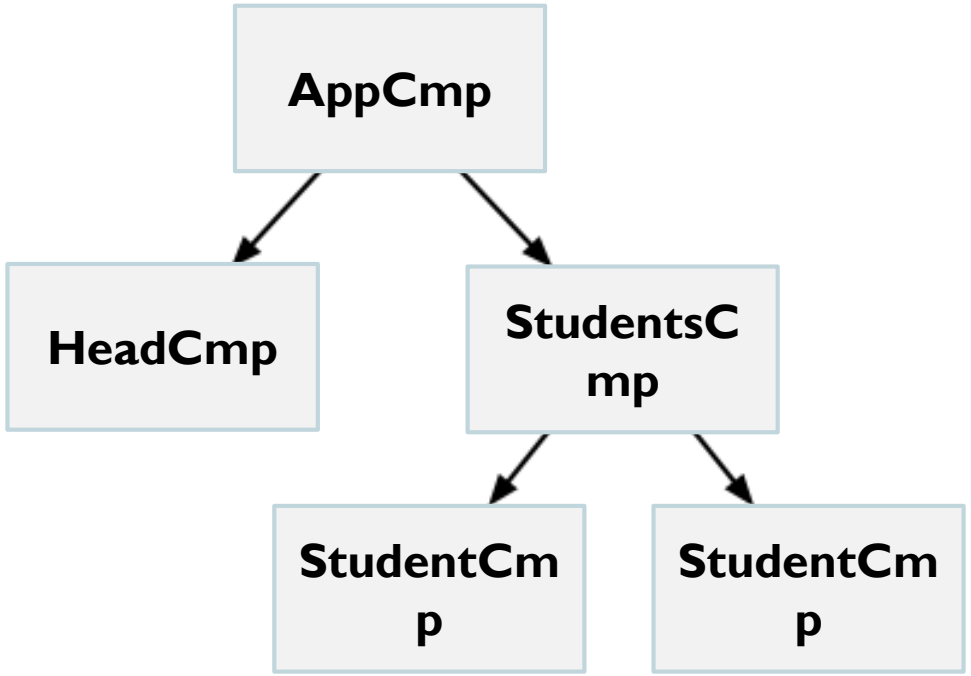
Components & Directives

Recall: Components

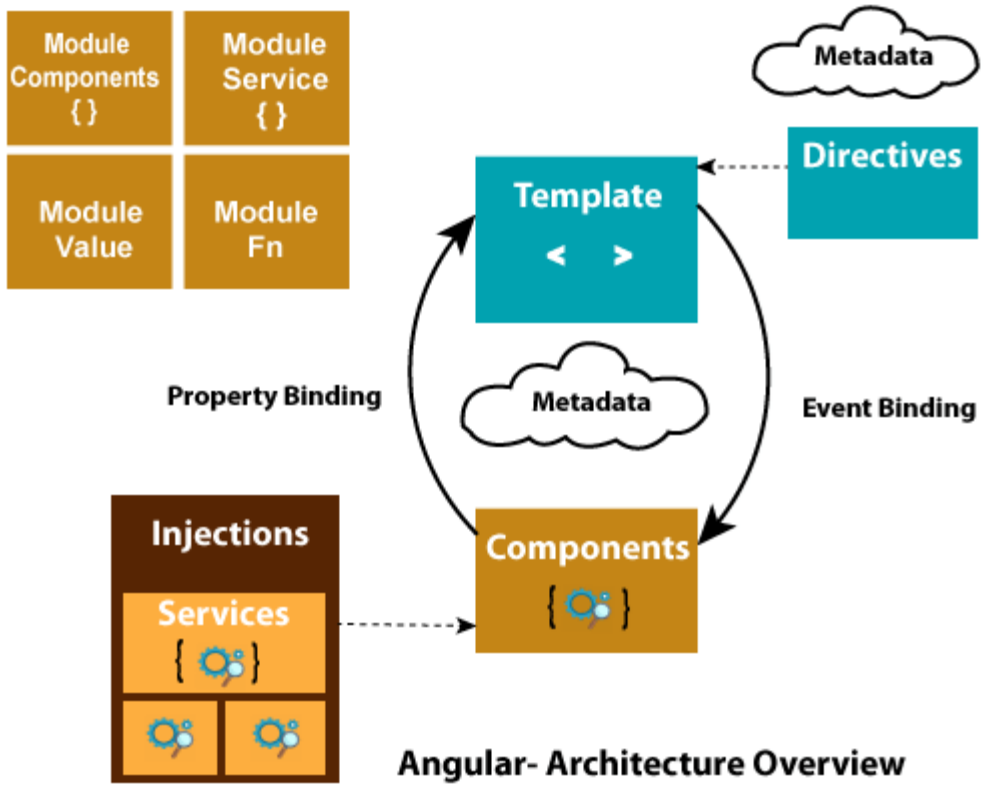
- Components and services both are simply classes with decorators that mark their types and provide metadata which guide Angular to do things.
- Every Angular application always has at least one component known as root component that connects a page hierarchy with page DOM.
- Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.



- Component contains 4 files:
  - controller: `*.component.ts`
  - View: `*.html`
  - Look & Feel: `*.css`
  - Unit testcase: `*.component.spec.ts`

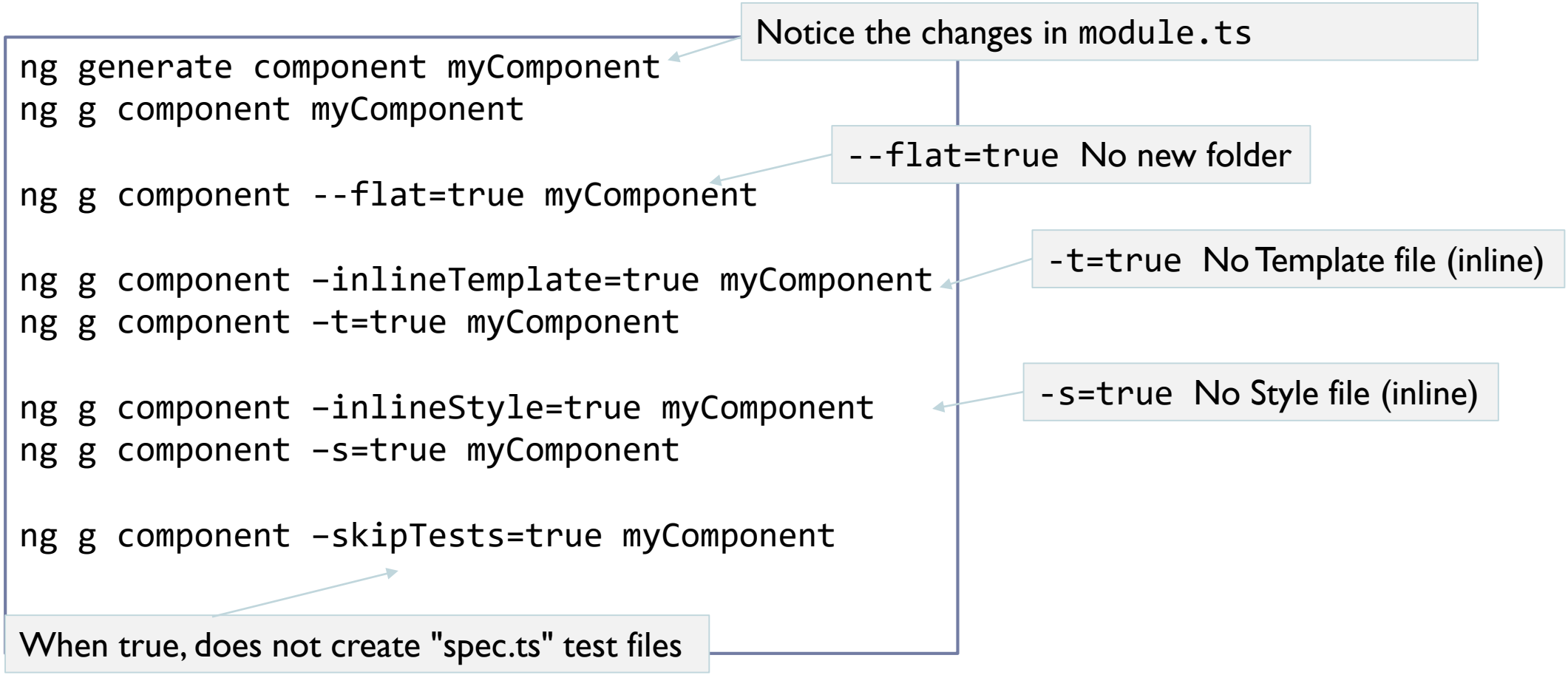
Metadata of Component class

- The metadata for a component class associates it with a template that defines a view.
  - A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.
- The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).



Creating a new Component

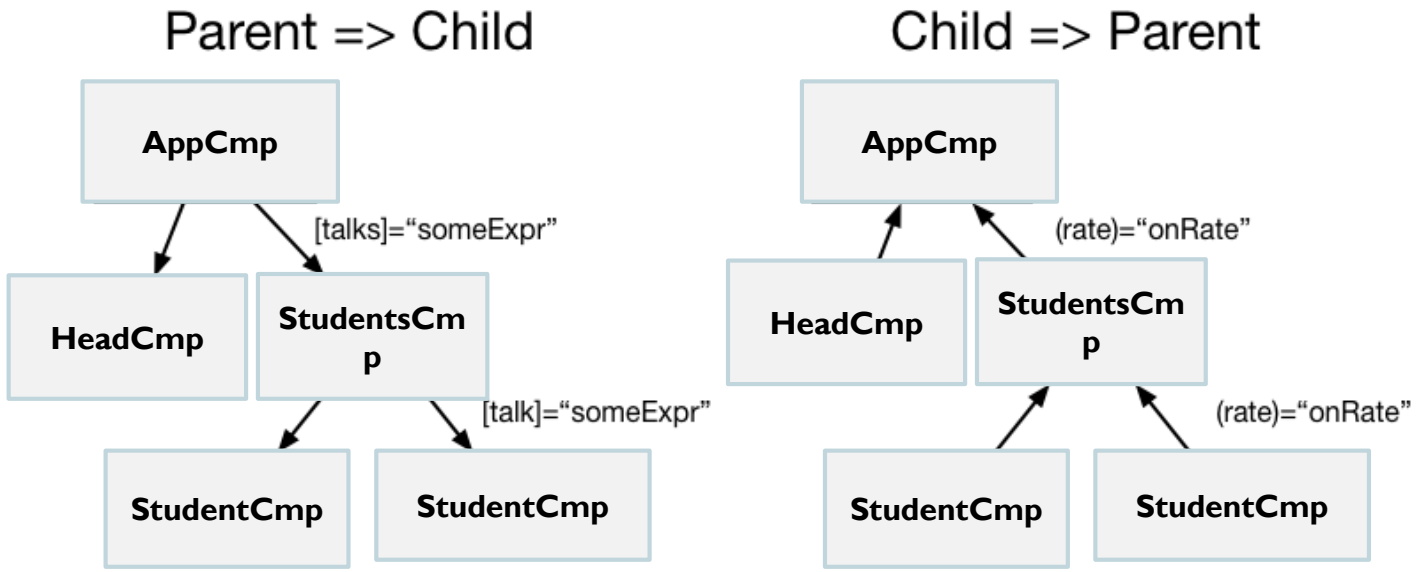
- You can build our component from scratch but it's easier to use Angular CLI



https://angular.io/cli/generate#component

Component Interaction

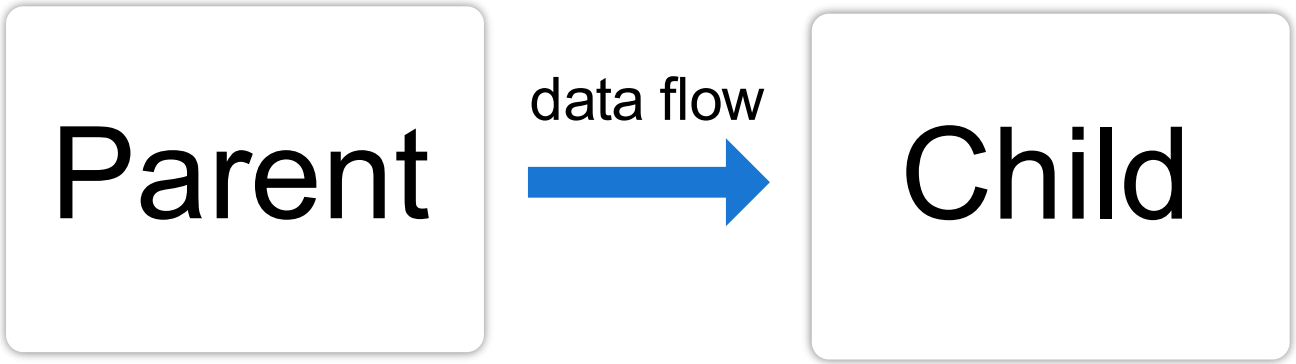
- A component has input and output properties, which can be defined in the component decorator or using property decorators.
- Data flows into a component via input properties. Data flows out of a component via output properties.
- `@Input()` and `@Output()` allow Angular to share data between the parent context and child directives or components. An `@Input()` property is writable while an `@Output()` property is observable.



@Input

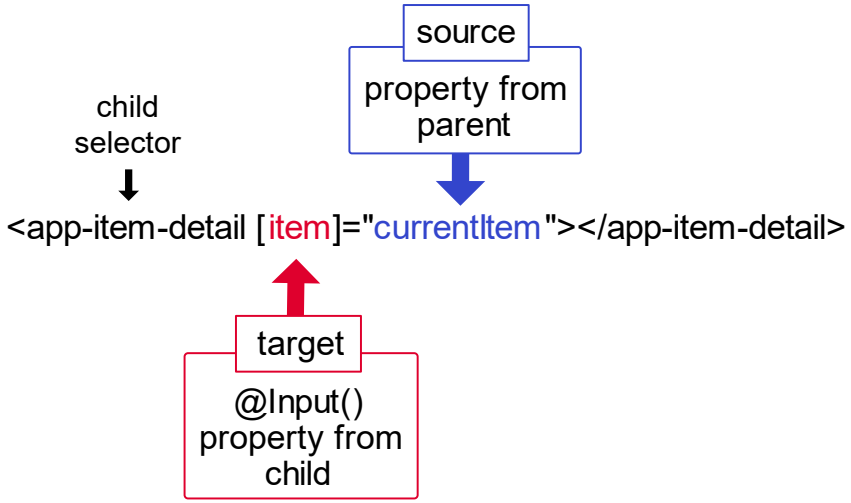
- Use the `@Input()` decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component.
- It helps to remember that the data flow is from the perspective of the child component.

@Input



@Input

```
@Component({
  selector: 'app-root',
  template: '<app-item-detail [item]="currentItem"></app-item-
detail>',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  currentItem = 'Television';
}
```




```
@Component({
  selector: 'app-item-detail',
  template: `
  <h2>Child component with @Input()</h2>
  <p>
    Today's item: {{item}}
  </p>
  `,
  styles: []
})
export class ItemDetailComponent {
  @Input() item: string;
}
```

Component inputs property

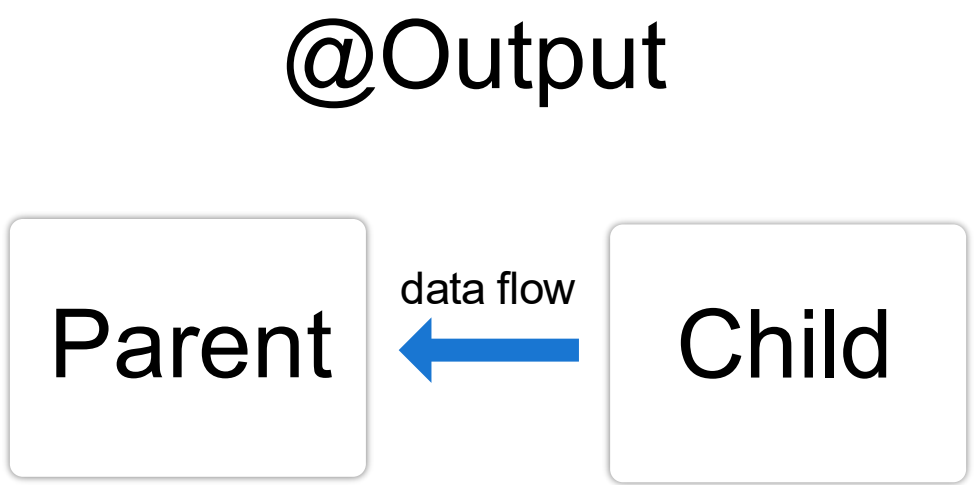
```
@Component({
  selector: 'app-root',
  template: '<app-item-detail [item]="currentItem"></app-item-
detail>',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  currentItem = 'Television';
}
```

With the `inputs` property, we specify the parameters we expect our component to receive. Inputs takes an array of strings which specify the input keys.
When we specify that a Component takes an input, it is expected that the definition class will have an instance variable that will receive the value.

```
@Component({
  selector: 'app-item-detail',
  template: `
  <h2>Child component with @Input()</h2>
  <p>
    Today's item: {{item}}
  </p>
  `,
  inputs: ['item']
})
export class ItemDetailComponent {
  item: string;
}
```

Output Property

- Use the `@Output()` decorator in the child component or directive to allow data to flow from the child out to the parent.
- An `@Output()` property should normally be initialized to an Angular EventEmitter with values flowing out of the component as events.
- You can also use `outputs: ['newItemEvent']` property instead of @Output decorator to achieve the same thing.



@Output

```
@Component({
  selector: 'app-root',
  template: `
    <app-item-output (newItemEvent)="addItem($event)"></app-item-output>
    <ul>
      <li *ngFor="let item of items">{{item}}</li>
    </ul>
  `,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  currentItem = 'Television';
  secondItem = "Computer";

  items = ['item1', 'item2', 'item3', 'item4'];

  addItem(newItem: string) {
    this.items.push(newItem);
  }
}
```

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

@Component({
  selector: 'app-item-output',
  template: `
  <label>Add an item: <input #newItem></label>
  <button (click)="addNewItem(newItem.value); newItem.value='';”>
      Add to parent's list
  </button>
  `,
  styles: []
})
export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }

}
```

Templates

- In a component, there are two ways to configure a template:
  - `templateUrl`
    - The relative path or absolute URL of a template file for an Angular component. If provided, do not supply an inline template using template.
  - `template`
    - An inline template for an Angular component. If provided, do not supply a template file using templateUrl.
    - Wins last

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
```

```
@Component({
  selector: 'app-root',
  template: `
    <app-item-detail [item]="currentItem"></app-item-detail>
`
})
```

Styles

- Use Bootstrap 4 with Angular
  - Install bootstrap: `npm install bootstrap -save`
  - Add to `src/styles.css`: `@import '~bootstrap/dist/css/bootstrap.css';`
  - This is Global Setting, applied on all components.
  - You also have other ways like configure in `angular.json`.
- Component Targeted Styles
  - `styleUrls: string[]`
    - One or more relative paths or absolute URLs for files containing CSS stylesheets to use in this component.
  - `styles: string[]`
    - One or more inline CSS stylesheets to use in this component.
    - Wins over `styleUrls`

Styles Example

```
@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
  styles: [`
    p {
      color: blue;
    }

    .p-content{
      background: red;
    }
  `],
  styleUrls: ['./item-list.component.css']
})
export class ItemListComponent {
}
```

Item-List works!

item-list.component.html

```
<style>
    p {
        color: Yellow;
    }

    .p-content {
        background: black;
    }
</style>

<p class="p-content">
    Item-List works!
</p>
```

item-list.component.css

```
p {
    color: pink;
}
```

View Encapsulation

- The 3 states of view encapsulation in Angular are:
  - `None`: All elements/styles are leaked - no Shadow DOM at all.
  - `Emulated`: Default Setting. Emulate Native scoping of styles by adding an attribute containing surrogate id to the Host Element and pre-processing the style rules provided via styles or `styleUrls`, and adding the new Host Element attribute to all selectors.
  - `ShadowDom`: Use Shadow DOM to encapsulate styles. Not supported by older browsers.

```
@Component({
        templateUrl: 'test.html',
        // encapsulation: ViewEncapsulation.ShadowDom
        // encapsulation: ViewEncapsulation.None
        // encapsulation: ViewEncapsulation.Emulated is default
})
```

Shadow DOM

- Shadow DOM will allow us to apply Scoped Styles to elements without affecting other elements.

```
<body>
    <p>This is a paragraph...</p>
    <div id="mydiv"></div>
    <script>
        let divElem = document.getElementById('mydiv');
        divElem.innerHTML = `
            <p>This is my second paragraph!!!!!!!!</p>
            <style>p{color: pink;}<style>
        `;
    </script>
</body>
```

This is a paragraph...

This is my second paragraph!!!!!!!!

```
<body>
    <p>This is a paragraph...</p>
    <div id="mydiv"></div>
    <script>
        let divElem = document.getElementById('mydiv');
        let shadowRoot = divElem.createShadowRoot();
        shadowRoot.innerHTML = `
            <p>This is my second paragraph!!!!!!!!</p>
            <style>p{color: pink;}<style>
        `;
    </script>
</body>
```

This is a paragraph...

This is my second paragraph!!!!!!!!

```
▼<body>
    <p>This is a paragraph...</p>
  ▼<div id="mydiv"> == $0
    ▼#shadow-root (open)
        <p>This is my second paragraph!!!!!!!!</p>
        <style>p{color: pink;}<style>
                </style>
    </div>
  ▶<script>…</script>
```

View Encapsulation Example

```
@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
  styles: [`
    p {
      color: blue;
    }

    .p-content{
      background: red;
    }
  `],
  styleUrls: ['./item-list.component.css'],
  // encapsulation: ViewEncapsulation.ShadowDom

  encapsulation: ViewEncapsulation.Emulated

  // encapsulation: ViewEncapsulation.None
})
export class ItemListComponent {
}
```

```
<app-item-list _ngcontent-gpo-c14>
  #shadow-root (open)
    <style>…</style>
    <style>…</style>
    <style>…</style>
    <style>…</style>
    <p class="p-content"> Item-List works!
    </p> == $0
  </app-item-list>
```

```
<app-item-list _ngcontent-acv-c14 _nghost-acv-c13>
  <p _ngcontent-acv-c13 class="p-content"> Item-List works!
  </p> == $0
</app-item-list>

<style> == $0
  p[_ngcontent-acv-c13] {
      color: Yellow;
  }

  .p-content[_ngcontent-acv-c13] {
      background: black;
  }
</style>
```

```
<app-item-list _ngcontent-cmx-c14>
  <p class="p-content"> Item-List works!
  </p> == $0
</app-item-list>

<style>
  p {
      color: Yellow;
  }

  .p-content {
      background: black;
  }
</style>
```

<ng-content>

- <ng-content>: used to create configurable components
- Well known as **Content Projection**.
- Components that are used in published libraries make use of <ng-content> to make themselves configurable.

```
@Component({
  selector: 'app-reusable-panel',
  templateUrl: './reusable-panel.component.html'
})
export class ReusablePanelComponent {
}

<div class="card">
    <div class="card-header">
        <ng-content></ng-content>
    </div>
    <div class="card-body">
        <ng-content select="p"></ng-content>
    </div>
</div>
```

app.component.html

```
<app-reusable-panel>
    <div>This is heading from App Component</div>
    <p>This is panel body from App Component using HTML
 element selector</p>
</app-reusable-panel>
```

```
▼<app-reusable-panel _ngcontent-otw-c15>
  ▼<div class="card">
    ▼<div class="card-header">
        <div _ngcontent-otw-c15>This is heading from App
        Component</div> == $0
      </div>
    ▼<div class="card-body">
        <p _ngcontent-otw-c15>This is panel body from App
        Component using HTML element selector</p>
      </div>
  </div>
</app-reusable-panel>
```

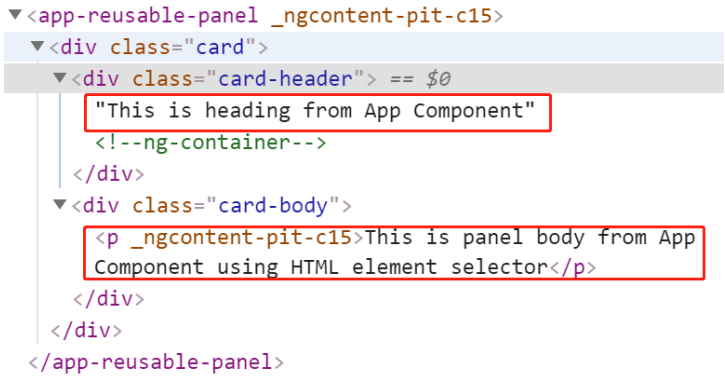
<ng-container>

- The `ng-container` directive provides us with an element that we can attach a structural directive to a section of the page, without having to create an extra element just for that

```
@Component({
  selector: 'app-reusable-panel',
  templateUrl: './reusable-panel.component.html'
})
export class ReusablePanelComponent {
}

<div class="card">
    <div class="card-header">
        <ng-content></ng-content>
    </div>
    <div class="card-body">
        <ng-content select="p"></ng-content>
    </div>
</div>
```

app.component.html

```
<app-reusable-panel>
    <ng-container>
        This is heading from App Component
    </ng-container>
    <p>
This is panel body from App Component using HTML element
 selector
    </p>
</app-reusable-panel>
```



Directives

Directives

- The Angular directives are used to manipulate the DOM. By using Angular directives, you can change the appearance, behavior or a layout of a DOM element. It also helps you to extend HTML.

- There are three kinds of directives in Angular:
  - Components — directives with a template. Any Component is a directive with a template.

  - Structural directives — change the DOM layout by adding and removing DOM elements. Structural directives start with a * sign.
    - ***ngIf Directive:** The ngIf allows us to Add/Remove DOM Element.
    - ***ngSwitch Directive:** The *ngSwitch allows us to Add/Remove DOM Element.
    - ***ngFor Directive:** The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).

  - Attribute directives — change the appearance or behavior of an element, component, or another directive.
    - **ngClass Directive:** The ngClass directive is used to add or remove CSS classes to an HTML element.
    - **ngStyle Directive:** The ngStyle directive facilitates you to modify the style of an HTML element using the expression. You can also use ngStyle directive to dynamically change the style of your HTML element

Structural directives - *ngIf

- The ngIf Directives is used to add or remove HTML Elements according to the expression.
- The expression must return a Boolean value. If the expression is false then the element is removed, otherwise element is inserted.

```
@Component({
  selector: 'app-directive-demos',
  templateUrl: './directive-
demos.component.html',
  styles: []
})
export class DirectiveDemosComponent {
  condition = true;
}
```

```
<p *ngIf="condition">
    condition is true and ngIf is true.
</p>
<p *ngIf="!condition">
    condition is false and ngIf is false.
</p>

<div *ngIf="condition; else elseBlock">
    Content to render when condition is true.
</div>
<ng-template #elseBlock>
    Content to render when condition is false.
</ng-template>
```

Structural directives - *ngFor

- The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).
- Some local variables like index, first, last, odd and even are exported by *ngFor directive.

```
export class DirectiveDemosComponent {
  courses: string[] = ["Node", "MongoDB", "Angular"];
}
```

```
<ul>
    <li *ngFor="let course of courses; index as i;">{{index}}-{{course}}</li>
</ul>
```

Structural directives
- [ngSwitch], *ngSwitchCase, *ngSwitchDefault

- ngSwitch is a structural directive which is used to Add/Remove DOM Element.
- The ngSwitch directive is applied to the container element with a switch expression.

```
export class DirectiveDemosComponent {
  items = [{ name: 'One', val: 1 }, { name: 'Two', val: 2 }, { name: 'Three', val: 3 }];
  selectedValue: string = 'two';
}
```

```
<div [ngSwitch]="selectedValue">
    <div *ngSwitchCase="'One'">One is Selected</div>
    <div *ngSwitchCase="'Two'">Two is Selected</div>
    <div *ngSwitchDefault>Default Option</div>
</div>
```

Attribute directives - [ngClass]

- The NgClass directive is used via [ngClass] selector
- Adds and removes CSS classes on an HTML element.

```
export class DirectiveDemosComponent {

  isSuccess = true;

  changeColor() {
    this.isSuccess = !this.isSuccess;
  }
}
```

```
<button class="btn"
    [ngClass]="{
    'btn-success': isSuccess,
    'btn-danger': !isSuccess
    }"
    (click)="changeColor()">
ngClass Save
</button>
```

Attribute directives - [ngStyle]

- An attribute directive that updates styles for the containing HTML element.
- Sets one or more style properties, specified as colon-separated key-value pairs.

```
export class DirectiveDemosComponent {

  isDiv = true;

}
```

```
<div
    [ngStyle]="{
    'backgroundColor': isDiv? 'gray': 'red',
    'color': isDiv? 'pink': 'black'}"
    >This is a div...</div>
```

Custom Directives

- To create a new custom Directive class from Angular CLI we use:
  - `ng generate directive <name> [options]`
  - `ng g directive <name> [options]`

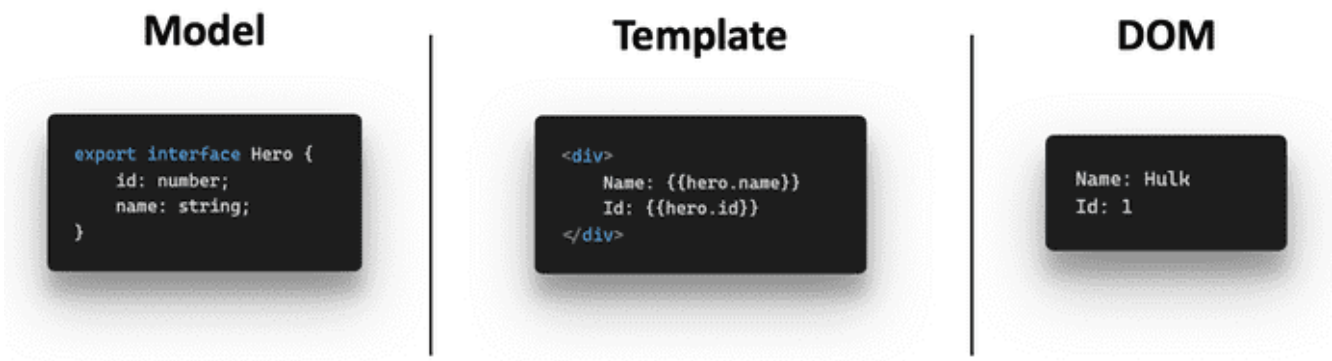
```
import { Directive, Renderer2, ElementRef } from '@angular/core';

@Directive({
  selector: '[appRoundBlock]'
})
export class RoundBlockDirective {

  constructor(renderer: Renderer2, elmRef: ElementRef) {
    renderer.setStyle(elmRef.nativeElement, 'border-radius', '100px');
  }

}
```

```
<div style="border: 1px solid blue" appRoundBlock>
        This is a custom diretive demo
</div>
```

Change Detection

What is Change Detection?

- The Angular framework needs to replicate the state of our application on the UI by combining the state and the template:



- It is also necessary to update the view if any changes happen to the state. This mechanism of syncing the HTML with our data is called "Change Detection".
- Change Detection: *The process of updating the view (DOM) when the data has changed*

What can cause the Change?

- Basically application state change can be caused by three things:
  - **Events** - click, submit, …
  - **XHR** - Fetching data from a remote server
  - MacroTasks - setTimeout(), setInterval()
  - MicroTasks - Promise.then()
  - Other async operations…

- They are all **asynchronous**. Which brings us to the conclusion that whenever some asynchronous operation has been performed, our application state should be changed. **This is when someone needs to tell Angular to update the view (Zones).**

Zones and execution contexts

- A zone provides an execution context that persists across async tasks.
- Execution Context is an abstract concept that holds information about the environment within the current code being executed.

The value of `this` in the callback of `setTimeout()` might differ depending on when `setTimeout()` is called.
Thus, you can lose the context in asynchronous operations.

```
const callback = function () {
    console.log('setTimeout callback context is', this);
}

const ctx1 = {
    name: 'ctx1'
};
const ctx2 = {
    name: 'ctx2'
};

const func = function () {
    console.log('caller context is', this);
    setTimeout(callback);
}

func.apply(ctx1);
func.apply(ctx2);
```

Zones and execution contexts

- A zone provides a new zone context other than this, the zone context that persists across asynchronous operations.

```
zone.run(() => {
            // now you are in a zone
            expect(zoneThis).toBe(zone);
            setTimeout(function() {
                // the zoneThis context will be the same zone
                // when the setTimeout is scheduled
                expect(zoneThis).toBe(zone);
            });
        });
```

This new context, `zoneThis`, can be retrieved from the `setTimeout()` callback function, and this context is the same when the `setTimeout()` is scheduled.

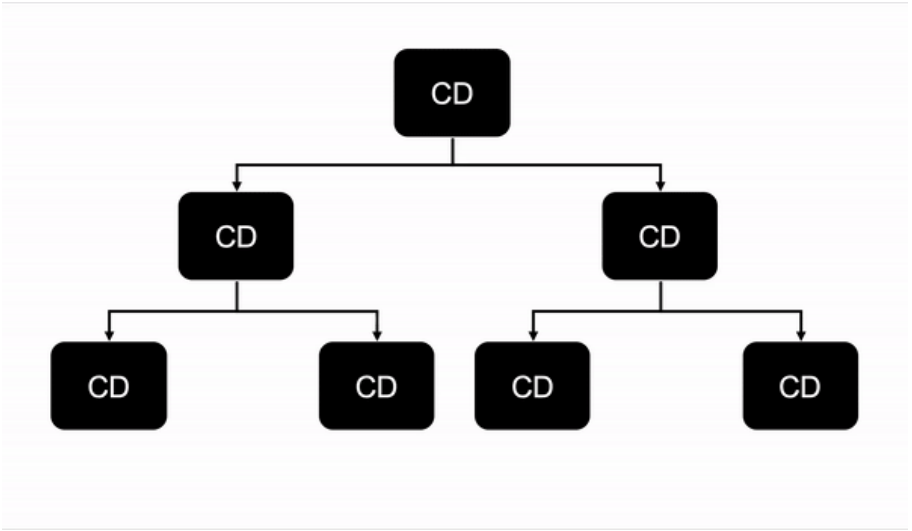
Zones notifies Angular about Changes

- 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 **change detection cycle**.

Somewhere in Angular source code, there's `ApplicationRef`, which listens to `NgZones onTurnDone` event. Whenever this event is fired, it executes a `tick()` function which essentially performs change detection.

How Change Detection Works?

- A change detection cycle can be split into two parts:
  - **Developer** updates the application model
  - **Angular** syncs the updated model in the view by re-rendering it
    - Angular detects the change
    - Change detection checks **every** component in the component tree from top to bottom to see if the corresponding model has changed
    - If there is a new value, it will update the component's view (DOM)



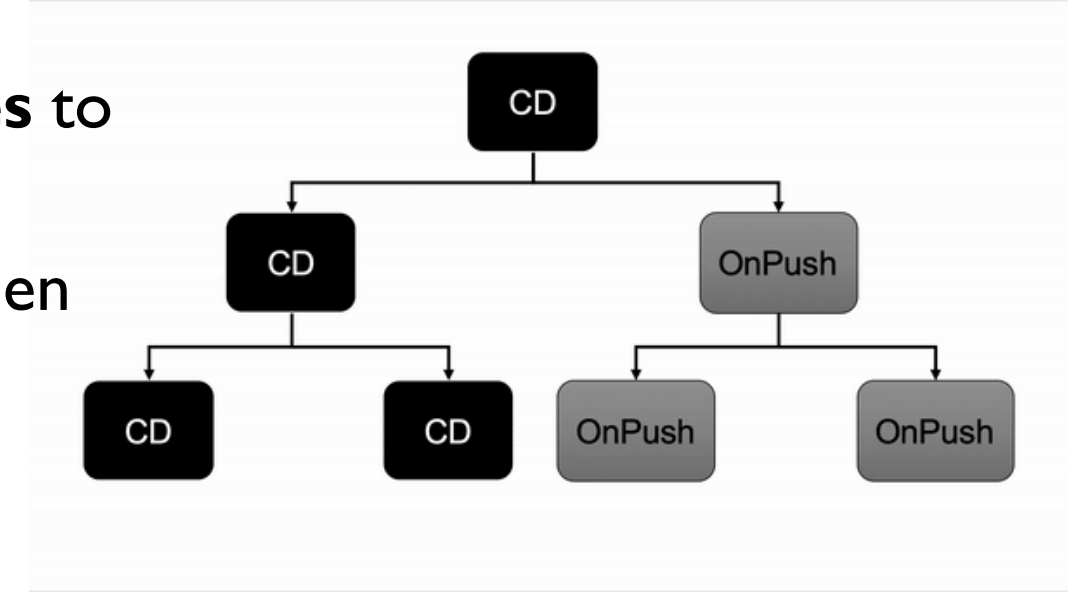
https://d33wubrfki0l68.cloudfront.net/43c03578c42f2333b28e9f2a6ab03b6d856f3f23/a7bdc/cf7351e3976cdc3041cadce5367fc318/cd-cycle.gif

Change Detection Strategies

- Angular provides two strategies to run change detections:
  - Default
    - Angular uses the `ChangeDetectionStrategy.Default` change detection strategy
    - **checks <u>every</u> component in the component tree from top to bottom** every time an event triggers change detection (like user event, timer, XHR, promise and so on)
    - negatively influence your application's performance in large applications which consists of many components
  - OnPush
    - See next Slides

OnPush Detection 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 can **skip change detection if its input hasn't changed**.
- You must use **Immutables** or **Observables** to use it.
- We can skip entire components subtrees when immutable objects are used and Angular is informed accordingly.




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

References

- Change Detection
  - https://angular.io/guide/zone
  - https://www.mokkapps.de/blog/the-last-guide-for-angular-change-detection-you-will-ever-need/
  - https://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html
- Directives
  - https://angular.io/guide/attribute-directives
  - https://angular.io/guide/structural-directives