CS572 Modern Web Applications Programming
Maharishi University of Management
Department of Computer Science
Assistant Professor Asaad Saad

PREV NEXT

OUTLINE

Search...


1. ---


2. Maharishi University of Management - Fairfield, Iowa


3. Data Binding


4. Bindings


5. One-way vs Two-ways Data Binding


6. Change Detection


7. One-way Data Binding in Two Phases


8. Why Using Two Phases in One-way Data Binding?

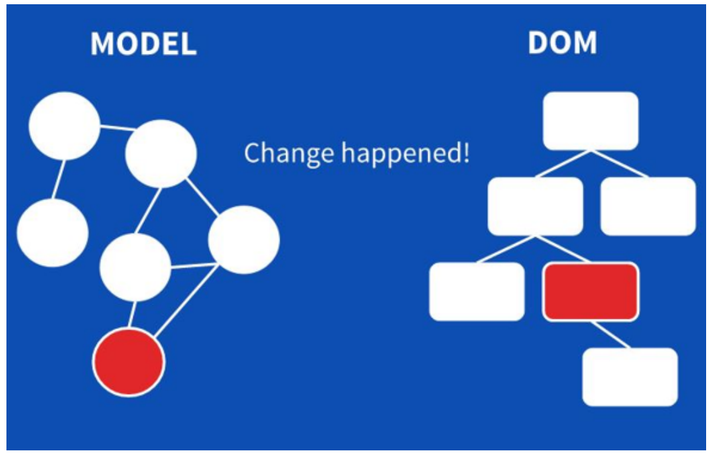

9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs

Data Binding

The basic task of data binding is to take the internal state of a program and make it somehow visible to the user interface. This state can be any kind of objects, arrays, primitives... just any kind of JavaScript data structures.



This state might end up as paragraphs, forms, links or buttons in the user interface (DOM). So basically we take data structures as input and generate DOM output to display it to the user. We call this process rendering.

OUTLINE

Search...

1. ---
2. Maharishi University of Management - Fairfield, Iowa
3. Data Binding
4. Bindings
5. One-way vs Two-ways Data Binding
6. Change Detection
7. One-way Data Binding in Two Phases
8. Why Using Two Phases in One-way Data Binding?
9. What has changed? Where?
10. React.js Virtual DOM
11. Angular Differs

Bindings

Property bindings, which can be added using the [] syntax, it reflects the state of the model in the view.

Event bindings, which can be added using the () syntax, it captures a browser event or component output to execute some function on a component or a directive.

Template bindings, which can be added using the {{}} syntax, it reflects the state of the model in the view.

PREV NEXT

OUTLINE

Search...


1. ---


2. Maharishi University of Management - Fairfield, Iowa


3. Data Binding


4. Bindings


5. One-way vs Two-ways Data Binding


6. Change Detection


7. One-way Data Binding in Two Phases


8. Why Using Two Phases in One-way Data Binding?


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs

One-way vs Two-ways Data Binding

In **AngularJS**, the default data binding method was **two-way data binding**. The problem with two-way data binding is that it often causes cascading effects throughout your application and makes it really difficult to trace data flow as your project grows.

One way data binding: we run CD one time only then it gets stable
Two-ways data binding: we must run CD many times UNTIL it gets stable

PREV NEXT

OUTLINE

Search...


1. ---


2. Maharishi University of Management - Fairfield, Iowa


3. Data Binding


4. Bindings


5. One-way vs Two-ways Data Binding


6. Change Detection


7. One-way Data Binding in Two Phases


8. Why Using Two Phases in One-way Data Binding?


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs

Change Detection

Let's define the application model that will store the state of our application.

Imagine an event changing the model: changing the teacher or number of students.

```
{
  "course": {"name": "MWA"},
  "details":
           { "id":"CS572",
             "teacher": "Asaad Saad",
             "block": "July",
             "students": 25
           }
}
```

At this point nothing has changed in the DOM. Only the Model has been updated.
At the end of the VM turn, magically, the **change detection** kicks in to propagate changes in the DOM.
Change detection goes through every component in the component tree to check if the model it depends on changed. Because from JS point of view , if this object was used as an input in another component, then there is **no change has happened to the object**! (same reference).

OUTLINE

Search...


1. ---


2. Maharishi University of Management - Fairfield, Iowa


3. Data Binding


4. Bindings


5. One-way vs Two-ways Data Binding


6. Change Detection


7. One-way Data Binding in Two Phases


8. Why Using Two Phases in One-way Data Binding?


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs

One-way Data Binding in Two Phases

An Angular application consists of nested components, so it will always have a component tree.

Angular separates updating the application model and reflecting the state of the model in the view into two distinct phases.

1. The developer is responsible for updating the application model.
2. Angular via bindings (observables), by means of **change detection**, is responsible for reflecting the state of the model in the DOM.

PREV NEXT

OUTLINE

Search...


1. ---


2. Maharishi University of Management - Fairfield, Iowa


3. Data Binding


4. Bindings


5. One-way vs Two-ways Data Binding


6. Change Detection


7. One-way Data Binding in Two Phases


8. Why Using Two Phases in One-way Data Binding?


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs

Why Using Two Phases in One-way Data Binding?

Using change detection only for updating the view state limits the number of places where the application model can be changed.

The major benefit of the separation is that it allows us to limit the options of who's contributing in updating the view state propagation process. This makes the system **more predictable**, and it also makes it a lot more performant.

The fact that the change detection graph in Angular can be modeled as a tree allowed us to get rid of digest loop (multiple digest runs until no changes occur). Now the system should get stable after a single pass (from top to bottom).

PREV NEXT

OUTLINE

Search...

1. ---
2. Maharishi University of Management - Fairfield, Iowa
3. Data Binding
4. Bindings
5. One-way vs Two-ways Data Binding
6. Change Detection
7. One-way Data Binding in Two Phases
8. Why Using Two Phases in One-way Data Binding?
9. What has changed? Where?
10. React.js Virtual DOM
11. Angular Differs

What has changed? Where?

It gets trickier when a change happens at runtime when the DOM has already been rendered. How do we figure out what has changed in our model, and where do we need to update the DOM?

Accessing the DOM tree is very expensive, so not only do we need to find out where updates are needed, but we also want to keep that access as tiny as possible.

One way to solve this, is simply making http request and **re-rendering the whole page**.

PREV NEXT

OUTLINE

Search...


1. ---


2. Maharishi University of Management - Fairfield, Iowa


3. Data Binding


4. Bindings


5. One-way vs Two-ways Data Binding


6. Change Detection


7. One-way Data Binding in Two Phases


8. Why Using Two Phases in One-way Data Binding?


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs

React.js Virtual DOM

The Virtual DOM is a collection of modules designed to provide a declarative way of representing the DOM for your app. So instead of updating the DOM when your application state changes, you simply create a virtual tree, which looks like the DOM state that you want.

The Virtual DOM allows you to update a view **whenever state changes** by **creating a full Virtual Tree of the view** and then **patching the DOM** efficiently to look exactly as you described it. This results in keeping manual DOM manipulation and previous state tracking out of your application code, promoting clean and maintainable rendering logic for web applications.

PREV NEXT

OUTLINE

Search...

1. ---
2. Maharishi University of Management - Fairfield, Iowa
3. Data Binding
4. Bindings
5. One-way vs Two-ways Data Binding
6. Change Detection
7. One-way Data Binding in Two Phases
8. Why Using Two Phases in One-way Data Binding?
9. What has changed? Where?
10. React.js Virtual DOM
11. Angular Differs

Angular Differs

In order to evaluate what changed, Angular provides ***differs***. Differs will evaluate a given property of your directive to determine what changed.

There are two types of built-in differs:

- **Iterable differs**
  - Iterable differs is used when we have a **list-like structure** and we're only interested on knowing things that were added or removed from that list.
- **Key-value differs**
  - Key-value differs is used for **dictionary-like structures**, and it works at the key level. This differ will identify changes when a new key is added, removed and changed.

PREV NEXT

OUTLINE

Search...

1. ---
2. Maharishi University of Management - Fairfield, Iowa
3. Data Binding
4. Bindings
5. One-way vs Two-ways Data Binding
6. Change Detection
7. One-way Data Binding in Two Phases
8. Why Using Two Phases in One-way Data Binding?
9. What has changed? Where?
10. React.js Virtual DOM
11. Angular Differs

Who Triggers a Change Detection Cycle

We want to display the value we enter on the **`input`** element in real-time.
We added an event handler to be triggered on every **keyup**. When any event happens in any component, Angular is going to check all bounded variables (**box**) and start a **Detection Change Cycle** for the component.

```
<input #box (keyup)='0' /> {{ box.value }}
```


OUTLINE

Search...

2. Maharishi University of Management - Fairfield, Iowa
3. Data Binding
4. Bindings
5. One-way vs Two-ways Data Binding
6. Change Detection
7. One-way Data Binding in Two Phases
8. Why Using Two Phases in One-way Data Binding?
9. What has changed? Where?
10. React.js Virtual DOM
11. Angular Differs
12. Who Triggers a Change Detection Cycle

What Causes the Change?

Basically application state change can be caused by three things:

- **Events** - click, submit, …
- **XHR** - Fetching data from a remote server
- **Timers** - setTimeout(), setInterval()

Notice how they are all **asynchronous**. Which brings us to the conclusion that whenever an asynchronous operation has been performed, our application state will be changed. **This is when someone needs to tell Angular to update the view (Zones).**

All async operations will run in an execution context that refers to the global scope, so how to know in which component the change has happened?

PREV NEXT

OUTLINE

Search...


3. Data Binding


4. Bindings


5. One-way vs Two-ways Data Binding


6. Change Detection


7. One-way Data Binding in Two Phases


8. Why Using Two Phases in One-way Data Binding?


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs


12. Who Triggers a Change Detection Cycle


13. What Causes the Change?

NgZone

Your code runs in Zone, which is simply an execution context for **async operations**. Angular **Monkey-Patch** all async code and add hooks around it.

```
a();
setTimeout(b, 0);
c();
```

```
a
c
b // async
```

What if we wanted to time how long this runs? This won't work!

```
start()
a();
setTimeout(b, 0);
c();
stop();
```

```
start
a
c
stop
b // missed
```

15

OUTLINE

Search...

4. Bindings
5. One-way vs Two-ways Data Binding
6. Change Detection
7. One-way Data Binding in Two Phases
8. Why Using Two Phases in One-way Data Binding?
9. What has changed? Where?
10. React.js Virtual DOM
11. Angular Differs
12. Who Triggers a Change Detection Cycle
13. What Causes the Change?
14. NgZone

How Zone works?

All async tasks are **Monkey-Patched** and they run in the **same Zone**.

```
Zone.run(function(){
   a();
   setTimeout(b, 0);
   c();
});
```

Monkey-Patch

```
Function a(){
   onZoneEnter();
   //code for function a
   onZoneLeave();
};
```

```
Function b(){
   onZoneEnter();
   //code for function b
   onZoneLeave();
};
```

```
Function c(){
   onZoneEnter();
   //code for function c
   onZoneLeave();
};
```

```
var start, time;
function onZoneEnter(){
   start = Date.now();
}
function onZoneLeave(){
   time += (Date.now() - start);
}
```


PREV NEXT

OUTLINE

Search...

6. Change Detection
7. One-way Data Binding in Two Phases
8. Why Using Two Phases in One-way Data Binding?
9. What has changed? Where?
10. React.js Virtual DOM
11. Angular Differs
12. Who Triggers a Change Detection Cycle
13. What Causes the Change?
14. NgZone
15. How Zone works?
16. Zones notifies Angular about Changes

Zones notifies Angular about Changes

Let’s assume that somewhere in our component tree an event is fired, maybe a button has been clicked. **Zones** execute the given handler and knows to which component it belongs because it's monkey-patched and notify Angular when the turn is done, which eventually causes Angular to perform a **change detection cycle**.

In React, the dev triggers setstate() to inform React that a change has happened. In angular, Zones notifies Angular about a change + where the change has happened.

OUTLINE

Search...


6. Change Detection


7. One-way Data Binding in Two Phases


8. Why Using Two Phases in One-way Data Binding?


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs


12. Who Triggers a Change Detection Cycle


13. What Causes the Change?


14. NgZone


15. How Zone works?

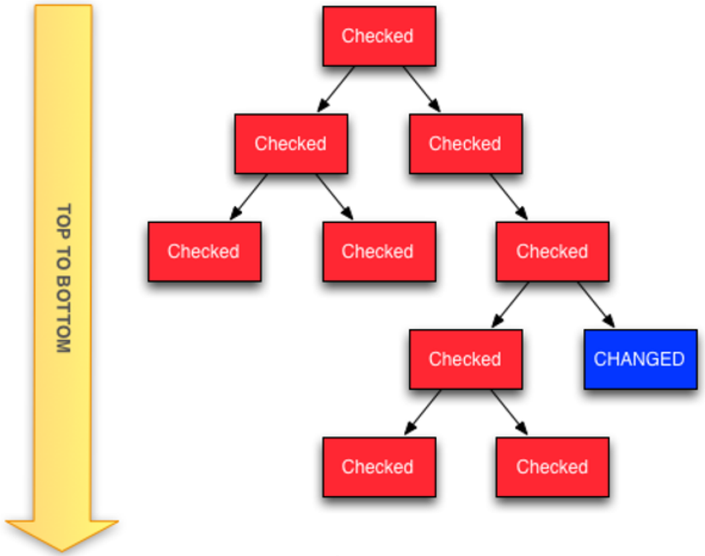

16. Zones notifies Angular about Changes

Change Detection Cycle Algorithm

When one of the components change, no matter where in the tree it is, a **change detection pass is triggered for the whole tree**.

Angular scans for changes from the top component node, all the way to the bottom leaves of the tree.

Each component has its own change detector.



Zones inform Angular that change has happened in this component

OUTLINE

Search...

7. One-way Data Binding in Two Phases
8. Why Using Two Phases in One-way Data Binding?
9. What has changed? Where?
10. React.js Virtual DOM
11. Angular Differs
12. Who Triggers a Change Detection Cycle
13. What Causes the Change?
14. NgZone
15. How Zone works?
16. Zones notifies Angular about Changes
17. Change Detection Cycle Algorithm

Example

```
import { Component, Input, OnChanges } from '@angular/core';
@Component({
   selector: 'app-child',
   template:`<p>Child Component: {{person.name}} lives at {{person.zipcode}}</p>`,
})
export class ChildComponent implements OnChanges {
   @Input() person: { name: string, zipcode: string };

   ngOnChanges(change) {
       console.log(`Change detected`)
   }
}
```

What happens when the App component (parent) changes **`zipcode`** by mutation vs without mutation?

OUTLINE

Search...


8. Why Using Two Phases in One-way Data Binding?


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs


12. Who Triggers a Change Detection Cycle


13. What Causes the Change?


14. NgZone


15. How Zone works?


16. Zones notifies Angular about Changes


17. Change Detection Cycle Algorithm


18. Example

Angular Change Detection Facts

- Change Detection Graph is a **directed tree.** It's performed in the same order starting from Root component.
- Way more predictable (Data flows from top to bottom)
- **Gets stable after a single pass**

Angular can perform hundreds of thousands of checks in milliseconds. Because **Angular generates VM friendly Monomorphic code**.

(component objects are **monomorphic**: Every components is a class with same properties type)

PREV NEXT

OUTLINE

Search...


9. What has changed? Where?


10. React.js Virtual DOM


11. Angular Differs


12. Who Triggers a Change Detection Cycle


13. What Causes the Change?


14. NgZone


15. How Zone works?


16. Zones notifies Angular about Changes


17. Change Detection Cycle Algorithm


18. Example


19. Angular Change Detection Facts

Monomorphic Type

Monomorphic use of operations is preferred over polymorphic operations. Operations are monomorphic if the hidden classes of inputs are always of the same type- otherwise they are **Polymorphic**

```
function add(x, y) {
        return x + y;
}
add(1, 2); // + in add is monomorphic
add("a", "b"); // + in add becomes polymorphic
```

The compiler will run Monomorphic functions much faster than polymorphic code.

OUTLINE

Search...


10. React.js Virtual DOM


11. Angular Differs


12. Who Triggers a Change Detection Cycle


13. What Causes the Change?


14. NgZone


15. How Zone works?


16. Zones notifies Angular about Changes


17. Change Detection Cycle Algorithm


18. Example


19. Angular Change Detection Facts


20. Monomorphic Type

Heuristic Detection

VMs do **heuristic detection** of "**hot functions**" (code that is executed hundreds or even thousands of times). If a function's execution count exceeds a predetermined limit, the VMs optimizer will pick up that bit of code and attempt to compile an optimized version based on the arguments passed to the function. In this case, it presumes your function will always be called with the **same *type* of arguments** (not necessarily the *same* objects).

On the other hand, if your code has to be written in a dynamic way (**polymorphic**: the shape of the objects isn't always the same), VMs will check every component no matter what its model structure looks like.

VMs don't like this sort of dynamic code, because they cannot optimize it.

PREV NEXT

OUTLINE

Search...


11. Angular Differs


12. Who Triggers a Change Detection Cycle


13. What Causes the Change?


14. NgZone


15. How Zone works?


16. Zones notifies Angular about Changes


17. Change Detection Cycle Algorithm


18. Example


19. Angular Change Detection Facts


20. Monomorphic Type


21. Heuristic Detection

Angular and Change Detection

Angular creates change detector classes at runtime for each component, which are **monomorphic**, because they know exactly what the shape of the component's model is. VMs can perfectly optimize this code, which makes it very fast to execute. The good thing is that we don't have to care about that because Angular does it automatically.

Smarter Change Detection?

Wouldn't it be great if we could tell Angular to only run change detection for the parts of the application that changed their state? Rather than running for the whole CD tree. There are data structures that give us some guarantees of when something has changed or not: **Immutables** and **Observables**.

OUTLINE

Search...


12. Who Triggers a Change Detection Cycle


13. What Causes the Change?


14. NgZone


15. How Zone works?


16. Zones notifies Angular about Changes


17. Change Detection Cycle Algorithm


18. Example


19. Angular Change Detection Facts


20. Monomorphic Type


21. Heuristic Detection


22. Angular and Change Detection

OnPush Strategy

When a component depends only on its input and this input was an immutable object, all we need to do is tell Angular that this component may **skip change detection if its input hasn't changed**.

```
@Component({
        changeDetection: ChangeDetectionStrategy.OnPush
})
```

You must use **Immutables** or **Observables** to use it.

We can tell Angular to skip change detection for this component's subtree if none of its inputs changed by setting the change detection strategy to `OnPush`.

We can skip entire components subtrees when immutable objects are used and Angular is informed accordingly.

OUTLINE

Search...


13. What Causes the Change?


14. NgZone


15. How Zone works?


16. Zones notifies Angular about Changes


17. Change Detection Cycle Algorithm


18. Example


19. Angular Change Detection Facts


20. Monomorphic Type


21. Heuristic Detection


22. Angular and Change Detection


23. OnPush Strategy

Better Solution

```
import { Component, Input, ChangeDetectionStrategy, OnChanges } from '@angular/core';
@Component({
   selector: 'app-child',
   template: ` <p>Child Component: {{person.name}} lives in {{address}} </p> `,
   changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent implements OnChanges {
   @Input() person: { name: string, zipcode: number };
   address: String;
   constructor() { console.info({ 'Constructor Input Value': this.person }) }
   ngOnInit() {
       console.info({ 'ngOnInit Input Value': this.person })
       this.address = this.locateAddress(this.person.zipcode);
   }
   ngOnChanges(change) {
       console.log(`Change detected, `)
       this.address = this.locateAddress(this.person.zipcode);
   }
}
```


OUTLINE

Search...


14. NgZone


15. How Zone works?


16. Zones notifies Angular about Changes


17. Change Detection Cycle Algorithm


18. Example


19. Angular Change Detection Facts


20. Monomorphic Type


21. Heuristic Detection


22. Angular and Change Detection


23. OnPush Strategy


24. Better Solution

What if...

- Change Detection starts
- Check **`A component`**
- Evaluate the expression **`{{name}}`** to the text **`Asaad Saad`**
- Update the DOM with this value
- Save the evaluated value in **`oldValues:`** `view.oldValues[1] = 'Asaad Saad';`
- Evaluate **`address`** expression to **`Fairfield`** and pass it down to the **`B component`**
- Save this value in **`oldValues:`** `view.oldValues[0] = 'Fairfield';`
- Run the same check for **`B component`** and call its lifecycle hooks
- Once the **`B component`** is checked, the current digest loop is finished.

```
@Component({
   selector: 'a-comp',
   template: `
      <span>{{name}}</span>
      <b-comp [address]="address"></b-comp>`
})
export class AComponent {
   name = 'Asaad Saad';
   address = 'Fairfield';
}
```

```
@Component({
   selector: 'b-comp',
   template: `
       <span>{{address}}</span> `
})
export class BComponent {
   @Input() address;
   ngAfterViewChecked() {
       this.address = 'Burlington';
   }
}
```


OUTLINE

Search...

15. How Zone works?
16. Zones notifies Angular about Changes
17. Change Detection Cycle Algorithm
18. Example
19. Angular Change Detection Facts
20. Monomorphic Type
21. Heuristic Detection
22. Angular and Change Detection
23. OnPush Strategy
24. Better Solution
25. What if...

What if...

If Angular is running in the **development mode** it then runs a second digest loop performing verification phase.

Angular runs a verification digest to check that properties value have not changed:

```
ACompView.instance.text === view.oldValues[0]; // false
          'Burlington' === 'Fairfield'
```

```
▼ERROR Error:                                BcomponentComponent.html:2
ExpressionChangedAfterItHasBeenCheckedError: Expression has
changed after it was checked. Previous value: 'null:
Fairfield'. Current value: 'null: Burlington'.
```

ExpressionChangedAfterItHasBeenCheckedError

OUTLINE

Search...

16. Zones notifies Angular about Changes
17. Change Detection Cycle Algorithm
18. Example
19. Angular Change Detection Facts
20. Monomorphic Type
21. Heuristic Detection
22. Angular and Change Detection
23. OnPush Strategy
24. Better Solution
25. What if...
26. What if...

Why Do We Need Verification Phase

Angular enforces unidirectional data flow from top to bottom. No component lower in hierarchy is allowed to update properties of a parent component after parent changes have been processed. This ensures that after the first digest loop the entire tree of components is stable. A tree is unstable if there are changes in the properties that need to be synchronized with the consumers that depend on those properties.

So why not run the change detection until the components tree stabilizes? because it may never stabilize and run forever. If a child component updates a property on the parent component as a reaction to this property change you will get an infinite loop.

OUTLINE

Search...


17. Change Detection Cycle Algorithm


18. Example


19. Angular Change Detection Facts


20. Monomorphic Type


21. Heuristic Detection


22. Angular and Change Detection


23. OnPush Strategy


24. Better Solution


25. What if...


26. What if...


27. Why Do We Need Verification Phase

Possible Fixes

- Use a component hook that's safe for change detection
- Asynchronous property update (both change detection and verification digests are performed synchronously)

```
ngAfterViewChecked() {
        setTimeout(() => { this.parent.name = 'Burlington'; });
}
```

- Forcing additional change detection cycle

```
constructor(private cd: ChangeDetectorRef) {}
ngAfterViewChecked() {
        this.cd.detectChanges();
}
```

The setTimeout function schedules a callback that will be executed in the next VM turn.

OUTLINE

Search...


18. Example


19. Angular Change Detection Facts


20. Monomorphic Type


21. Heuristic Detection


22. Angular and Change Detection


23. OnPush Strategy


24. Better Solution


25. What if...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes

Main Points

- When somewhere in our component tree an event is fired. Zones execute the given handler and notify Angular, which eventually causes Angular to perform change detection.
- Each component has its own change detector, and an Angular application consists of a component tree, so we have change detector tree too. This tree can also be viewed as a directed graph where data always flows from top to bottom.
- Unidirectional data flow is more predictable than cycles. We always know where the data we use in our views comes from.
- Change detection gets stable after a single pass. If one of our components causes any additional change after the first run and during change detection, Angular will throw an error.

PREV NEXT

OUTLINE

Search...


19. Angular Change Detection Facts


20. Monomorphic Type


21. Heuristic Detection


22. Angular and Change Detection


23. OnPush Strategy


24. Better Solution


25. What if...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points

Main Points

- Angular creates change detector classes at runtime for each component, which are monomorphic, because they know exactly what the shape of the component's model is. VMs can perfectly optimize this code, which makes it very fast to execute.
- Angular has to check every component every single time an event happens because maybe the application state has changed.
- When using Immutable objects or Observables we can optimize the Change Detection Algorithm.

PREV NEXT

OUTLINE

Search...


20. Monomorphic Type


21. Heuristic Detection


22. Angular and Change Detection


23. OnPush Strategy


24. Better Solution


25. What if...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points


30. Main Points

Main Points

- Angular separates updating the application model and updating the view.
- Event bindings are used to update the application model.
- Change detection uses property bindings to update the view.
  Updating the view is unidirectional and top-down. This makes the system more predictable and performant.
- We make the system more efficient by using the OnPush change detection strategy for the components that depend on immutable input and only have local mutable state.

PREV NEXT

OUTLINE

Search...


21. Heuristic Detection


22. Angular and Change Detection


23. OnPush Strategy


24. Better Solution


25. What if...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points


30. Main Points


31. Main Points

Built-in Directives

Directives add behavior to their host elements.

The built-in directives are imported and made available to your components automatically when importing **`BrowserModule`**.

- **`ngIf`**
- **`ngSwitch`**
- **`ngStyle`**
- **`ngClass`**
- **`ngFor`**
- **`ngNonBindable`**

OUTLINE

Search...


22. Angular and Change Detection


23. OnPush Strategy


24. Better Solution


25. What if...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points


30. Main Points


31. Main Points


32. Built-in Directives

Attribute Directives & Structural Directives

Attribute Directives

They are applied like HTML attributes and impact the element they are attached. (DOM friendly)

Structural Directives

They change the structure of the DOM, not only the element on which they sit.
They usually have a * before their name. (DOM un-friendly)

Components

Yes components are directives but with template.

OUTLINE

Search...


23. OnPush Strategy


24. Better Solution


25. What if...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points


30. Main Points


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives

ngIf

The condition is determined by **the result of the expression** that you pass in to the directive.

If the result of the expression returns a **`false`** value, the element will be removed from the DOM.

```
<div *ngIf="false"></div> <!-- never displayed -->
<div *ngIf="a > b"></div> <!-- displayed if a is more than b -->
<div *ngIf="myFunc()"></div> <!-- displayed if myFunc returns true -->
```

The * indicates that this directive treats the component/tag as a **template**, ngIF is a **Structural Directive**

OUTLINE

Search...


24. Better Solution


25. What if...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points


30. Main Points


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. nglf

De-Sugaring of Structural Directives

Angular provides a nice syntax for structural directives (`*ngIf`). It transforms the * syntax into a less beautiful syntax behind the scenes.

```
<div *ngIf="false"></div>
```

Desugared

```
<ng-template [ngIf]="false">
        <div></div>
</ng-template>
```

The directive css selector name is `[ngIf]` but the * is just to tell angular to create the template for me

OUTLINE

Search...


25. What if...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points


30. Main Points


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. ngIf


35. De-Sugaring of Structural Directives

ngIf else

```
<div *ngIf="isValid; else other_content">
        content here ...
</div>

<ng-template #other_content>other content here...</ng-template>
```


OUTLINE

Search...


26. What if...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points


30. Main Points


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. nglf


35. De-Sugaring of Structural Directives


36. nglf else

ngStyle

To set a given DOM element CSS properties

```
<div [style.background-color]="'yellow'">Yellow background</div>
```

Use it to set one css property.

Value is a literal string.

```
<span [style.font-size.px]="16">
        Red text
</span>
```

Specify units

`[style.fontSize.px] [style.fontSize.em] [style.fontSize.%]`

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
        White text on blue background
</div>
```

Use it to set multiple css properties.

Value is a JavaScript Object literal (quote invalid keys)

OUTLINE

Search...


27. Why Do We Need Verification Phase


28. Possible Fixes


29. Main Points


30. Main Points


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. ngIf


35. De-Sugaring of Structural Directives


36. ngIf else


37. ngStyle

ngClass

We pass an **object literal**. The object is expected to have the **keys** as the **class names** and the **values** should be a **truthy/falsy** value to indicate whether the class should be applied or not.

```
.special { margin: 1px;  color: #eee; }
```

```
<div [ngClass]="{special: false}">class special will not be applied</div>
<div [class.special]="false">class special will not be applied</div>
```

We can also use a list of class names to specify multiple classes should be added to the element.

```
<div class="base" [ngClass]="['blue', 'round']">
        This will always have base, blue, round classes
</div>
```


OUTLINE

Search...


28. Possible Fixes


29. Main Points


30. Main Points


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. ngIf


35. De-Sugaring of Structural Directives


36. ngIf else


37. ngStyle


38. ngClass

ngFor

To repeat a given DOM element. The syntax is:

```
*ngFor="let item of items"
```

- The `let item` syntax specifies a (template) variable that's receiving each element of the `items` array
- The `items` is the collection of items from your controller.

```
@Component({
        template: `<ul>
                        <li *ngFor="let name of names">Hello {{ name }}</li>
                   </ul>`
})
export class MyComponent {
        public names: string[]
        constructor() {
                this.names = ['Asaad', 'Mike', 'Mada'];
        }
}
```


OUTLINE

Search...


29. Main Points


30. Main Points


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. ngIf


35. De-Sugaring of Structural Directives


36. ngIf else


37. ngStyle


38. ngClass


39. ngFor

Getting an Index

```
@Component({
        template: `<ul>
                        <li *ngFor="let name of names; let num = index">
                                {{ num+1 }} - Hello {{ name }}
                        </li>
                    </ul>`
})

export class MyComponent {
        constructor( public names: string[]) {
                names = ['Asaad', 'Mike', 'Mada'];
        }
}
```


OUTLINE

Search...


30. Main Points


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. nglf


35. De-Sugaring of Structural Directives


36. nglf else


37. ngStyle


38. ngClass


39. ngFor


40. Getting an Index

ngNonBindable

We use ngNonBindable when we don't want to compile or bind a particular section of our page.

```
<div>
   <span ngNonBindable>This {{ Hello }} will not be evaluated</span>
</div>
```


OUTLINE

Search...


31. Main Points


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. ngIf


35. De-Sugaring of Structural Directives


36. ngIf else


37. ngStyle


38. ngClass


39. ngFor


40. Getting an Index


41. ngNonBindable

Creating our Custom Directive

To create a new custom Directive class from Angular CLI we use:

```
ng g d directiveName
```

Notice

- Angular creates a Directive using the **@Directive()** decorator
- Directives don't have template/styles
- The selector is a **CSS selector [attribute]**
- CLI will add our directive class to the **`declarations[]`** array in **`module.ts`** along with all other components to be instantiated.

OUTLINE

Search...


32. Built-in Directives


33. Attribute Directives & Structural Directives


34. ngIf


35. De-Sugaring of Structural Directives


36. ngIf else


37. ngStyle


38. ngClass


39. ngFor


40. Getting an Index


41. ngNonBindable


42. Creating our Custom Directive

Services we usually use within Directives

All these Services can be imported from `@angular/core`

Reference to the Element we are applying the directive on

Reference to a Service (helper Object)

```
constructor(   private element: ElementRef,
               private renderer2: Renderer2  ){

        element.nativeElement.style.fontSize = '22px';
        renderer2.setStyle(element.nativeElement, 'font-size', '22px');
}
```

When you set an **input** property in your directive, you cannot read its value from the constructor, cause it's not been created yet! You will have to wait for the constructor to create the directive object and set this property first, then you may read it within another stage of the directive lifecycle (ngOnInit).

OUTLINE

Search...


33. Attribute Directives & Structural Directives


34. ngIf


35. De-Sugaring of Structural Directives


36. ngIf else


37. ngStyle


38. ngClass


39. ngFor


40. Getting an Index


41. ngNonBindable


42. Creating our Custom Directive


43. Services we usually use within Directives

Host Element

To turn an Angular component into something rendered in the DOM you have to associate an Angular component with a DOM element. We call such elements host elements.

A directive can interact with its host DOM element in the following ways:

- It can listen to its events.
- It can update its properties.
- It can invoke methods on it.

Angular automatically checks host property bindings during change detection. If a binding changes, it will update the host element of the directive.

OUTLINE

Search...


34. ngIf


35. De-Sugaring of Structural Directives


36. ngIf else


37. ngStyle


38. ngClass


39. ngFor


40. Getting an Index


41. ngNonBindable


42. Creating our Custom Directive


43. Services we usually use within Directives


44. Host Element

Example

```
@HostBinding('style.backgroundColor') myBG;

@HostListener('mouseenter') foo(){
        this.element.nativeElement.style.color = 'red',
        this.myBG = 'yellow'
}
@HostListener('mouseleave') bar(){
        this.element.nativeElement.style.color = 'black',
        this.myBG = 'white’
}
@HostListener("input", "$event.target.value") onChange(updatedValue: string) {
        this.value = updatedValue.trim();
}
```

This is not Platform friendly, tightly coupled with DOM.

OUTLINE

Search...

35. De-Sugaring of Structural Directives
36. ngIf else
37. ngStyle
38. ngClass
39. ngFor
40. Getting an Index
41. ngNonBindable
42. Creating our Custom Directive
43. Services we usually use within Directives
44. Host Element
45. Example

DOM Interaction

We don’t interact with the DOM directly. Angular aims to provide a higher-level API, so the native platform, the DOM, will just reflect the state of the Angular application. This is useful for a couple of reasons:

- It makes components easier to refactor.
- It allows unit testing most of the behavior of an application without touching the DOM.
- It allows running Angular applications in a web worker, server, or other platforms where a native DOM isn’t present.

OUTLINE

Search...


36. ngIf else


37. ngStyle


38. ngClass


39. ngFor


40. Getting an Index


41. ngNonBindable


42. Creating our Custom Directive


43. Services we usually use within Directives


44. Host Element


45. Example


46. DOM Interaction

Main Point

There are three kinds of directives in Angular: Components, Structural directives and Attribute directives.

- **Components** are the most common of the three directives.
- **Structural Directives** change the structure of the view. Two examples are NgFor and NgIf in the Template Syntax page.
- **Attribute directives** are used as attributes of elements. The built-in NgStyle directive in the Template Syntax page, for example, can change several element styles at the same time.

OUTLINE

Search...


37. ngStyle


38. ngClass


39. ngFor


40. Getting an Index


41. ngNonBindable


42. Creating our Custom Directive


43. Services we usually use within Directives


44. Host Element


45. Example


46. DOM Interaction


47. Main Point

Pipes

Pipes transform displayed values within a template.

Example: *When data arrives in our component, we could just push their raw values directly to the view. That might make a bad user experience. Everyone prefers a simple birthday date like April 15, 1988 to the original raw string format — Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time).*

```
{{ dateObj | date }} // output is 'Apr 15, 1988'
```


OUTLINE

Search...


38. ngClass


39. ngFor


40. Getting an Index


41. ngNonBindable


42. Creating our Custom Directive


43. Services we usually use within Directives


44. Host Element


45. Example


46. DOM Interaction


47. Main Point


48. Pipes

Built-in Pipes

@angular/common

- async
- currency
- date
- decimal
- json
- lowercase
- percent
- slice
- uppercase

Examples:

```
<p>{{ myValue | uppercase }}</p>
<p>{{ myDate | date:"MM/dd/yy" }}</p>
<p>{{ myValue | slice:3:7 | uppercase }}</p>
```


OUTLINE

Search...


39. ngFor


40. Getting an Index


41. ngNonBindable


42. Creating our Custom Directive


43. Services we usually use within Directives


44. Host Element


45. Example


46. DOM Interaction


47. Main Point


48. Pipes


49. Built-in Pipes

Custom Pipes

To create a custom pipe from CLI: **`ng g p myPipe`** or by adding the **`@Pipe`** decorator to a class.

Custom pipes should be declared in the **`declarations[]`** array at **`app.module.ts`**

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'double' })
export class DoublePipe implements PipeTransform {
        transform(value: any, args?: any): any { return value * 2; }
}

<input type="text" #input (keyup)="0">
<p>{{ input.value | double }}</p>
```


OUTLINE

Search...


40. Getting an Index


41. ngNonBindable


42. Creating our Custom Directive


43. Services we usually use within Directives


44. Host Element


45. Example


46. DOM Interaction


47. Main Point


48. Pipes


49. Built-in Pipes


50. Custom Pipes

Pure and Impure Pipes

When Pipe is pure, it means that Angular will **NOT re-run** them on the value they are applied to upon each change detection cycle. This behavior makes sense, as it saves performance.

If you need to **re-run the pipe on each change detection cycle**, you may mark your pipe as impure by setting **'pure' to false**.

```
@Pipe({ name: 'myPipe', pure: false })
export class myPipe implements PipeTransform { ... }
```


OUTLINE

Search...

41. ngNonBindable
42. Creating our Custom Directive
43. Services we usually use within Directives
44. Host Element
45. Example
46. DOM Interaction
47. Main Point
48. Pipes
49. Built-in Pipes
50. Custom Pipes
51. Pure and Impure Pipes

Async Pipe

The **async pipe** (a built-in pipe) is an impure pipe. Its job is to fetch asynchronously returned data from Promises or Observables.

Therefore, the async pipe is a great helper if you want to print some data to the screen which isn't available upon component initialization.

```
@Component({
        template: `<p>{{asyncValue | async}}</p> `
})
export class AppComponent {
        asyncValue = new Promise((resolve, reject) => {
                        setTimeout(() => resolve('Data is here!'), 2000); });
}
```

What's going to happen if we don't add the async pipe?

OUTLINE

Search...


42. Creating our Custom Directive


43. Services we usually use within Directives


44. Host Element


45. Example


46. DOM Interaction


47. Main Point


48. Pipes


49. Built-in Pipes


50. Custom Pipes


51. Pure and Impure Pipes


52. Async Pipe

keyvalue Pipe

You may pipe an Object through the **keyvalue** pipe, which will give you an array suitable for use within an ***ngFor**.

```
@Component({
   template: `<div *ngFor="let item of data | keyvalue">
                 {{item.key}} - {{item.value}}
              </div>`
})
export class MyComponent {
   data = { "key": "value", "key2": "value2" };
}
```


OUTLINE

Search...


43. Services we usually use within Directives


44. Host Element


45. Example


46. DOM Interaction


47. Main Point


48. Pipes


49. Built-in Pipes


50. Custom Pipes


51. Pure and Impure Pipes


52. Async Pipe


53. keyvalue Pipe