value changes in the attached form element with changes in the data model, allowing you to respond to user input with input validation and error handling.

NgForm

Creates a top-level FormGroup instance and binds it to a <form> element to track aggregated form value and validation status. As soon as you import FormsModule, this directive becomes active by default on all <form> tags. You don't need to add a special selector.

NgModelGroup
Creates and binds a FormGroup instance to a DOM element.

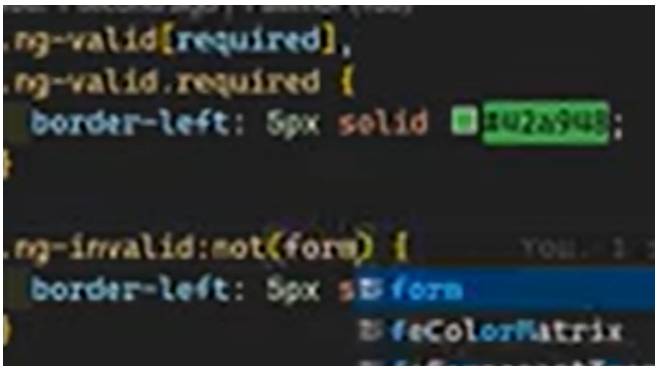
Implementation example >>>



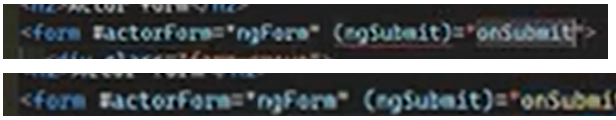
Track Control States & Show Validation Messages >>>

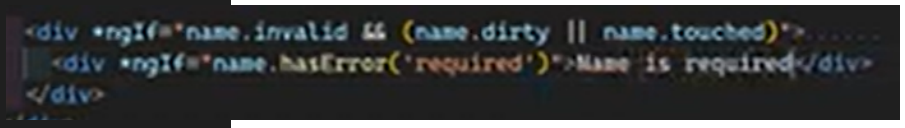
Ng-submitted class appended when the form was submitted and each form control have set of classes for present different events

Ng-pristine turn into ng-dirty when something typed on input



Add the Submitted State in the template >>>





Add Dynamic Rules to Template Forms >>>

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

export function forbiddenNameValidator(forbiddenName: RegExp):
ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null {
    const forbidden = forbiddenName.test(control.value);
    return forbidden ? {forbiddenName: {value: control.value}} :
    null;
  }
}
```

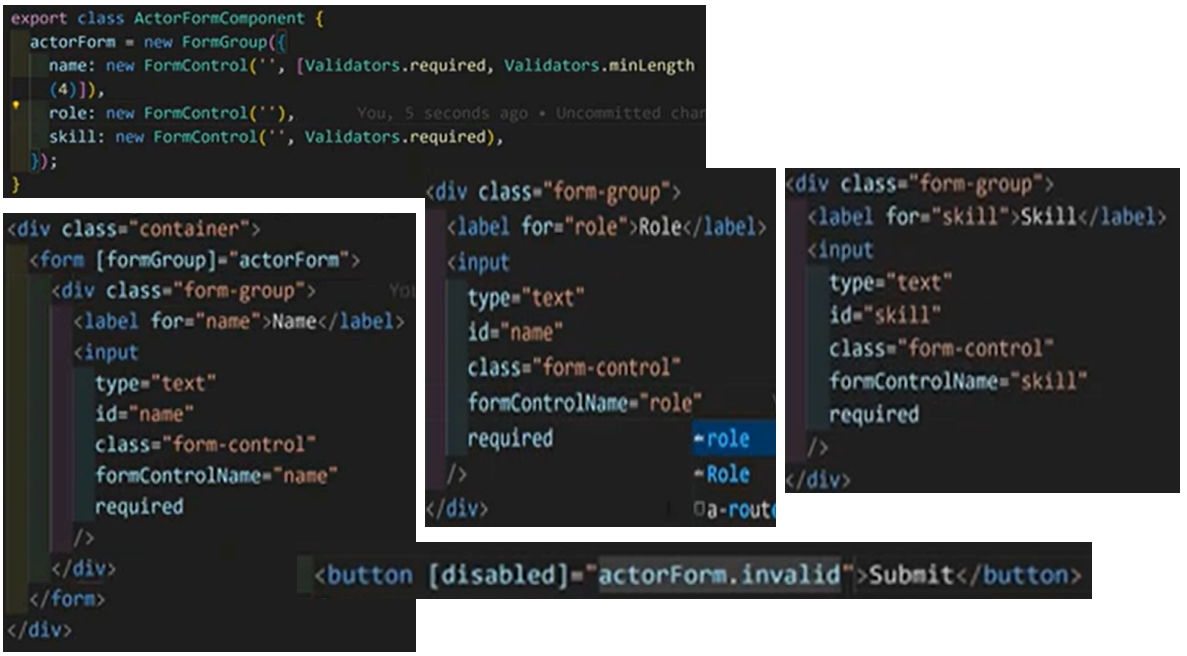
This function does not interact directly with the template

```
@Directive({
  selector: '[appForbiddenName]',
  standalone: true,
  providers: [
    {
      provide: NG_VALIDATORS,
      useExisting: forbiddenNameDirective,
      multi: true,
    },
  ],
})
export class forbiddenNameDirective implements Validator {
  @Input('appForbiddenName') forbiddenName = '';
  validate(control: AbstractControl): ValidationErrors | null {
    return this.forbiddenName
      ? forbiddenNameValidator(new RegExp(control.value))(control)
      : null;
  }
}
```

```
<input
  type="text"
  class="form-control"
  required
  minlength="4"
  appForbiddenName="leela"
  [(ngModel)]="actor.name"
  #name="ngModel"
/>
```

```
<div *ngIf="name.hasError('forbiddenName')">Invalid Name</
```

Validate in the reactive forms >>>



Custom Validators >>>

From Pristine to Touched >>>

Angular automatically mirrors many control properties onto the form control element as CSS classes. Use these classes to style form control elements according to the state of the form.

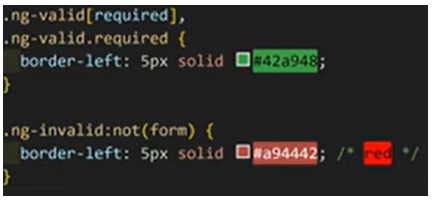
Angular attaches CSS classes to form elements to indicate their validation and interaction states. These classes make it easy to style form controls conditionally.

Validation state classes:

.ng-valid: Control is valid.
.ng-invalid: Control is invalid.
.ng-pending: Control is in a pending state, often while asynchronous validation is processing.

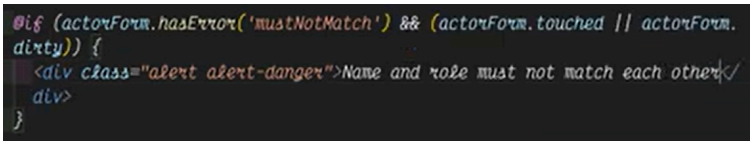
Interaction state classes:

.ng-pristine: Control has not been interacted with yet.
.ng-dirty: Control has been modified by the user.
.ng-untouched: Control has not been touched (i.e., not yet focused and blurred).
.ng-touched: Control has been touched (focused and blurred at least once).



Cross-Field Validation >>>

One form control depend on another control. In here role should not be equal to name attribute



Cross-Field Validation in Template Driven forms >>>

Create directive using previous validator

```ts
import { Directive } from '@angular/core';
import {
  AbstractControl,
  NG_VALIDATORS,
  ValidationErrors,
  Validator,
} from '@angular/forms';
import { mustNotMatchValidator } from '../actor-form/mustNotMatch.
validator';

@Directive({
  selector: '[appmustNotMatch]',
  standalone: true,
  providers: [
    { provide: NG_VALIDATORS, useExisting: mustNotMatchDirective,
    multi: true },
  ],
})
export class mustNotMatchDirective implements Validator {
  validate(control: AbstractControl): ValidationErrors | null {
    return mustNotMatchValidator(control);
  }
}
```

```html
@if (actorForm.hasError('mustNotMatch') && (actorForm.touched ||
actorForm.dirty)) {

<div class="alert alert-danger">Name and role must not match</div>
}
```

```html
<form #actorForm="ngForm" appmustNotMatch>
```

Asynchronous Validations in reactive forms >>>

```ts
import { Injectable } from '@angular/core';
import { delay, of } from 'rxjs'; 143.9k (gzipped: 28.6k)

@Injectable({
  providedIn: 'root',
})
export class ActorService {
  roles = ['hero', 'villain'];

  isRoleTaken(role: string) {
    return of(this.roles.includes(role)).pipe(delay(1000));
  }
}
```

```ts
import { inject } from '@angular/core';
import { AbstractControl, AsyncValidatorFn } from '@angular/forms';
import { ActorService } from './actor.service';
import { catchError, map, of } from 'rxjs'; 144k (gzipped: 28.6k)

export function uniqueRoleValidator(
  actorService: ActorService
): AsyncValidatorFn {
  return (control: AbstractControl) => {
    return actorService.isRoleTaken(control.value).pipe(
      map((isTaken) => (isTaken ? { uniqueRole: true } : null)),
      catchError((error) => of(null))
    );
  };
}
```

```html
<div *ngIf="role.pending">Checking availability of role</div>
<div *ngIf="role.hasError('uniqueRole')">
  The role is not unique. Select another role
</div>
```

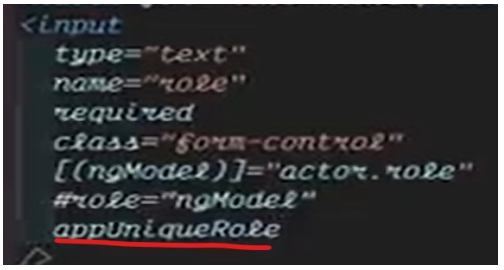
```ts
get role() {
  return this.actorForm.controls.role;
}
```

```ts
role: new FormControl('', {
  validators: [Validators.required],
  asyncValidators: [uniqueRoleValidator(this.actorService)],
}),
```

```ts
role: new FormControl('', {
  validators: [Validators.required],
  asyncValidators: [uniqueRoleValidator(this.actorService)],
  updateOn: 'blur',
}),
```

Asynchronous Validations in Template Driven forms >>>

```
@Directive({
  selector: '[appUniqueRole]',
  standalone: true,
  providers: [
    {
      provide: NG_ASYNC_VALIDATORS,
      useExisting: UniqueRoleDirective,
      multi: true,
    },
  ],
})
export class UniqueRoleDirective implements AsyncValidator {
  actorService = inject(ActorService);
  validate(
    control: AbstractControl
  ): Promise<ValidationErrors | null> | Observable<ValidationErrors |
  null> {
    return uniqueRoleValidator(this.actorService)(control);
  }
}
```



```
<input
  type="text"
  name="role"
  required
  class="form-control"
  [(ngModel)]="actor.role"
  #role="ngModel"
  appUniqueRole
```

```
[ngModelOptions]="{ updateOn: 'blur' }"
```

```
@if (role.pending){
<div class="alert alert-danger">Checking availability...</div>
} @if (role.hasError('uniqueRole')) {
<div class="alert alert-danger">Role is already taken.</div>
} @if (role.invalid && (role.touched || role.dirty)) { @if
(role.hasError('required')) {
<div class="alert alert-danger">Role is required</div>
```

Building Dynamic Reactive Forms —----------

```
Dynamic FOrm


what are the elements ->

let external

questions = [
{
key:
value:
label:
required:
order:
controlType:
type:
options: {}
},


]
```

```
export class QuestionBase<T> {
  value: T | undefined;
  key: string;
  label: string;
  required: boolean;
  order: number;
  controlType: string;
  type: string;
  options: { key: string; value: string }[];

  constructor(options: {
    value?: T;
    key?: string;
    label?: string;
    required?: boolean;
    order?: number;
    controlType?: string;
    type?: string;
    options?: { key: string; value: string }[];
  } = {}
}
```

Service class

```
import { Injectable } from '@angular/core';
import { QuestionBase } from './question-base';

@Injectable()
export class QuestionService {
  getQuestions() {
    const questions: QuestionBase<string>[] = [];
  }
}
```

Textbox class

```
import { QuestionBase } from './question-base';

export class Textbox extends QuestionBase<string> {
  override controlType = 'textbox';
}
```

Drop down class

```
import { QuestionBase } from './question-base';

export class DropDown extends QuestionBase<string> {
  override controlType = 'dropdown';
}
```

Initializing service

```
export class QuestionService {
  getQuestions() {
    const questions: QuestionBase<string>[] = [
      new DropDown({
        key: 'favoriteAnimal',
        label: 'Favorite Animal',
        options: [
          { key: 'cat', value: 'Cat' },
          { key: 'dog', value: 'Dog' },
          { key: 'horse', value: 'Horse' },
          { key: 'capybara', value: 'Capybara' },
        ],
        order: 3,
      }),
      new Textbox({
        key: 'firstName',
        label: 'First name',
        value: 'Alex',
        required: true,
        order: 1,
      }),
    ];
  }
}
```

```
getQuestions() {
  const questions: QuestionBase<string>[] = [
      order: 3,
    }),
    new Textbox({
      key: 'firstName',
      label: 'First name',
      value: 'Alex',
      required: true,
      order: 1,
    }),
    new Textbox({
      key: 'emailAddress',
      label: 'Email',
      type: 'email',
      order: 2,
    }),
  ];

  return of(questions.sort((a, b) => a.order - b.order));
}
```

App Component

```
export class AppComponent {
  title = 'angular18-forms';
  questionService = inject(QuestionService);

  questions$ = this.questionService.getQuestions();
}
```

Dynamic Form component

```
export class DynamicFormComponent {
  @Input() questions!: QuestionBase<string>[];
}
```

Invoking Dynamic Form Component within the App Component

```
<app-dynamic-form [questions]="questions$ | async"></app-dynamic-form>
```

Creating service to build dynamic form group(questionControlls)

```
import { Injectable } from '@angular/core';
import { QuestionBase } from './question-base';
import { FormControl, FormGroup, Validators } from '@angular/forms';

@Injectable()
export class QuestionControlService {
  toFormGroup(questions: QuestionBase<string>[]) {
    const group: any = {};
    questions.forEach((question) => {
      group[question.key] = question.required
        ? new FormControl(question.value || '', Validators.required)
        : new FormControl(question.value || '');
    });
    return new FormGroup(group);
  }
}
```

Leverage the service and initialize formgroup within dynamicForm COmponent

```
@Component({
  selector: 'app-dynamic-form',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule, DynamicFormQuestionComponent],
  providers: [QuestionControlService],
  templateUrl: './dynamic-form.component.html',
  styleUrl: './dynamic-form.component.css',
})
export class DynamicFormComponent {
  @Input() questions: QuestionBase<string>[] = [];
  qcs = inject(QuestionControlService);
  form!: FormGroup;
  ngOnInit() {
    this.form = = this.qcs.toFormGroup(this.questions);
  }
}
```

Loop the dynamic formgroup within DynamicForm Component and pass each form control to separate component

```
<form [formGroup]="form">
  @for (question of questions; track question.key) {
  <app-dynamic-form-question
    [queston]="question"
    [form]="form"
  ></app-dynamic-form-question>
  }
</form>
```

Separate component class file

```
@Component({
  selector: 'app-dynamic-form-question',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './dynamic-form-question.component.html',
  styleUrl: './dynamic-form-question.component.css',
})
export class DynamicFormQuestionComponent {
  @Input() question!: QuestionBase<string>;
  @Input() form!: FormGroup;
}
```

Separate component template

```
<div class="form-group" [formGroup]="form">
  <label [attr.for]="question.key">{{ question.label }}</label>
  @if (question.controlType == 'textbox') {
  <input type="text" [formControlName]="question.key" [id]="question.key" />
  } @if (question.controlType == 'dropdown') {
  <select [id]="question.key" [formControlName]="question.key">
    @for (opt of question.options; track opt.key) {
    <option [value]="opt.value">{{ opt.key }}</option>
    }
  </select>
  }
</div>
```

Track errors in formcontrol separate component

```
export class DynamicFormQuestionComponent {
  @Input() question!: QuestionBase<string>;
  @Input() form!: FormGroup;

  get isValid() {
    return this.form.controls[this.question.key].valid;
  }
}
```

Display error message in template

```
<div *ngIf="!isValid" class="alert alert-danger">
  {{ question.label }} is required
</div>
```

Http Client in Standalone Components

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/
core';
import { provideRouter } from '@angular/router';
import { provideHttpClient } from '@angular/common/http';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(),
  ],
};
```

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class ConfigService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';
  constructor(private http: HttpClient) {}

  getPosts() {
    return this.http.get(this.apiUrl);
  }
}
```

```
@Component({
  selector: 'app-post-list',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './post-list.component.html',
  styleUrl: './post-list.component.css',
})
export class PostListComponent {
  private configService = inject(ConfigService);

  posts$ = this.configService.getPosts();
}
```

```
<h2>Posts</h2>
<ul>
  @for (post of posts$ | async; track post.id) {
  <li>{{ post.title }}</li>
  }
</ul>
```

Angular's HttpClient uses the XMLHttpRequest API to make HTTP requests.

However, Angular offers an optional feature configuration, withFetch, that allows switching HttpClient to use the fetch API instead. Let's explore this feature in detail, along with a step-by-step example.

fetch ()

HttpClient is Angular's service for making HTTP requests, such as GET, POST, PUT, DELETE, etc.
To enable HttpClient in an Angular application, you use the provideHttpClient function in the app configuration.
provideHttpClient accepts optional configuration functions (e.g., withFetch) to modify or extend the client's behavior.

By default, HttpClient relies on the XMLHttpRequest (XHR) API.
The withFetch feature changes the underlying HTTP request method to use the modern fetch API instead.

The fetch API is widely supported in modern browsers and environments, and it has some benefits, like better alignment with JavaScript's Promise-based syntax.
However, fetch does not support some XHR features, such as upload progress events.

```
provideHttpClient(withFetch()),
```

```ts
export class AppComponent {
  title = 'angular18-http';
  private http = inject(HttpClient);
  data: any;

  fetchData() {
    this.http
      .get('https://jsonplaceholder.typicode.com/todos/1')
      .subscribe((response) => {
        this.data = response;
      });
  }
}
```

```html
<h1>Fetch Api with HttpClient</h1>

<button (click)="fetchData()">Fetch Data</button>

<div *ngIf="data">
  <pre>{{ data | json }}</pre>
</div>
```

Functional vs DI-Based Interceptors —----------

interceptors for handling HTTP requests: functional interceptors and DI-based (dependency injection-based) interceptors. T

Using withInterceptors, you define interceptors as functions that handle requests and responses directly. This approach offers a more predictable ordering of interceptors and doesn't rely on DI, which can simplify configuration and make the sequence of interceptors clearer.

Interceptors are configured as functions instead of classes.

The order of interceptors is more predictable since they are explicitly listed.
Recommended over DI-based interceptors for their simplicity and ordering benefits.

Function based interceptors >>>

```ts
import { HttpInterceptorFn } from '@angular/common/http';

export const customHeaderInterceptor: HttpInterceptorFn = (request, next) => {
  const modifiedRequest = request.clone({
    setHeaders: {
      'custom-header': 'keeawebdev',
    },
  });
  return next(modifiedRequest);
};
```

```ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(withInterceptors([customHeaderInterceptor])),
  ],
};
```

The withInterceptorsFromDi method is an older style of configuring interceptors through DI. Here, interceptors are provided as classes and managed through Angular's DI system. While still supported, it is recommended to use functional interceptors due to the predictability in their ordering.

Interceptors are registered as injectable classes.
Order of interceptors can be harder to manage since DI-based ordering depends on injection order, which can be less explicit.

DI based interceptors >>>

```ts
@Injectable()
export class customHeaderInterceptor implements HttpInterceptor {
  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    const modifiedRequest = req.clone({
      setHeaders: {
        'custom-header': 'learnwebdev',
      },
    });
    return next.handle(modifiedRequest);
  }
}
```

```ts
provideHttpClient(withInterceptorsFromDi()),
{
  provide: HTTP_INTERCEPTORS,
  useClass: customHeaderInterceptor,
  multi: true,
},
```

withInterceptors is recommended for new applications due to its predictability and simplicity in managing interceptor ordering.

withInterceptorsFromDi continues to support DI-based class interceptors, useful in applications where this setup is already in place or when there's a preference for DI classes.

Call Parent & Child Interceptors —----------

```ts
import { HttpInterceptorFn } from '@angular/common/http';

export const LoggingInterceptor: HttpInterceptorFn = (req, next) => {
  console.log('Logging Interceptor for users path');
  return next(req);
};
```

new custom interceptor

```ts
export const routes: Routes = [
  {
    path: 'users',
    providers: [provideHttpClient(withInterceptors([LoggingInterceptor]))],
    loadComponent: () =>
      import('./Users/users/users.component').then(
        (users) => users.UsersComponent
      ),
  },
];
```

only the logging interceptor will be placed to users path. Not the customInterceptor (in config file) will not be executed. Custom interceptor will be executed within other routes except the users route

```ts
export const routes: Routes = [
  {
    path: 'users',
    providers: [
      provideHttpClient(
        withInterceptors([LoggingInterceptor]),
        withRequestsMadeViaParent()
      ),
    ],
    loadComponent: () =>
      import('./Users/users/users.component').then(
        (users) => users.UsersComponent
      ),
  },
```

This will be executed both child interceptors and parent interceptors, but the child interceptor will be executed first.

Fetch JSON, Images & More with ResponseType —----------

Understanding responseType: The responseType option lets you specify the expected response data type for your HTTP request. Here are the possible values:

'json': (Default) Parses the response as JSON and returns it as an object. This is commonly used for APIs returning JSON data.

'text': Returns the response as a plain string. Useful for APIs returning text data (e.g., HTML or plain text).

'arraybuffer': Returns the raw bytes of the response as an ArrayBuffer. This is used for binary data, such as images, audio, or video files.

'blob': Returns the response as a Blob object. This is also used for binary data, particularly when downloading files.

Setting responseType as a Literal: Since the value of responseType determines the type returned by HttpClient, it must be specified as a literal type (not a generic string) so TypeScript knows exactly what type to expect. This happens automatically if you provide the responseType inline as part of the request options. However, if you're using a variable for the request options, specify responseType explicitly as a literal using as const.

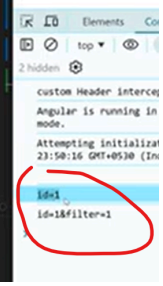
```
fetchTextData() {
  this.http.get('test.txt', { responseType: 'text' }).subscribe((data) => {
    this.textData = data;
  });
}

fetchImageData() {
  this.http
    .get('image.txt', { responseType: 'arraybuffer' })
    .subscribe((data) => {
      this.imageData = data;
    });
}
```

HttpParams & Object Literals —----------

```
ngOnInit() {
  const params = { id: 1 };
  this.http
    .get('https://jsonplaceholder.typicode.com/posts', {
    params })
    .subscribe((response) => {
      console.log(response);
      this.data = response;
    });
}
```

```
ngOnInit() {
  const httpParams = new HttpParams();
  let params = httpParams.set('id', 1);
  this.http
    .get('https://jsonplaceholder.typicode.com/posts', {
    params: { id: 1 } })
    .subscribe((response) => {
      console.log(response);
      this.data = response;
    });
}
```

```
ngOnInit() {
  let params1 = new HttpParams();
  let params2 = params1.set('id', 1);
  let params3 = params2.set('filter', 1);

  this.http
    .get('https://jsonplaceholder.typicode.com/posts', {
      params: params2.set('userId', 1),
    })
    .subscribe((response) => {
      console.log(params1.toString());
      console.log(params2.toString());
      console.log(params3.toString());
      this.data = response;
    });
}
```



Those are immutables object and creates new objects at each time

HTTP Requests with Custom ParamCodec —----------

Encoding url parameters

```
import { HttpParameterCodec } from '@angular/common/http';

export class CustomHttpParameterCodec implements HttpParameterCodec {
  encodeKey(key: string): string {
    return encodeURIComponent(key).replace(/%20/g, '+');
  }

  encodeValue(value: string): string {
    return encodeURIComponent(value).replace(/%20/g, '+');
  }

  decodeKey(key: string): string {
    return decodeURIComponent(key).replace('+', '%20');
  }

  decodeValue(value: string): string {
    return decodeURIComponent(value.replace(/\+/g, '%20'));
  }
}
```

```
ngOnInit() {
  const url = 'https://jsonplaceholder.typicode.com/todos/1';

  let httpParams = new HttpParams({
    encoder: new CustomHttpParameterCodec(),
  });
  httpParams = httpParams.set('id', 1);
  httpParams = httpParams.set('user Id', 1);

  this.http.get(url, { params: httpParams }).subscribe((response) => {
    this.data = response;
  });
}
```

Immutable HttpHeaders —----------

```
Request headers are additional pieces of information sent with HTTP requests to
provide metadata, authentication, or other information that the server might need to
process the request correctly. In Angular, the HttpClient module is used to send HTTP
requests, and we can configure request headers using its options.

Using a simple object literal.
Using an instance of HttpHeaders.
```

```
this.http
  .get(url, {
    params: httpParams,
    headers: {
      'X-Debugug-Level': 'verbose',
    },
  })
  .subscribe((response) => {
    this.data = response;
  });
```

```
const baseHeaders = new HttpHeaders().set('x-Debug-Level', 'minimal');
const updatedHeaders = baseHeaders.set('x-Debug-Level', 'verbose');

this.http
  .get(url, {
    params: httpParams,
    headers: updatedHeaders,
  })
  .subscribe((response) => {
    console.log(response);
    this.data = response;
  });
```

Observe Full Response —----------

```
this.http
  .get(url, {
    observe: 'response' as const,
    params: httpParams,
    headers: baseHeaders,
  })
  .subscribe((response) => {
    console.log(response);
    this.data = response;
  });
```

File Upload - Track Progress & Handle Events —----------

Http client can return steam of raw events that represent various stages of http request

Request send
Upload progress
Response header received
Download progress

```ts
export class FileLoaderComponent {
  formGroup = new FormGroup({
    file: new FormControl(null),
  });
  selectedFile: File | null = null;

  onFileSelected(event: Event) {
    const input = event.target as HTMLInputElement;
    if (input.files?.length) {
      this.selectedFile = input.files[0];
    }
  }
}
```

```html
<div>
  <h1>File Upload Event</h1>

  <form [formGroup]="formGroup" (ngSubmit)="onSubmit()">
    <div><label for="file">UPload FIle</label></div>
    <div>
      <input
        type="file"
        id="file"
        formControlName="file"
        (change)="onFileSelected($event)"
      />
    </div>

    <div>
      <button>Submit</button>
    </div>
  </form>
</div>
```

```ts
onSubmit() {
  if (!this.selectedFile) return;

  const formData = new FormData();
  formData.append('file', this.selectedFile);

  this.http
    .post('http://localhost:3000/api/upload', formData, {
      reportProgress: true,
      observe: 'events' as const,
    })
    .subscribe((event) => {});
}
```

```ts
.subscribe((event) => {
  switch (event.type) {
    case HttpEventType.UploadProgress:
      if (event.total) {
        this.progress = Math.round((event.loaded / event.total) * 100);
      }
  }
});
```

```ts
.subscribe((event) => {
    case HttpEventType.Response:
      this.uploadComplete = true;
      this.progress = 100;
      console.log('upload complete', event.body);
      break;
    default:
      break;
  }
});
```

Fix HTTP Failures in with RxJS Operators —----------

When working with HTTP requests in Angular, you may encounter two types of failures:
Network/Connection Errors: These occur when the request doesn't reach the server due to issues like no internet connection or a timeout. In this case:

The HTTP status code is 0.
The error property is an instance of ProgressEvent.

Backend Errors: These occur when the request reaches the backend, but the server fails to process it correctly. In this case:

The HTTP status code is set by the server (e.g., 404, 500).
The error property contains the response from the server.

The Angular HttpClient returns all errors wrapped in an HttpErrorResponse. We can handle these errors effectively using RxJS operators like catchError and retry.

1. catchError Operator
The catchError operator is used to intercept and handle errors in an Observable stream. It can:

Transform the error into a value suitable for the UI.
Log the error or perform any side effects.

. retry Operator
The retry operator automatically retries a failed request a specified number of times. It is useful for handling transient issues like network interruptions.

```
export class ErrorHandlerComponent {
  data: any;
  errorMessage: string | null = null;
  http = inject(HttpClient);

  fetchData() {
    this.errorMessage = null;
    this.data = null;
    const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';

    this.http.get(apiUrl).pipe(retry(3), catchError(this.handleError));
  }                                    catchError(this.handleError.bind(this)))

  handleError(error: HttpErrorResponse) {

    return throwError(
      () => new Error('An Error Occured. Please try again later')
    );
  }
}
```

```
export class ErrorHandlerComponent {
  http = inject(HttpClient);

  fetchData() {
    this.errorMessage = null;
    this.data = null;
    const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';

    this.http
      .get(apiUrl)                    catchError(this.handleError.bind(this)))
      .pipe(retry(3), catchError(this.handleError))
      .subscribe({
        next: (response) => {
          this.data = response;
        },
        error: (err) => {
          console.error('unhandled error:', err);
        },
      });
  }

  handleError(error: HttpErrorResponse) {
    return throwError(
      () => new Error('An Error Occured. Please try again later')
    );
```

```
handleError(error: HttpErrorResponse) {
  if (error.status === 0) {
    this.errorMessage =
      'Netweork Error: Please check your internet connection';
  } else {
    this.errorMessage = `Backend returned code ${error.status}. ${error.message}`;
  }
  return throwError(
    () => new Error('An Error Occured. Please try again later')
  );
}
```

Dynamic retry >>>

```
retry({
  count: 3,
  delay: (error, retryCount) => {
    if (![500, 503].includes(error.status)) {
      throw error;
    }
    console.log(`retry attempt ${retryCount}`);
    return timer(1000);
  },
}),
```

Angular 19 Server Side Rendering

Create a New Angular Project with SSR
ng new my-angular-ssr-app —standalone —ssr
This command creates a new Angular app with SSR preconfigured. It adds essential files like server.ts and main.server.ts

Add SSR to an Existing Angular Project
ng add @angular/ssr
This modifies the existing app to support SSR by adding the required files and configurations

server.ts:
Starts the server to handle SSR requests.
main.server.ts
"Bootstraps the Angular application on the server."

app.config.server.ts

"Provides configurations specific to the server-side application."

Run the SSR Application
Build for SSR

npm run build:ssr

Run the Server
npm run serve:ssr

Hybrid Rendering APIs

Angular 19 introduces developer preview APIs for hybrid rendering. These APIs combine SSR and CSR seamlessly. Stay tuned for updates!

Better Core Web Vitals (CWV):
"CWV metrics like First Contentful Paint (FCP) and Largest Contentful Paint (LCP) are improved. This results in a smoother experience."

HttpClient Caching for both Get & Post Requests using includePostRequests —----------

"When an Angular app is running on the server (for example, during SSR), HttpClient can automatically cache outgoing HTTP requests. This cache is transferred to the browser as part of the initial HTML.

In the browser, during the initial rendering, Angular checks the cache to reuse the data instead of making fresh HTTP requests, saving time and resources.

Once the app becomes 'stable'—which means the app's initial rendering is complete—the HttpClient stops using the cache and falls back to regular behavior."

"By default, HttpClient caches all HEAD and GET requests that do not contain Authorization or Proxy-Authorization headers.

For example, if you're fetching a list of products or users with a GET request, Angular caches that during SSR and reuses it in the browser.

But what if you want to include other request types like POST? Angular allows you to configure caching behavior with the withHttpTransferCacheOptions function."

```
export class HttpCacheComponent {
  posts$: Observable<Post[]> | null = null;

  http = inject(HttpClient);
  ngOnInit() {
    this.getPosts();
  }

  getPosts() {
    this.posts$ = this.http.get<Post[]>(
      `https://jsonplaceholder.typicode.com/posts`
    );
  }
}
```

Initial http request(when the page is loaded) not made since it was cached, After the button click getPosts() will be executed and http request also executed normally. But the Post requests behaves normally without chaching by default

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(),
    provideClientHydration(
      withEventReplay(),
      withHttpTransferCacheOptions({
        includePostRequests: true,      You, 1 second ago
      })
    ),
  ],
};
```

In order to chase the post requests also

SSR afterNextRender for implementing Browser-Only Logic —----------

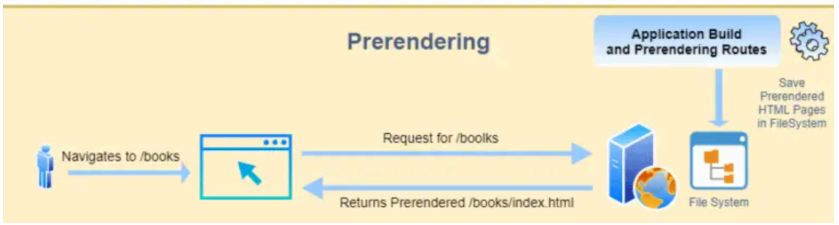
```
<div>
  <h1>Authoring Server-Compatible Components</h1>
  <p #content class="content">
    This is some content whose height we want to calculate. It only works in the
    browser.
  </p>      You, 6 seconds ago • Uncommitted changes
</div>

<app-http-cache></app-http-cache>
```

```
@ViewChild('content') contentRef!: ElementRef;

constructor() {
  afterNextRender(() => {
    const contentHeight = this.contentRef.nativeElement.scrollHeight;
    console.log('Content height in browser:', contentHeight);
  });
}      You, now • Uncommitted changes
```

Prerendering in Angular 19 - Standalone Components for Static Pages —----------

Angular applications can use Server-Side Rendering (SSR) for dynamic, per-request rendering, and prerendering (Static Site Generation or SSG) for generating static HTML at build time. Both improve performance (Time to First Byte) and SEO, but serve different use cases and can be combined in a hybrid approach

Mode	Description	Best For	Requires Server at Runtime?
SSR (Server)	Renders HTML on the server for each request at runtime.	Dynamic content that changes frequently or is user-specific.	Yes (e.g., Node.js/Express).
Prerender (SSG)	Renders static HTML files at build time for specific routes.	Static content like documentation, blog posts, or landing pages.	No, static files can be served from a CDN/static host.
Client (CSR)	Renders entirely in the browser.	User-specific content or highly interactive components where initial SEO isn't critical.	No.



"Prerendering is a process where Angular generates static HTML for your application during the build phase. This boosts SEO and improves page load times since static files are served directly to the browser."

Example :

```
export const routes: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'httpcache',
    component: HttpCacheComponent,
  },
  {
    path: 'products/:id',
    component: ProductComponent,
  },
];
```

sample routes

Authoring Server-Compatible Components

This is some content whose height we want to calculate. It only works in the browser.

- Home
- Http Cache
- Product 1
- Product 3

product works!

trying to access product/:id routes with/without prerendering

Building express server inside angular project - creating Server.js at the project root level

```
const express = require("express");
const cors = require("cors"); 4.5k (gzipped: 1.9k)

const app = express();

app.use(cors());

app.get('/api/products/:id', (req, res) => {

})

app.listen("3000", () => {
  console.log("Server listening to port 3000");
});
```

```
app.get("/api/products/:id", (req, res) => {
  const product = {
    id: req.params.id,
    title: `procust title ${req.params.id}`,
    description: `products description ${req.params.id}`,
  };
  res.send(product);
});
```

Inside page or component

```
export class ProductComponent {
  http = inject(HttpClient);
  route = inject(ActivatedRoute);
  product$: Observable<Product> | null = null;

  ngOnInit() {
    this.product$ = this.route.params.pipe(
      switchMap((params) => {
        const id = params['id'];
        return this.http.get<Product>(
          `http://localhost:3000/api/products/${id}`
        );
      })
    );
  }
}
```

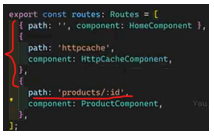
```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>

  <div>title: {{ product.title }}</div>
  <div>Description: {{ product.description }}</div>
</div>
```

So this implementation without prerendering concept…

Prerender unparameterized static routes in Angular 19 SSR using config object with Routes File

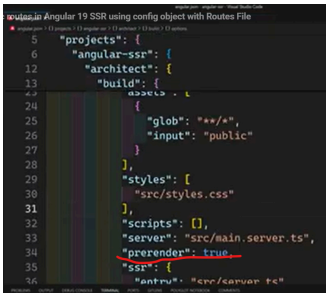
---

Npm run build - create build output , dist folder



Ng build
Prerender options by default will be true - all the static routes prerendered (non parameterized routes)

Angular.json file



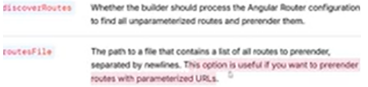

```
"prerender": {
  "discoverRoutes": true
},
```

When we build the project with this property true - dist folder will be include the prerender static files
If we set this property to false and then build the project it will not be create prerender static files

If we need to control what are the routes should be prerenderd

```
"prerender": {
  "discoverRoutes": false,
  "routesFile": "routes.txt"
},
```

```
1 /httpcache
2
```



mention the routes in here (/ , /product, /users likewise)

Prerender the Parametrized dynamic routes in Angular 19 SSR for SEO Load Time Improve

---

only this two prerendered

When the user directly lands on a specifi page, the prerendered page will be rendered. But we navigate normally between the pages then we will get the updated request

Hybrid Rendering in Angular, Combine SSR, CSR & SSG

```
Hybrid Rendering.

Define Hybrid Rendering: "Hybrid rendering combines server-side rendering (SSR),
client-side rendering (CSR), and pre-rendering (SSG), giving developers the flexibility
to optimize performance, SEO, and user experience. With Angular 19's new server
rendering APIs, you have complete control over rendering strategies at the route level."

Why Use Hybrid Rendering?
Flexibility: "Choose SSR for SEO-heavy pages, CSR for dynamic experiences, or SSG for
static, fast-loading pages."
Built-in i18n Support: "Easily localize your application."
Environment Agnostic: "Run it on any JavaScript runtime, not just Node.js."
```

```
D:\angular19>ng new --ssr --server-routing
√ What name would you like to use for the new workspace and initial project? angular-server
√ Which stylesheet format would you like to use? CSS            [ https://developer.mozilla.org/docs/Web/CSS
```

Npm run build
Npm run server:ssr:Angular-server

```
export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'contact', component: ContactComponent },
];
```

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideHttpClient(withFetch()),
    provideRouter(routes),
    provideClientHydration(withEventReplay()),
  ],
};
```

```
<ul>
  <li>
    <a [routerLink]="['/']" routerLinkActive="router-link-active">Home</a>
  </li>
  <li>
    <a [routerLink]="['/contact']" routerLinkActive="router-link-active"
      >Home</a
    >
  </li>
</ul>
```

```
import { mergeApplicationConfig, ApplicationConfig } from '@angular/core'; Calculatin
import { provideServerRendering } from '@angular/platform-server'; Calculating ...
import { provideServerRoutesConfig } from '@angular/ssr'; Calculating ...
import { appConfig } from './app.config';
import { serverRoutes } from './app.routes.server';

const serverConfig: ApplicationConfig = {
  providers: [
    provideServerRendering(),
    provideServerRoutesConfig(serverRoutes)
  ]
};
```

```
import { RenderMode, ServerRoute } from '@angular/ssr';

export const serverRoutes: ServerRoute[] = [
  {
    path: '**',
    renderMode: RenderMode.Prerender
  }
];
```

```
(alias) enum RenderMode
import RenderMode
```

```
const serverConfig: ApplicationConfig = {
  providers: [
    provideServerRendering(),
    provideServerRoutesConfig(serverRoutes)
  ]
};

export const config = mergeApplicationConfig(appConfig, serverConfig);
```

```
import { RenderMode, ServerRoute } from '@angular/ssr'; 889 (gzipped: 494)

export const serverRoutes: ServerRoute[] = [
  { path: '', renderMode: RenderMode.Client },
  { path: 'contact', renderMode: RenderMode.Server },
];
```

```
Client
Prerender
Server
```

```
Optimized Performance: "Choose the best rendering strategy for each route."
Enhanced SEO: "Pre-rendered and server-rendered pages boost search engine rankings."
Reduced Server Costs: "SSG minimizes server load."


Hybrid rendering is a game-changer for Angular developers. It gives you full control
over how your app is delivered, balancing performance, scalability, and user
experience. Try it out, and let me know your thoughts in the comments!"
```

Customizing build-time prerendering using Parameterized routes & Fallback strategies

---

```
RenderMode.Prerender
Enables prerendering for specific routes. You can define this mode in your route configuration.
```

```
You can prerender routes dynamically for parameters like /post/:id. This is achieved using the getPrerenderParams function.
```

```
Handle requests for paths that aren't prerendered using strategies like:

Server: Fallback to SSR (default).
Client: Fallback to Client-Side Rendering (CSR).
None: No fallback (return a 404 for missing paths).
```

Example:

```
export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'product/:id', component: ProductComponent },
];
```

```
export const serverRoutes: ServerRoute[] = [
  { path: '', renderMode: RenderMode.Server },
  { path: 'contact', renderMode: RenderMode.Client },
  { path: 'product/:id', renderMode: RenderMode.Server },
];
```

```
export class ProductComponent {
  route = inject(ActivatedRoute);

  ngOnInit() {
    this.route.params.subscribe((params) => {
      console.log(params);
    });
  }
}
```

Make only several dynamic routes prerendered

```
export const serverRoutes: ServerRoute[] = [
  { path: '', renderMode: RenderMode.Server },
  { path: 'contact', renderMode: RenderMode.Client },
  {
    path: 'product/:id',
    renderMode: RenderMode.Prerender,
    async getPrerenderParams() {
      return [{ id: '1' }, { id: '2' }];
    },
    fallback: PrerenderFallback.Server,
  },
];
```

```
export const serverRoutes: ServerRoute[] = [
  { path: '', renderMode: RenderMode.Server },
  { path: 'contact', renderMode: RenderMode.Client },
  {
    path: 'product/:id',
    renderMode: RenderMode.Prerender,
    async getPrerenderParams() {
      const posts = inject(PostService);
      const ids = await posts.getPosts();
      return ids.map((id) => ({ id }));
    },
    fallback: PrerenderFallback.Server,
  },
];
```

```
fallback: PrerenderFallback.None,
```

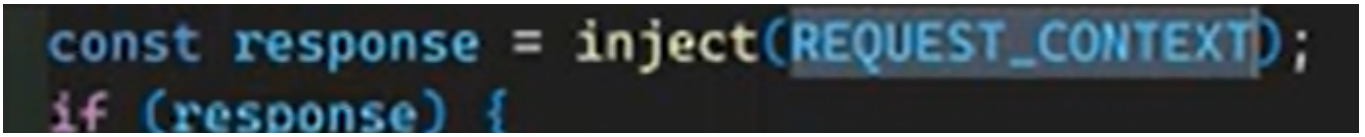
Angular SSR: REQUEST, RESPONSE_INIT, and REQUEST_CONTEXT —----------

Gives access to critical request and response informations

```
"In Angular, server-side rendering allows you to pre-render your app on the server before
sending it to the browser. Angular provides three key tokens in the @angular/core package
to interact with the SSR environment:

REQUEST: Gives access to the current HTTP request object. With this, you can access
headers, cookies, and other details.
RESPONSE_INIT: Enables you to set headers and status codes dynamically in the server
response.
REQUEST_CONTEXT: Provides additional context about the current request for custom handling.
"
"These tokens make it easy to create applications that rely on server-specific data during
rendering. But remember, these tokens are available only on the server during SSR and will
be null in client-side rendering or static site generation."
```

These token only available in server side rendering

server show this information



Gives info about requesting device

Hydration

In brief, Angular Hydration is the process of restoring a server-side rendered application on the client side, helping to enhance performance by eliminating the need to recreate DOM nodes.

While Angular Hydration brings several advantages, it also comes with certain limitations and potential challenges.

What is Angular Hydration?

Angular Hydration is the process of reactivating a server-side rendered application on the client side.

It involves reusing the DOM elements rendered on the server, preserving the application state, transferring any data already fetched by the server, and other related tasks.

The goal of hydration is to boost performance by reusing existing DOM nodes, reducing the need to regenerate them, and preventing UI flickers during the transition.

Re creating Dom nodes cause to

This process could cause a visible UI flicker and harm performance metrics such as First Input Delay (FID) and Largest Contentful Paint (LCP).

With Angular Hydration enabled, instead of replacing the DOM, the client-side Angular application attempts to align the existing DOM elements with the application structure at runtime.

By reusing these DOM elements, Angular Hydration helps prevent UI flickers and boosts performance.

To enable Angular Hydration, you need to import provideClientHydration from @angular/platform-browser and add it to your application's bootstrapping providers list or your root app module's provider list.

Hydration enables only for the server side applications - can be debugged with angular devtools

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideHttpClient(withFetch()),
    provideRouter(routes),
    provideClientHydration(),
  ],
};
```

```
provideClientHydration(withEventReplay()),
```

withEventReply() will store the all user actions and event during the hydration process ongoing and applies all queued events once the hydration process was done

Hydration Constraints —----------

The client browser dom structure and the html content returned by the server should be exactly same In order to complete the hydration process properly

```
If you have components that manipulate the DOM using native DOM APIs or use innerHTML
or outerHTML, the hydration process will encounter errors. Specific cases where DOM
manipulation is a problem are situations like accessing the document, querying for
specific elements, and injecting additional nodes using appendChild. Detaching DOM
nodes and moving them to other locations will also result in errors.

This is because Angular is unaware of these DOM changes and cannot resolve them during
the hydration process. Angular will expect a certain structure, but it will encounter a
different structure when attempting to hydrate. This mismatch will result in hydration
failure and throw a DOM mismatch erro


document.getELements
```

As well as invalid html structures and syntaxes should be avoided

```
@Component({
  selector: 'app-home',
  host: { ngSkipHydration: 'true' },
  imports: [],
  templateUrl: './home.component.html',
  styleUrl: './home.component.css',
})
export class HomeComponent {
  http = inject(HttpClient);
```

```
provideClientHydration(withEventReplay(), withI18nSupport()),
```

Defer Loading —----------

Deferred loading allows you to optimize application performance by postponing the loading of components, directives, and pipes until they're needed. This approach reduces the initial bundle size and improves metrics like Largest Contentful Paint (LCP) and Time to First Byte (TTFB).

Using the @defer block, you can:

Delay loading of non-critical resources.
Specify content to display during various loading states like placeholder, loading, or error.

```
@defer {
<app-large-component></app-large-component>
}
```

```
@defer {
  <app-large-component></app-large-component>
} @placeholder {
  <div>Loading ... PLease wait</div>
}
```

```
@defer {
<app-large-component></app-large-component>
} @placeholder(minimum 500ms) {
<div>Loading ... PLease wait</div>
}
```

```
@defer {
<app-large-component></app-large-component>
} @loading {
<div>fetching content ... </div>
} @placeholder(minimum 500ms) {
<div>Loading ... PLease wait</div>
}
```

```
@defer {
<app-large-component></app-large-component>
} @loading (after 100 ms; minimum 1s){
<div>fetching content ... </div>
} @placeholder(minimum 500ms) {
<div>Loading ... PLease wait</div>
}
```

```
@defer {
<app-large-component></app-large-component>
} @loading (after 100ms; minimum 1s){
<div>fetching content ... </div>
} @placeholder(minimum 500ms) {
<div>Loading ... PLease wait</div>
} @error {
  <p>failed to load content. PLeas</p>
}
```

```
@defer (on viewport){
<app-large-component />
} @placeholder (minimum 500ms) {
```

The @defer block allows you to load and display content lazily in Angular. While waiting for the content to load, a placeholder can be displayed. This improves app performance and user experience by delaying heavy content rendering until specific conditions are met.

The @defer block is triggered by conditions specified in two ways:

on: Specifies conditions like idle, viewport, interaction, etc., which determine when the block is loaded.
when: A custom conditional expression that determines when to load the block.

```
<div #triggerELement></div>
<div class="class-a">
  @defer (on viewport(triggerELement)){
  <app-large-component />
  } @placeholder (minimum 500ms) {
```

Try to loads when ever the triggerElement comes to viewport

```
@defer (on interaction){
<app-large-component />
} @placeholder (minimum 500ms) {
```

User interaction need to load the component, click on it

```
@defer (on hover){
<app-large-component />
} @placeholder (minimum 500ms) {
```

```
@defer (on timer(20)){
<app-large-component />
} @placeholder (minimum 500ms) {
```

```
@defer (when showLargeCOmponen){
<app-large-component />
} @placeholder (minimum 500ms) {
```

Showloargecomponent is a variable in class file

```
@defer (on interaction; prefetch on idle){
<app-large-component />
} @placeholder (minimum 500ms) {
```

Incremental Hydration

What is it?
Incremental hydration is a performance optimization technique in Angular.

It builds on full application hydration, which means that instead of hydrating (activating interactivity for) the entire application at once, it does so incrementally, or piece by piece.

Why is it useful?
Smaller Initial Bundles:
Reduces the size of JavaScript bundles that are initially sent to the browser. This decreases the initial page load time, which is critical for performance.

Improved Performance Metrics:

First Input Delay (FID): Measures the time from when a user interacts with the site (e.g., clicks a button) to when the browser responds. Smaller bundles mean quicker interaction.

Cumulative Layout Shift (CLS): Measures unexpected layout shifts. Incremental hydration helps avoid these shifts, especially with deferrable views.

Full Application Hydration:
In Angular, hydration refers to making a server-rendered application interactive by attaching event listeners and enabling dynamic features on the client side. In full application hydration, this process happens all at once.

Limitations of Full Hydration:

- Larger JavaScript bundles.
- Content loading above the fold (the visible part of the page) can result in layout shifts if placeholder content is replaced dynamically.

Deferrable Views (@defer)
What is it?

@defer is a mechanism to load parts of the application lazily. It allows developers to specify parts of the UI that can be loaded later, depending on certain conditions or triggers.

Before Incremental Hydration:

If @defer content was placed above the fold, the placeholder would render first, followed by the main content, causing layout shifts.

With Incremental Hydration:

Angular ensures that the main template of the @defer block is rendered during hydration, without placeholder transitions, avoiding layout shifts.

Enabling Incremental Hydration
Prerequisites:
Server-Side Rendering (SSR):
This ensures the application is rendered on the server and sent as HTML to the client for faster initial display. Follow Angular's SSR Guide to enable this.

Hydration:

Ensures the server-rendered app becomes interactive on the client side. Follow Angular's Hydration Guide for setup.

How to enable Incremental Hydration?

Use the provideClientHydration() function from @angular/platform-browser.
Pass the withIncrementalHydration() configuration to it.

```
import { bootstrapApplication, provideClientHydration,
withIncrementalHydration, } from
'@angular/platform-browser';

bootstrapApplication(AppComponent,
        { providers:
    [provideClientHydration(withIncrementalHydration
    ())],
});
```

What happens when enabled?

Angular uses incremental hydration instead of full hydration. Event replay (covered below) is enabled automatically.

Event Replay
What is it?

A mechanism that queues browser events triggered by users (e.g., clicks, input) before hydration completes and replays them once hydration is finished.

What happens when enabled?

Angular uses incremental hydration instead of full hydration. Event replay (covered below) is enabled automatically.

Why is it important?

Without event replay, users may interact with the page before hydration finishes, leading to unresponsive behavior.

Relation to Incremental Hydration:

Incremental hydration automatically enables event replay.
If you were previously using withEventReplay(), you can remove it once withIncrementalHydration() is enabled.

Hydrate Triggers
What are they?
Triggers define when deferred content should be hydrated (loaded and made interactive).
Example triggers could include:

- Scrolling into view.
- User interactions, like clicks.

How do they work with Incremental Hydration?
During server-side rendering:

Angular loads and renders the main template of the @defer block instead of placeholders.

During client-side rendering:

- Dependencies for @defer content are deferred and stay inactive until the trigger fires.
- Once the trigger fires:
- Fetch dependencies.
- Hydrate the content.
- Replay any queued events.

How Does Incremental Hydration Work?

Core Principles:

Combines Existing Features:
Builds on full hydration, deferrable views, and event replay.

Defines Hydration Boundaries:
Using @defer blocks with hydrate triggers, Angular hydrates only the required parts.
Renders Main Templates Early:
Prevents placeholder content from being displayed, reducing layout shifts.

Optimized Client-Side Behavior:
Dependencies are only fetched and hydrated when needed, saving resources and improving load times.

Summary:
Incremental hydration is a significant performance improvement in Angular, focusing on: Smaller initial bundles for faster loads.

Avoiding layout shifts using @defer with hydrate triggers.
Ensuring seamless user interaction through event replay.

This optimization is crucial for applications that need to balance performance and interactivity, especially when dealing with large, complex UIs.

Controlling hydration of content with triggers - Incremental Hydration —----------

```
hydrate on idle
hydrate on viewport
hydrate on interaction
hydrate on hover
hydrate on immediate
hydrate on timer
```

```
provideClientHydration(withIncrementalHydration()),
```

Enable incremental hydration. By default eventReply was enabled