Day01 Change Detection and State Management

Table of Contents

[Part 1 Project Setup 3](#_Toc152251037)

[Part 2 Change Detection with Classes 5](#_Toc152251038)

[Part 3 Implementing ChangeDectectorRef 8](#_Toc152251039)

[Part 4 Change Detection with Lists 9](#_Toc152251040)

[Part 5 List Actions and ChangeDetectionStrategy 11](#_Toc152251041)

[Part 6 List Actions and ChangeDetectorRef 13](#_Toc152251042)

[Part 7 Running Asynchronous Tasks Inside Zone.js 16](#_Toc152251043)

[Part 8 Running Asynchronous Tasks Outside Zone.js 18](#_Toc152251044)

[Appendix A – Angular Life-Cycle Hooks 21](#_Toc152251045)

[Appendix B – Enable Automatic Save in VSCode 23](#_Toc152251046)

# 

# Part 1 Project Startup

There is a starter project for today. It is called skills-ng16-CDSM-Day01-starter.zip. Unzip that file, rename the folder to just skills and run the npm install command against that folder. After that run ng build or ng serve. **You must use the name *skills***.

The project folder contains a basic boilerplate app and two other components. These files are almost the same from a previous boot camp, *components and templates*. Here we have a parent and two children components. From the parent app component, we call the first child. At the same time, we bind to a property on that child called messageFromParent. That child property is bound to the parents messageToChildren property, which is just a simple string for now.

In first-child we use the @Input() decorator to create the messageFromParent property and make it available to receive data from the parent app. Then in the child’s template we display that message using simple interpolation.

Run the application and click on the blue Change Message button to change the Hello message in the child component.

1. Just to demonstrate how this change system works, we can add log statements in strategic places. One of the easiest ways to do this is to use the ngDoCheck() method but AfterViewChecked() and ngOnChanges() can also be used:

|  |  |
| --- | --- |
| ngDoCheck(){  console.log("first-chilld change detected");  } | ngDoCheck(){  console.log("second-chilld change detected");  } |

Add these methods to the first-child and second-child components. In both cases they can go just below the ngOnInit() methods.

1. Run the application again by actually refreshing the browser. You will see the methods from #1 firing and logging the string:

|  |
| --- |
| A screenshot of a computer error message  Description automatically generated |

1. Now, clear the Console window and click the Change Message blue button:

|  |
| --- |
| A screenshot of a computer error message  Description automatically generated |

As you can see, the ngDoCheck() method fired in both children components.

1. There are two other situations where change detection occurs. One is running code outside of the zone.js zone and the other is when an http request completes. This will be an example of the first situation. This code can be inserted in any of the components, but I put it in the first-child.component.ts file:

|  |
| --- |
| constructor() { }  ngOnInit(): void {  setTimeout ( ( ) => {  console.log("This is printed after 2 seconds");  }, 2000 );  }  ngDoCheck(){ |

Run the app after inserting the setTimeout() method. You will see that change detection runs in both children components after two seconds.

1. The second scenario is when one of our components request data. For this demo, add the HttpClientModule package to your app.module.ts file, then add the module itself to the imports array:

|  |
| --- |
| import { BrowserModule } from '@angular/platform-browser';  import { HttpClientModule } from '@angular/common/http';  import { AppComponent } from './app.component';  …  ],  imports: [  BrowserModule,  HttpClientModule  ],  providers: [], |

1. Then in the second-child.component.ts file, import the HttpClient from @angular/common/http and inject the class via the constructor:

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { HttpClient } from "@angular/common/http"; … export class SecondChildComponent implements OnInit {  constructor( private http:HttpClient ) { } |

1. Still in the second-child.component.ts file, add this function:

|  |
| --- |
| getData(){  this.http.get<any>('https://jsonplaceholder.typicode.com/posts/1')  .subscribe({  next: data => {  console.log(data);  },  error: error => { console.log(error); }  })  } |

1. Now just trigger that function with a button on the template:

|  |
| --- |
| <p>second-child works!</p>  <div class="clearing">  <button (click) = 'getData()' > Get Data </button>  </div> |

1. Run the app, wait for the setTimeout() function to execute, clear the Console window and then click the Get Data button from #7. You should see something like in the image below:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

# Part 2 Change Detection with Classes

For Part2, you should delete the skills folder you created in Part1. Do this especially if you are on Mac or Linux. If you are using VS Code in these environments, always close the folder before moving on to a different project.

There is a separate starter file for Part 2. This is a zipped file named skills-ng16-CDSM-Day01-starter-for-part02. You may copy that to a folder and unzip it. Rename the folder to just skills. Please run the npm install command just like you did for Part 1. After that you may run the Angular command ng build to make sure there are no errors. Once you are satisfied, open the project folder, skills, in your VS Code or other editor.

You will notice that this project folder contains a new file, employee.ts. This file is in the app folder. We will use this file to demonstrate how change detection differs between objects and just plain properties. In other words, it behaves differently with a reference to an object, as opposed to a primitive type like numbers, strings and Booleans.

The project folder also contains an extra component first-child-child. It is a child component of first-child. This means that it is a grand-child of the app component. If you run the app now, you will see that even this grand-child runs the change detection cycle. The ngDoCheck() function has been activated in this component.

1. Start with the app.component.ts file and import Employee from the employee.ts file:

|  |
| --- |
| import { Component } from '@angular/core';  import { Employee } from "./employee"; |

1. In the class of the same file, create an instance of the Employee class and in the constructor assign a new value to the empname property of Employee:

|  |
| --- |
| export class AppComponent {  title : string = 'skills';  employee : Employee = new Employee();  constructor() {  this.employee.empname = 'Axle';  } |

Take a look at employee.ts to see the structure of that Employee object.

1. In the changeMessage() function, assign a new value to the empname property:

|  |
| --- |
| changeMessage(){  this.employee.empname = "Axle Barr";  } |

1. Move to the first-child component and also import the Employee class like you did for the app component. Change the @Input() decorator to the following:

|  |
| --- |
| export class FirstChildComponent implements OnInit {  @Input() employee! : Employee;  constructor() { } |

1. Then in the template of the first-child component, change the interpolation to the following:

|  |
| --- |
| <div>  <p>first-child works!</p>  <p> {{ employee.empname }} </p>  </div>  <div class = "grandChild"> |

1. Finally for the changes in the app.component.html file, change the app-first-child binding to the following:

|  |
| --- |
| <div>  <app-first-child  class = "firstChild"  [employee] = "employee"  >  </app-first-child>  <app-second-child |

1. Now run the application and it works just like before. Changes are being propagated just like before. You can remove all references to messageToChildren or messageFromParent. I left theses references for anyone who remembers the **Components** boot camp.
2. As you run this app, notice that all of the ngDoCheck() methods execute, even in the grand child. Also the template view changes, we go from just *Axle* to *Axle Barr*. However, if we now update the @Component decorator function to include a change detection strategy of OnPush, the template of the child component does not update. Do this in first-child.component.ts file:

|  |
| --- |
| @Component({  selector: 'app-first-child',  templateUrl: './first-child.component.html',  styleUrls: ['./first-child.component.css'],  changeDetection: ChangeDetectionStrategy.OnPush,  })  export class FirstChildComponent implements OnInit { |

Note that the ngDoCheck() message fires, it is only the template that is not updated. However the grand child is not affected. Change detection stops at first-child. This strategy has to be imported from @angular/core.

However in the app component, if we now update a reference to the employee object, the behavior changes. In the changeMessage() method of the app component, comment out the existing line and add in these lines to change the entire object:

|  |
| --- |
| changeMessage(){  //~~this.employee.empname = "Axle Barr";~~  this.employee = {  id : 1,  empname : "Axle Barr",  emppass : "Axle"  }  } |

The conclusion is that change detection with OnPush will only work with non reference types. Part3 below will show how non reference types can be coerced into firing change detection.

# Part 3 Implementing ChangeDectectorRef

1. Before continuing revert the changeMessage() function in the app component back to just changing the empname property. Leave the object itself but add comment lines to the entire object. Uncomment the first line. See below.
2. Comment all other lines except the first line in the changeMessage() method in app.component.ts file:

|  |
| --- |
| changeMessage(){  this.employee.empname = "Axle Barr";  ~~// this.employee = {~~  ~~// id : 1,~~  ~~// empname : "Axle Barr",~~  ~~// emppass : "Axle"~~  ~~// }~~  } |

1. Once again, with this change and the OnPush strategy in place, the grand child will not be affected by change detection. Also the view will not be updated, the name shown does not get changed.
2. We can now add a new method that will target and run change detection even though we have the OnPush strategy engaged and we are **not** working with references. In the first-child.component.ts file, import the ChangeDetectorRef Package from @angular/core:

|  |
| --- |
| import { Component, ChangeDetectorRef } from '@angular/core';  import { Employee } from "./employee";  @Component({ |

1. Inject the ChangeDetectorRef into the component via the constructor:

|  |
| --- |
| employee : Employee = new Employee();  constructor( private cd : ChangeDetectorRef ) { }  ngOnInit(): void { } |

1. At this time we still will not trigger change detection. We must find a method in the child component that will help with this action. Since we know that ngDoCheck() runs for sure, we can insert the detectChanges() method there:

|  |
| --- |
| constructor(private cd : ChangeDetectorRef ) { }  ngOnInit(): void { }  ngDoCheck(){  console.log("first-child change detected");  this.cd.detectChanges();  } |

We know ngDoCheck() runs since it prints in the console window.

# Part 4 Change Detection with Lists

For this part there is a new starter package called skillcards-ng16-CDSM-Day01-starter-for-part04.zip. Unzip that file into a folder and rename the unzipped folder to just skillcards.

If you are using VS Code, open a terminal and run npm install first. You may run a build or just go to serve.

This part will simulate a master-detail scenario and involve just four files. In modern web development, this scenario is sometimes referred to as cards. So, each detail is a card. A CSS file was provided along with an employee class file that structures our employee data.

1. Build the card now by adding a new component. Run the command   
   ng g c vacation --skip-tests -s. You should now have a new component in its own folder called vacation. The module file would be updated as well.
2. Begin to shape this card component by designing the main features. We will start with an outer pair of <div> tags and on the inside, insert a child <div> and an <input> tag:

|  |
| --- |
| <p>vacation works!</p>  <div>  <div></div>  <input />  </div> |

You can remove the initial <p> tag, we won’t b using it.

1. From this child component, we will communicate with the parent or app component via the @Input() decorator function:

|  |
| --- |
| export class VacationComponent {  @Input() employee!:Employee; |

Remember to import the Input class from @angular/core. Also, import the Employee class that’s in the starter folder.

1. With this in place, we can add code in the vacation template to read the various parts of the Employee class:

|  |
| --- |
| <p>vacation works!</p>  <div>  <div> {{employee.username}} </div>  <input type="checkbox" /> {{employee.onvacation}}  </div> |

We won’t be using the checkbox but this is a good feature to have as an employee’s status can be toggled using this box.

1. Call the vacation component from the parent template and pass the employee object. Since we have only three employees, we get three cards:

|  |  |
| --- | --- |
| <div>  <h1>On Vacation (´•‿•`)</h1>  </div>  <app-vacation  \*ngFor="let employee of employees;"  [employee]="employee"  >  </app-vacation> | A white background with black text and black text  Description automatically generated |

1. Lets add a CSS class in the vacation template to make the three rows more streamlined:

|  |
| --- |
| ~~<p>vacation works!</p>~~  <div class=”vaca”>  <div> {{employee.username}} </div>  <input type="checkbox" /> {{employee.onvacation}}  </div> |

Also remove the <p> tag and its contents

1. To log the number of times each card renders, we can add this piece of code to the template:

|  |
| --- |
| <div class="vaca">  <div>{{employee.username}}</div>  <input type="checkbox" /> {{employee.onvacation}}  <div> {{ rendered() }} </div>  </div> |

This is a very simple way to call a method on the component

1. That rendered() function will be in the component:

|  |
| --- |
| export class VacationComponent {  @Input() employee!:Employee;  rendered(){  console.log("Rendered");  } |

1. You may see something like this in the Console window:

|  |
| --- |
| A screenshot of a computer  Description automatically generatedNote: Angular checks twice in developer mode. |

# Part 5 List Actions and ChangeDetectionStrategy

We will now add various actions to the code so far. These actions will show how Angular handles the two main change detection strategies, OnPush and ChangeDetectorRef.

With a list like employees, we may want to add or subtract from this list and hence the database. We may also want to edit the list. Lets try to edit the list first, so add this method in the app component:

|  |
| --- |
| employees! : Employee[];  editEmployees(): void {  this.employees[1].username="Janice";  }  ngOnInit(): void { |

Here we change the name of the second employee from Jane to Janice. The employee list is in the app.component.ts file.

1. In the parent template, add code to call this method. Previously we used simple interpolation but now we can add a button to do this:

|  |
| --- |
| </app-vacation>  <div class="topp">  <button (click)="editEmployees()">Change array</button>  </div> |

1. Run the app and immediately clear the console window. This way you can see the log events once the button from #2 is clicked:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Two things happened, the name changed and each component card was re-rendered six times in developer mode. In production, it would be three. Still a lot.

1. Recall the discussion about reference vs primitive changes. Angular treats them differently. Lets now change our method to change the entire object:

|  |
| --- |
| employees! : Employee[];  editEmployees(): void {  //this.employees[1].username="Janice";  this.employees[1] = {...this.employees[0], username:"Janice"};  }  ngOnInit(): void { |

In this scenario, we use the spread operator to first make a deep copy of the entire object located at position 1 in the array. Then we create an entire object to replace that original object. The number of *rendered* is the same, but see below:

1. Change the change detection strategy on the child component to OnPush:

|  |
| --- |
| templateUrl: './vacation.component.html',  changeDetection: ChangeDetectionStrategy.OnPush,  })  export class VacationComponent { |

Remember to import the ChangeDetectionStrategy from @angular/core

1. When you run the app now, you see a major difference:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

The name did change and only one *rendered* was registered on my machine.

1. Reverse what you did in step #3 above, so add comments for the object change and uncomment the line where just the property was changed:

|  |
| --- |
| editEmployees(): void {  this.employees[1].username="Janice";  //this.employees[1] = {...this.employees[0], username:"Janice"};  } |

When ran, no changes were made on my system.

# Part 6 List Actions and ChangeDetectorRef

After change in Part5 #6 above, there should no longer be any rendering happening. But this may not be what you want. We can implement a technique based on the ChangeDctorRef like we did in Part3 #4, 5 and 6:

1. Continuing with the files from Part5, Inject the ChangeDetectorRef via the constructor of the vacation.component.ts file:

|  |
| --- |
| constructor ( private cd: ChangeDetectorRef ) { }  rendered(){  console.log("Rendered");  } |

Remember to import the ChangeDetectorRef from @angular/core.

1. Also add a click event to the checkbox on the vacation template:

|  |
| --- |
| <div class="vaca">  <div> {{employee.username}} </div>  <input type="checkbox" (click)="rendered()" /> {{employee.onvacation}}  <div> {{ rendered() }} </div>  </div> |

1. If you click on the Change Array button, nothing happens as we changed the inner component, not the parent. If you click on the checkbox the log fires and we see *rendered* in the Console window. However the name did not change!

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

An interesting situation here, if you click the Change Array button then check the box, the name changes.

1. Try putting this line in the rendered() function. The behavior should not change from #3 above:

|  |
| --- |
| constructor(private cd: ChangeDetectorRef) { }  rendered(){  console.log("Rendered");  this.cd.markForCheck();  } |

1. What would make more sense is to put the highlighted line in the ngOnChanges() life cycle hook method:

|  |
| --- |
| rendered(){  console.log("Rendered");  //this.cd.markForCheck();  }  ngOnChanges() {  this.cd.markForCheck;  } |

The behavior is the same, but this is the recommended way to trigger change in collaboration with the OnPush strategy. Notice that clicking the Change Array button does not trigger a render, but checking the individual checkboxes does.

Optional

The next few changes are optional if you want to see how to actually call a parent function from a child component. We covered this in the **Templates and Components** boot camp.

1. Start with the child component and add an @Output() decorator. Give your decorator a name, here it is toggleEmployee. Point that to a new EventEmitter object.

|  |
| --- |
| export class VacationComponent {  @Input() employee!:Employee;  @Output() toggleEmployee = new EventEmitter();  constructor(private cd: ChangeDetectorRef) { } |

There are several imports, let the editor guide you.

1. With this method we have to actually *emit* something, or at least call the emit() method, so write a new method for this in the same file:

|  |
| --- |
| ngOnChanges() {  this.cd.markForCheck;  }  changeName(){  this.toggleEmployee.emit();  } |

We will call this method from our template.

1. Over in the template, change the checkbox code to call the changeName() function:

|  |
| --- |
| <div class="vaca">  <div> {{employee.username}} </div>  <input type="checkbox" (click)="changeName()" /> {{employee.onvacation}}  <div> {{ rendered() }} </div>  </div> |

1. Final change for this section is to bind the child toggleEmployee emission to the editEmployees() function:

|  |
| --- |
| <app-vacation  \*ngFor="let employee of employees;"  [employee]="employee"  (toggleEmployee)="editEmployees()"  >  </app-vacation> |

Do this in the app.component.html file. So now, when the vacation component checks the checkbox, it emits a blank value. However that triggers the editEmployees() method in the parent class. Now if you check the Jane row, the name will change to Janice.

# Part 7 Running Asynchronous Tasks Inside Zone.js

We already know that NgZone is used by Angular to listen for events that require change detection. In core Angular code, Angular waits for the onTurnDone event. When this event fires, Angular runs change detection and updates the UI.

Note however that NgZone is a class that's useful to programmers also. It is one way of running asynchronous processes outside the Angular zone to improve performance. Of course we can run code inside the zone as well. This example illustrates both situations. Running code inside the zone is the default but it could slow down our app due to the many refresh cycles. Asynchronous task is similar to a parallel task.

We can run asynchronous tasks *outside* the default Angular zone. You would do this when you don't want change detection to keep updating the template.

In this example, we continue with the previous code including the optional part. We will only be working in the app component, not in the vacation component. We will simulate a long running process and show how NgZone can be used to control some aspects of this asynchronous call.

1. We start in the app.component.ts file and import the necessary classes:

|  |
| --- |
| import { Component, OnInit, NgZone, ChangeDetectorRef } from '@angular/core';  import { Employee } from "./employee"; |

1. Add the following properties. The counter variable will increment, displayDiv will be used to show/hide a pair of <div> tags and interval will eventually represent our asynchronous object:

|  |
| --- |
| export class AppComponent implements OnInit{  title = 'skillcards';  employees! : Employee[];  counter : number = 0;  displayDiv : number = 0;  interval : any;  editEmployees(): void { |

1. Also add a constructor and inject the two classes we imported in #1:

|  |
| --- |
| interval : any;  constructor(private ngZone : NgZone, private cdr : ChangeDetectorRef){}  editEmployees(): void { |

1. Add the first major function. This function will increment the counter we added in #2. Once the counter becomes equal to 10 or greater, the interval timer will be deleted and garbage collected behind the scenes. At this point we determine which of the three <div> tags we want to show:

|  |
| --- |
| constructor(private ngZone : NgZone, private cdr : ChangeDetectorRef){}  updateCount(){  this.counter++;  if ( this.counter >= 10 ) {  clearInterval(this.interval);  }  } |

Notice the counter, it increments every time this method gets called.

1. The final line here will display in the console window how many times the view was rendered:

|  |
| --- |
| clearInterval(this.interval);  }  console.log("Rendered: " + this.counter);  } |

1. The next important function is the startCount() function. This method will create our timer object. Once the timer is created, we pass it a function that points to the updateCount method from #4. This means that updateCount() will be called every half a second:

|  |
| --- |
| constructor(private ngZone : NgZone, private cdr : ChangeDetectorRef){}  startCount(){  this.counter = 0;  let intervalFn = () => {this.updateCount()};  this.interval = setInterval(intervalFn, 500);  }  updateCount(){ |

Note that when we create the interval counter object, we also store a reference to it and identify this reference with interval. We need this reference when we destroy the interval object. We do this in #4. I could have done this method in just one line, in fact I could reduce the number of methods. However, this way it is better for understanding.

1. Now the most important function, the one that decides if we call the other two functions the default, in-zone way or out of the zone. First we try in-zone:

|  |
| --- |
| constructor(private ngZone : NgZone, private cdr : ChangeDetectorRef){}  countDaysInZone(){  this.ngZone.run(() => {this.startCount()});  }  startCount(){ |

1. To test this method, we now have to organize our template. Let us do this in two steps. At the top of the parent html file, add these lines:

|  |
| --- |
| <div>  <h1>On Vacation (´•‿•`) for {{counter}} days!</h1>  </div> |

1. Below those lines add this pair of <div>s:

|  |
| --- |
| </div>  <div>  <button (click)="countDaysInZone()" class="topp">Start Count (in Angular)</button>  </div> |

1. The app should now run and update the counter on the HTML template. It should also show how many times the page was re-rendered in the console window. Note, if you press the Start Count button on the template again, it will just go to 11, then 12 and so on. To get it to restart you must re-initialize the counter, manually, in startCount() method. Obviously, this is just a learning experiment.

# Part 8 Running Asynchronous Tasks Outside Zone.js

Continuing from part 7, we now switch to running the startCount() method which in turn calls the updateCount() method, outside of the NgZone.

Now, the template has to work together with the TS code. Angular 17+ has a much better way of handling this type of situation. However, we will need to add lots of code to the front end to accomplish the various modes or states of our various processes.

Note, this part, has a gif that I created using just MS Paint. It is available inside of an assets folder and you can find it in the code for today.

Starting with the app template, add these three lines to the top of the HTML template:

|  |
| --- |
| <ng-container \*ngIf="displayDiv == 0; then onStart"></ng-container>  <ng-container \*ngIf="displayDiv == 1; then during"></ng-container>  <ng-container \*ngIf="displayDiv == 2; then allDone"></ng-container> <div>  <button (click)="countDaysInZone()" class="topp">Start Count (in Angular)</button>  </div> |

You may overwrite the initial line we had in part 1 or comment it out. The identifier to the right of the then in the above code will correspond to the three display modes in #2 below, via the template references in the lines below.

1. Add the following <ng-template> tags below the code from #1 above:

|  |
| --- |
| <ng-template #onStart>  <div>  <h1>On Vacation (´•‿•`) for 0 days!</h1>  </div>  </ng-template>  <ng-template #during>  <div>  <h1>On Vacation (´•‿•`) for <img src="./../assets/smiley.gif"> days!</h1>  </div>  </ng-template>  <ng-template #allDone>  <div>  <h1>On Vacation (´•‿•`) for {{counter}} days!</h1>  </div>  </ng-template> |

So, based on the value of displayDiv from #1, only one of these pairs of   
ng-template code will run. We add displayDiv in #5 below. Copy the smiley.gif file from the assets folder and place it in the assets folder of the current app.

1. Add this button to the button you already have. Notice that it calls a different starter function which we will code soon:

|  |
| --- |
| </div>  <div>  <button (click)="countDaysInZone()" class="topp">Start Count (in Angular)</button>  <button (click)="countDaysOutZone()" class="topp">Start Count (out Angular)</button>  </div> |

1. Now for the TS file. So in app.component.ts file, add this method:

|  |
| --- |
| }  countDaysOutZone(){  this.displayDiv = 1;  this.ngZone.runOutsideAngular(() => {this.startCount()});  }  startCount(){ |

So, now #3 and #4 are in sync. Notice that we start the display on the browser by applying mode 0 as the page loads (see #4 below). But now we change to 1 as the count starts. This mode will change depending on the part of the process we are in. Remember to remove the original <h1> we had in the beginning of the part.

1. Add a new property of the AppComponent to handle the display changes, so modes as I call them:

|  |
| --- |
| export class AppComponent implements OnInit{  title = 'skillcards';  employees! : Employee[];  counter : number = 0;  displayDiv : number = 0;  interval : any; |

So this is the starting or default mode, just zero. This corresponds to the ng-container in #1 and #2 above. So in this mode, zero, the #onStart <div> should show.

1. The updateCount() method now has to be adjusted. Since we are running outside of NgZone, we have to force the app to re-render the view. There are two ways to do this either with ChangeDetectorRef or with NgZone:

|  |
| --- |
| constructor(private ngZone : NgZone, private cdr : ChangeDetectorRef){}  updateCount(){  this.counter++;  if ( this.counter >= 10 ) {  clearInterval(this.interval);  this.displayDiv = 2;  this.cdr.detectChanges();  }  } |

Remember we added the ChangeDetectorRef class in part 7 #1.

1. You may now test the app. Notice how the message at the top of the browser window changes. It should go from the default to the animation and finally at the number of days being 10. The second method of forcing the template to re-render after the count, is to use NgZone:

|  |
| --- |
| updateCount(){  this.counter++;  if(this.counter >= 10){  clearInterval(this.interval);  this.displayDiv = 2;  //this.cdr.detectChanges();  this.ngZone.run(()=>{this.counter}); |

# Appendix A – Angular Life-Cycle Hooks

Angular life-cycle hooks are well documented on their website located at: <https://angular.io/guide/lifecycle-hooks>

In this section I will just touch on the hooks that affect the Component and give a few

The constructor is invoked as soon as an instance of the component is created. However, at this time the class properties aren’t initialized yet in the constructor.

When the constructor’s work is done, Angular will invoke the following lifecycle hooks. As a developer you can take advantage of these callback functions.

ngOnChanges()**—** As noted on the Angular website, this method fires on a regular basis. Once when the component is built and then once again anytime a bound property is changed. This method should be used with caution since it can slow down your App’s performance. If the component has no input properties, ngOnChanges() isn’t invoked.

This is the only life cycle method that takes a parameter. Once executed, this method will look for a SimpleChanges object passed in as a parameter. This parameter holds current and previous values. Think of it as keeping state at least temporarily.

This method is called **before** ngOnInit() if the component has bound inputs. So basically, any property that has the @Input decorator in front is the trigger for this method. Whenever these properties change this method fires.

ngOnInit()**—** This method is called AFTER ngOnChanges() and of course, the constructor method. By the time ngOnInit() is invoked, the component properties will have been created and initialized. This is usually a place where http requests are made. However, none of the projected content is available. So if there are any properties in the class that were decorated with @ViewChild, @ViewChildren, @ContentChild and @ContentChildren, those properties will not be available and may produce null results.

ngDoCheck()**—** Use the DoCheck hook to detect and react to changes that Angular doesn't catch on its own. In other words force Angular to run certain code. This method is very similar to ngOnChange(). It will run every time an event is fired whether a value got changed, or not.

This method is Called immediately after ngOnChanges() on every change detection cycle, and just after ngOnInit() the first time the component is created.

This is a good hook to write a custom change detection algorithm or add some debug code,

ngAfterContentInit()**—**This method is really only called if you used <ng-content> in your component’s HTML section. It is invoked when the child component’s properties are initialized and any projection completes.

ngAfterContentChecked()**—** again, this is run after Angular verifies that projected content is complete. This projected content can be for a component or a directive.

ngAfterViewInit()**—** Invoked after a component’s view or it’s children’s view has been fully initialized.

ngAfterViewChecked()**—** Fires after Angular checks the component's views and child views. It also runs after the view that contains the directive is checked. This callback may be called more than once as the result of modifications in the current component or in other components.

ngOnDestroy()**—** This method is fired just before the component, directive, pipe, or service is destroyed. The method must be configured on the object that has to be destroyed, but can be called from a parent object.

Just like the *finally* block, in a try/catch enclosure, this method is used to perform cleanup work:

Unsubscribe from Observables and/or DOM events

Stop any interval timers

Unregister any callbacks that the directive registered in the global space

Notify some other part of the application that the component will no longer be a part of the application

# Appendix B – Enable Automatic Save in VSCode

In VS Code, open Preferences by going to *File -> Preferences -> Settings*.

Once there, search for auto save.

A screenshot of a computer

Description automatically generated

Resources

<https://mokkapps.de/blog/the-last-guide-for-angular-change-detection-you-will-ever-need>

<https://www.thinktecture.com/en/angular/whats-the-hype-onpush/>