Youssef RAFII

Angular

What We'll Cover Over the Next Three Days

1

Foundation & Modern Features

Standalone components and Signals.

Best practices in Angular.

2

State & Async Operations

Service-based state management.

RxJS and async operations.

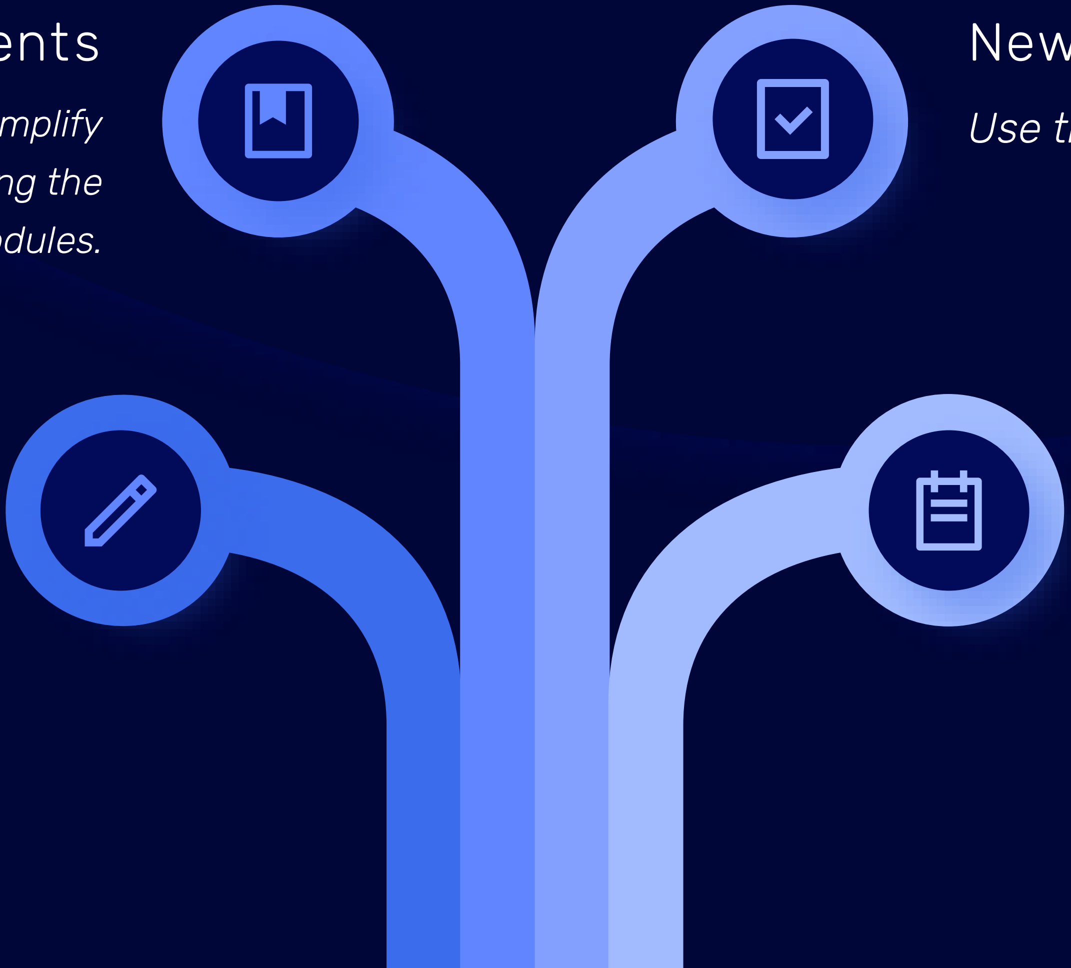
3

Performance & Advanced Routing

Advanced routing and lazy loading.

Deferrable views and performance optimization.

Day 1: Building a Strong Foundation with Modern Angular



Standalone Components

Standalone components simplify Angular development by reducing the need for NgModules.

New flow control

Use the new flow controls

Signals

Signals provide a simpler and more efficient way to manage state compared to RxJS.

They improve performance by optimizing change detection.

Best Practices in Angular

Following best practices ensures your application is scalable, maintainable, and performant.

Standalone Components

Introduction to Standalone Components

- Components are now self-contained.
- Declare dependencies using imports: [] within @Component.
- No need for declarations in NgModule.
- With Angular 15, this feature became stable.
- Can be imported directly into other components.

The NgModule Problem

Scenario: Create a reusable button

- Traditional Approach:
  1. Create ButtonComponent
  2. Declare in NgModule
  3. Export component from module
  4. Import module everywhere

- Issues:
  - Tight coupling
  - Hidden dependencies
  - Less efficient tree-shaking
  - Complex maintenance at scale

Bundle Size Impact

NgModules:

- 📦 Module (10 components) → Uses 3 → Often loads all 10

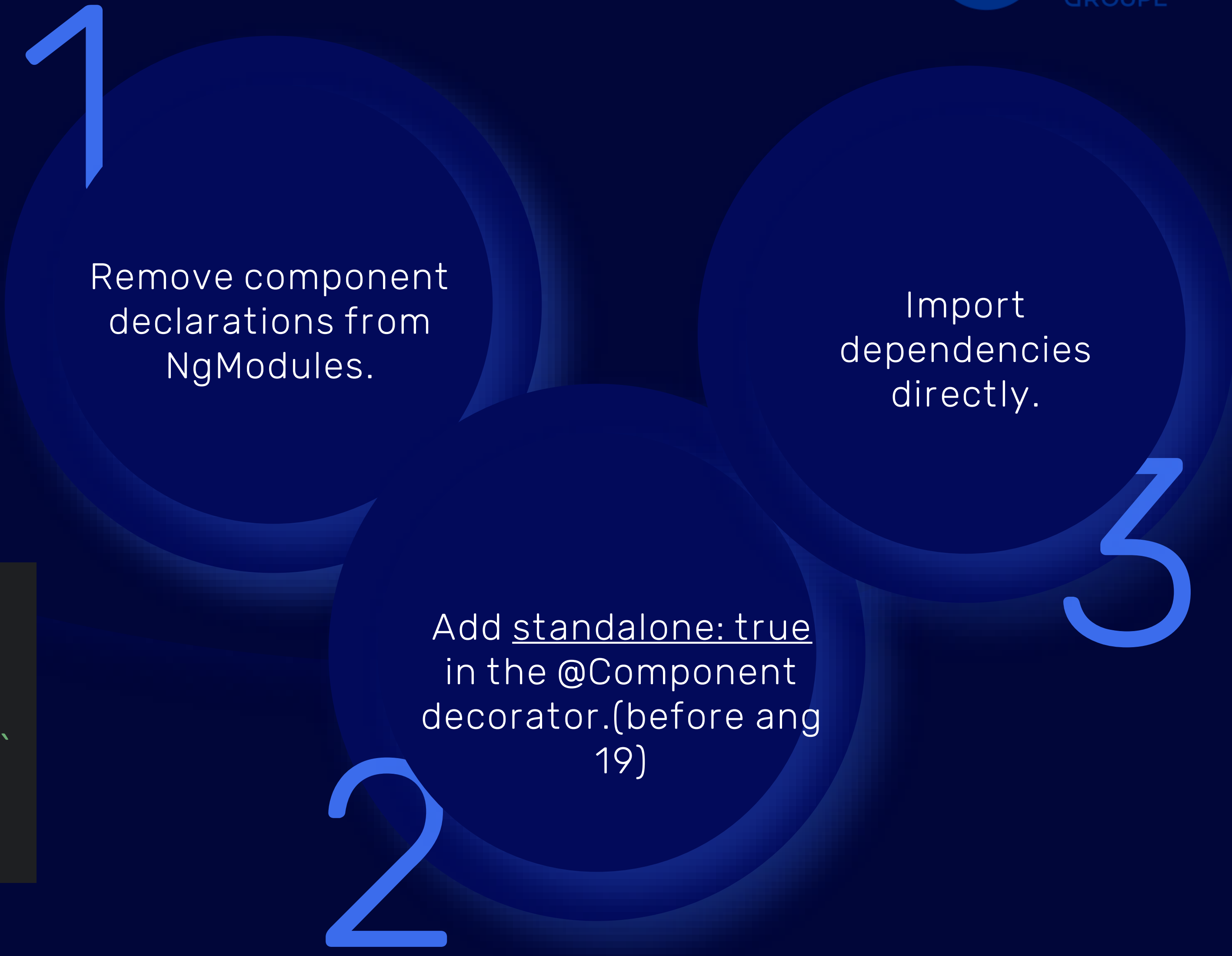
Standalone:

- 📦 Precise imports → Uses 3 → Loads exactly 3

Measurable Results:

- Bundle size reduction
- More efficient tree-shaking

Migration from NgModules



1

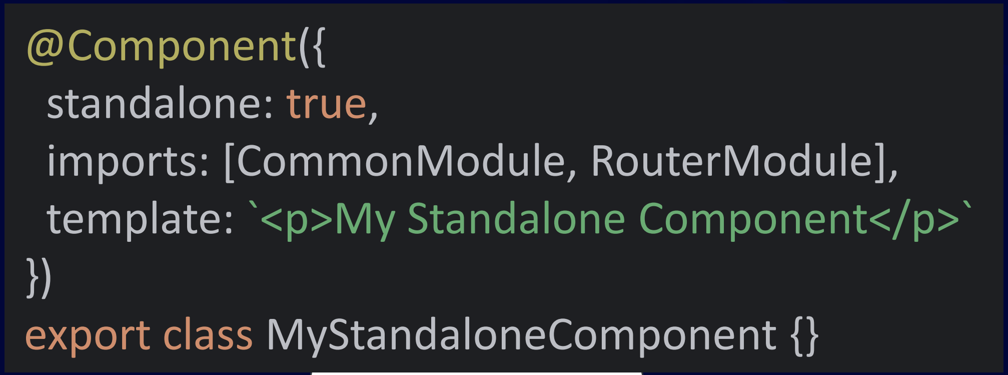
Remove component declarations from NgModules.

2

Add standalone: true in the @Component decorator.(before ang 19)

3

Import dependencies directly.

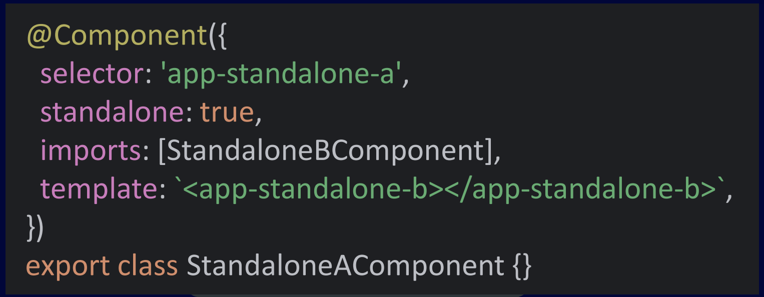


```
@Component({
  standalone: true,
  imports: [CommonModule, RouterModule],
  template: `<p>My Standalone Component</p>`
})
export class MyStandaloneComponent {}
```

Using Standalone Components

Standalone -> Standalone

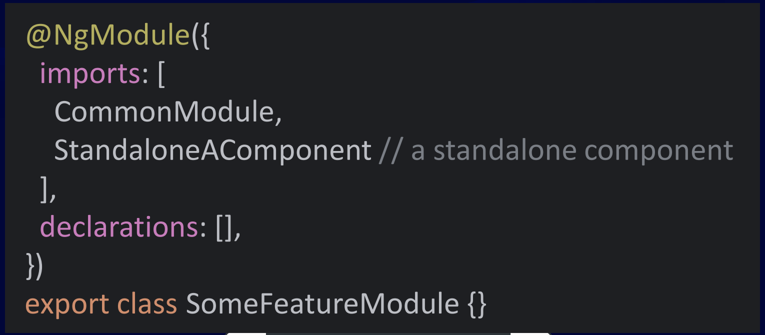
- *Direct Import: One standalone component can directly import another using its imports: [] array.*
- *No NgModule Needed: Purely standalone, each component declares its dependencies.*



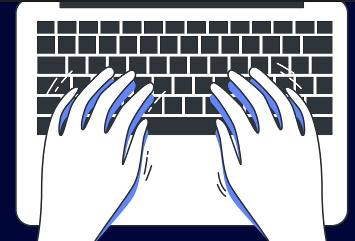
```
@Component({
  selector: 'app-standalone-a',
  standalone: true,
  imports: [StandaloneBComponent],
  template: `<app-standalone-b></app-standalone-b>`,
})
export class StandaloneAComponent {}
```

Standalone -> Standard

- *Declare as if it were a Module Export: In the NgModule's imports array, you can just reference the standalone component directly.*

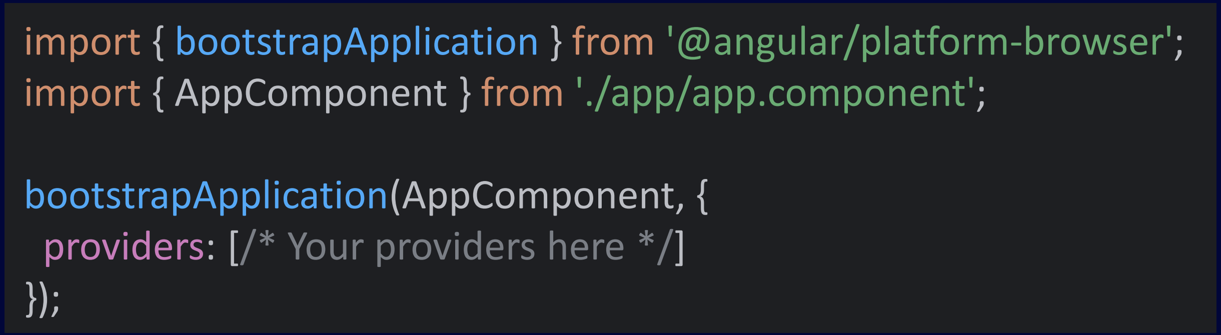


```
@NgModule({
 imports: [
   CommonModule,
   StandaloneAComponent // a standalone component
 ],
 declarations: [],
})
export class SomeFeatureModule {}
```



Bootstrapping Standalone Applications

- Delete or rename your AppModule file (e.g., app.module.ts)
- Add standalone: true in the @Component decorator.
- Modify main.ts to Use bootstrapApplication:
  - Replace any platformBrowserDynamic().bootstrapModule(AppModule) code with bootstrapApplication(AppComponent, {...}).
  - Add Providers for services if needed (e.g., HTTP interceptors, global services).



```
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, {
  providers: [/* Your providers here */]
});
```

QUIZ

Exercise

Built-in control flow

Built-in Control Flow Syntax

Why the change?

- Old: *ngIf, *ngFor, *ngSwitch (structural directives)- New
- @if, @for, @switch (built-in syntax)

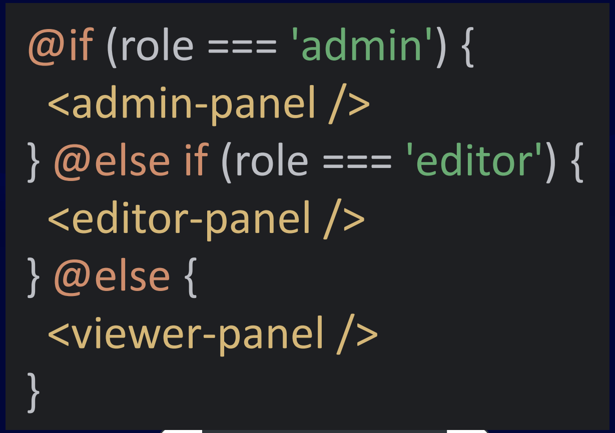
Key Benefits:

- Better performance (no structural directives overhead)
- More intuitive syntax
- Better type checking
- Easier to learn for beginners

@if and @else

Key Features:

- No ng-template required
- Supports else if chains
- Better type narrowing

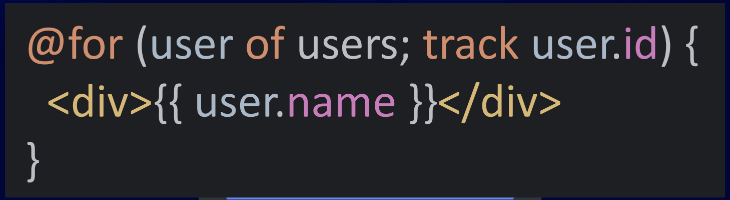


```
@if (role === 'admin') {
  <admin-panel />
} @else if (role === 'editor') {
  <editor-panel />
} @else {
  <viewer-panel />
}
```

@for

Why 'track' is Required:

- Mandatory in new syntax (not optional!)
- Improves performance for list updates
- Angular knows which items changed
- Use unique identifier (id, index if no id)

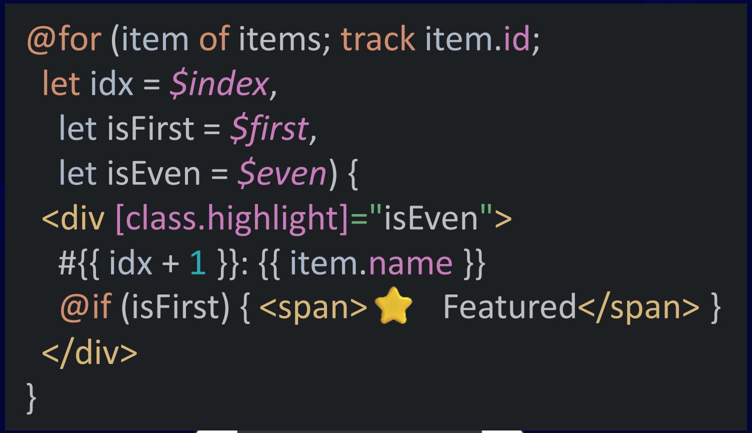


```
@for (user of users; track user.id) {
  <div>{{ user.name }}</div>
}
```

$index and other contextual variables

- Inside @for contents, several implicit variables are always available:

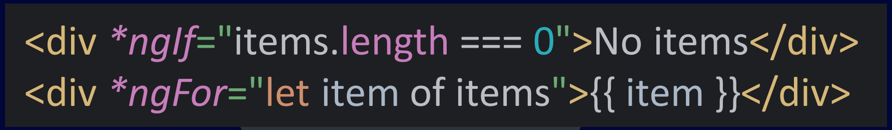
VARIABLE	MEANING
$count	Total number of items
$index	Current iteration index (0-based)
$first	True if first item
$last	True if last item
$even	True if even index
$odd	True if odd index



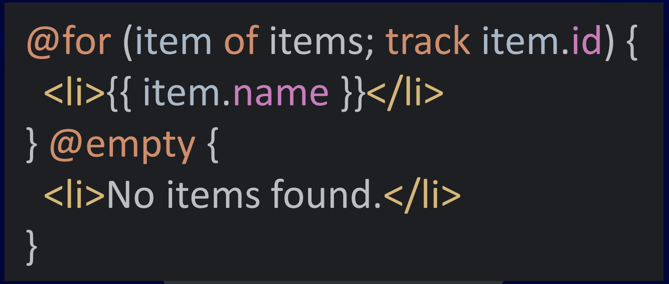
```
@for (item of items; track item.id;
 let idx = $index,
  let isFirst = $first,
  let isEven = $even) {
 <div [class.highlight]="isEven">
  #{{ idx + 1 }}: {{ item.name }}
  @if (isFirst) { <span>⭐  Featured</span> }
 </div>
}
```

empty block

- You can optionally include an @empty section immediately after the @for block content.
- No need for separate length check
- Cleaner code
- Better performance (single evaluation)



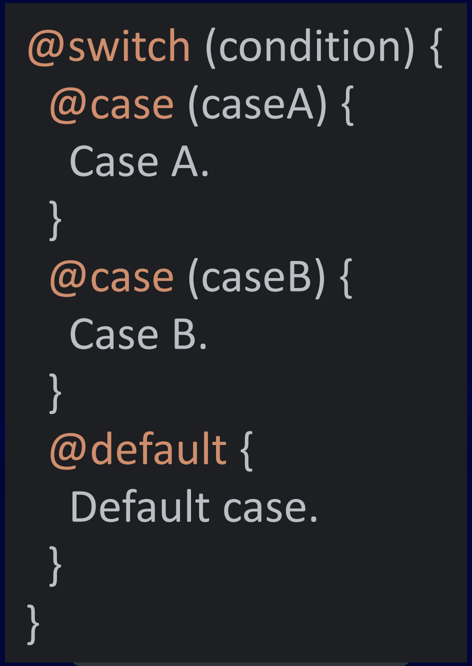
```
<div *ngIf="items.length === 0">No items</div>
<div *ngFor="let item of items">{{ item }}</div>
```



```
@for (item of items; track item.id) {
  <li>{{ item.name }}</li>
} @empty {
  <li>No items found.</li>
}
```

@switch

- The @switch syntax is a new control flow feature in Angular, replacing the traditional *ngSwitch directive.
- It provides a cleaner and more intuitive way to handle multiple conditional cases in templates.



```
@switch (condition) {
  @case (caseA) {
    Case A.
  }
  @case (caseB) {
    Case B.
  }
  @default {
    Default case.
  }
}
```

Performance Benefits of New Control Flow

➜ Built into Compiler

- No structural directive overhead
- Optimized at compile time

➜ Better Change Detection:

- Works seamlessly with signals
- Reduced rendering cycles

Exercise

Best Practices In Angular

Introduction to Best Practices

Why Best Practices Matter:

- Maintainable codebase
- Better performance
- Easier collaboration
- Reduced technical debt

Logical Folder Structure

Key Principles:

- Feature-based organization
- Clear separation of concerns
- Flat structure when possible

```
src/
  app/
    core/    # Singleton services
      guards/
      interceptors/
      services/
    shared/ # Reusable components
      components/
      pipes/
      directives/
      utils/
    features/   # Feature modules
      users/
        components/
        services/
        models/
        users.routes.ts
      …
```

Single Responsibility Principle (SRP)

- Each component handles one specific task
- Simplifies testing and maintenance
  - Example:
    - A UserListComponent displays users
    - A separate UserDetailComponent handles user details

```
@Component({
  selector: 'app-driver-list',
  template: `
<app-driver-item   *ngFor="let drv of drivers"    [driver]="drv">
</app-driver-item>`
})
export class DriverListComponent implements OnInit {
  drivers: Driver[] =  [ …];
}
```

```
@Component({
  selector: 'app-driver-item’,
  template: `
 <h3>Name: {{driver.name}}</h3>
 ` })
export class DriverItemComponent {
  @Input() driver: Driver;
}
```

Automatic Subscription Management

Problem: Memory Leaks

❌ Manual subscriptions without cleanup

```
ngOnInit() {
  this.userService.getUsers().subscribe(users => {
    this.users = users;
  }); // LEAK! Never unsubscribed
}
```

Solution 1: Async Pipe (Recommended)

✅ Automatic subscription + unsubscription

```
users$ = this.userService.getUsers();

<div *ngFor="let user of users$ | async">
  {{ user.name }}
</div>
```

Solution 2: takeUntilDestroyed (Angular 16+)

✅ Automatic cleanup

```
users = signal<User[]>([]);

constructor() {
  this.userService.getUsers()
    .pipe(takeUntilDestroyed())
    .subscribe(users => this.users.set(users));
}
```

Solution 3: toSignal (Best with Signals)

✅ Convert Observable to Signal

```
users = toSignal(this.userService.getUsers(),
  { initialValue: [] });
```

Use trackBy Function for *ngFor

- Improves list rendering performance
- Helps Angular track item identity

```
<li *ngFor="let item of items; trackBy: trackById">
          {{ item.name }}
</li>
```

```
trackById(index: number, item: any): number {
  return item.id;
}
```

- The trackBy function helps Angular identify items in a list when the data changes.
- Without trackBy, Angular re-renders the entire list even if only one item changes.
- Using trackBy improves performance, especially for large lists.

Day 2: Mastering State & Asynchrony in Angular

Change Detection

Harnessing Angular's Core Re-rendering Mechanism

Service-Based State Management

Simplifying Shared Data with Lightweight Patterns

Error Handling Patterns

Building Resilient Apps through Strategic Failure Management

Signals

Why Angular introduced Signals

Problems with Zone.js:

- Detects ALL browser events globally
- Triggers change detection for entire app
- Hard to debug and optimize
- "Black box" reactivity

Problems with RxJS for State:

- Complex learning curve
- Manual subscription management
- Not optimized for synchronous values

Signals Solution:

✅ Update only the Elements that reads the Signals

✅ Better performance

✅ Simpler model(like a variable)

✅ Compiler optimizations

What Signals are?

Definition:

- A signal is a wrapper around a value that notifies consumers when that value changes.

Mental Model:

- Think of a signal as a "smart variable" that knows:
  - Who is reading it
  - When to notify readers about changes
  - How to minimize unnecessary updates(grouping changes and removing redundants)

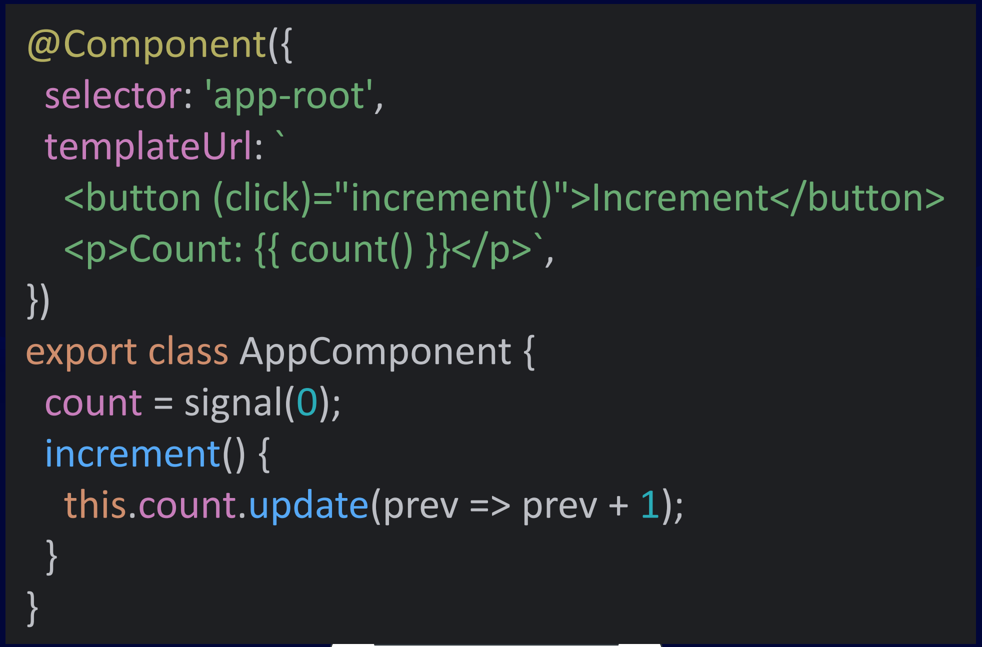
Key Characteristics:

- Container for a value (primitive or object)
- Read by calling it as a function: signal()
- Write with .set() or .update()
- Synchronous (not async like Observables)
- Fine-grained reactivity(if two signals exists in the same component only the changed will be updated)

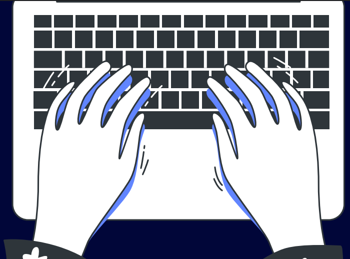
Creating a signal

How to Create a Signal:

- Use the signal() function.
- Pass an initial value.
- Signals are getters: count() returns 0 initially.



```
@Component({
 selector: 'app-root',
 templateUrl: `
  <button (click)="increment()">Increment</button>
  <p>Count: {{ count() }}</p>`,
})
export class AppComponent {
 count = signal(0);
 increment() {
  this.count.update(prev => prev + 1);
 }
}
```



Reading a Signal

How to Read a Signal:

- Call the signal function to read its value.
- In templates: Simply use {{ count() }} or {{ user().name }}. No async pipe needed.

Automatic Dependency Tracking:

When you read a signal:

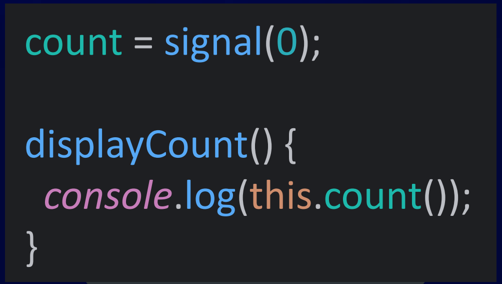
1. Angular records "this code depends on count"

2. When count changes → Angular re-runs this code

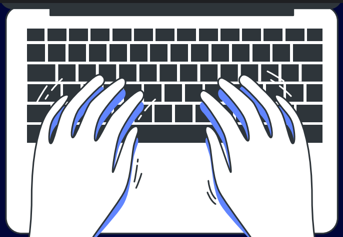
3. Works in: templates, computed, effects

Key Points

- Automatic updates: When a component's template reads a signal, Angular marks that component as needing update whenever the signal changes



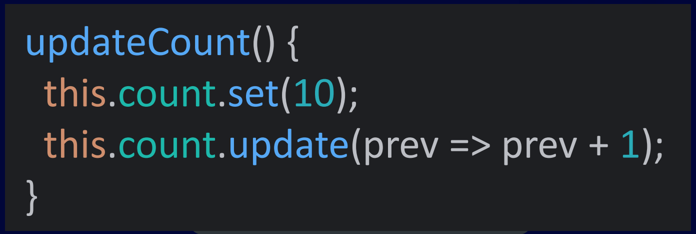
```
count = signal(0);

displayCount() {
 console.log(this.count());
}
```



Updating a Signal

How to Update a Signal:

- **set(newValue):** Replace the signal's value entirely
- **update(fn):** Compute a new value based on current value



```
updateCount() {
  this.count.set(10);
  this.count.update(prev => prev + 1);
}
```

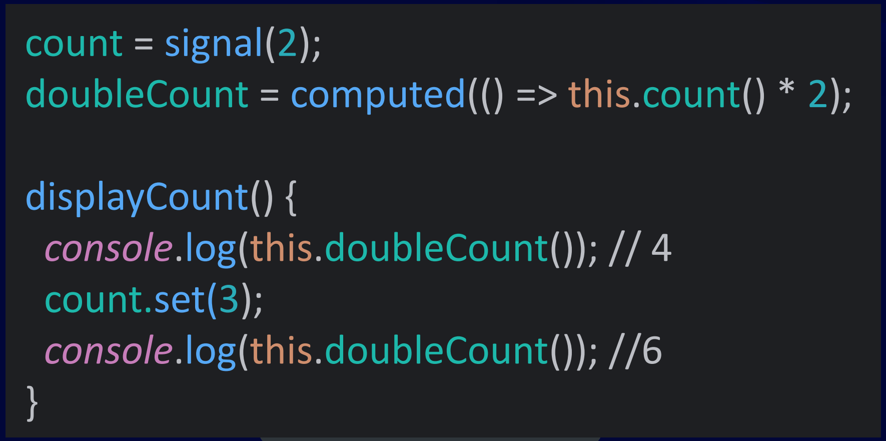
Key Points

- Signals are immutable (you don't modify them directly).
- After calling .set() or .update(), any component or computed signal depending on it is notified and updates its value/render.

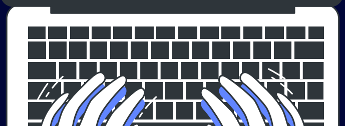
Computed Signals: Derived State

What Are Computed Signals?

- Signals that derive their value from other Signals.
- Automatically update when dependencies change.



```
count = signal(2);
doubleCount = computed(() => this.count() * 2);

displayCount() {
  console.log(this.doubleCount()); // 4
  count.set(3);
  console.log(this.doubleCount()); //6
}
```



Use Case:

- Calculate derived values (e.g., totals, formatted data).
- Not writable: You cannot .set() a computed signal. It always derives from dependencies.
- Lazily evaluated & memorized: The function only runs when you read double() and its inputs changed. The result is cached

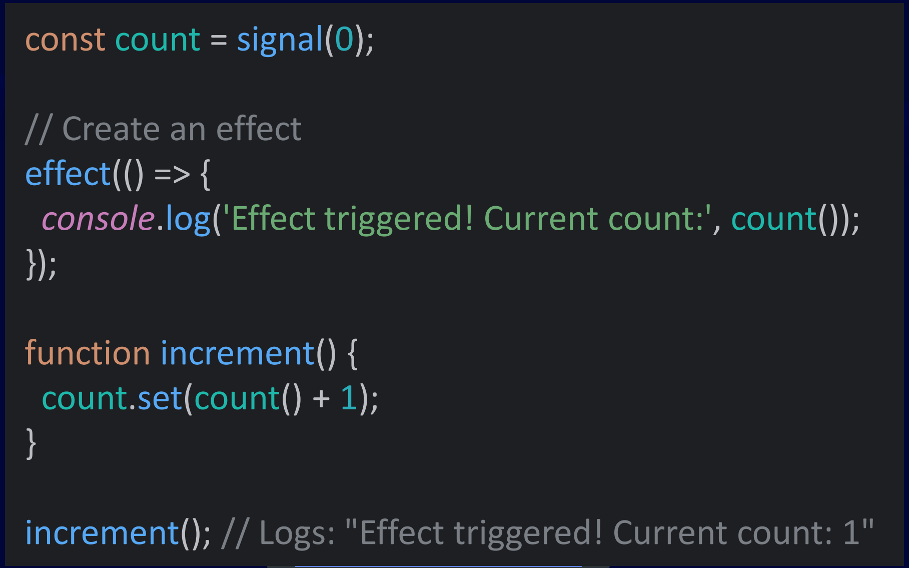
Effect Function: Side Effects

What Is an Effect?

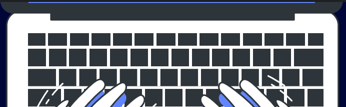
- effect(fn) automatically runs whenever the signal it depends on changes
- Behavior:
  - Initial execution on creation, then re-executes on relevant changes.
  - Tracks dependencies dynamically, like computed.
  - Runs asynchronously (after change detection)

Use Case:

- Logging or analytics when data changes.
- Syncing a signal’s value to external APIs (e.g. localStorage).



```
const count = signal(0);

// Create an effect
effect(() => {
 console.log('Effect triggered! Current count:', count());
});

function increment() {
 count.set(count() + 1);
}

increment(); // Logs: "Effect triggered! Current count: 1"
```



Signal Inputs and Outputs: Simplifying Component Communication

What Are Signal Inputs and Outputs?

- Signal Inputs: Reactive inputs that automatically update.
- Signal Outputs: Reactive outputs that emit values based on Signal changes.

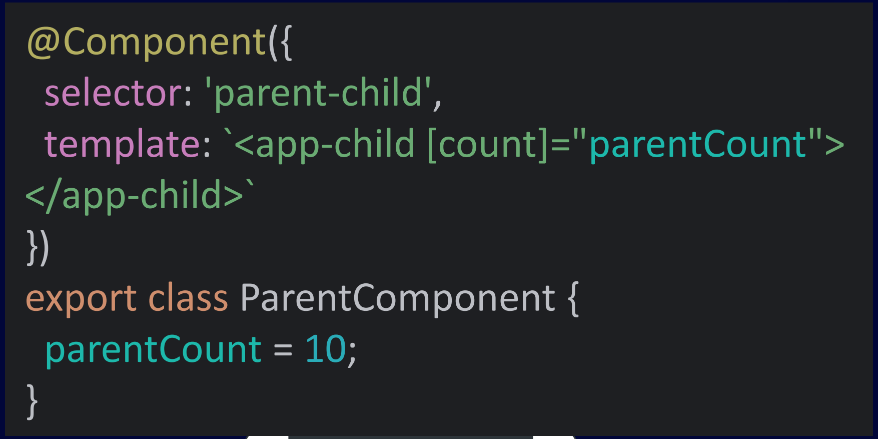
Why Use Them?

- Simplify parent-child communication.
- Make component interactions more reactive.

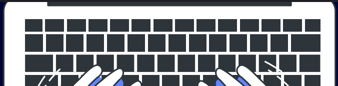
Signal-based component inputs

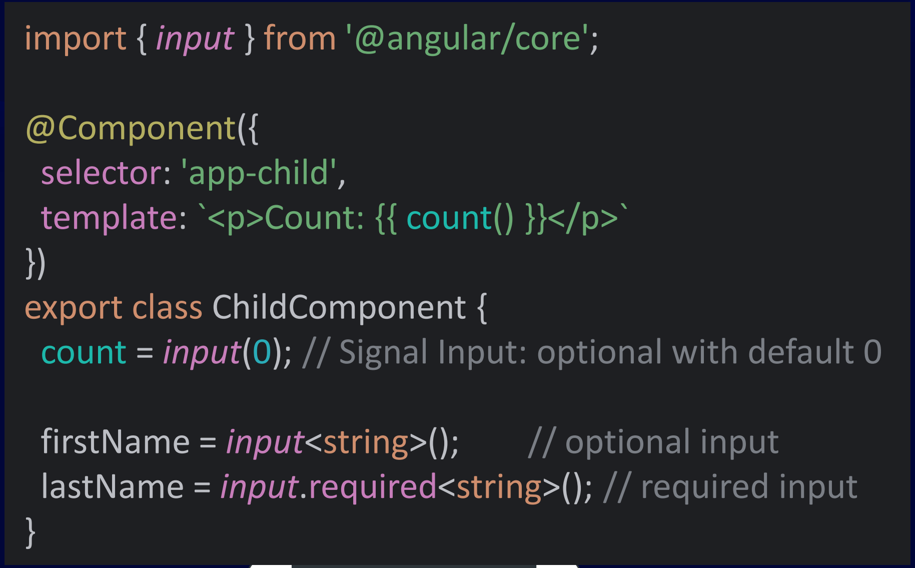
What Are Signal Inputs?

- Inputs exposed as Signals.
- Use the input() function instead of @Input. These create signals for inputs.

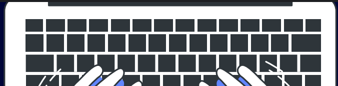


```
@Component({
  selector: 'parent-child',
  template: `<app-child [count]="parentCount">
</app-child>`
})
export class ParentComponent {
  parentCount = 10;
}
```





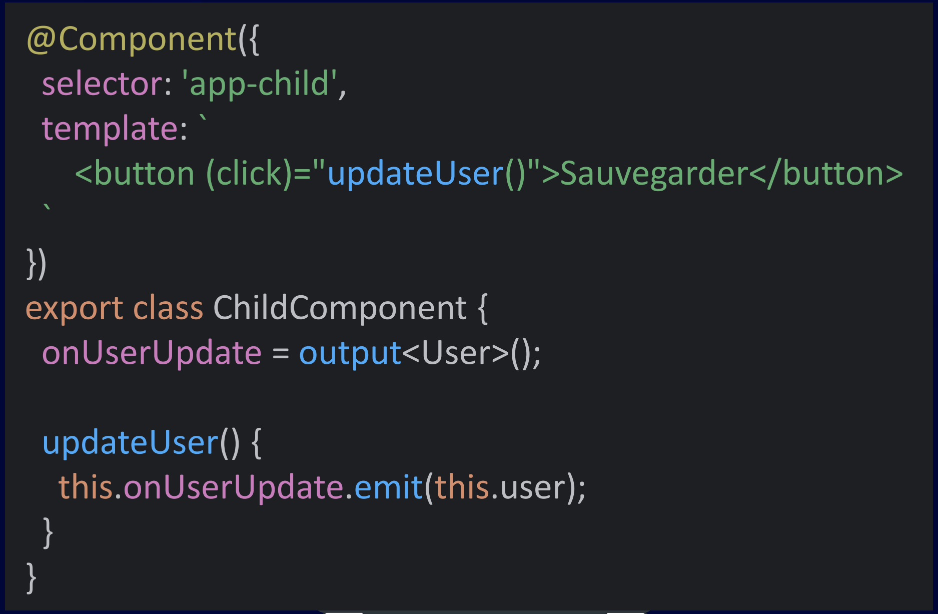
```
import { input } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `<p>Count: {{ count() }}</p>`
})
export class ChildComponent {
  count = input(0); // Signal Input: optional with default 0

  firstName = input<string>();        // optional input
  lastName = input.required<string>(); // required input
}
```



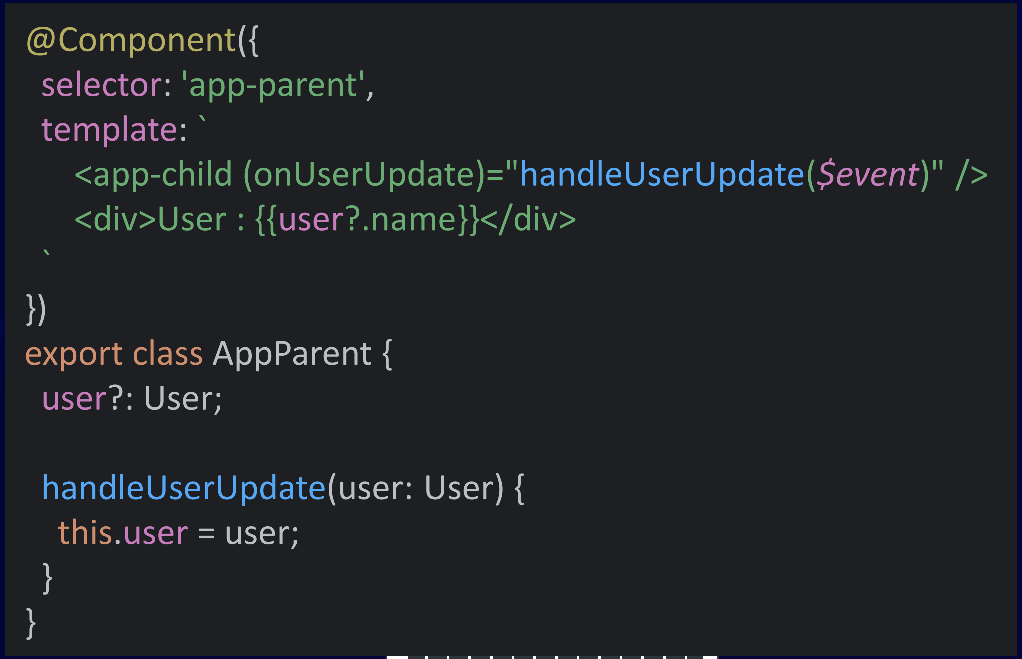
Signal-based component output

What Are Signal Outputs?

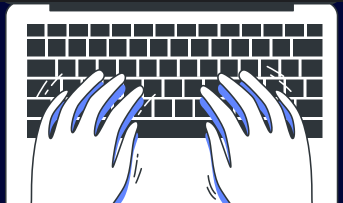
- Outputs that emit values based on Signal changes.
- Use output() to replace @Output.

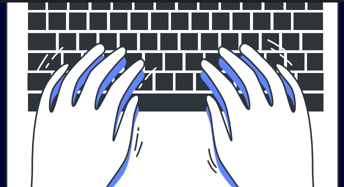


```
@Component({
 selector: 'app-child',
 template: `
   <button (click)="updateUser()">Sauvegarder</button>
 `
})
export class ChildComponent {
 onUserUpdate = output<User>();

 updateUser() {
  this.onUserUpdate.emit(this.user);
 }
}
```



```
@Component({
 selector: 'app-parent',
 template: `
   <app-child (onUserUpdate)="handleUserUpdate($event)" />
   <div>User : {{user?.name}}</div>
 `
})
export class AppParent {
 user?: User;

 handleUserUpdate(user: User) {
  this.user = user;
 }
}
```





Signal vs. RxJS (BehaviorSubject, Observable)

Signals for state, RxJS for streams

- Signals hold synchronous state, Observables handle async streams/events. They solve overlapping but distinct problems

Direct value access

- signal() returns value on demand. A BehaviorSubject exposes current value via .value, but deriving new values (e.g. with pipe(map)) yields an Observable that must be subscribed to

Derived values

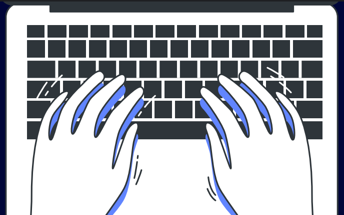
- Signals can be read anywhere; Observables often need async pipes or manual subscription and unsubscription.

Interoperability: toSignal, toObservable

toSignal(observable)

- Signals hold synchronous state, Observables handle async streams/events. They solve overlapping but distinct problems
- Behaves like async pipe but usable in code.
- Subscribes automatically; unsubscribes when context (component/service) is destroyed

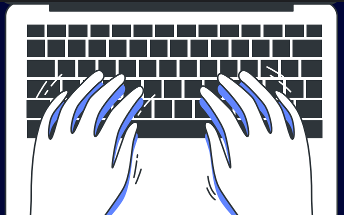
```
import { toSignal } from '@angular/core/rxjs-interop';
const counter$ = interval(1000);
const counter = toSignal(counter$, { initialValue: 0 });
```



toObservable(signal)

- Converts a signal to an RxJS Observable
- Under the hood it uses an effect and emits new values when the signal changes.
- Emits synchronously the first value (if available) and then only after the signal stabilizes.

```
import { toObservable } from '@angular/core/rxjs-interop';
const query = this.searchService.query;  // a Signal<string>
const query$ = toObservable(query);
query$.pipe(debounceTime(300)).subscribe(q => this.search(q));
```



When to use Signals vs RxJS

➔ Signals for state, RxJS for events

➔ Signals are synchronous

- They always have the current value, making them ideal for local state.

➔ RxJS for async complexity

- Observables provide operators for debouncing, throttling, error handling, etc.

➔ Guiding principle

- If you need a stream of values or handling asynchronous sequence, use RxJS. If you need to store or derive state reactively, signals are simpler.

QUIZ

Exercise

Change Detection in Angular

Change Detection in Angular

What is Change Detection?

- Synchronizes component state with the DOM
- Angular checks for changes and updates view
- Happens automatically after events

When should Angular update the screen?

1. Traditional: Zone.js + Default/OnPush
2. Modern: Signals

The Traditional Problem: Zone.js

How Zone.js Works:

1. ALL Async Event are intercepted
   a) Events (click, input, mouse…)
   b) Timers (setTimeout, setInterval)
   c) Promises & HTTP requests
2. Triggers change detection on ENTIRE app

The Problem:

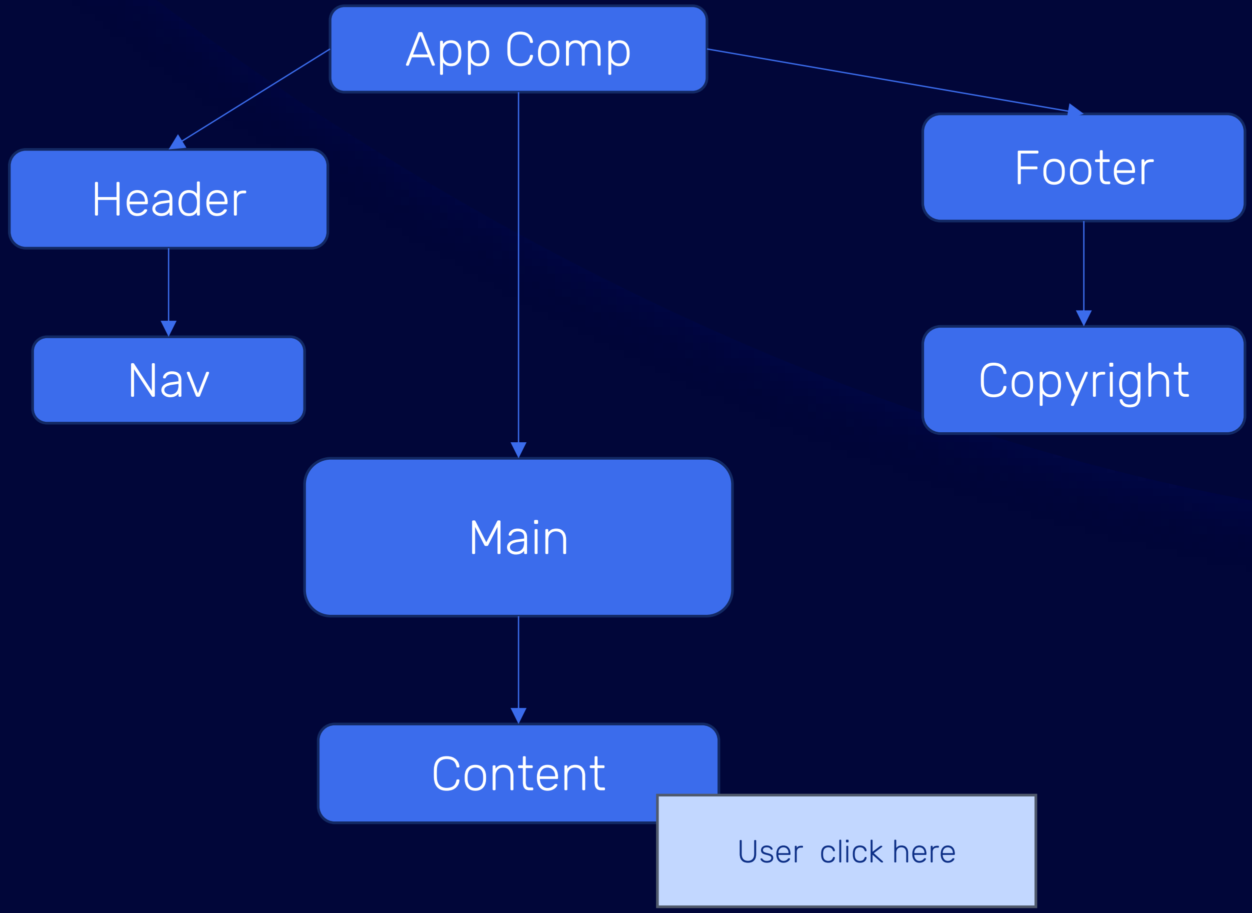
❌ User clicks ONE button

❌ Angular checks EVERY component

❌ Even components with no changes

❌ Performance degrades as app grows

Default Change Detection Strategy



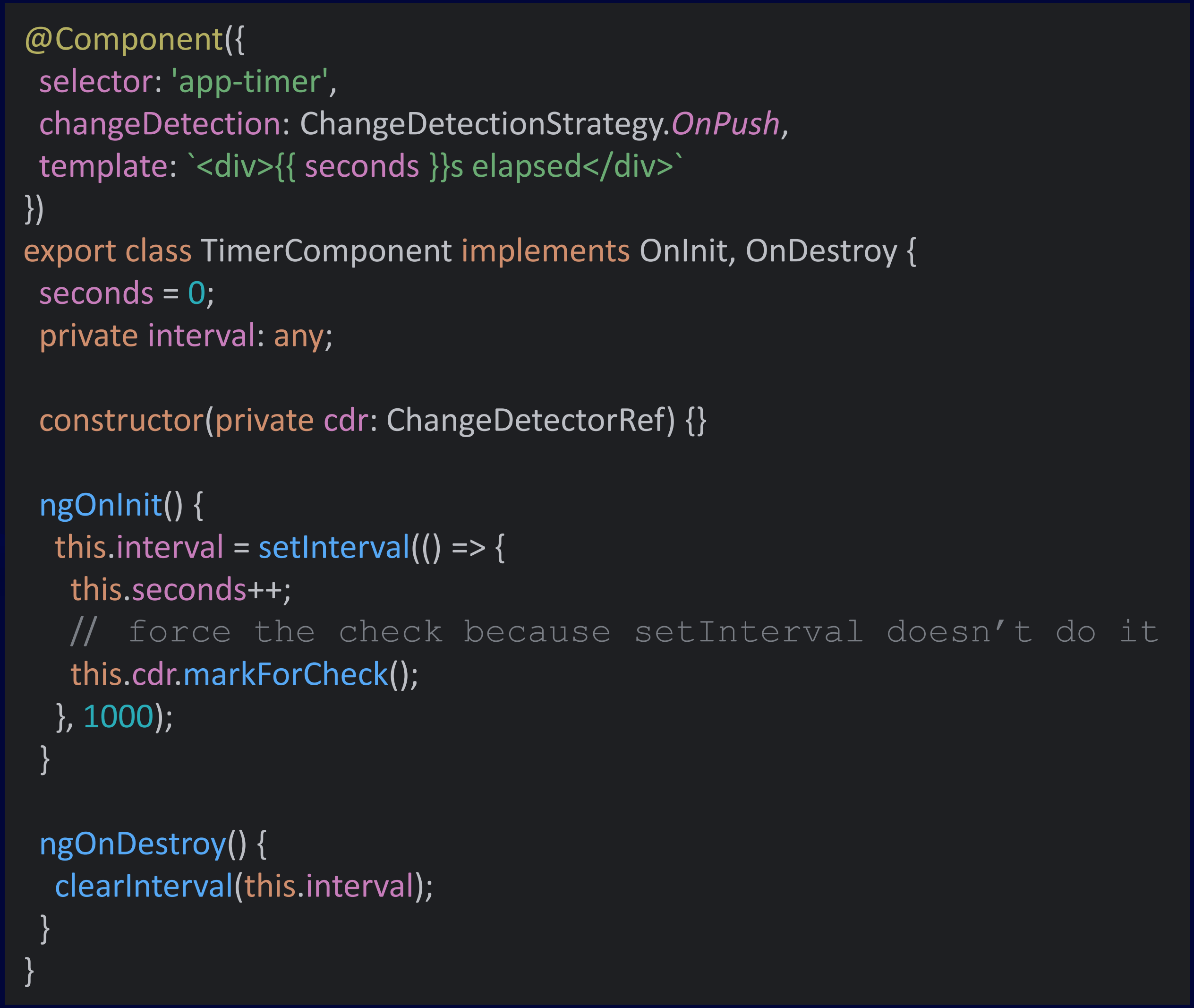
What Happens:

- 1. User clicks button
- 2. Zone.js intercepts event
- 3. Angular checks ALL components (top to bottom)
- 4. Updates DOM where needed

Solution 1 - OnPush Strategy

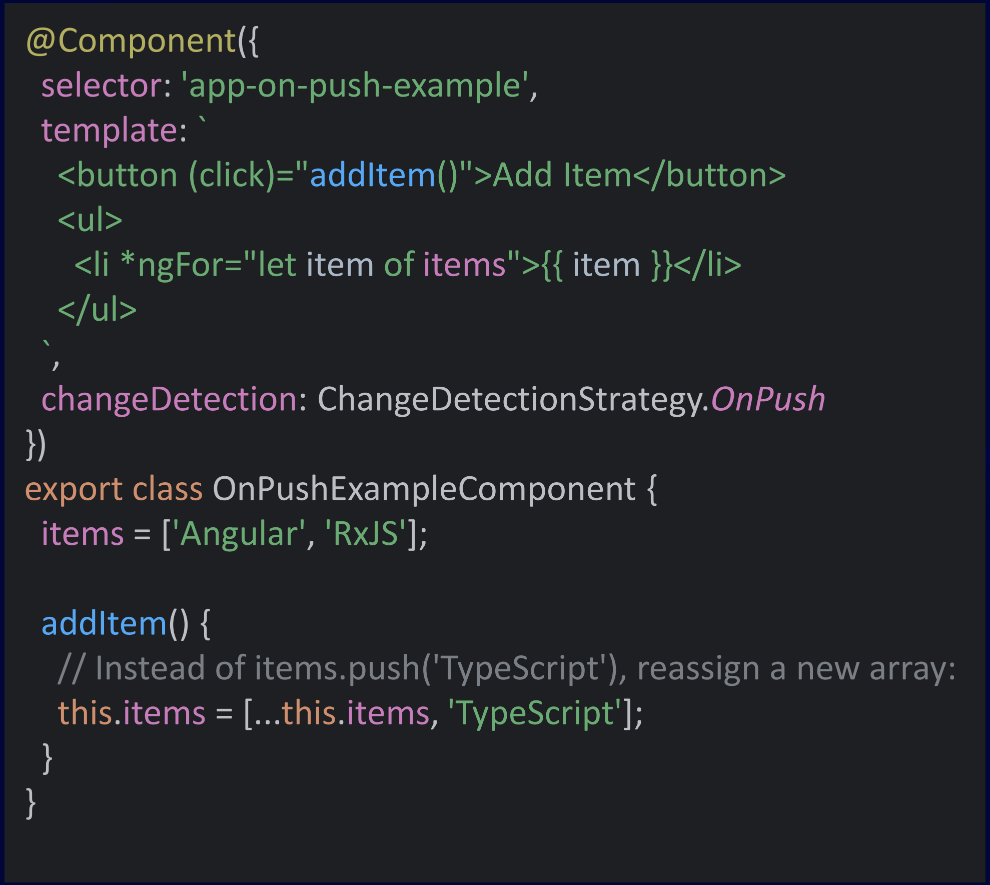
The Component checked ONLY when:

- ✓ @Input() reference changes (new object/array)
- ✓ Event in component or its children
- ✓ Async pipe emits new value
- ✓ Manual trigger (markForCheck)



```
@Component({
  selector: 'app-timer',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `<div>{{ seconds }}s elapsed</div>`
})
export class TimerComponent implements OnInit, OnDestroy {
  seconds = 0;
  private interval: any;

  constructor(private cdr: ChangeDetectorRef) {}

  ngOnInit() {
    this.interval = setInterval(() => {
      this.seconds++;
      //  force the check because setInterval doesn’t do it
      this.cdr.markForCheck();
    }, 1000);
  }

  ngOnDestroy() {
    clearInterval(this.interval);
  }
}
```

OnPush Code Example



```
@Component({
  selector: 'app-on-push-example',
  template: `
    <button (click)="addItem()">Add Item</button>
    <ul>
      <li *ngFor="let item of items">{{ item }}</li>
    </ul>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class OnPushExampleComponent {
  items = ['Angular', 'RxJS'];

  addItem() {
    // Instead of items.push('TypeScript'), reassign a new array:
    this.items = [...this.items, 'TypeScript'];
  }
}
```

Critical Rule with OnPush:

❌ items.push(newItem)   // Mutation - not detected!

✅ items = [...items, newItem] // New reference - detected!

Why?

OnPush compares references, not values

Same reference = "nothing changed" (even if content different)

New reference = "something changed" → check component

The Problem with OnPush

1. Easy to Forget Immutability

- One mutation bug breaks everything
- Hard to debug "why didn't it update?"

2. Team Discipline Required

- Everyone must understand the rules
- Code reviews must catch violations

3. Still Not Optimal

- Still checks entire component
- Just less frequently

Solution 2 - Signals

The Signal Approach:

- Fine-grained reactivity at expression level
- Angular knows EXACTLY what to update
- No Zone.js overhead
- No global change detection

```
cart = signal([item1, item2]);
total = computed(() =>
  cart().reduce((sum, item) => sum + item.price, 0)
);
```

```
<div>Items: {{ cart().length }}</div>
<div>Total: {{ total() }}</div>
```

When cart changes:

- Both expressions update automatically
- No change detection cycle
- No checking other components
- Instant, precise updates

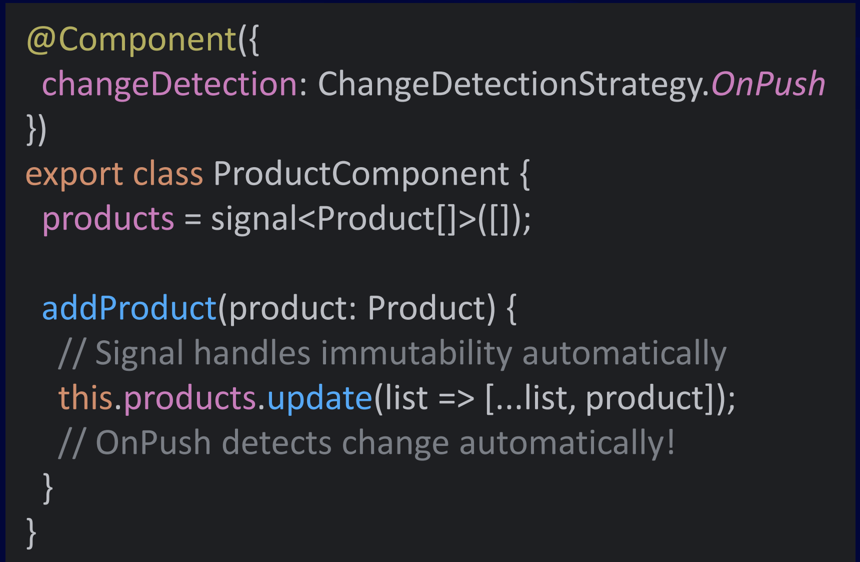
OnPush + Signals = Perfect Match

Traditional OnPush Challenge:

- Must remember immutability rules
- Easy to forget and create bugs
- Manual change detection sometimes needed

Signals Solution:

- Signals are immutable by design
- .set() and .update() create new references automatically
- Change detection "just works"



```
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ProductComponent {
  products = signal<Product[]>([]);

  addProduct(product: Product) {
    // Signal handles immutability automatically
    this.products.update(list => [...list, product]);
    // OnPush detects change automatically!
  }
}
```

Avoid Methods & Getters in Templates

Problem

- Calling a Method or Getter directly in the template can cause multiple calls per change detection cycle.
- Performance Hit: If the method or getter has expensive logic, it re-runs for every binding check.

```
❌ Bad - Method call
<p>Total: {{ calculateTotal() }}</p>
// Called 50+ times per second!
```

Angular calls calculateTotal() repeatedly whenever it checks this component.

```
✅ Good - Computed signal
total = computed(() =>
  this.items().reduce((sum, i) => sum + i.price, 0)
);

<p>Total: {{ total() }}</p>
// Calculated once, cached until items changes
```

Signals solve this automatically:

- Computed caches result
- Recalculates only when dependencies change
- No manual optimization needed

QUIZ

Error Handling Patterns

Error Handling in Angular – Overview

- Why Error Handling?: Prevent crashes, improve user experience

- Scope: Local vs. Global

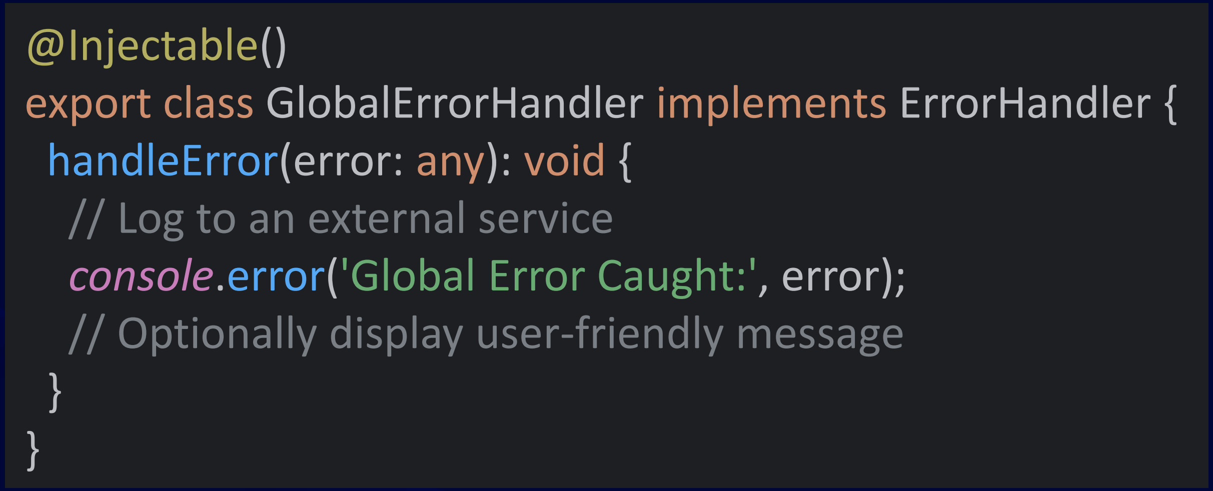
- Angular’s Built-in Mechanisms:
  - ErrorHandler
  - HttpInterceptor
  - RxJS Operators

Local Error Handling with RxJS Operators

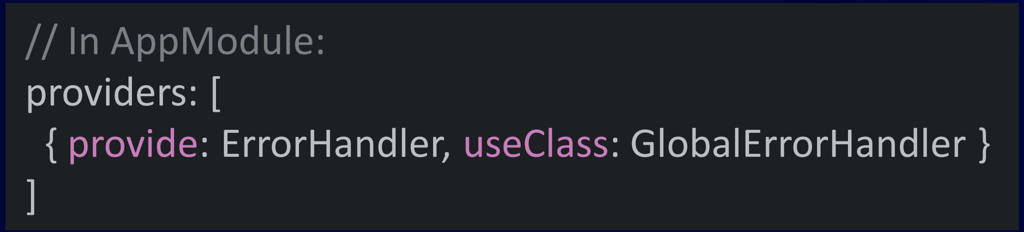
```
this.http.get('/api/data').pipe(
  retry(2),
  catchError((err) => {
    console.error('HTTP Error:', err);
    return of([]); // Fallback or empty result
  })
).subscribe(data => {
  this.myData = data;
});
```

- retry(n): Re-attempt an operation n times
- catchError: Transform or handle errors in the pipeline
- of([]): Return fallback Observable

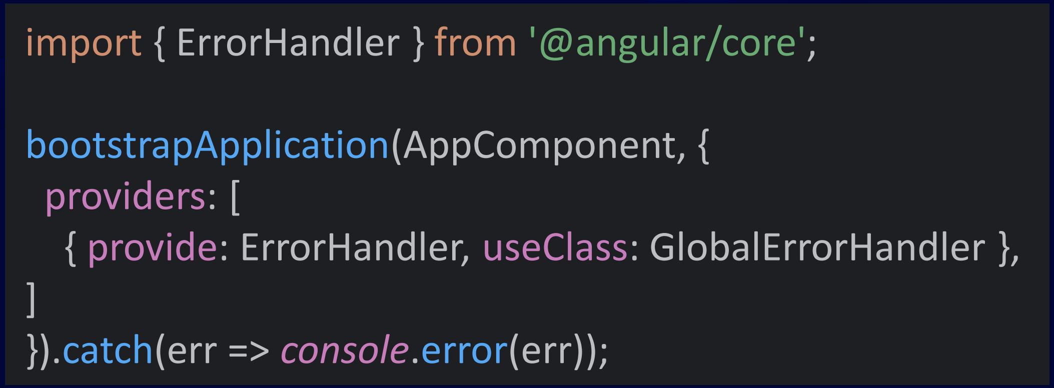
Global Error Handler (ErrorHandler)



```
@Injectable()
export class GlobalErrorHandler implements ErrorHandler {
  handleError(error: any): void {
    // Log to an external service
    console.error('Global Error Caught:', error);
    // Optionally display user-friendly message
  }
}
```

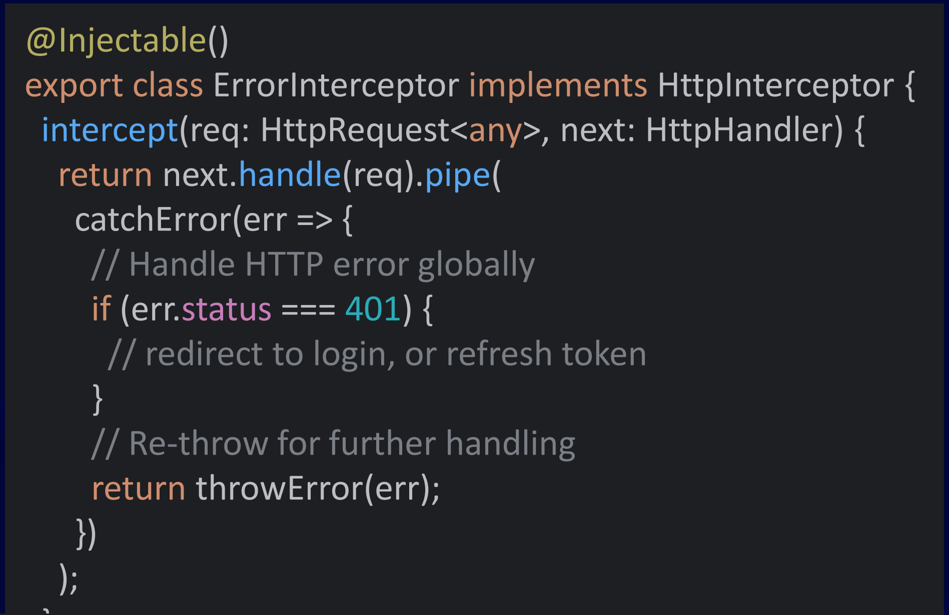


```
// In AppModule:
providers: [
  { provide: ErrorHandler, useClass: GlobalErrorHandler }
]
```

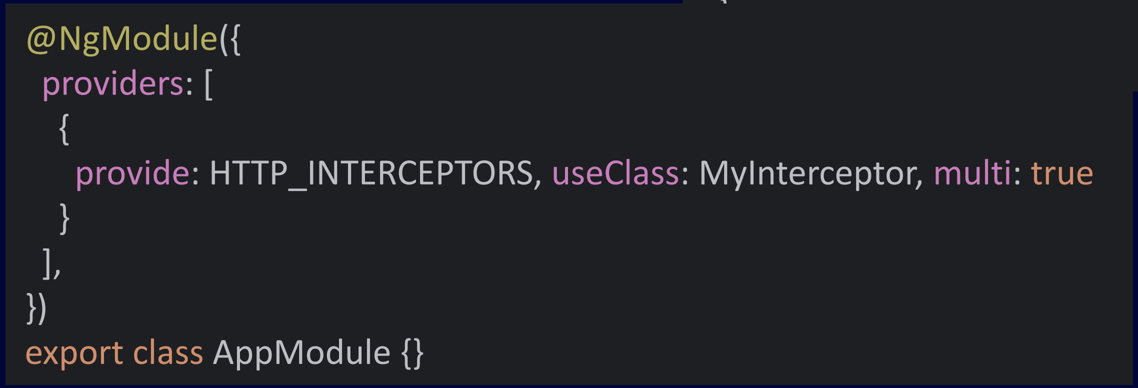


```
import { ErrorHandler } from '@angular/core';

bootstrapApplication(AppComponent, {
  providers: [
    { provide: ErrorHandler, useClass: GlobalErrorHandler },
]
}).catch(err => console.error(err));
```

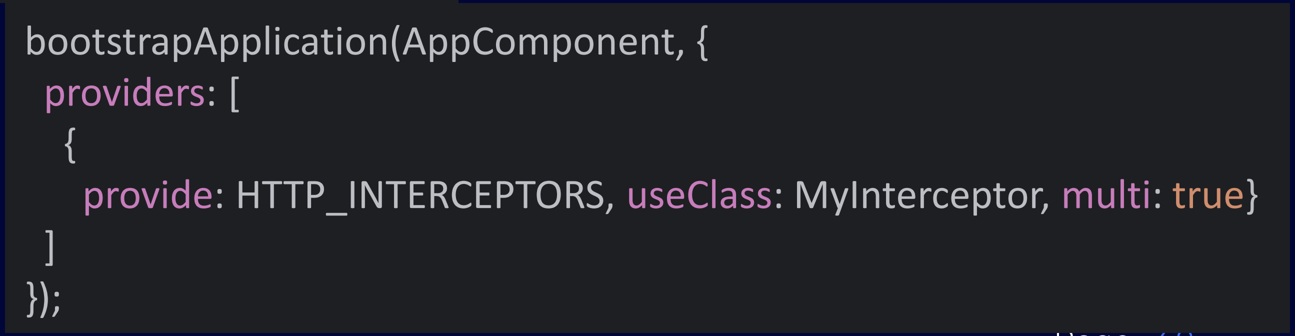
Http Interceptor(ErrorHandler)



```
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
 intercept(req: HttpRequest<any>, next: HttpHandler) {
  return next.handle(req).pipe(
   catchError(err => {
    // Handle HTTP error globally
    if (err.status === 401) {
     // redirect to login, or refresh token
    }
    // Re-throw for further handling
    return throwError(err);
   })
  );
```



```
@NgModule({
 providers: [
  {
   provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true
  }
 ],
})
export class AppModule {}
```



```
bootstrapApplication(AppComponent, {
 providers: [
  {
   provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true}
 ]
});
```

Service-Based State Management

What is State Management?

Definition:

- Managing data that multiple components need to access and modify

How do two components share the same data?

1. Input/Output chain (messy for distant components)
2. Service-based state (centralized)
   - Traditional: RxJS Subjects
   - Modern: Signal Stores

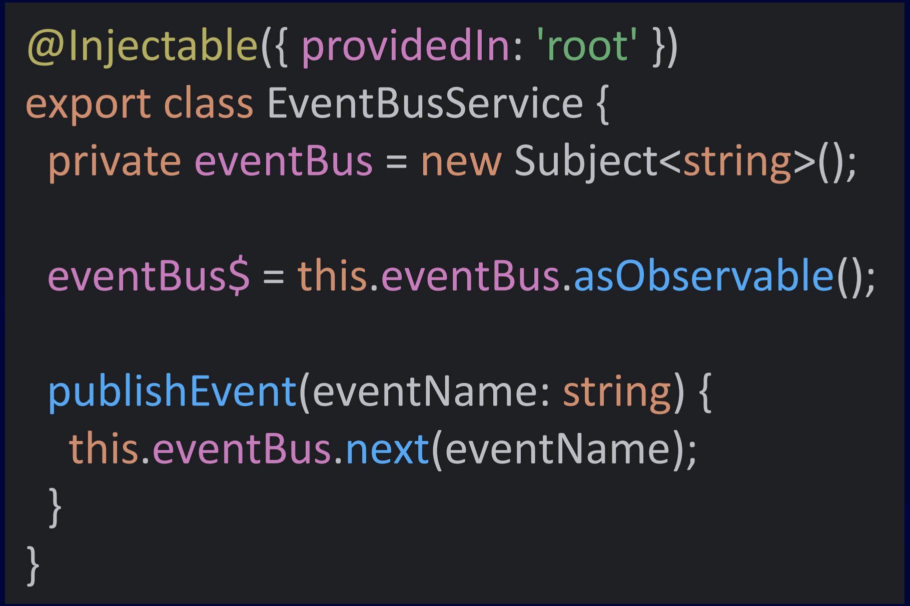
Traditional Approach: RxJS Subjects

- Subject
  - Basic event bus
  - No initial value
  - New subscribers miss past emissions
  - Use: Events, notifications

- BehaviorSubject
  - Holds a current value, instantly given to new subscribers
  - Common for shared state (e.g., a user’s profile or a task list)
  - Use: State management (most common)

- ReplaySubject
  - Replays a specified number of past values to new subscribers
  - Use: History tracking, caching

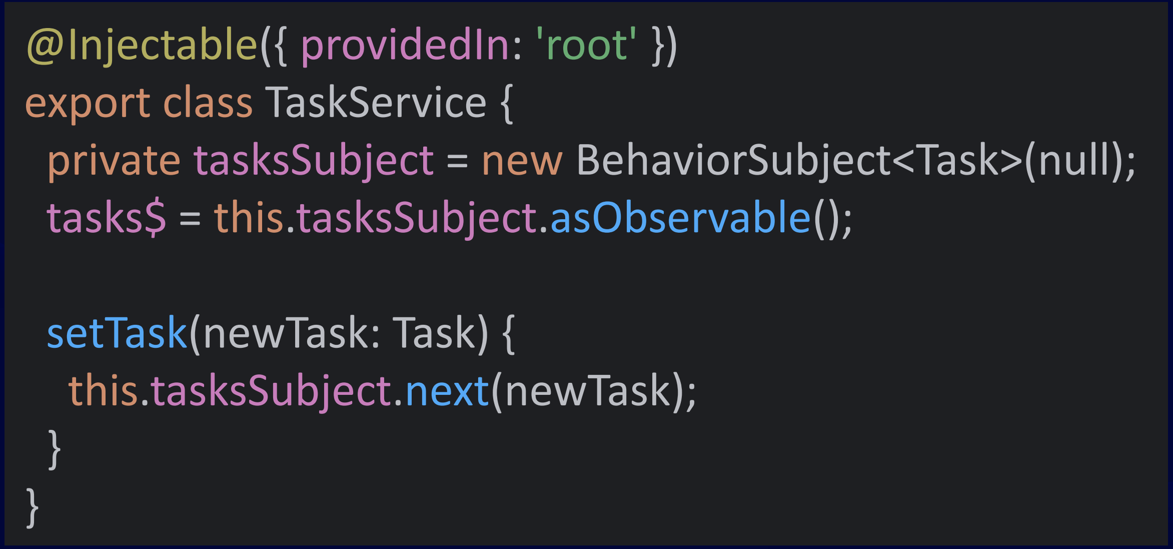
Subject Example - Broadcasting Events



```
@Injectable({ providedIn: 'root' })
export class EventBusService {
  private eventBus = new Subject<string>();

  eventBus$ = this.eventBus.asObservable();

  publishEvent(eventName: string) {
    this.eventBus.next(eventName);
  }
}
```

- No stored state
- Each subscriber only sees new events

BehaviorSubject Example - Shared Task State



```
@Injectable({ providedIn: 'root' })
export class TaskService {
  private tasksSubject = new BehaviorSubject<Task>(null);
  tasks$ = this.tasksSubject.asObservable();

  setTask(newTask: Task) {
    this.tasksSubject.next(newTask);
  }
}
```

- Has a Default value
- New subscribers get the current task

ReplaySubject Example - History Tracking

```
@Injectable({ providedIn: 'root' })
export class LogService {
  private logSubject = new ReplaySubject<string>(3);
  logs$ = this.logSubject.asObservable();

  addLog(entry: string) {
    this.logSubject.next(entry);
  }
}
```

```
// Early: Emit logs
logService.addLog('App started');
logService.addLog('User logged in');
logService.addLog('Data loaded');

// Late subscriber gets last 3 logs
ngOnInit() {
  this.logService.logs$.subscribe(logs => {
    console.log(logs); // Receives all 3 past logs!
  });
}
```

- Stores the last 3 entries
- New subscribers get recent history

Modern Approach - Signal Stores

The Problem with RxJS for State:

- ❌ Manual subscription management
- ❌ Async pipe or takeUntilDestroyed everywhere
- ❌ Boilerplate code
- ❌ Easy to create memory leaks
- ❌ Verbose immutability management

Signal Store Solution:

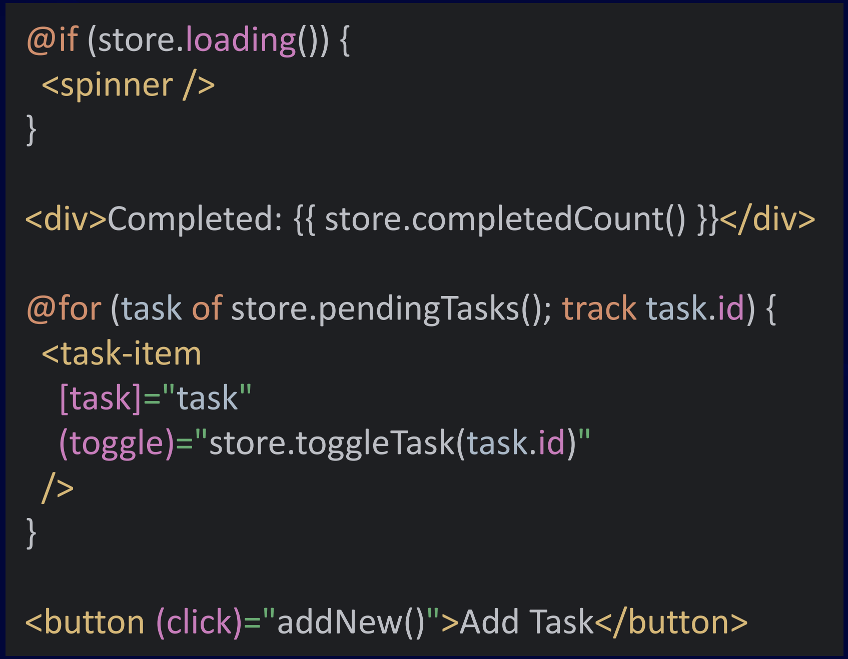
- ✅ No subscriptions needed
- ✅ Automatic reactivity
- ✅ Synchronous state access
- ✅ Immutability built-in
- ✅ Simpler, cleaner code

Signal Store Pattern

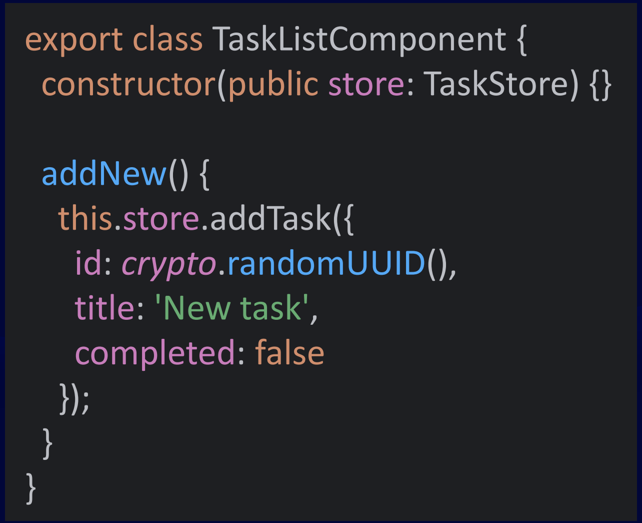
```
@Injectable({ providedIn: 'root' })
export class TaskStore {
  // 1. Private writable signals
  private _tasks = signal<Task[]>([]);
  private _loading = signal(false);

  // 2. Public readonly signals
  readonly tasks = this._tasks.asReadonly();
  readonly loading = this._loading.asReadonly();

  // 3. Computed derived state
  readonly completedCount = computed(() =>
    this.tasks().filter(t => t.completed).length
  );

 readonly pendingTasks = computed(() =>
    this.tasks().filter(t => !t.completed)
  );
```

```
  // 4. Actions (methods that modify state)
  addTask(task: Task) {
    this._tasks.update(tasks => [...tasks, task]);
  }

  toggleTask(id: string) {
    this._tasks.update(tasks =>
      tasks.map(t => t.id === id
        ? {...t, completed: !t.completed}
        : t
      )
    );
  }

  async loadTasks() {
    this._loading.set(true);
    const tasks = await this.http.get<Task[]>(...);
    this._tasks.set(tasks);
    this._loading.set(false);
  }
}
```

Using Signal Store in Components



```
@if (store.loading()) {
  <spinner />
}

<div>Completed: {{ store.completedCount() }}</div>

@for (task of store.pendingTasks(); track task.id) {
  <task-item
    [task]="task"
    (toggle)="store.toggleTask(task.id)"
  />
}

<button (click)="addNew()">Add Task</button>
```



```
export class TaskListComponent {
  constructor(public store: TaskStore) {}

  addNew() {
    this.store.addTask({
      id: crypto.randomUUID(),
      title: 'New task',
      completed: false
    });
  }
}
```

Code quality

Angular Style Guide

- Key Conventions:
    - File Naming: Use kebab-case for file names (e.g., my-component.component.ts).
- Naming Conventions
    - Variables and Functions:
        - Use camelCase for variables and functions.
    - Classes and Interfaces:
        - Use PascalCase for class and interface names.
    - Constants:
        - Use UPPER_CASE with underscores.

Clean Code Principles

- DRY (Don't Repeat Yourself)
  - Definition: Avoid code duplication by abstracting common functionality.
  - Strategies:
    - Use functions and services for reusable logic.
    - Create shared modules and components.
- KISS (Keep It Simple, Stupid)
  - Definition: Simplicity should be a key goal; avoid unnecessary complexity.
  - Approach:
    - Write clear and straightforward code.
    - Break down complex problems into simpler parts.
- YAGNI (You Aren't Gonna Need It)
  - Definition: Don't add functionality until it's necessary.
  - Application:
    - Focus on current requirements.
    - Avoid speculative features.

Performance Considerations

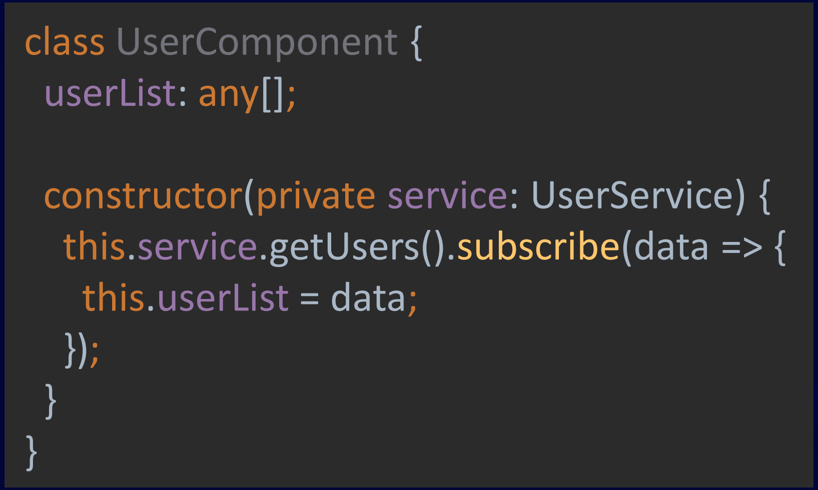
- Optimizing Change Detection
  - OnPush Change Detection Strategy:
    - Reduces unnecessary checks.
    - Improves performance in components with immutable data.
  - Detaching Change Detectors:
    - Manually control when change detection runs.
    - Useful in performance-critical components.
- Avoiding Memory Leaks
  - Unsubscribe from Observables:
    - Prevents memory leaks by releasing resources.
    - Use async pipe.
  - Proper Use of ngOnDestroy:
    - Clean up subscriptions and resources.

Refactoring Techniques

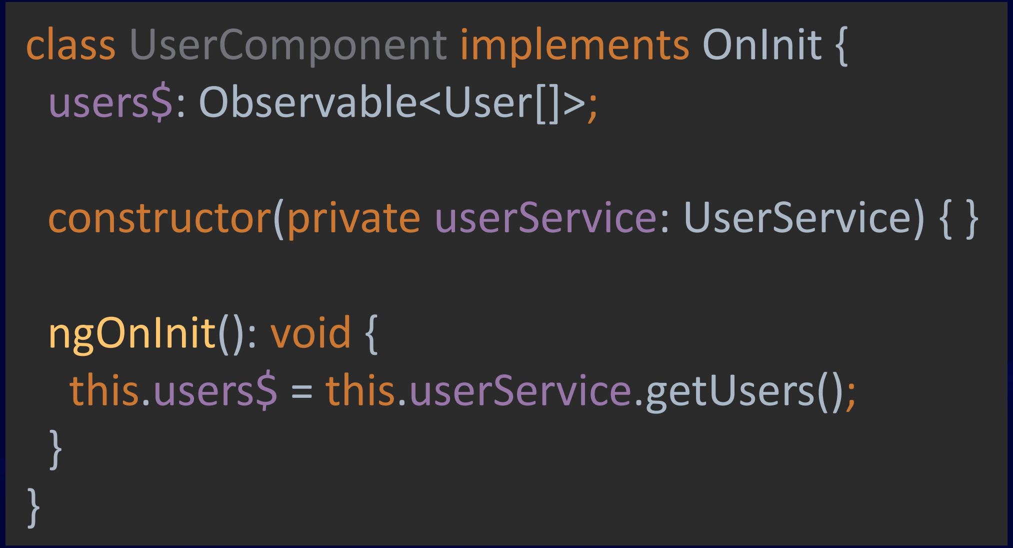
- Identifying Code Smells
  - Common Code Smells:
    - **Long Methods:** Methods that are too long and complex.
    - **Duplicated Code:** Similar code blocks scattered throughout the codebase.
    - **Large Classes:** Classes that handle too many responsibilities.
  - Impact:
    - Makes code harder to understand.
    - Increases the risk of bugs.

- Refactoring Patterns
  - Extract Method:
    - Break down large methods into smaller, reusable ones.
  - Rename for Clarity:
    - Use meaningful names for variables, methods, and classes.
  - Simplify Conditionals:
    - Refactor complex conditional logic into simpler structures.

Exemple 1



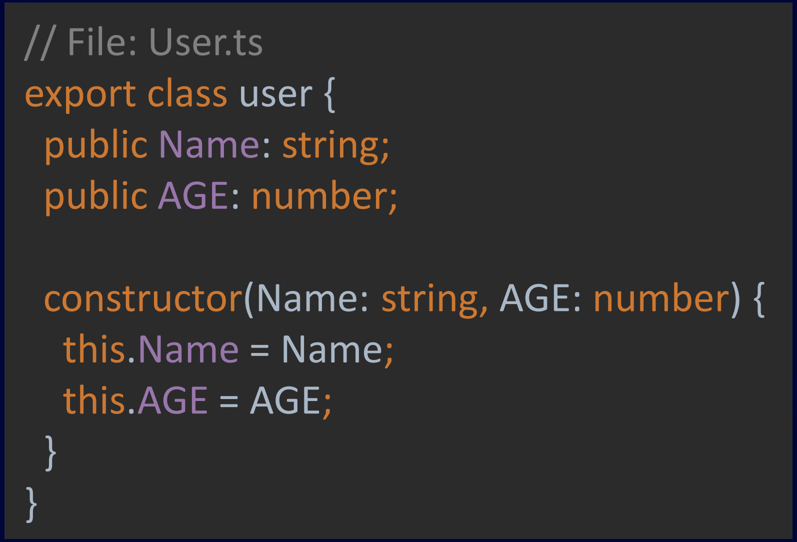
```
class UserComponent {
 userList: any[];

 constructor(private service: UserService) {
  this.service.getUsers().subscribe(data => {
   this.userList = data;
  });
 }
}
```



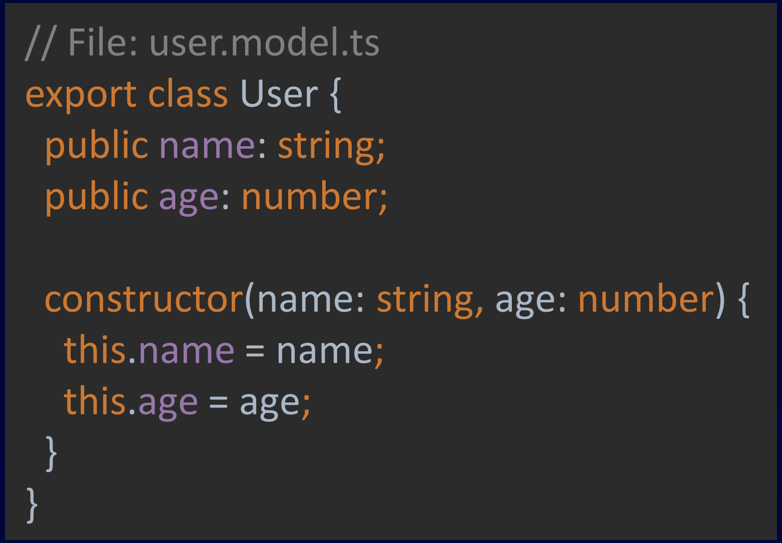
```
class UserComponent implements OnInit {
 users$: Observable<User[]>;

 constructor(private userService: UserService) { }

 ngOnInit(): void {
  this.users$ = this.userService.getUsers();
 }
}
```

- Explanation:
  - Uses Strong Typing: User[] instead of any[].
  - Follows Clean Code Principles: Uses OnInit, avoids direct subscription in the constructor.

Exemple 2



```
// File: User.ts
export class user {
  public Name: string;
  public AGE: number;

  constructor(Name: string, AGE: number) {
    this.Name = Name;
    this.AGE = AGE;
  }
}
```



```
// File: user.model.ts
export class User {
  public name: string;
  public age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
}
```

- Principle: Follow Consistent Naming Conventions
  - Classes and Interfaces: Use PascalCase.
  - Variables and Methods: Use camelCase.
  - File Names: Use kebab-case and include the type suffix (e.g., .model.ts).

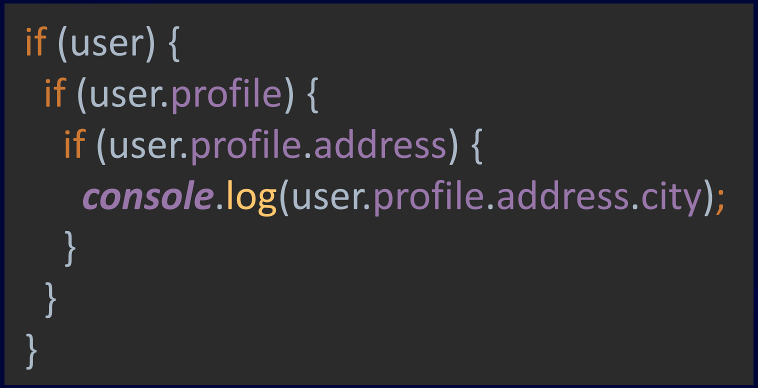
Exemple 3

```
if (user.roleId === 1) {
  // Admin privileges
}
```

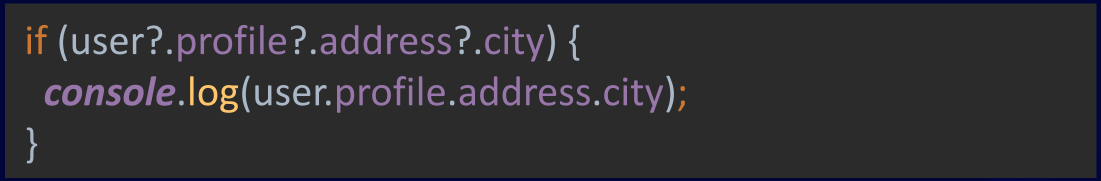
```
// File: user.model.ts
export class User {
  public name: string;
  publicenum UserRole {
  Admin = 1,
  Editor,
  Viewer,
}

if (user.roleId === UserRole.Admin) {
  // Admin privileges
}
 age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
}
```

- Principle: Use Constants and Enums Instead of Magic Numbers
  - Magic Numbers: Hard-coded numbers or strings with unexplained meanings.
  - Solution: Define constants or enums to represent these values.

Exemple 4



```
if (user) {
 if (user.profile) {
  if (user.profile.address) {
   console.log(user.profile.address.city);
  }
 }
}
```



```
if (user?.profile?.address?.city) {
 console.log(user.profile.address.city);
}
```

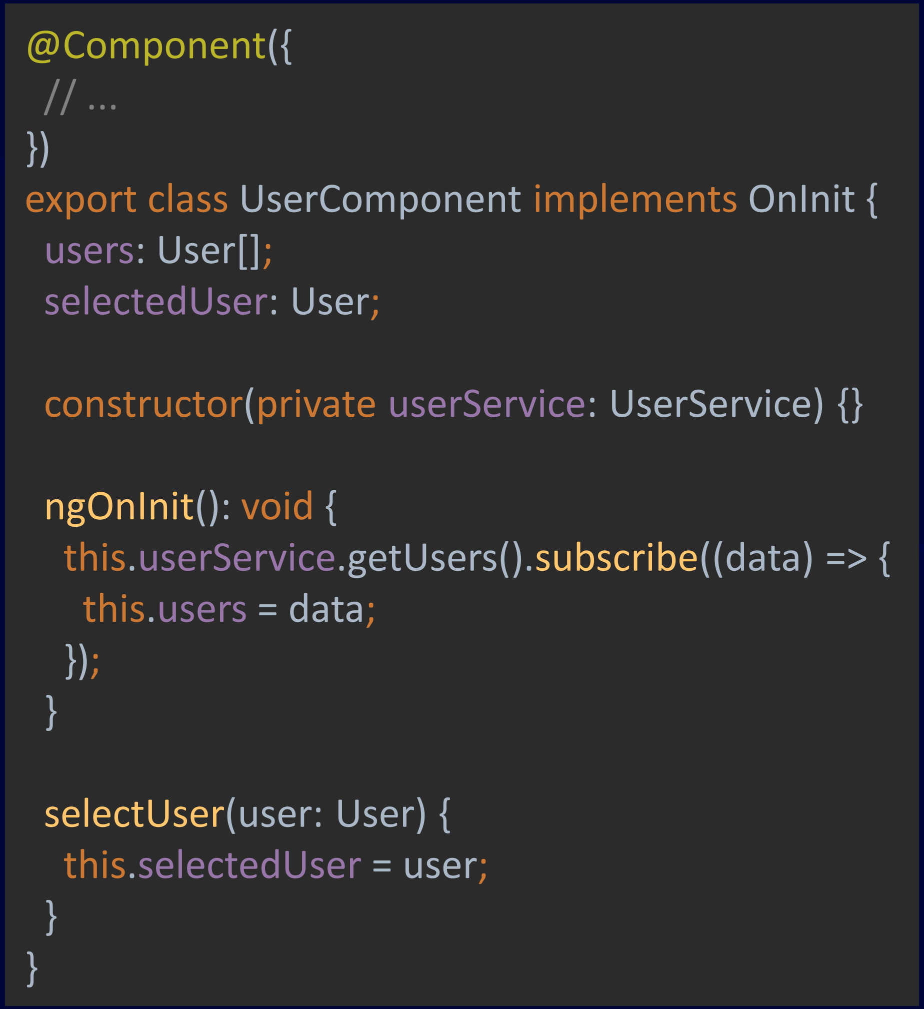
- Principle: Simplify Code with Optional Chaining
  - Optional Chaining (?.) reduces the need for multiple nested if statements.
  - Simplifies code and improves readability.

Exemple 5 (1/2)

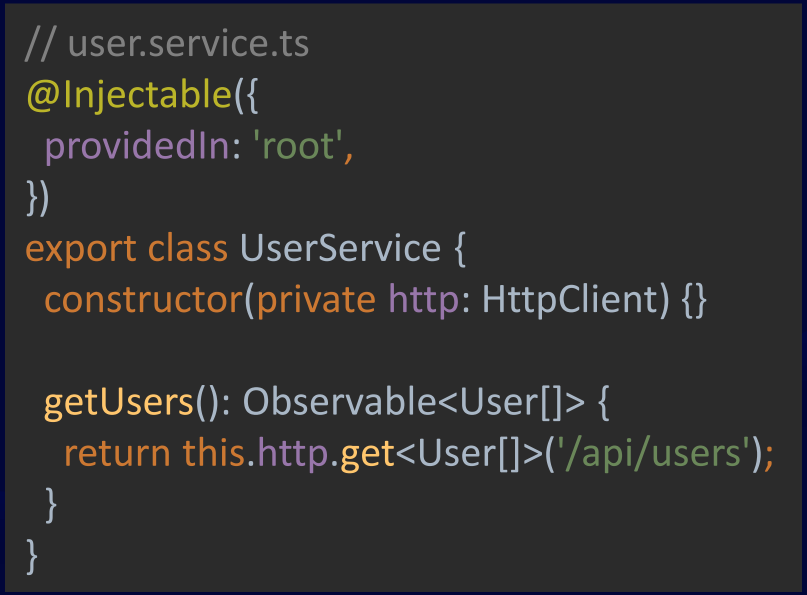
```
@Component({
 // ...
})
export class UserComponent {
 users: User[];
 selectedUser: User;

 constructor(private http: HttpClient) {
  this.http.get<User[]>('/api/users').subscribe((data) => {
   this.users = data;
  });
 }

 selectUser(user: User) {
  this.selectedUser = user;
 }
}
```

- Principle: Separate Concerns by Using Services
- Components should focus on the view and user interaction.
- Services handle data fetching and business logic.

Exemple 5 (2/2)



```
@Component({
  // ...
})
export class UserComponent implements OnInit {
  users: User[];
  selectedUser: User;

  constructor(private userService: UserService) {}

  ngOnInit(): void {
    this.userService.getUsers().subscribe((data) => {
      this.users = data;
    });
  }

  selectUser(user: User) {
    this.selectedUser = user;
  }
}
```



```
// user.service.ts
@Injectable({
  providedIn: 'root',
})
export class UserService {
  constructor(private http: HttpClient) {}

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>('/api/users');
  }
}
```

Exemple 6

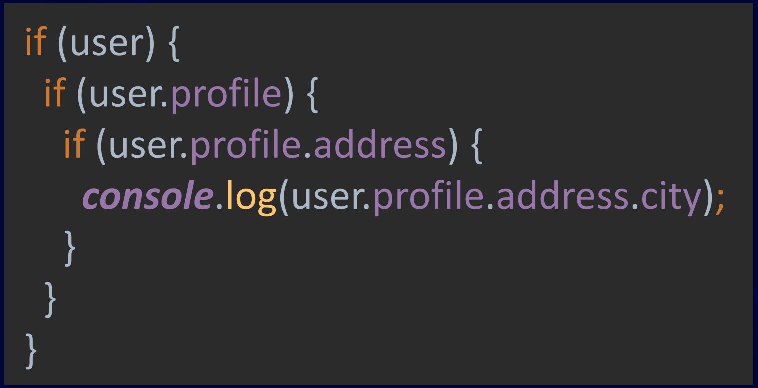
```html
<!-- component.html -->
<div style="color: red; font-size: 20px;">
  {{ message }}
</div>
```

```html
<!-- component.html -->
<div class="message">
  {{ message }}
</div>
```

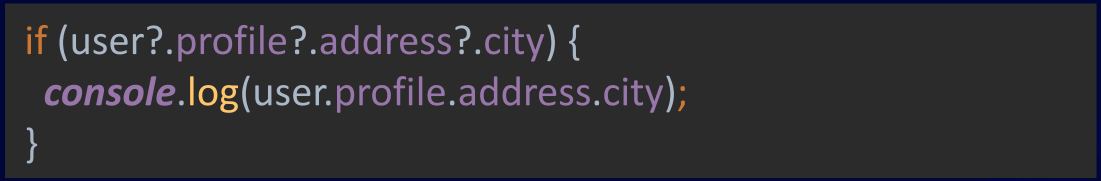
```css
/* component.css */
.message {
  color: red;
  font-size: 20px;
}
```

- Principle: Use External Stylesheets and Templates
  - Separate style and markup from logic.
  - Benefits:
    - Improves maintainability.
    - Enables reuse and theming.

Exemple 7



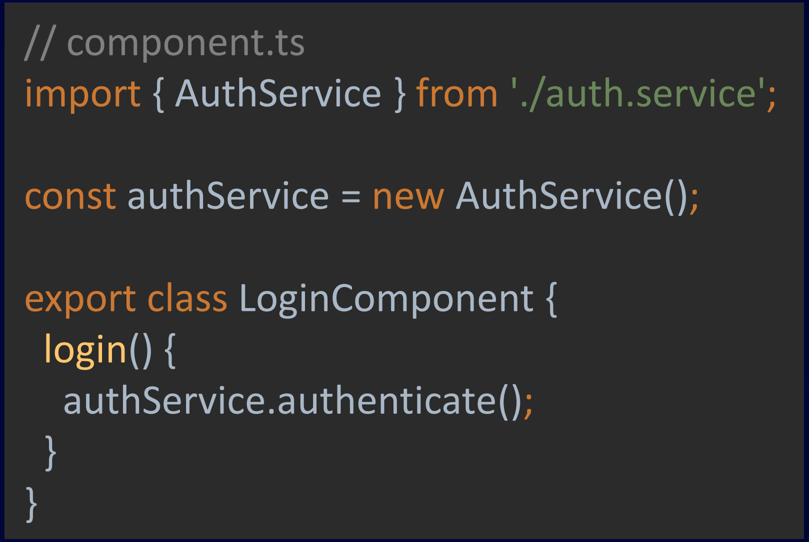
```
if (user) {
  if (user.profile) {
    if (user.profile.address) {
      console.log(user.profile.address.city);
    }
  }
}
```



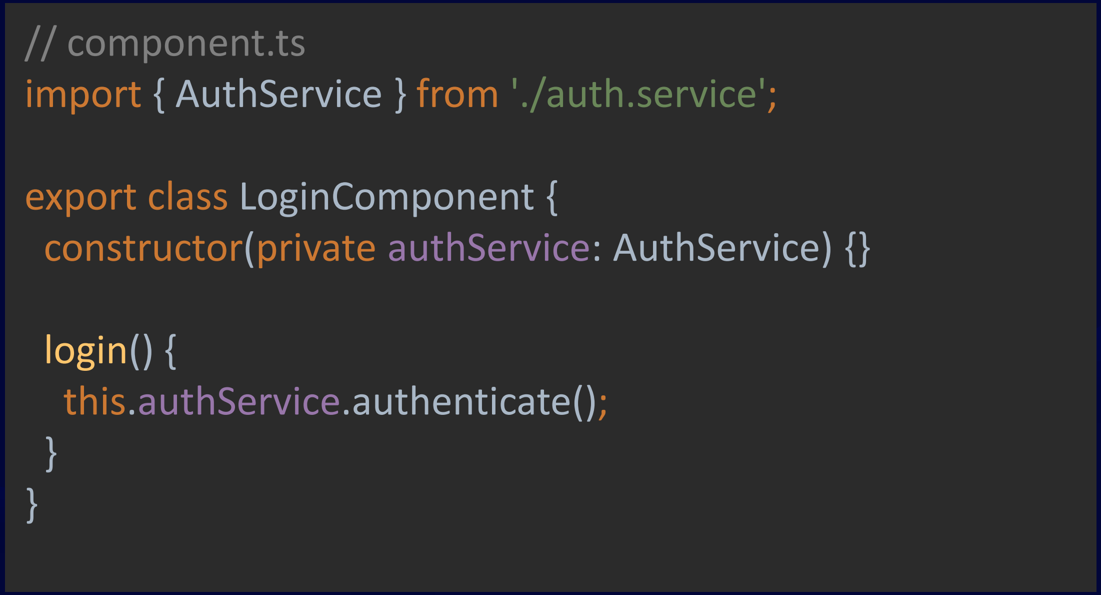
```
if (user?.profile?.address?.city) {
  console.log(user.profile.address.city);
}
```

- Principle: Simplify Code with Optional Chaining
  - Optional Chaining (?.) reduces the need for multiple nested if statements.
  - Simplifies code and improves readability.

Exemple 8



```
// component.ts
import { AuthService } from './auth.service';

const authService = new AuthService();

export class LoginComponent {
  login() {
    authService.authenticate();
  }
}
```



```
// component.ts
import { AuthService } from './auth.service';

export class LoginComponent {
  constructor(private authService: AuthService) {}

  login() {
    this.authService.authenticate();
  }
}
```

- Principle: Use Angular's Dependency Injection System
  - Inject services via the constructor.
  - Or using inject()

Exemple 9

```
// component.ts
import { AuthService } from './auth.service';

const authService = new AuthService();

export class LoginComponent {
  login() {
    authService.authenticate();
  }
}
```

```
// component.ts
import { AuthService } from './auth.service';

export class LoginComponent {
  constructor(private authService: AuthService) {}

  login() {
    this.authService.authenticate();
  }
}
```

- Principle: Use Angular's Dependency Injection System
  - Inject services via the constructor.
  - Or using inject()

Exemple 10

```
let data: any;
data = this.getData();
```

```
interface Data {
  id: number;
  value: string;
}

let data: Data;
data = this.getData();
```

- Principle: Use Strong Typing for Type Safety
  - Avoid any type unless absolutely necessary.
  - Define interfaces or types for data structures.

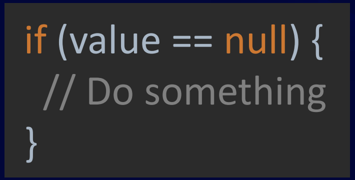
Exemple 11

```
<!-- component.html -->
<p>Welcome to our application!</p>
```

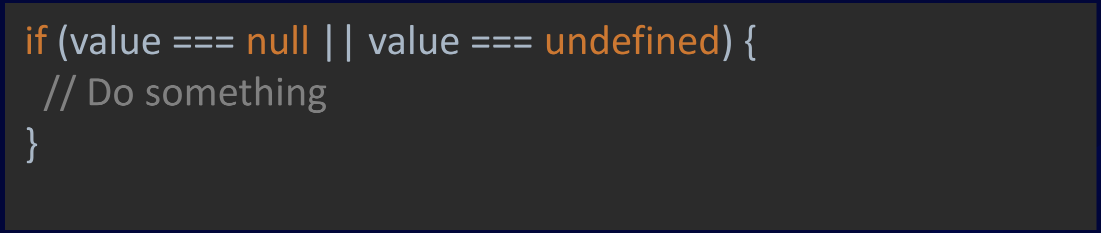
```
<!-- component.html -->
<p>{{ 'WELCOME_MESSAGE' | translate }}</p>


// en.json (translation file)
{
"WELCOME_MESSAGE": "Welcome to our application!"
}
```

- Principle: Use Internationalization (i18n) Practices
  - Utilize translation pipes or services for text.
  - Benefits:
    - Facilitates localization.
    - Improves scalability for multi-language support.

Exemple 12



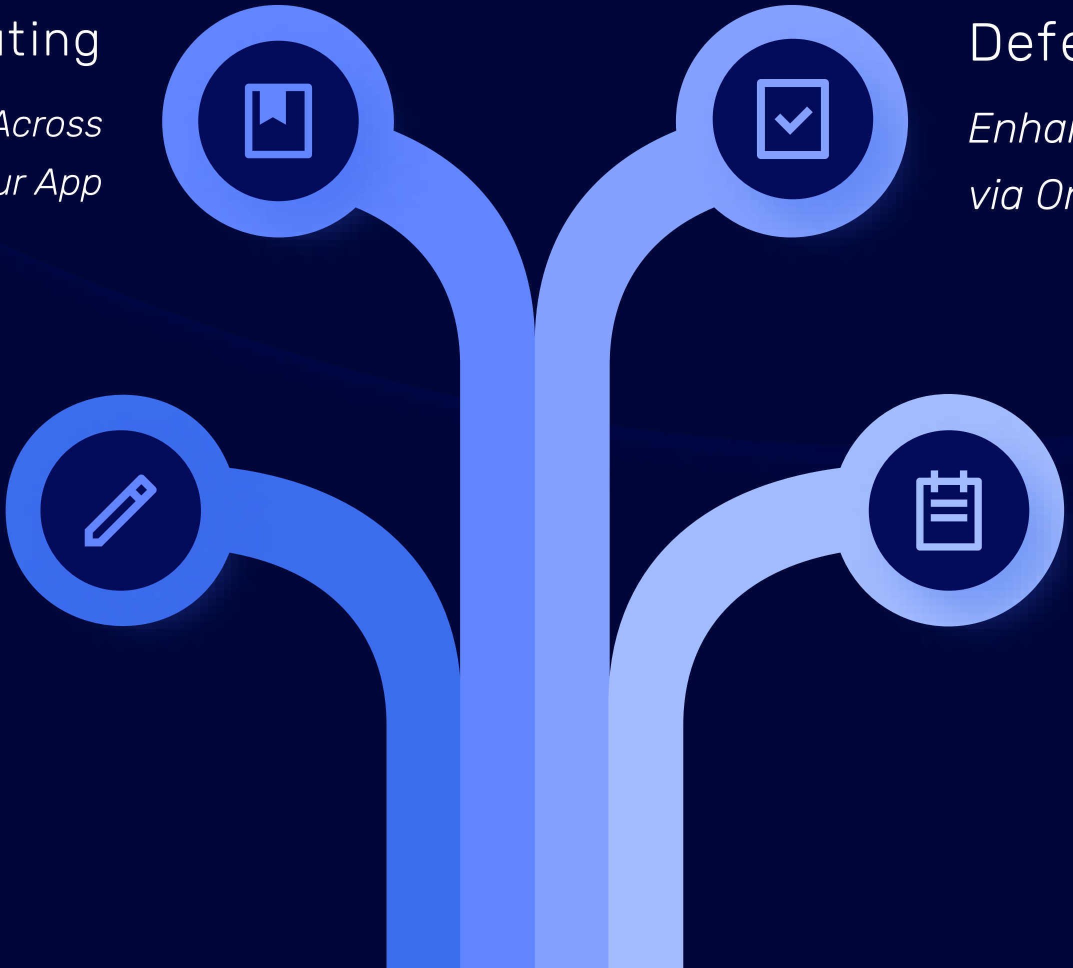
```
if (value == null) {
  // Do something
}
```



```
if (value === null || value === undefined) {
  // Do something
}
```

- Principle: Prefer Strict Equality (===) Over Abstract Equality (==)
  - Avoid using == and != as they perform type coercion.
  - Benefits:
    - Prevents unexpected behavior.
    - Improves code reliability.

Day 3: Optimizing & Scaling Your Angular Application



Advanced Routing

Structuring Navigation & Data Flows Across Your App

Deferrable Views

Enhancing Perceived Performance via On-Demand Rendering

Quality of Code

Maintaining Readability & Reliability in Larger Projects

Error Handling Patterns

Building Resilient Apps through Strategic Failure Management

Asynchronous Operations & Advanced RxJS Operators

Recap: RxJS & Angular Integration

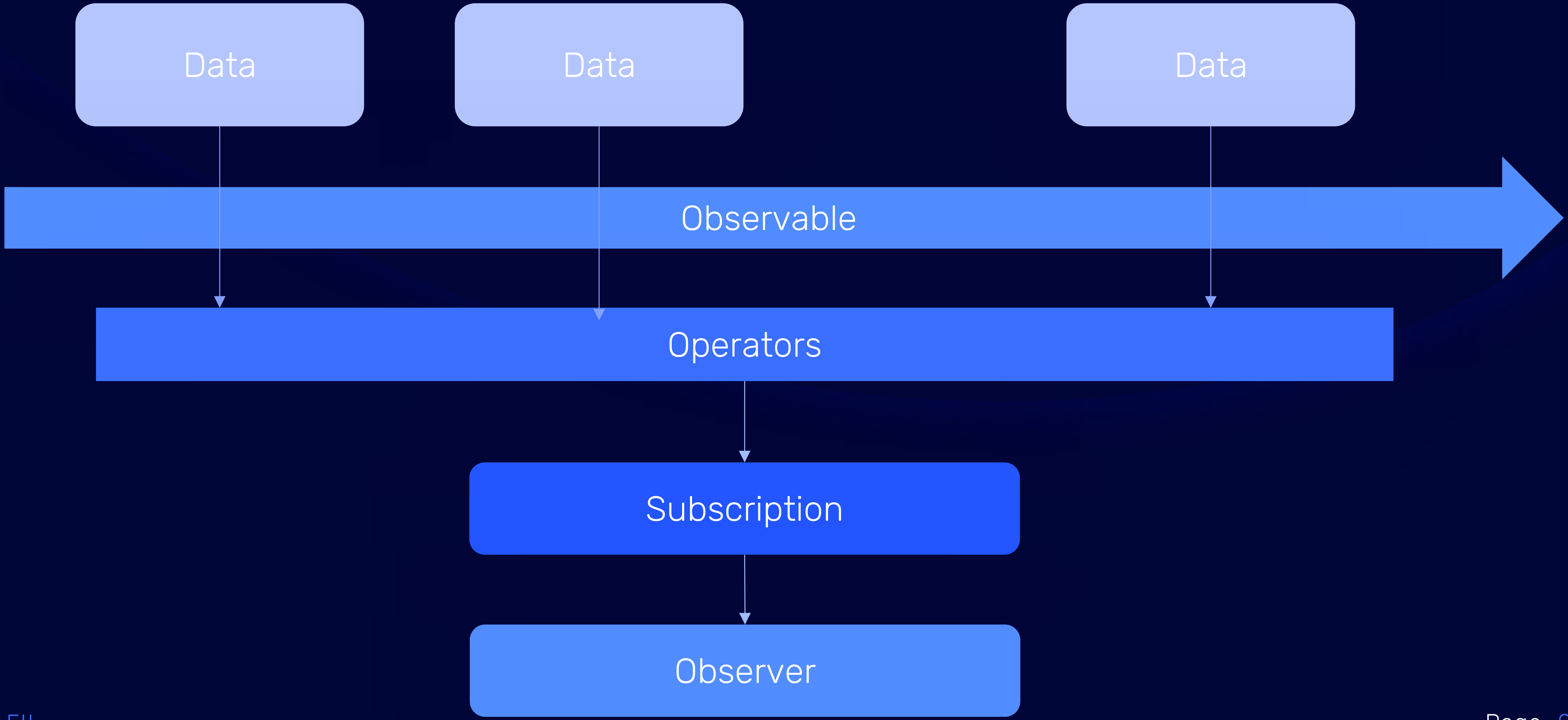
- RxJS: Library for reactive programming

- Angular uses Observables for:
  - HTTP calls
  - Reactive Forms
  - Event handling & more

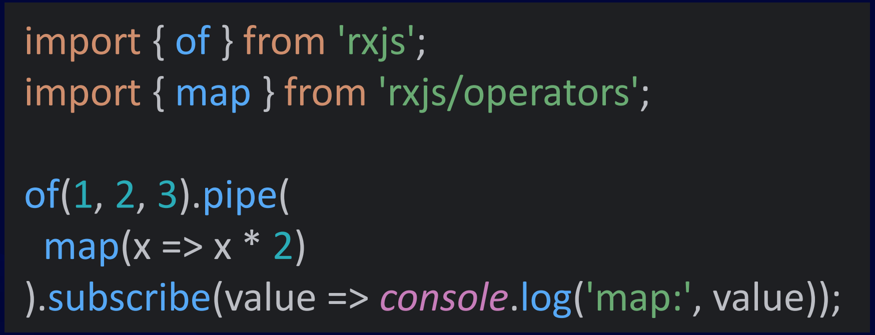
Definition of an Observable

- Observable is a lazy push collection of multiple values over time
- Subscribe to receive emissions (next/error/complete)
- Operators to transform/filter/combine streams
- Unsubscribe to stop receiving

Operators


Data
Data
Data
Observable
Operators
Subscription
Observer

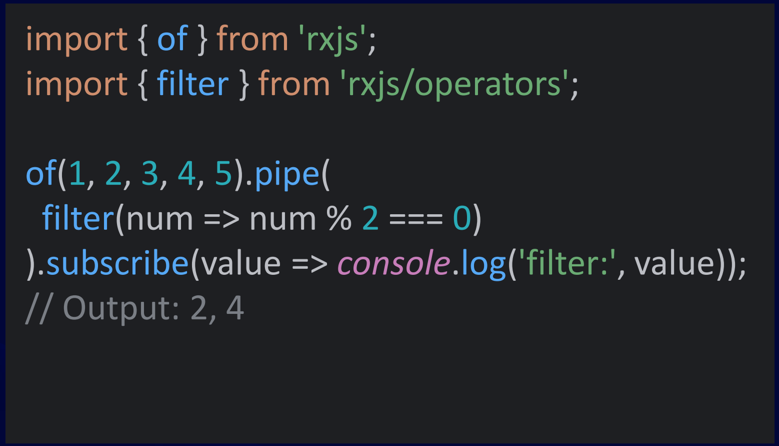
map



```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

of(1, 2, 3).pipe(
  map(x => x * 2)
).subscribe(value => console.log('map:', value));
```

- Transforms each emission
- Useful for calculations or object shaping

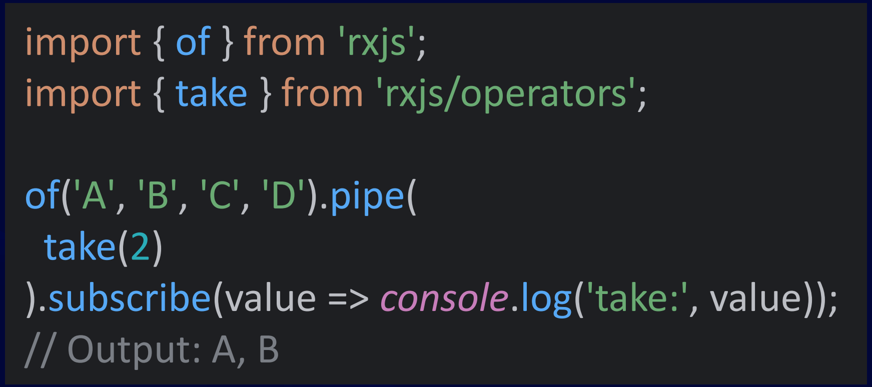
filter



```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';

of(1, 2, 3, 4, 5).pipe(
  filter(num => num % 2 === 0)
).subscribe(value => console.log('filter:', value));
// Output: 2, 4
```

- Filters out emissions that don’t match a condition

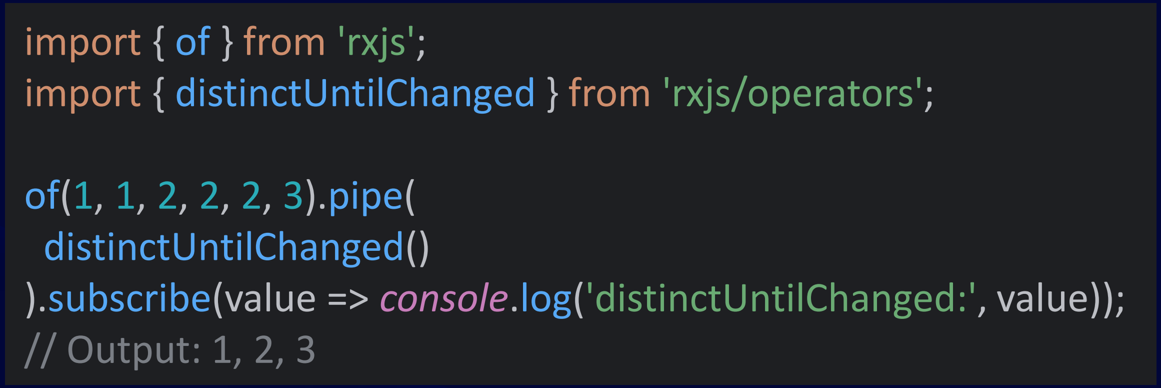
take



```
import { of } from 'rxjs';
import { take } from 'rxjs/operators';

of('A', 'B', 'C', 'D').pipe(
  take(2)
).subscribe(value => console.log('take:', value));
// Output: A, B
```

- Emits only the first N values

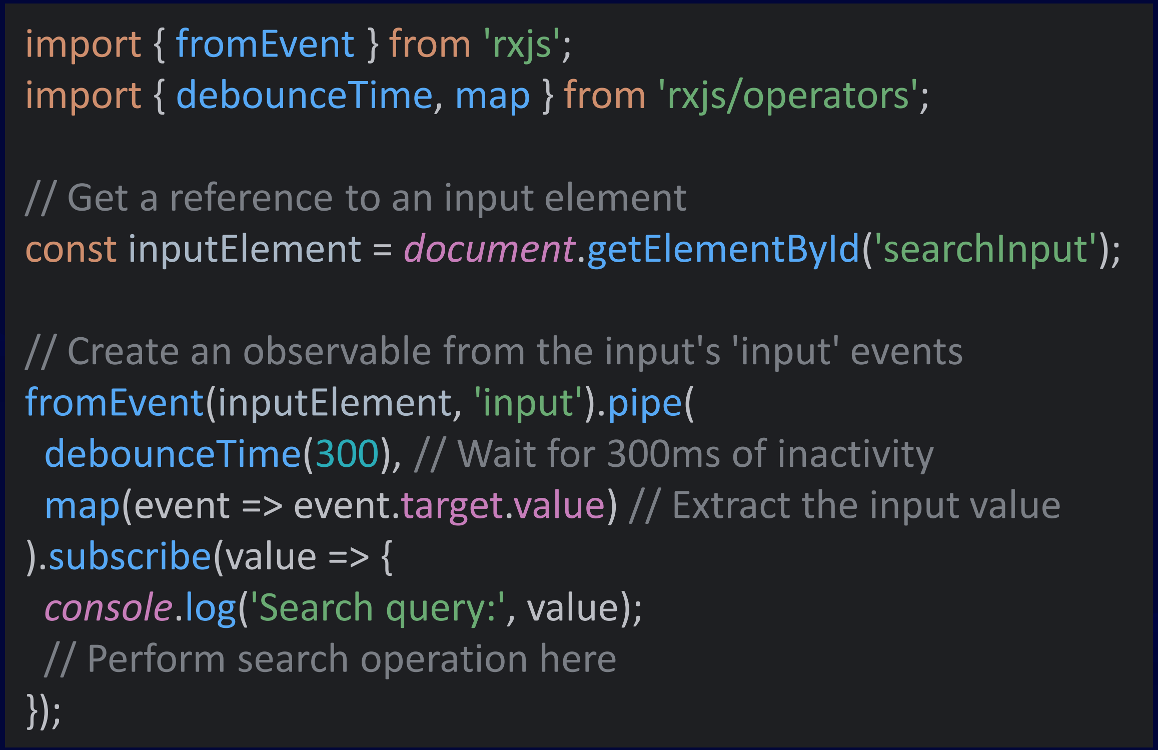
distinctUntilChanged



```
import { of } from 'rxjs';
import { distinctUntilChanged } from 'rxjs/operators';

of(1, 1, 2, 2, 2, 3).pipe(
  distinctUntilChanged()
).subscribe(value => console.log('distinctUntilChanged:', value));
// Output: 1, 2, 3
```

- Only emits new values if they’re different from the previous emission

debounceTime



```
import { fromEvent } from 'rxjs';
import { debounceTime, map } from 'rxjs/operators';

// Get a reference to an input element
const inputElement = document.getElementById('searchInput');

// Create an observable from the input's 'input' events
fromEvent(inputElement, 'input').pipe(
  debounceTime(300), // Wait for 300ms of inactivity
  map(event => event.target.value) // Extract the input value
).subscribe(value => {
  console.log('Search query:', value);
  // Perform search operation here
});
```

- It effectively "throttles" the emissions by only allowing the latest value to pass through after a period of inactivity.

Deferrable Views

Deferrable Views in Angular – Introduction

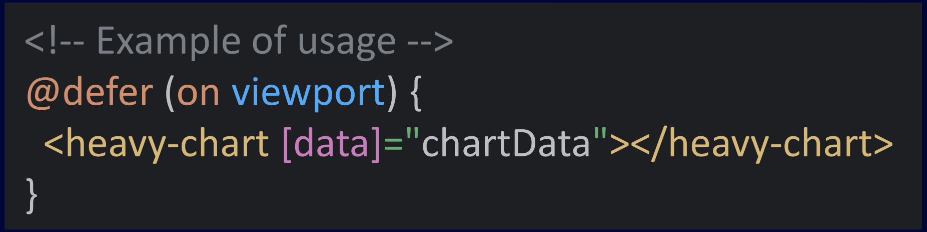
- What Are Deferrable Views?
  - A new feature allowing you to defer loading or rendering part of the template.
  - Helps optimize performance by loading content only when necessary.

- Key Use Cases:
  - Large components or features that aren't initially needed.
  - Improving perceived performance by offloading content to "idle" time or certain user actions.

- Key Benefits:
  - Smaller Initial Bundle
  - Faster Time to Interactive
  - Load on Demand

How @defer Works

- @defer directive is placed in the template to wrap the content to be postponed.
- Triggers determine when the deferred content finally loads
- Prefetch can optionally prepare resources before the actual load.
- Separate chunk: heavy-chart-lazy.js



```
<!-- Example of usage -->
@defer (on viewport) {
  <heavy-chart [data]="chartData"></heavy-chart>
}
```

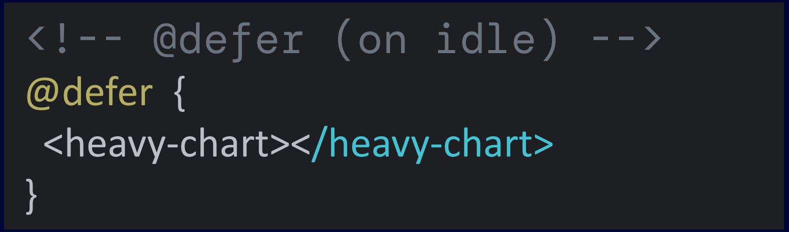
@defer Syntax

- @defer – Main Deferred Content: The actual component that loads lazily
- @placeholder – Before Loading: Shows BEFORE loading starts
- @loading – During Loading: Shows WHILE component is loading
- @error – Loading Failed: Shows if loading fails (network error, etc.)

```
@defer (trigger) {
  <!-- Main content - loads lazily -->
  <heavy-component />
}
@placeholder {
  <!-- Shown before loading starts -->
  <div>Click to load</div>
}
@loading (minimum 500ms; after 100ms)
{
  <!-- Shown during loading -->
  <spinner />
}
@error {
  <!-- Shown if loading fails -->
  <error-message />
}
```

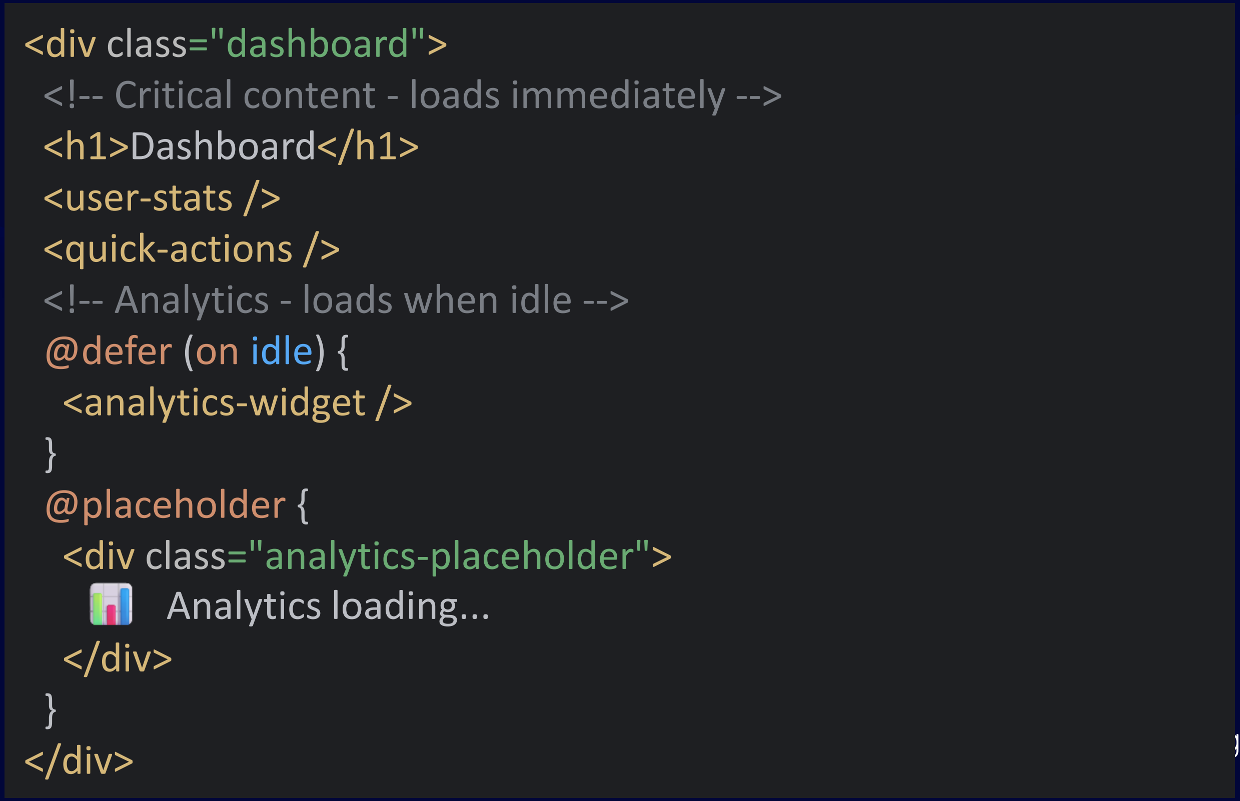
On Idle Trigger

- Triggers when browser is idle (the default behavior).
- Great for large content that isn't immediately needed.
- Postpones rendering until the main thread is less busy.



```
<!-- @defer (on idle) -->
@defer {
 <heavy-chart></heavy-chart>
}
```

- Benefits:
  - Third-party scripts don't block main content
  - Page feels instant
  - Social widgets load in background



```
<div class="dashboard">
 <!-- Critical content - loads immediately -->
 <h1>Dashboard</h1>
 <user-stats />
 <quick-actions />
 <!-- Analytics - loads when idle -->
 @defer (on idle) {
  <analytics-widget />
 }
 @placeholder {
  <div class="analytics-placeholder">
   📊 Analytics loading...
  </div>
 }
</div>
```

On ViewPort Trigger

- Triggers when the specified content enters the viewport (i.e., is scrolled into view).
- Ideal for lazy-loading images or sections below the fold.
- Improves initial load by skipping content that’s off-screen.

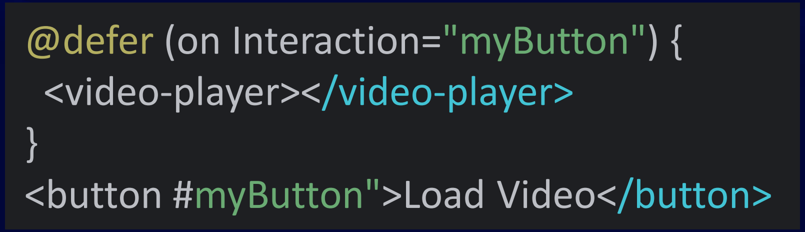
```
@defer (on viewport) {
  <img src="heavy-image.jpg" alt="Deferred Image">
}
```

- Benefits:
  - Thumbnail loads immediately (small)
  - Full gallery defers until visible
  - Saves bandwidth if user doesn't scroll

On Interaction Trigger

- Triggers when the user interacts with a specific element (e.g., a button).
- Useful for content behind a “Click to load” approach.
- Defers overhead until the user explicitly wants it.

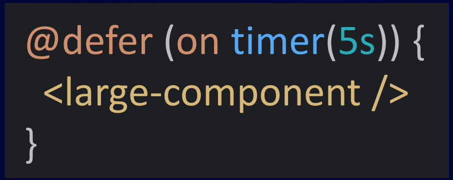
- Interaction Types Detected:
  - Click
  - dblclick
  - Mousedown
  - Touchstart
  - Keydown
  - focus



```
@defer (on Interaction="myButton") {
 <video-player></video-player>
}
<button #myButton">Load Video</button>
```

On timer Trigger

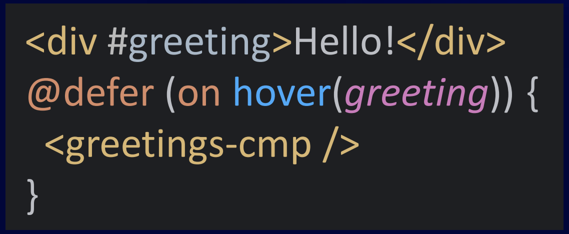
- Triggers after a specific duration (e.g. 5 seconds).
- Schedules loading for content that can wait but is still needed eventually.
- E.g., load an ad or secondary widget after initial page setup.



```
@defer (on timer(5s)) {
  <large-component />
}
```

On Hover Trigger

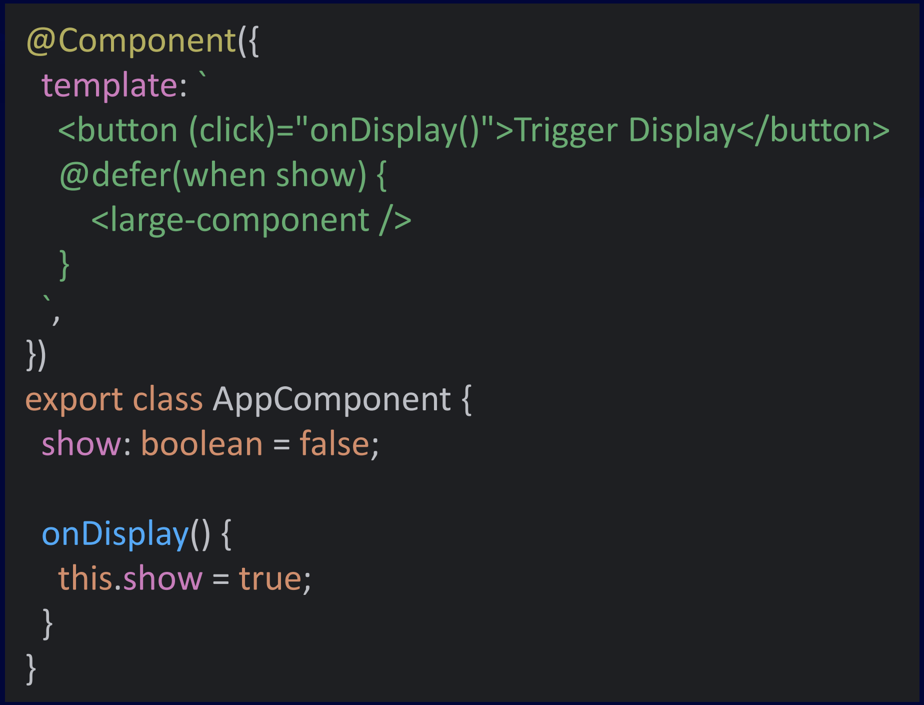
- Triggers when the mouse hovers over a specified area.
- Could be used for large tooltips or advanced popovers.
- Defers overhead until the user actually points at the element.



```
<div #greeting>Hello!</div>
@defer (on hover(greeting)) {
  <greetings-cmp />
}
```

when Trigger

- Accepts a custom condition expression, loads deferred content once the condition is truthy.
- Perfect for advanced logic—like “only load if user is logged in” or “only load after a form is valid.”
- Offers full control over when the view loads.

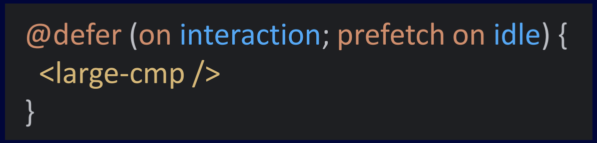


```
@Component({
 template: `
  <button (click)="onDisplay()">Trigger Display</button>
  @defer(when show) {
    <large-component />
  }
 `,
})
export class AppComponent {
 show: boolean = false;

 onDisplay() {
  this.show = true;
 }
}
```

Prefetching with @defer

- Concept:
  - Load resources or data behind a deferred block in the background so that when the user (or condition) triggers the @defer block, it appears quickly.

- Why Prefetch?
  - Faster Display: If you know the user is likely to want the deferred content soon, prefetching avoids a sudden load delay.
  - Better Perceived Performance: The user sees the content instantly once @defer is triggered.



```
@defer (on interaction; prefetch on idle) {
  <large-cmp />
}
```

Advanced Routing

Functional Guards with inject()

- Benefits:
  - ✅ Less boilerplate (simple function)
  - ✅ Easier to test (pure function)
  - ✅ Better tree-shaking
  - ✅ inject() gives access to DI

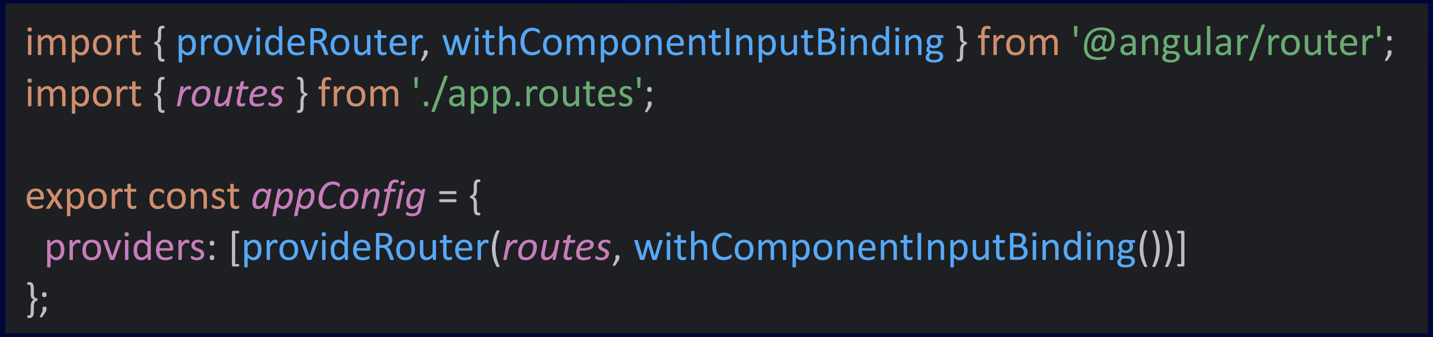
```
export const authGuard: CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isLoggedIn()) {
    return true;
  }

  router.navigate(['/login']);
  return false;
};
```

Signal-Based Route Parameters - No More ActivatedRoute!

- Problems:
  - Verbose (subscribe, takeUntilDestroyed)
  - Manual type conversion (params.get('id'))
  - Multiple observables to manage

Signal-Based Route Parameters- Enabling

- Enable Input Binding in the Router Configuration:
  - Use withComponentInputBinding() when setting up the router to automatically bind route parameters to component inputs.



```
import { provideRouter, withComponentInputBinding } from '@angular/router';
import { routes } from './app.routes';

export const appConfig = {
  providers: [provideRouter(routes, withComponentInputBinding())]
};
```

Signal-Based Route Parameters: The use

- Declare the Input in the Component:
  - Use input.required<string>() to define a required input that will receive the route parameter.

```
import { Component, input } from '@angular/core';

@Component({
  selector: 'app-user',
  template: `<p>User ID: {{ driverId() }}</p>`
})
export class UserComponent {
  driverId = input.required<string>(); // get the param of : /driver/:driverId
}
```

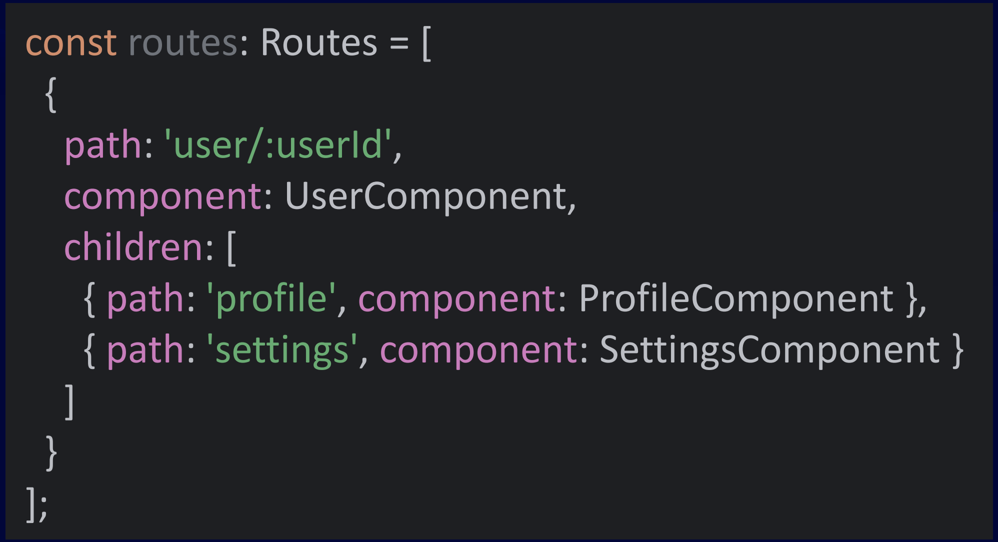
Accessing Parent Route Parameters in Child Routes(1/3)

- In nested routes, child components often need access to parameters defined in the parent route (e.g., userId in /user/:userId/profile).
- With Angular's modern features, we can achieve this declaratively using input.required<string>() and withRouterConfig.
- Enable Parameter Inheritance in Router Configuration:

```
export const appConfig = {
  providers: [
    provideRouter(
      routes,
      withRouterConfig({
        paramsInheritanceStrategy: 'always' // Inherit parent params
      })
    )
  ]
};
```

Accessing Parent Route Parameters in Child Routes(2/3)

- Define a parent route with a parameter (e.g., :userId) and child routes.



```
const routes: Routes = [
  {
    path: 'user/:userId',
    component: UserComponent,
    children: [
      { path: 'profile', component: ProfileComponent },
      { path: 'settings', component: SettingsComponent }
    ]
  }
];
```

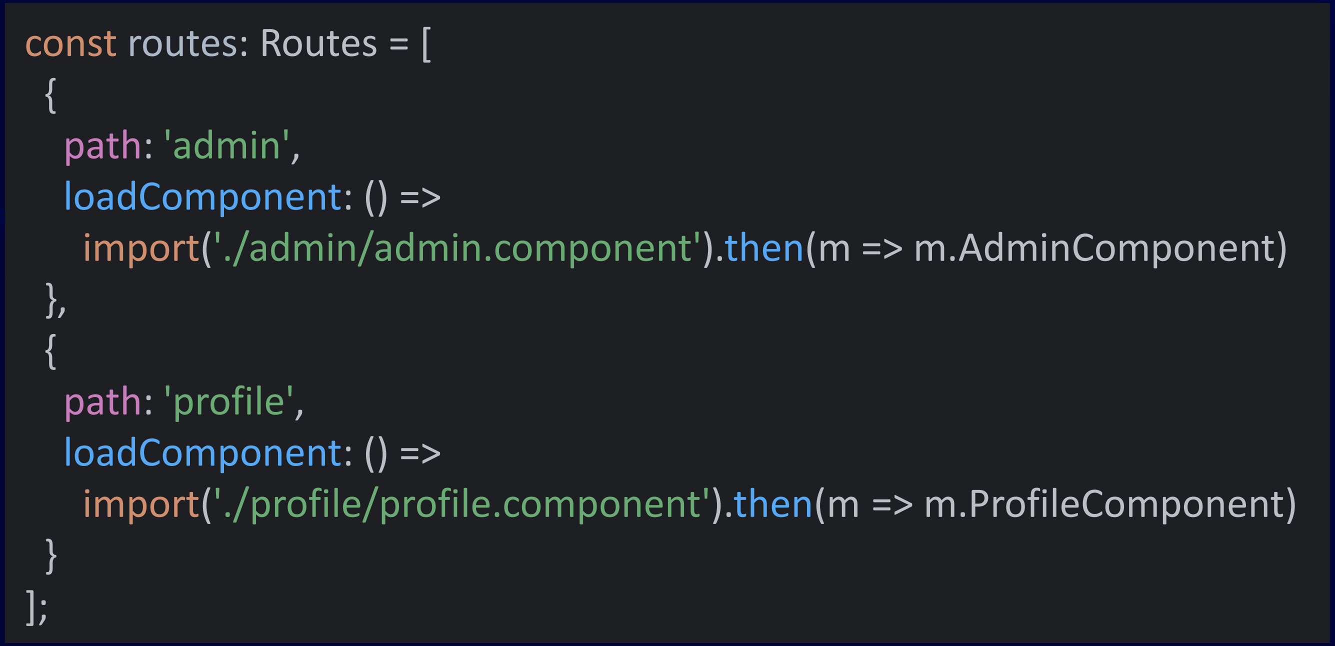
Accessing Parent Route Parameters in Child Routes(3/3)

- In the child component, use input.required<string>() to access the parent route parameter.

```
import { Component, input } from '@angular/core';

@Component({
  selector: 'app-profile',
  template: `<p>User ID: {{ userId() }}</p>`
})
export class ProfileComponent {
  userId = input.required<string>(); // Access parent route parameter
}
```

Modern Lazy Loading - New Way 1: loadComponent (Single Component)

- Benefits:
  - ✅ No NgModule needed
  - ✅ Smaller bundle chunks
  - ✅ Simpler code



```
const routes: Routes = [
 {
   path: 'admin',
   loadComponent: () =>
     import('./admin/admin.component').then(m => m.AdminComponent)
 },
 {
   path: 'profile',
   loadComponent: () =>
     import('./profile/profile.component').then(m => m.ProfileComponent)
 }
];
```

Modern Lazy Loading - New Way 2: loadChildren with Routes (Child Routes)

```
const routes: Routes = [
 {
  path: 'admin',
  loadChildren: () =>
   import('./admin/admin.routes').then(m => m.ADMIN_ROUTES)
 }
];

// admin/admin.routes.ts
export const ADMIN_ROUTES: Routes = [
 { path: '', component: AdminDashboardComponent },
 { path: 'users', component: UserManagementComponent },
 { path: 'settings', component: SettingsComponent }
];
```