Advanced Components

Throughout this book, we've learned how to use Angular's built-in directives and how to create components of our own. In this chapter we'll take a deep dive into **advanced** features we can use to make components.

In this chapter we'll learn the following concepts:

- Styling components (with encapsulation)
- Modifying host DOM elements
- Modifying templates with *content projection*
- Accessing neighbor directives
- Using lifecycle hooks
- Detecting changes

How to Use This Chapter

This chapter gives a tour of advanced Angular APIs. It's assumed the reader is familiar with the basics of creating components, using built-in directives, and organizing component files.

As this is an intermediate/advanced level chapter, it's assumed the reader is able to fill in some of the basics (such as importing dependencies).

This chapter comes with a runnable code, found in the `advanced-components` folder. If at any time you feel you're lacking context, checkout the example code for this chapter.

To run the demos in this chapter, change into the project folder and run:

```
npm install
npm start
```

Then open your browser to http://localhost:42000

Styling

Angular provides a mechanism for specifying component-specific styles. CSS stands for *cascading style sheet*, but sometimes we **don't** want the cascade. Instead we want to provide styles for a component that won't leak out into the rest of our page.

Angular provides two attributes that allow us to define CSS classes for our component.

To define the style for our component, we use the View attribute `styles` to define in-line styles, or `styleUrls`, to use external CSS files. We can also declare those attributes directly on the `Component` decorator.

Let's write a component that uses inline styles:

code/advanced-components/src/app/styling/inline-style/inline-style.component.ts

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-inline-style',
5    styles: [`
6    .highlight {
7      border: 2px solid red;
8      background-color: yellow;
9      text-align: center;
10     margin-bottom: 20px;
11   }
12   `],
13   template: `
14   <h4 class="ui horizontal divider header">
15     Inline style example
16   </h4>
17
18   <div class="highlight">
19     This uses component <code>styles</code>
20     property
21   </div>
22   `
23 })
24 export class InlineStyleComponent {
25 }
```

In this example we defined the styles we want to use by declaring the `.highlight` class as an item on the array on the `styles` parameter.

Further on in the template we reference that class on the div using `<div class="highlight">`.

And the result is exactly what we expect - a div with a red border and yellow background:

Inline style example

This uses component `styles` property

Example of component using styles

Another way to declare CSS classes is to use the `styleUrls` property. This allows us to declare our CSS on an external file and just reference them from the component.

Let's write another component that uses this, but first let's create a file called `external.css` with the following class:

code/advanced-components/src/app/styling/external-style/external-style.component.css

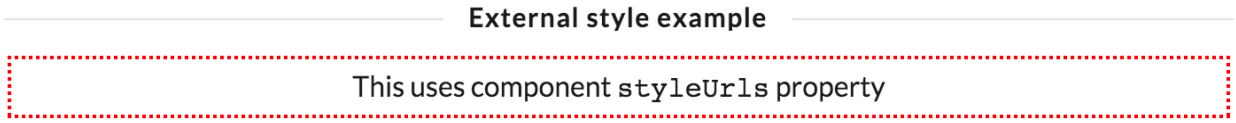
```
.highlight {
  border: 2px dotted red;
  text-align: center;
  margin-bottom: 20px;
}
```

Then we can write the code that references it:

code/advanced-components/src/app/styling/external-style/external-style.component.ts

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

@Component({
  selector: 'app-external-style',
  styleUrls: ['./external-style.component.css'],
  template: `
<h4 class="ui horizontal divider header">
  External style example
</h4>

<div class="highlight">
  This uses component <code>styleUrls</code>
  property
</div>
`
})
export class ExternalStyleComponent {
}
```

And when we load the page, we see our div with a dotted border:

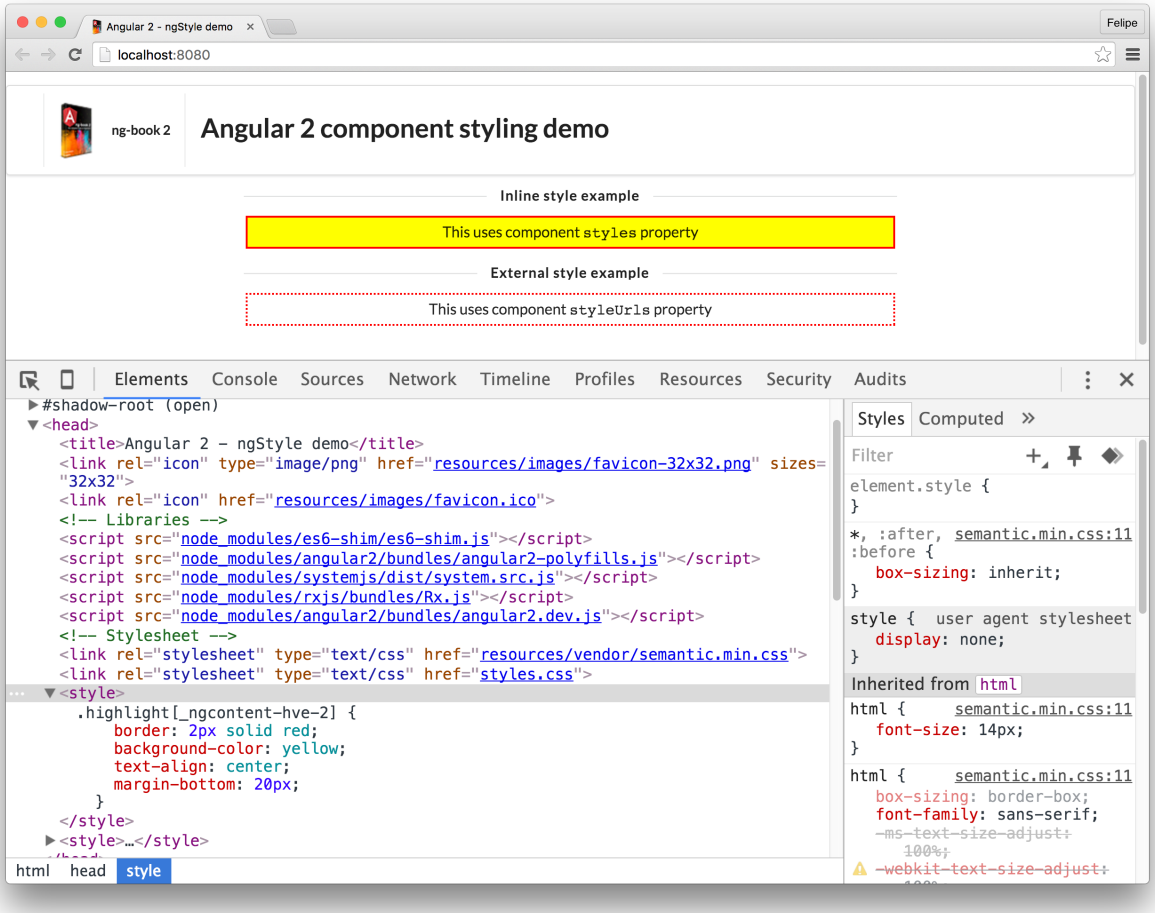


Example of component using styleUrls

View (Style) Encapsulation

One interesting thing about this example is that both components define a class called `highlight` with different properties, but the attributes of one didn't leak into the other.

This happens because Angular styles are **encapsulated by the component context** by default. If we inspect the page and expand the `<head>`, we'll notice that Angular injected a `<style>` tag with our style:

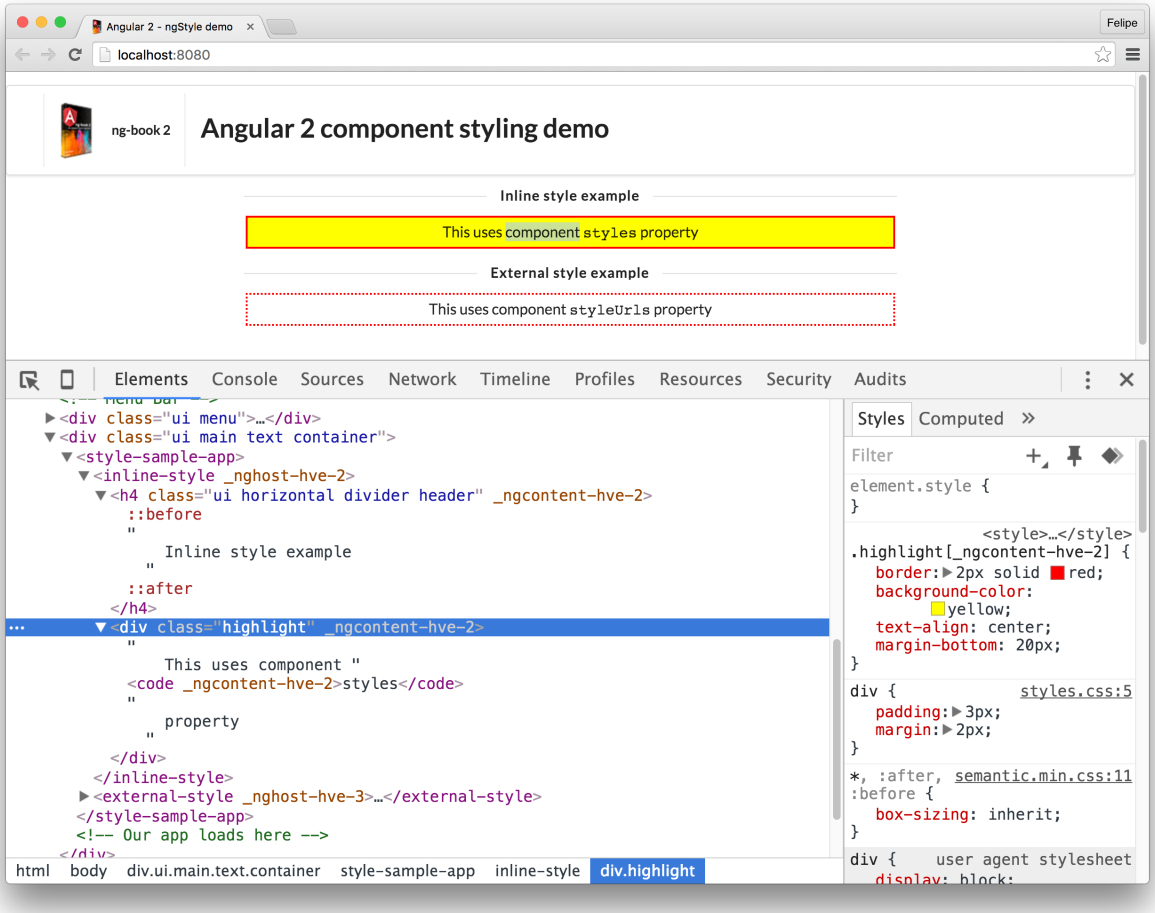


Injected style

You'll also notice that the CSS class has been scoped with `_ngcontent-hve-2`:

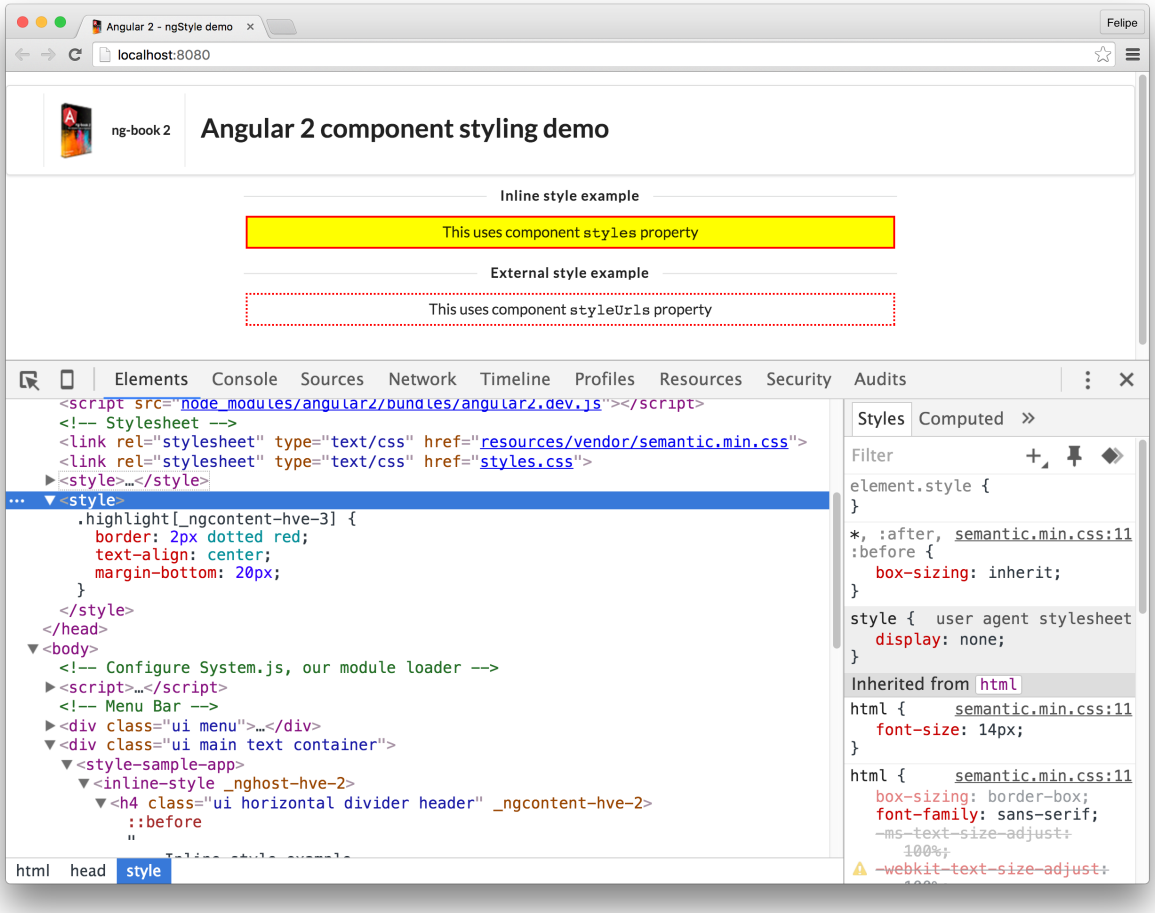
```
.highlight[\_ngcontent-hve-2] {
  border: 2px solid red;
  background-color: yellow;
  text-align: center;
  margin-bottom: 20px;
}
```

And if we check how our `<div>` is rendered, you'll find that `_ng-content-hve-2` was added:



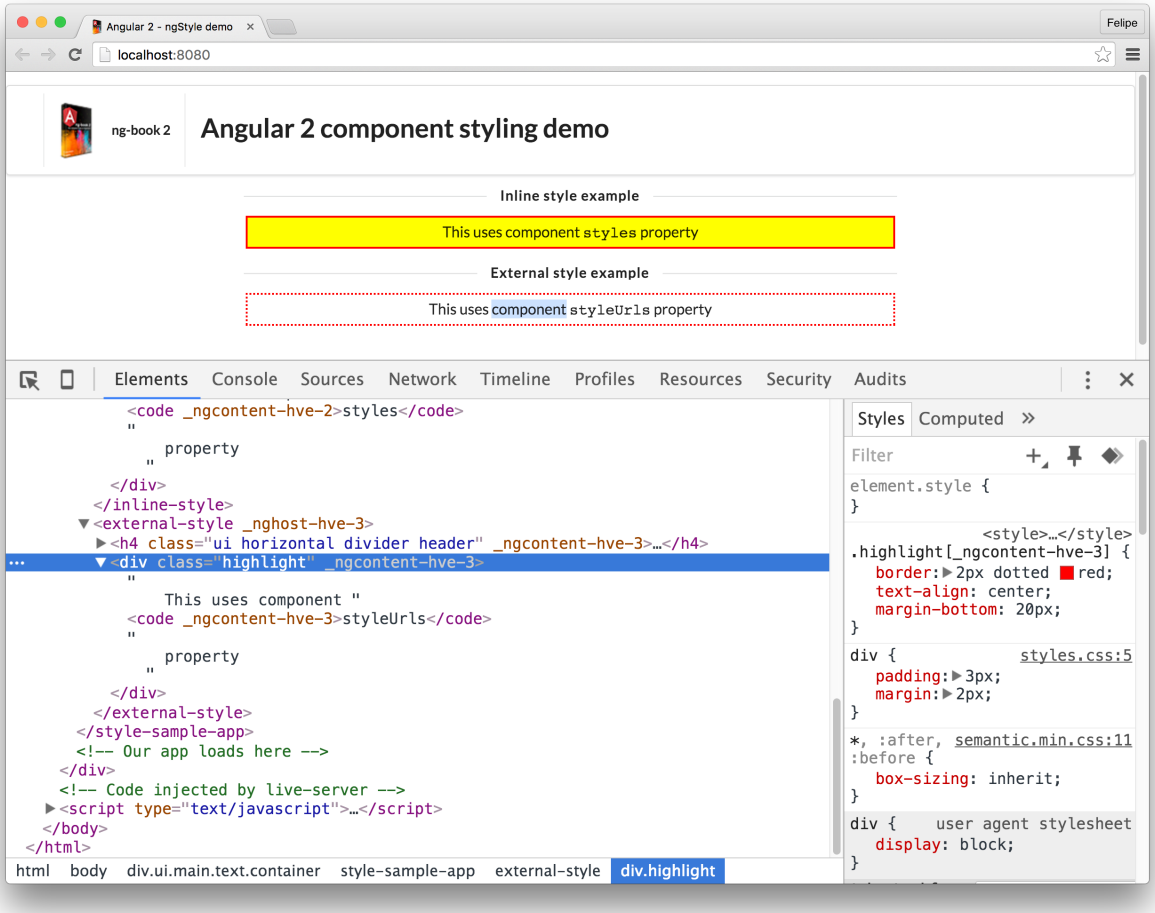
Injected style

The same thing happens for our external style:



External style

and:



External style

Angular allows us to change this behavior, by using the `encapsulation` property.

This property can have the following values, defined by the `ViewEncapsulation` enum:

- **Emulated** - this is the default option and it will encapsulate the styles using the technique we just explained above
- **Native** - with this option, Angular will use the Shadow DOM (more on this below)
- **None** - with this option set, Angular won't encapsulate the styles at all, allowing them to leak to other elements on the page

Shadow DOM Encapsulation

You might be wondering: what is the point of using the Shadow DOM? By using the Shadow DOM the component we **use a unique DOM tree that is hidden from the other elements on the page**. This allows styles defined within that element to be invisible to the rest of the page.

For a deep dive into Shadow DOM, please check this guide by Eric Bidelman[131].

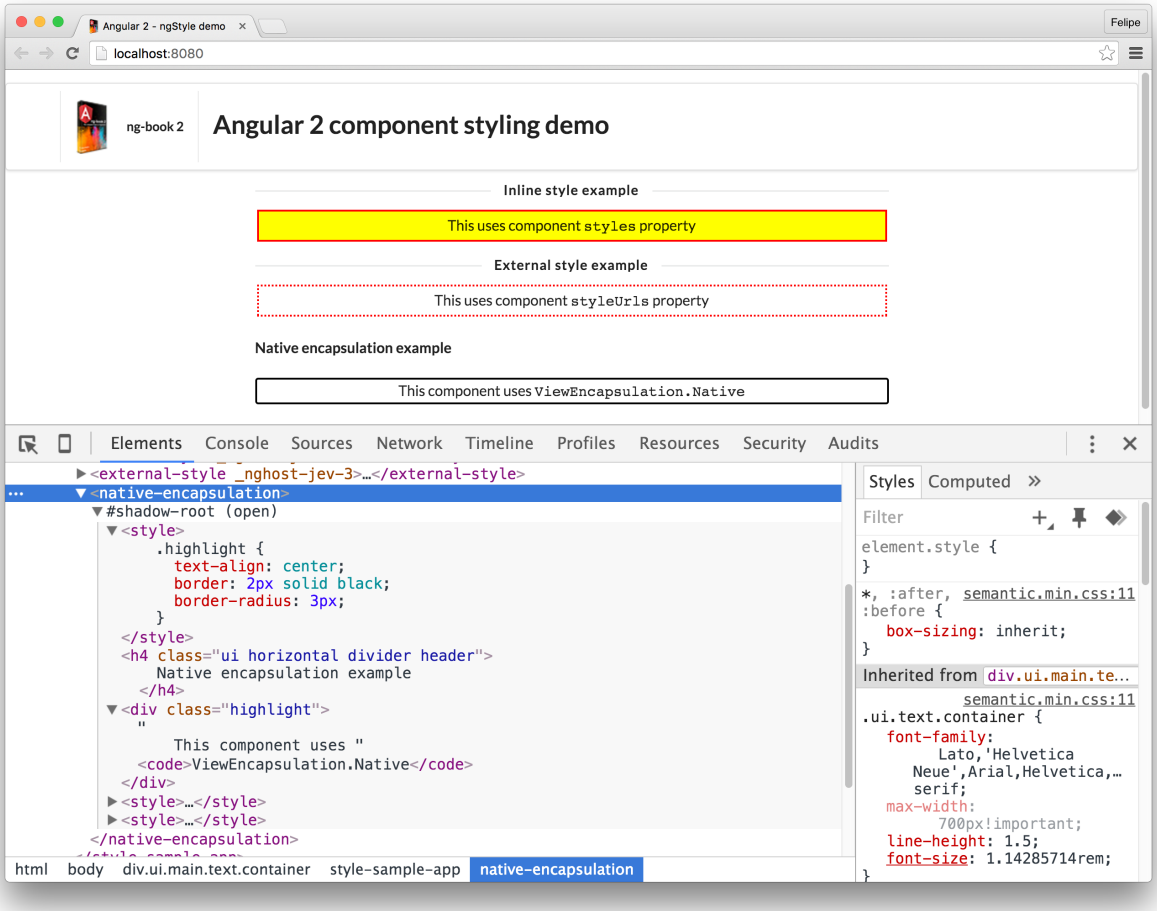
Let's create another component that uses the **Native** encapsulation (Shadow DOM) to understand how this works:

code/advanced-components/src/app/styling/native-encapsulation/native-encapsulation.component.ts

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

@Component({
  selector: 'app-native-encapsulation',
  styles: [`
  .highlight {
    text-align: center;
    border: 2px solid black;
    border-radius: 3px;
    margin-botton: 20px;
  }`],
  template: `
  <h4 class="ui horizontal divider header">
    Native encapsulation example
  </h4>

  <div class="highlight">
    This component uses <code>ViewEncapsulation.Native</code>
  </div>
  `,
  encapsulation: ViewEncapsulation.Native
})
export class NativeEncapsulationComponent {
}
```

In this case, if we inspect the source code, we'll see:

[131] http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/



Native encapsulation

Everything inside the `#shadow-root` element has been encapsulated and isolated from the rest of the page.

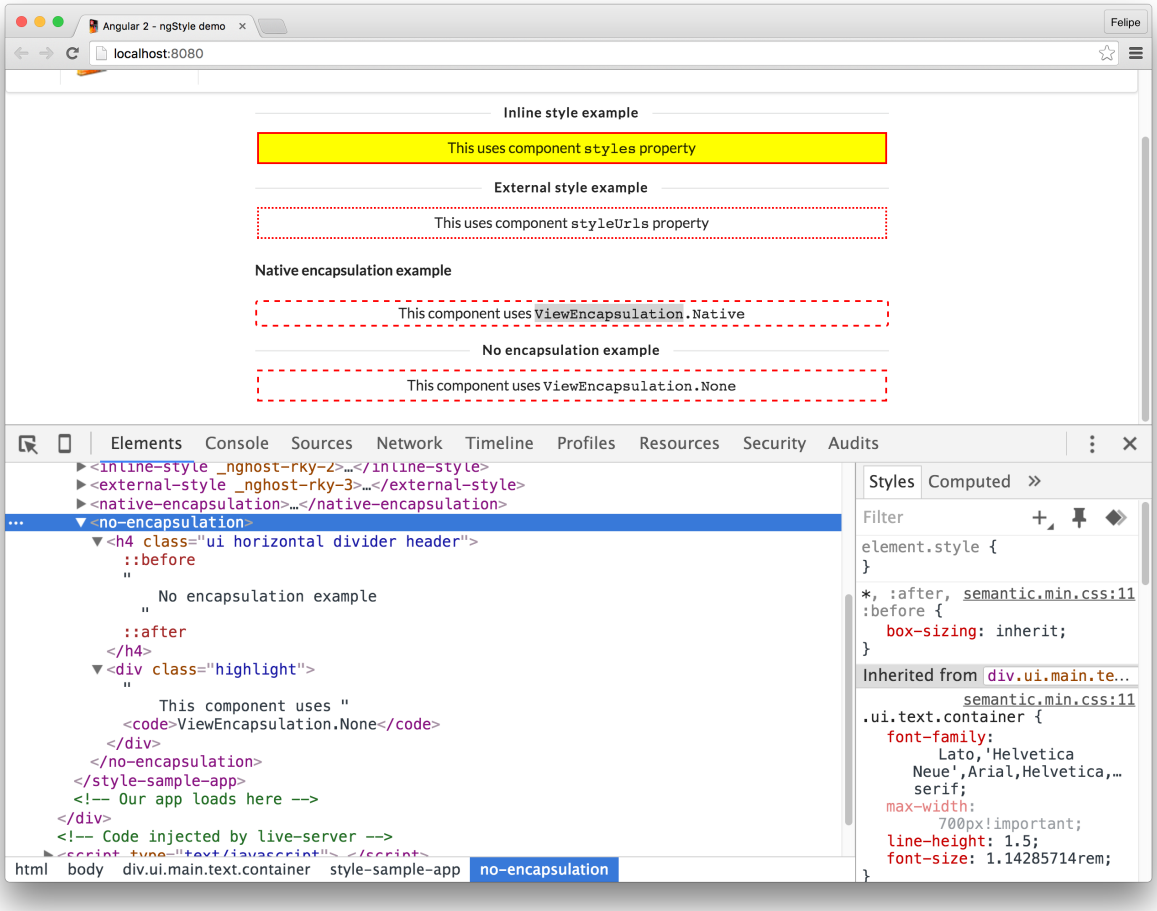
No Encapsulation

Finally, if we create a component that specifies `ViewEncapsulation.None`, no style encapsulation will be added:

code/advanced-components/src/app/styling/no-encapsulation/no-encapsulation.component.ts

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

@Component({
  selector: 'app-no-encapsulation',
  styles: [`
  .highlight {
    border: 2px dashed red;
    text-align: center;
    margin-bottom: 20px;
  }
  `],
  template: `
  <h4 class="ui horizontal divider header">
    No encapsulation example
  </h4>

  <div class="highlight">
    This component uses <code>ViewEncapsulation.None</code>
  </div>
  `,
  encapsulation: ViewEncapsulation.None
})
export class NoEncapsulationComponent {
}
```

When we inspect the element:



No encapsulation

We can see that nothing was injected on the HTML. Also on the header we can find that the `<style>` tag was also injected exactly like we defined on the `styles` parameter:

```
1  .highlight {
2    border: 2px dashed red;
3    text-align: center;
4    margin-bottom: 20px;
5  }
```

One side-effect of using `ViewEncapsulation.None` is that, since we don't have any encapsulation, this style "leaks" into other components. If we check the picture above, the `ViewEncapsulation.Native` component style was affected by this new component's style. But sometimes this can be exactly what you want.

You can comment out the `<app-no-encapsulation></app-no-encapsulation>` code on the `StyleSampleApp` template to see the difference.

Creating a Popup - Referencing and Modifying Host Elements

The *host element* is the element to which the directive or component is bound. Sometimes we have a component that needs to attach markup or behavior to its host element.

In this example, we're going to create a `Popup` directive that will attach behavior to its host element which will display a message when clicked.

Components vs. Directives - What's the difference?

Components and directives are closely related, but they are slightly different.

You may have heard that "components are directives with a view". This isn't exactly true. Components come with functionality that makes it easy to add views, but directives can have views too. In fact, **components are implemented with directives**.

One great example of a directive that renders a conditional view is `NgIf`.

But we can attach behaviors to an element **without a template** by using a *directive*.

Think of it this way: Components are Directives and Components always have a view. Directives may or may not have a view.

If you choose to render a view (a template) in your Directive, you can have more control over how that template is rendered. We'll talk more about how to use that control later in this chapter.

Popup Structure

Now let's write our first directive. We want this directive to **show an alert when we click a DOM element** that includes the attribute `popup`. The message displayed will be identified by the element's `message` attribute.

Here's what we want it to look like:

```
1  <element popup message="Some message"></element>
```

In order to make this directive work, there are a couple of things we need to do:

- receive the `message` attribute *from* the host
- be notified when the host element is clicked

Let's start coding our directive:

code/advanced-components/src/app/host/popup-demo/steps/host-1.ts

```
@Directive({
  selector: '[popup]'
})
export class PopupDirective {
  constructor() {
    console.log('Directive bound');
  }
}
```

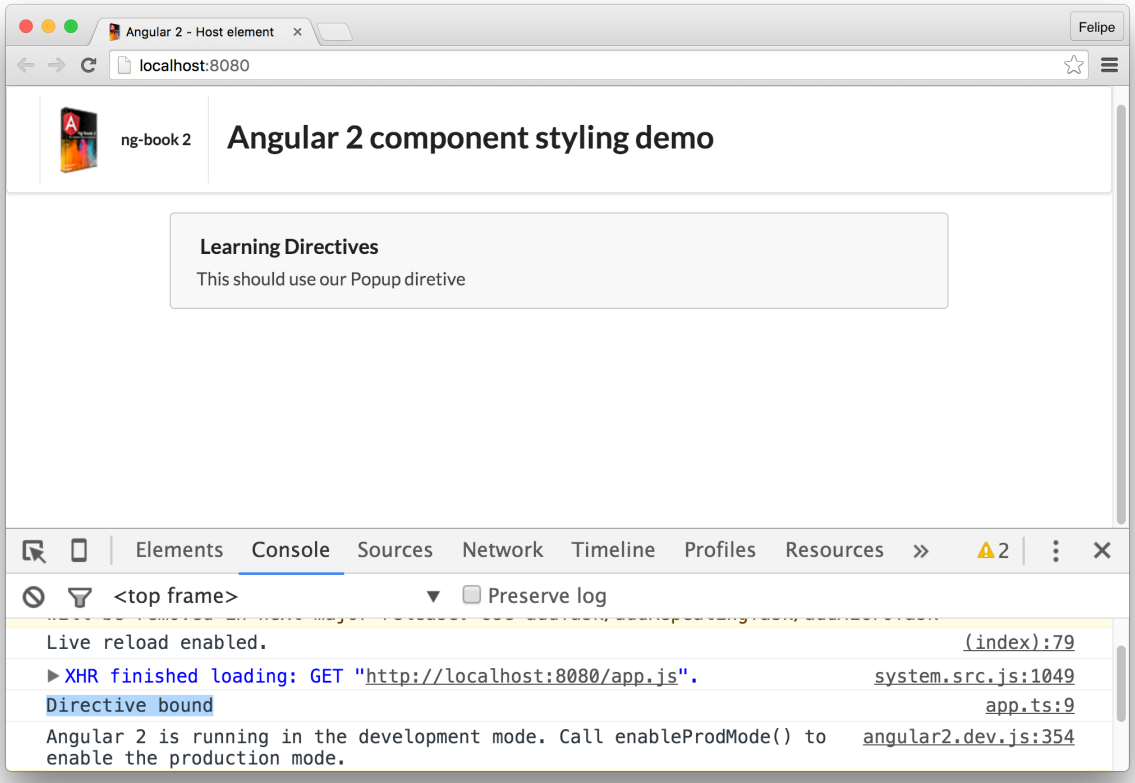
We use the `Directive` decorator and set the `selector` option to `[popup]`. This will make this directive bind to any elements that define the popup attribute.

Now let's create an app that has an element that has the `popup` attribute:

code/advanced-components/src/app/host/popup-demo/steps/host-1.ts

```
@Component({
  selector: 'app-popup-demo',
  template: `
  <div class="ui message" popup>
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>
  `
})
export class PopupDemoComponent1 {
}
```

When we run this application, we expect the message `Directive bound` to be logged on the console, indicating we have successfully bound to the first `<div>` in our template:



Binding to host element

Using `ElementRef`

If we want to learn more about the host element a directive is bound to, we can use the built-in `ElementRef` class.

This class holds the information about a given Angular element, including the native DOM element using the `nativeElement` property.

In order to see the elements our directive is binding to, we can change our directive constructor to receive the `ElementRef` and log it to the console:

code/advanced-components/src/app/host/popup-demo/steps/host-2.ts

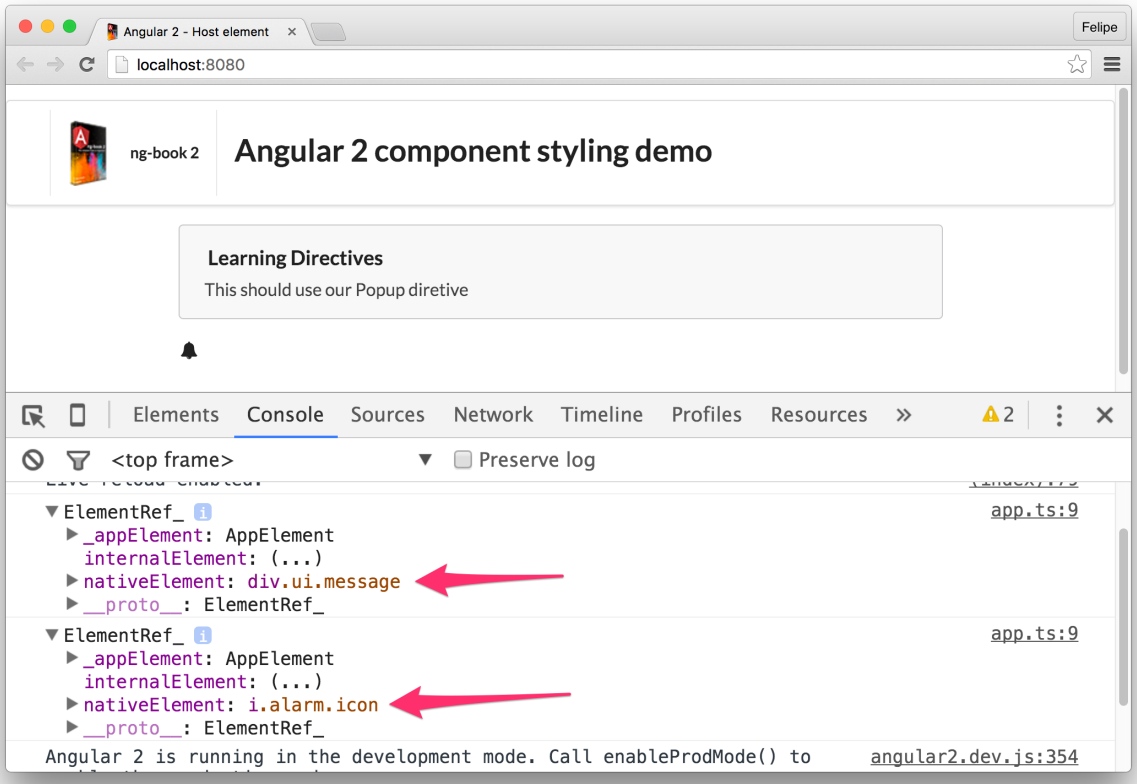
```
import { Component, Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[popup]'
})
export class PopupDirective {
  constructor(_elementRef: ElementRef) {
    console.log(_elementRef);
  }
}
```

We can also add a second element to the page that uses our directive, so we can see two different `ElementRefs` logged to the console:

code/advanced-components/src/app/host/popup-demo/steps/host-2.ts

```
@Component({
  selector: 'app-pop-demo',
  template: `
  <div class="ui message" popup>
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>

  <i class="alarm icon" popup></i>
  `
})
export class PopupDemoComponent2 {
}
```

When we run our app now, we can see two different `ElementRefs`: one with `div.ui.message` and the other with `i.alarm.icon`. This means that the directive was successfully bound to two different host elements:



ElementRefs

Binding to the `host`

Moving on, our next goal is to do something when the host element is clicked.

We learned before that the way we bind events in elements in Angular is using the `(event)` syntax.

In order to bind events of the host element, we'll do something very similar, but the syntax is different. In order to bind the directive to a host's `click` event, we're going to use the decorator `HostListener`.

The `HostListener` decorator allows a directive to listen to events on its host element.

We'll do this by decorating a function on the component with the `@HostListener()` decoration.

We also want the host element to define what message will pop up when the element is clicked, using the `message` attribute.

First, let's add an `inputs` attribute to the directive. We'll do this by importing `Input` and using the `@Input` decorator with the property we will use for this input:

```
import { Component, Input } from '@angular/core';
...
class Popup {
  @Input() message: String;
  ...
}
```

We're saying that we're having a property with the name `message` and expect to receive an input with the same name.

Then, let's add the `HostListener` decoration. We'll do this by adding `@HostLisener('click')` on the function we want to call when the host is clicked:

code/advanced-components/src/app/host/popup-demo/steps/host-3.ts

```
  HostListener
} from '@angular/core';

@Directive({
  selector: '[popup]'
})
export class PopupDirective {
  @Input() message: String;

  constructor(_elementRef: ElementRef) {
    console.log(_elementRef);
  }

  @HostListener('click') displayMessage(): void {
    alert(this.message);
  }
}
```

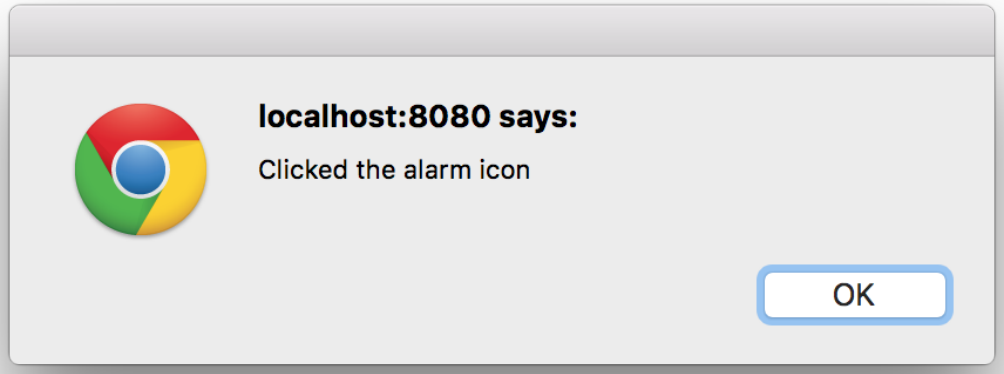
Then when the host element is clicked we'll call the directive's `displayMessage` method, which will display the message the host element defines.

And finally, we need to change our app template a bit to add the message we want displayed for each element:

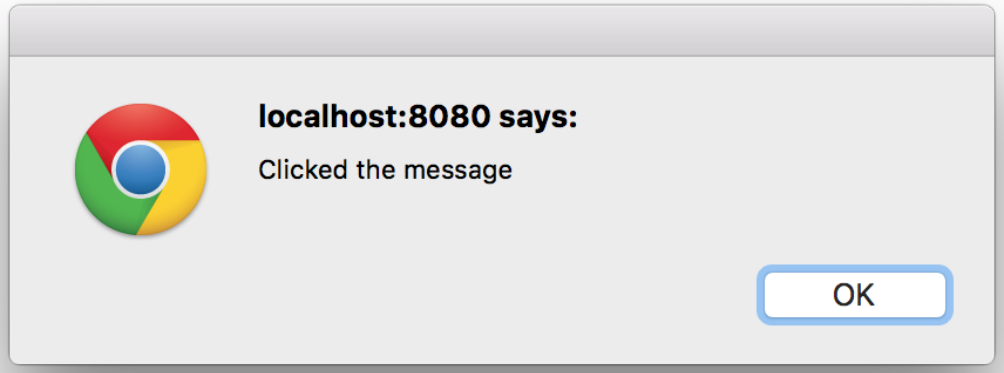
code/advanced-components/src/app/host/popup-demo/steps/host-3.ts

```
@Component({
  selector: 'app-popup-demo',
  template: `
  <div class="ui message" popup
       message="Clicked the message">
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>

  <i class="alarm icon" popup
     message="Clicked the alarm icon"></i>
  `
})
export class PopupDemoComponent3 {
}
```

Notice that we use the `popup` directive twice, and we pass a different `message` each time we use it. This means when we run the app, we're able to click either on the message or on the alarm icon, and we'll see different messages:



Popup 1



Popup 2

Adding a Button using `exportAs`

Now let's say we have a new requirement: we want to trigger the alert manually by clicking a button. How could we trigger the popup message from **outside** the host element?

In order to achieve this, we need to make the **directive available from elsewhere in the template**.

As we discussed in previous chapters, the way to reference a component is by using **template reference variable**. We can reference directives the same way.

In order to give the templates a reference to a directive we use the `exportAs` attribute. This will allow the host element (or a child of the host element) to define a template variable that references the directive using the `#var="exportName"` syntax.

Let's add the `exportAs` attribute to our directive:

code/advanced-components/src/app/host/popup-demo/steps/host-4.ts

```
17 @Directive({
18   selector: '[popup]',
19   exportAs: 'popup',
20 })
21 export class PopupDirective {
22   @Input() message: String;
23
24   constructor(_elementRef: ElementRef) {
25     console.log(_elementRef);
26   }
27
28   @HostListener('click') displayMessage(): void {
29     alert(this.message);
30   }
31 }
```

And now we need to change the two elements to export the template reference:

code/advanced-components/src/app/host/popup-demo/steps/host-4.ts

```
35   template: `
36   <div class="ui message" popup #popup1="popup"
37        message="Clicked the message">
38     <div class="header">
39       Learning Directives
40     </div>
41
42     <p>
43       This should use our Popup diretive
44     </p>
45   </div>
46
47   <i class="alarm icon" popup #popup2="popup"
48      message="Clicked the alarm icon"></i>
```

See that we used the template var `#popup1` for the `div.message` and `#popup2` for the `icon`.

Now let's add two buttons, one to trigger each popup:

code/advanced-components/src/app/host/popup-demo/steps/host-4.ts

```
49   <div style="margin-top: 20px;">
50     <button (click)="popup1.displayMessage()" class="ui button">
51       Display popup for message element
52     </button>
53
54     <button (click)="popup2.displayMessage()" class="ui button">
55       Display popup for alarm icon
56     </button>
57   </div>
```

Now reload the page and click each of the buttons and each message will appear as expected.

Creating a Message Pane with Content Projection

Sometimes when we are creating components we want to pass inner markup as an argument to the component. This technique is called *content projection*. The idea is that it lets us specify a bit of markup that will be expanded into a bigger template.

Angular 1 digged deep in the dictionary and called this *transclusion*.

Let's create a new directive that will render a nicely styled message like this:

Popup 1

Our goal is to write markup like this:

```
1  <div message header="My Message">
2    This is the content of the message
3  </div>
```

Which will render into the more complicated HTML like:

```
<div class="ui message">
  <div class="header">
    My Message
  </div>

  <p>
    This is the content of the message
  </p>
</div>
```

We have two challenges here: we need to change the host element `<div>` to add the `ui` and `message` CSS classes, and we need to add the div's contents to a specific place in our markup.

Changing the Host's CSS

To add attributes to the host element, we use a new decorator, similar to when we listened to events on the host: the `HostBinding` decorator. But now, instead of using specifying the event name we want to listen for, we'll define the attribute name we want to 'bind' to. In this component, it looks like this:

```
@HostBinding('attr.class') cssClass = 'ui message';
```

This decoration tells angular that we want the value of `cssClass` to be kept in sync with the host's attribute `class`.

Using `ng-content`

Our next challenge is to include the original host element children in a specific part of a view. To do that, we use the `ng-content` directive.

Since this directive needs a template, let's use a component instead and write the following code:

code/advanced-components/src/app/content-projection/content-projection-demo/messageo.component.ts

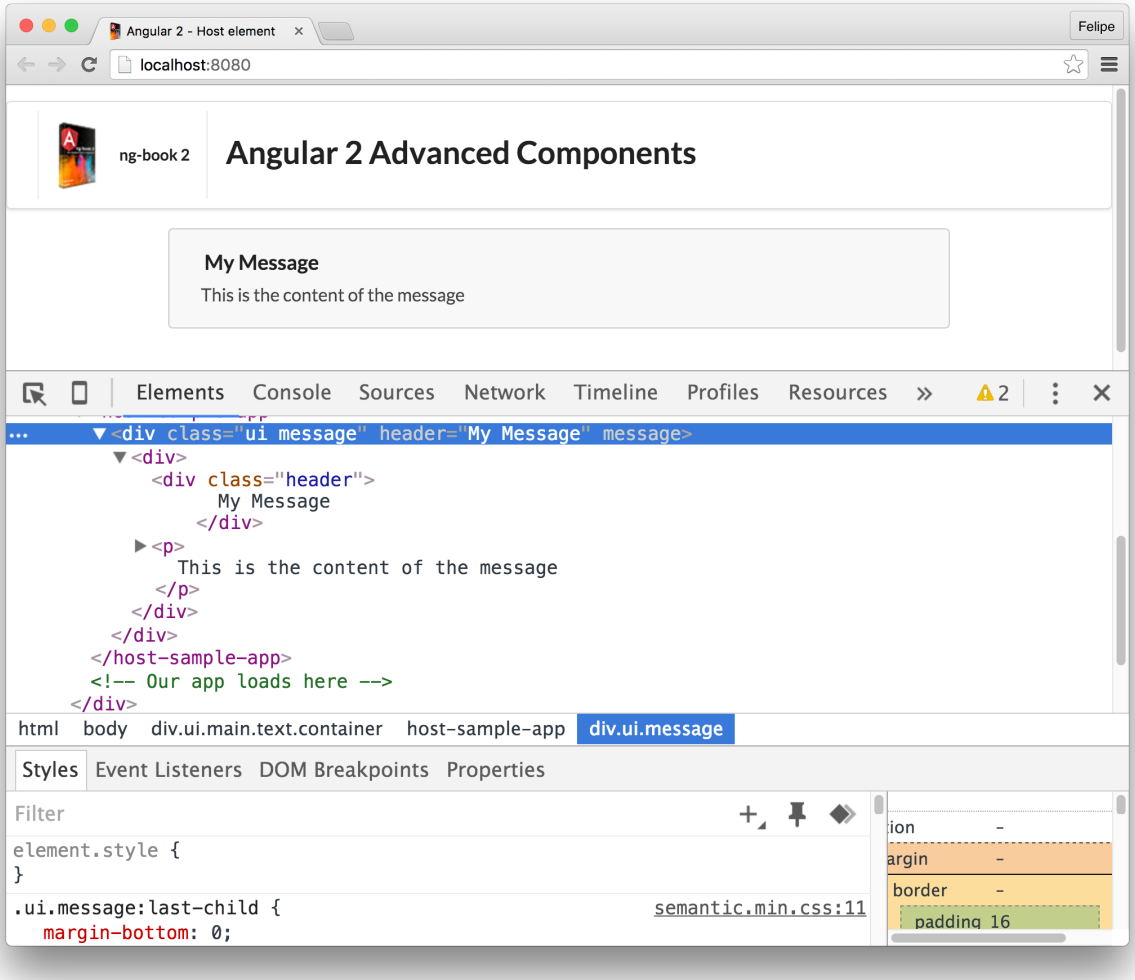
```
/* tslint:disable:component-selector */
import {
  Component,
  OnInit,
  Input,
  HostBinding
 } from '@angular/core';

@Component({
```

```
  selector: '[app-message]',
  template: `
    <div class="header">
      {{ header }}
    </div>
    <p>
      <ng-content></ng-content>
    </p>
  `
})
export class MessageComponent implements OnInit {
  @Input() header: string;
  @HostBinding('attr.class') cssClass = 'ui message';

  ngOnInit(): void {
    console.log('header', this.header);
  }
}
```

A few highlights:

- We use the `@Input` decorator to indicate we want to receive a `header` attribute, set on the host element
- We set the host element's `class` attribute to `ui message` using the `host` attribute of our component
- We use `<ng-content></ng-content>` to project the host element's children into a specific location of our template

When we open the app in the browser and inspect the message div, we see it worked exactly like we planned:



projected content

Querying Neighbor Directives - Writing Tabs

It's great when you can create a component that fully encapsulates its own behavior.

However, as a component grows in features, it might make sense to split it up into several smaller components that work together.

A great example of components that work together is a tab pane that has multiple tabs. The tab panel or *tabset*, as it's usually called, is composed of multiple *tabs*. In this scenario we have a parent component (the tabset) and multiple child components (the tabs). The tabset and the tabs don't make sense separately, but putting all of the logic in one component is cumbersome. So in this example, we're going to cover how to make separate components that work together.

Let's start writing those components in a way that we'll be able to use the following markup:

```
<tabset>
  <tab title="Tab 1">Tab 1</tab>
  <tab title="Tab 2">Tab 2</tab>
  ...
</tabset>
```

We're going to use Semantic UI Tab styles[132] to render the tabs.

ContentTabComponent

Let's start by writing the `ContentTabComponent`

code/advanced-components/src/app/tabs/content-tabs-demo/content-tab.component.ts

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

@Component({
  selector: 'tab',
  templateUrl: './content-tab.component.html'
})
export class ContentTabComponent implements OnInit {
  @Input() title: string;
  active = false;
  name: string;

  constructor() { }

  ngOnInit() { }
}
```

and the template:

[132] http://semantic-ui.com/modules/tab.html#/examples

code/advanced-components/src/app/tabs/content-tabs-demo/content-tab.component.html

```
<div class="ui bottom attached tab segment"
  [class.active]="active">

  <ng-content></ng-content>

</div>
```

There are not many new concepts here. We're declaring a component that will use the `ContentTabComponent` selector, and it will allow a `title` input to be set.

Then we're rendering a `<div>` and using the content projection concept we learned on the previous section to inline the contents of the `<tab>` directive inside the `div`.

Next we declare 3 properties on our components: *title*, *active* and *name*. One thing to notice is the `@Input('title')` decorator we added to the `title` property. This decorator is a way to ask Angular to automatically bind the value of the *input* `title` into the *property* `title`.

`ContentTabsetComponent` Component

Now let's move on to the `ContentTabsetComponent` component that will be used to wrap the tabs:

code/advanced-components/src/app/tabs/content-tabs-demo/content-tabset.component.ts

```
import {
  Component,
  AfterContentInit,
  QueryList,
  ContentChildren
} from '@angular/core';

import { ContentTabComponent } from './content-tab.component';

@Component({
  selector: 'tabset',
  templateUrl: './content-tabset.component.html'
})
export class ContentTabsetComponent implements AfterContentInit {
  @ContentChildren(ContentTabComponent) tabs: QueryList<ContentTabComponent>;

  ngAfterContentInit(): void {
    this.tabs.toArray()[0].active = true;
  }
```

```
20
21    setActive(tab: ContentTabComponent): void {
22      this.tabs.toArray().forEach((t) => t.active = false);
23      tab.active = true;
24    }
25
26    constructor() { }
27  }
```

and the template:

code/advanced-components/src/app/tabs/content-tabs-demo/content-tabset.component.html

```
1  <div class="ui top attached tabular menu">
2    <a *ngFor="let tab of tabs"
3       class="item"
4       [class.active]="tab.active"
5       (click)="setActive(tab)">
6
7      {{ tab.title }}
8
9    </a>
10 </div>
11 <ng-content></ng-content>
```

Let's break down the implementation so we can learn about the new concepts it introduces.

`ContentTabsetComponent` `@Component` Decorator

The `@Component` section doesn't have many new ideas. We're using the `<tabset>` tab as our selector.

The template itself uses `ngFor` to iterate through the tabs and if the tab has the *active* flag set to true, it will add the *active* CSS class to the `<a>` element that renders the tab.

We also specify that we are rendering the tabs themselves after the initial *div*, right where `ng-content` is.

`ContentTabsetComponent` class

Now let's turn our attention to the `ContentTabsetComponent` class. The first new idea we see here is that the `ContentTabsetComponent` class is implementing `AfterContentInit`. This *lifecycle hook* will tell Angular to call a method of our class (`ngAfterContentInit`) once the contents of the child directives has been initialized.

`ContentTabsetComponent ContentChildren` and `QueryList`

Next thing we do is declare the `tabs` property that will hold every `ContentTabComponent` component we declare inside the `ContentTabsetComponent`. Notice that instead of declaring this list as an array of `ContentTabComponents`, we use the class `QueryList`, passing a generic of `ContentTabComponent`. Why is this?

`QueryList` is a class provided by Angular and when we use `QueryList` with a `ContentChildren` Angular populates this with the **components that match the query** and then **keeps the items up to date** if the state of the application changes.

However, `QueryList` requires a `ContentChildren` to populate it, so let's take a look at that now.

On the `tabs` instance variable, we add the `@ContentChildren(Tab)` decorator. This decorator will tell Angular to inject all the direct child directives (of the `ContentTabComponent` type) into the `tabs` parameter. We then assign it to the `tabs` property of our component. With this **we now have access to all the child `ContentTabComponent` components**.

Initializing the `ContentTabsetComponent`

When this component is initialized, we want to make the first tab active. To do this we use the `ngAfterContentInit` function (that is described by the `AfterContentInit` hook). Notice that we use `this.tabs.toArray()` to cast the Angular's `QueryList` into a native TypeScript array.

`ContentTabsetComponent setActive`

Finally we define a `setActive` method. This method is used when we click a tab on our template e.g. using `(click)="setActive(tab)"`. This function will iterate through all the tabs, setting their `active` properties to false. Then we set the tab we clicked active.

Using the `ContentTabsetComponent`

Now the next step is to code the application component that makes use of both of the components we created. Here's how we write the component:

code/advanced-components/src/app/tabs/content-tabs-demo/content-tabs-demo.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-content-tabs-demo',
5    templateUrl: './content-tabs-demo.component.html'
6  })
7  export class ContentTabsDemoComponent implements OnInit {
8    tabs: any;
```

```
9
10    constructor() { }
11
12    ngOnInit() {
13      this.tabs = [
14        { title: 'About', content: 'This is the About tab' },
15        { title: 'Blog', content: 'This is our blog' },
16        { title: 'Contact us', content: 'Contact us here' },
17      ];
18    }
19
20  }
```

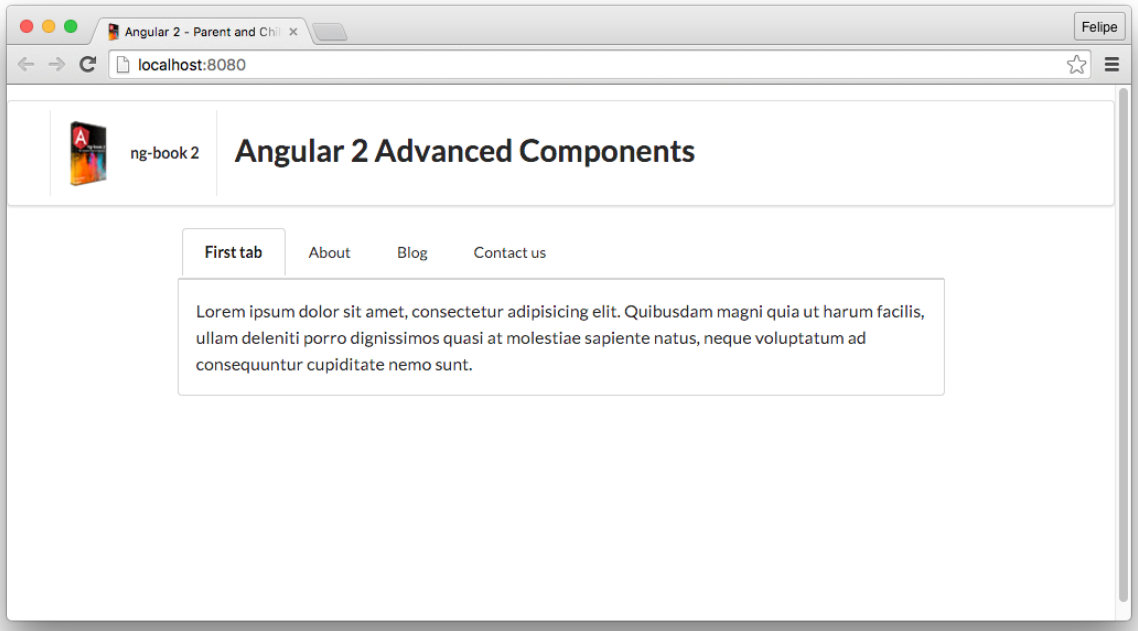
and template:

code/advanced-components/src/app/tabs/content-tabs-demo/content-tabs-demo.component.html

```
1  <tabset>
2    <tab title="First tab">
3      Lorem ipsum dolor sit amet, consectetur adipisicing elit.
4      Quibusdam magni quia ut harum facilis, ullam deleniti porro
5      dignissimos quasi at molestiae sapiente natus, neque voluptatum
6      ad consequuntur cupiditate nemo sunt.
7    </tab>
8
9    <tab
10      *ngFor="let tab of tabs"
11      [title]="tab.title">
12     {{ tab.content }}
13   </tab>
14 </tabset>
```

We're declaring that we're using `tabs-sample-app` as our component's selector and using the `ContentTabsetComponent` and `ContentTabComponent` components.

On the template we then create a `ContentTabsetComponent` and we add first a static tab (First tab) and we add a few more tabs from the `tabs` property of the component controller class, to illustrate how we can render tabs dynamically.



Tabset application

Lifecycle Hooks

Lifecycle hooks are the way Angular allows you to add code that runs before or after each step of the directive or component lifecycle.

The list of hooks Angular offers are:

- `OnInit`
- `OnDestroy`
- `DoCheck`
- `OnChanges`
- `AfterContentInit`
- `AfterContentChecked`
- `AfterViewInit`
- `AfterViewChecked`

Using these hooks each follow a similar pattern:

In order to be notified about those events you

1. declare that your directive or component class `implements` the interface and then
2. declare the `ng` method of the hook (e.g. `ngOnInit`)

Every method name is `ng` plus the name of the hook. For example, for `OnInit` we declare the method `ngOnInit`, for `AfterContentInit` we declare `ngAfterContentInit` and so on.

When Angular knows that a component implements these functions, it will invoke them at the appropriate time.

Let's take a look at each hook individually and when we would use each of them.

It is actually not mandatory for the class to `implement` the interface, one could just create the method of the hook. But it is considered good practice[133] and has benefits from strong typing and editor tooling.

`OnInit` and `OnDestroy`

The `OnInit` hook is called when your directive properties have been initialized, and before any of the child directive properties are initialized.

Similarly, the `OnDestroy` hook is called when the directive instance is destroyed. This is typically used if we need to do some cleanup every time our directive is destroyed.

In order to illustrate let's write a component that implements both `OnInit` and `OnDestroy`:

code/advanced-components/src/app/lifecycle/on-init/on-init.component.ts

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

@Component({
  selector: 'app-on-init',
  template: `
  <div class="ui label">
    <i class="cubes icon"></i> Init/Destroy
  </div>
  `
})
export class OnInitComponent implements OnInit, OnDestroy {
```

[133] https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html

```
16    constructor() { }
17
18    ngOnInit(): void {
19      console.log('On init');
20    }
21
22    ngOnDestroy(): void {
23      console.log('On destroy');
24    }
25  }
```

For this component, we're just logging *On init* and *On destroy* to the console when the hooks are called.

Now in order to test those hooks let's use our component in our app component using `ngFor` to conditionally display it based on a boolean property. Let's also add a button that allows us to toggle that flag. This way, when the flag is false, our component will be *removed* from the page, causing the `OnDestroy` hook to be called. Similarly when the flag is toggled to true, the `OnInit` hook will be called.

Here's how our app component will look:

code/advanced-components/src/app/lifecycle/on-init/on-init-demo.component.ts

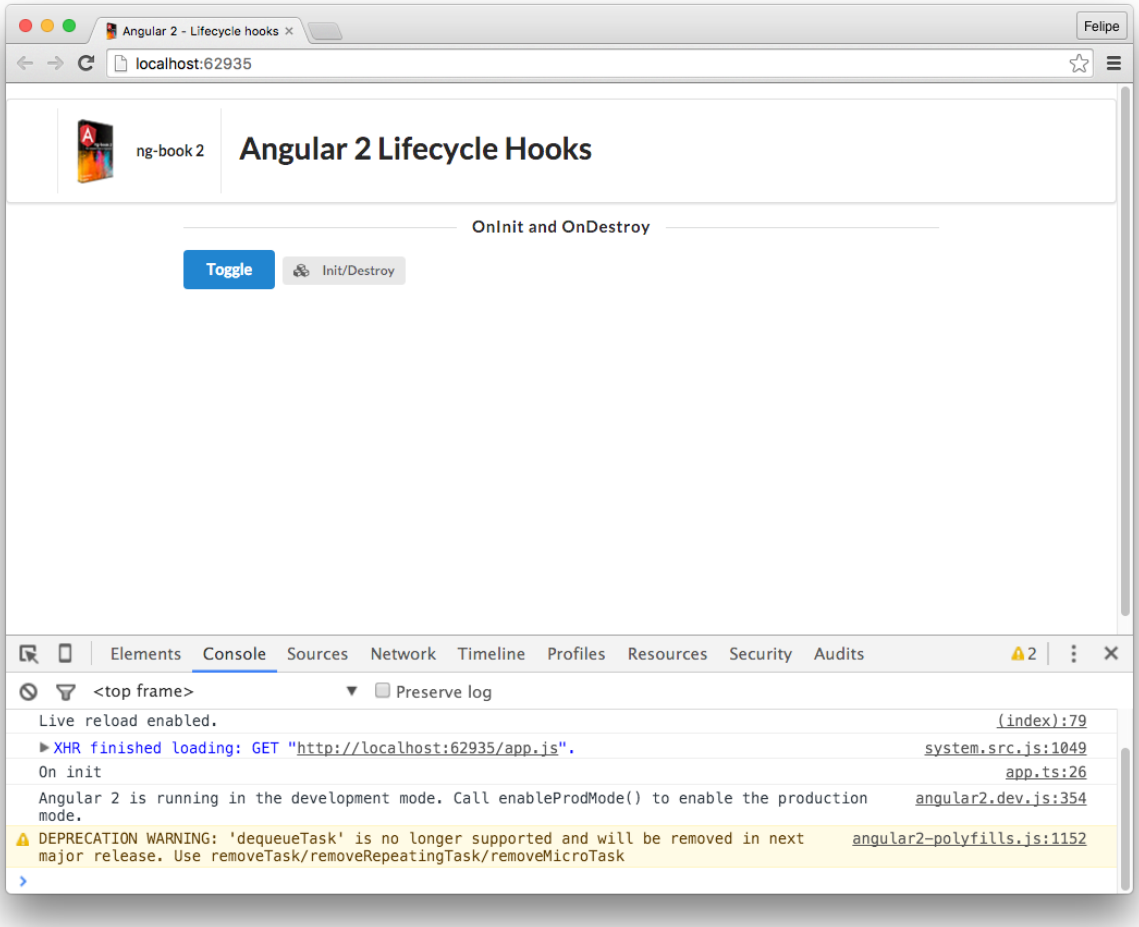
```
1   import { Component } from '@angular/core';
2
3   @Component({
4     selector: 'app-on-init-demo',
5     templateUrl: './on-init-demo.component.html'
6   })
7   export class OnInitDemoComponent {
8     display: boolean;
9
10    constructor() {
11      this.display = true;
12    }
13
14    toggle(): void {
15      this.display = !this.display;
16    }
17  }
```

and the template:

code/advanced-components/src/app/lifecycle/on-init/on-init-demo.component.html

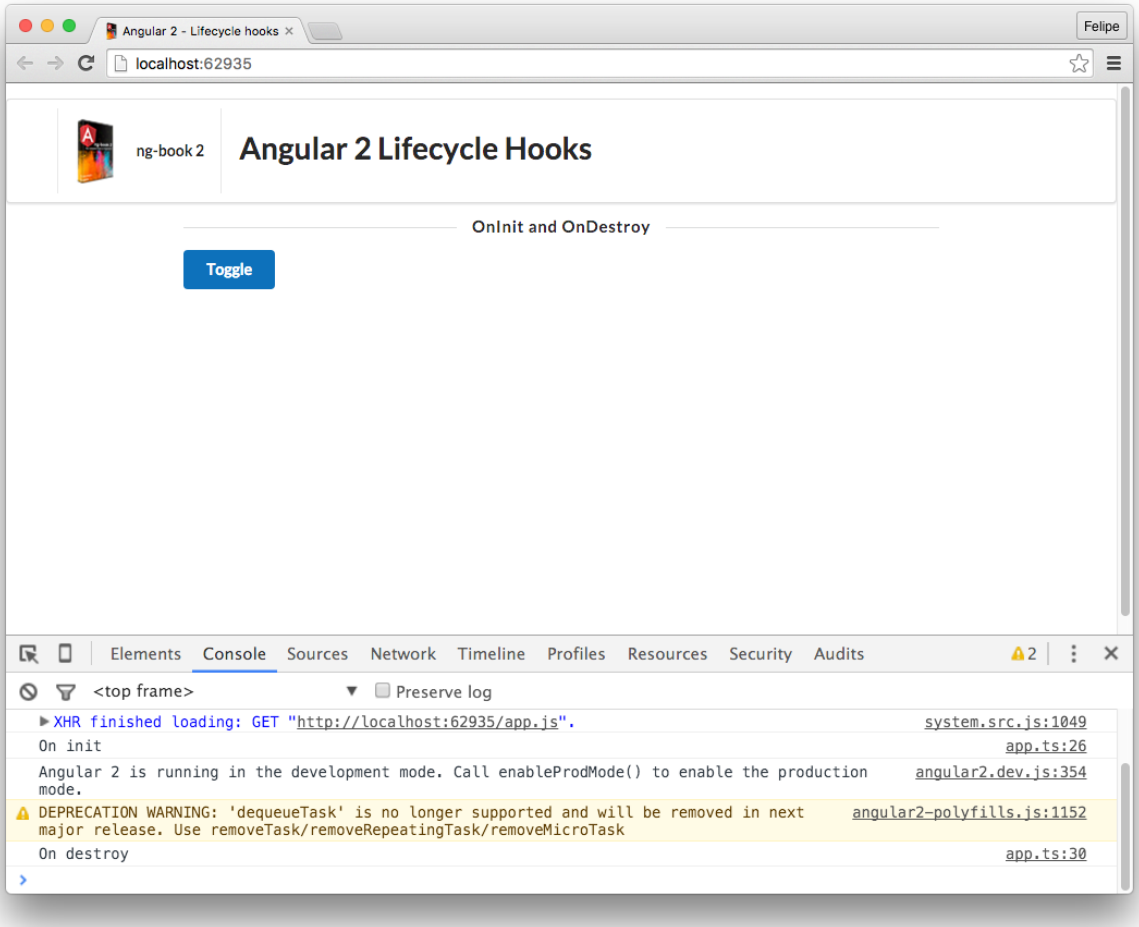
```
<h4 class="ui horizontal divider header">
  OnInit and OnDestroy
</h4>

<button class="ui primary button" (click)="toggle()">
  Toggle
</button>
<app-on-init *ngIf="display"></app-on-init>
```

When we first run the application, we can see that the `OnInit` hook was called when the component was first instantiated:



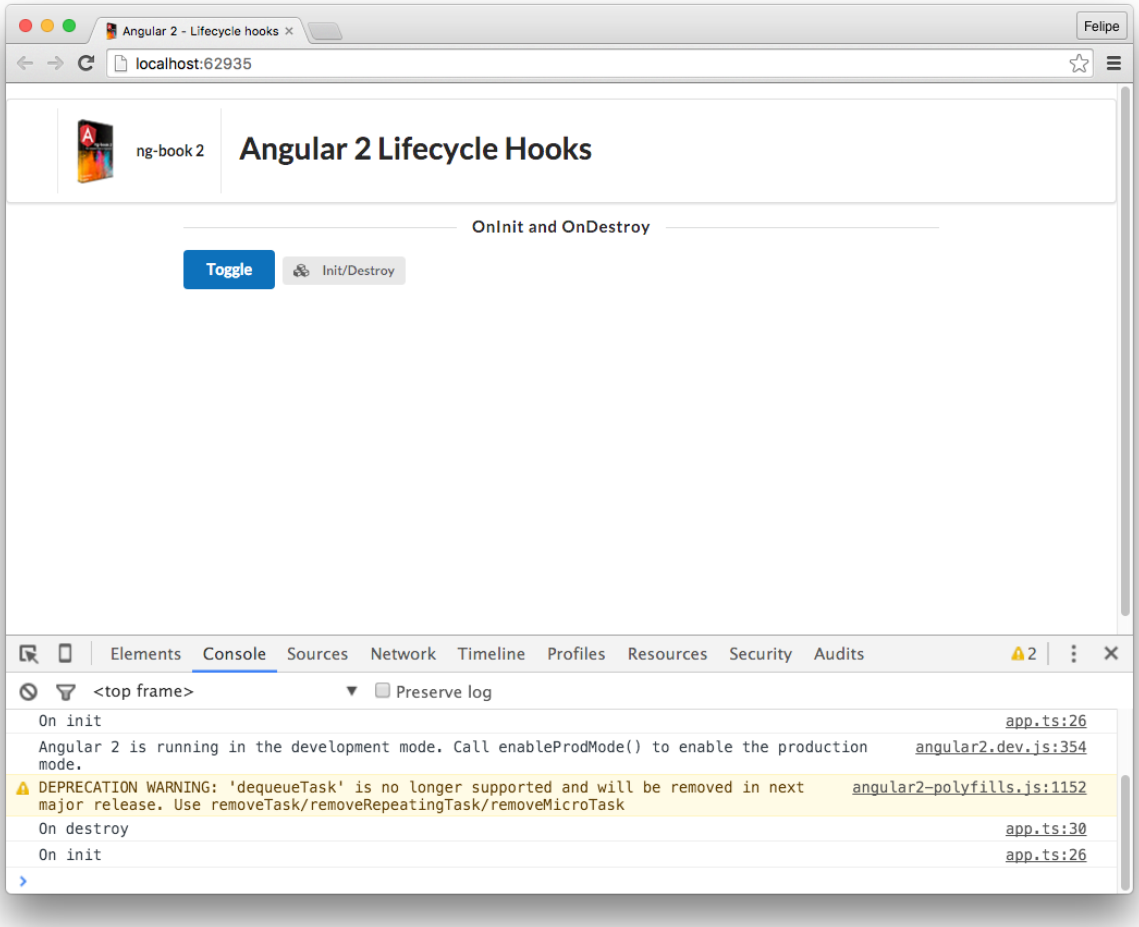
Initial state of our component

When I click the `Toggle` button for the first time, the component is destroyed and the hook is called as expected:



OnDestroy hook

And if we click it another time:



OnDestroy hook

OnChanges

The `OnChanges` hook is called after one or more of our component properties have been changed. The `ngOnChanges` method receives a parameter which tells which properties have changed.

To understand this better, let's write a comment block component that has two inputs: *name* and *comment*:

code/advanced-components/src/app/lifecycle/on-changes/on-changes.component.ts

```
import {
  Component,
  OnInit,
  OnChanges,
  Input,
  SimpleChange
} from '@angular/core';

@Component({
  selector: 'app-on-changes',
  templateUrl: './on-changes.component.html'
})
export class OnChangesComponent implements OnChanges {
  @Input('name') name: string;
  @Input('comment') comment: string;

  ngOnChanges(changes: {[propName: string]: SimpleChange}): void {
    console.log('Changes', changes);
  }
}
```

and template:

code/advanced-components/src/app/lifecycle/on-changes/on-changes.component.html

```
<div class="ui comments">
  <div class="comment">
    <a class="avatar">
      <img src="assets/images/avatars/matt.jpg">
    </a>
    <div class="content">
      <a class="author">{{name}}</a>
      <div class="text">
        {{comment}}
      </div>
    </div>
  </div>
</div>
```

The important thing about this component is that it implements the `OnChanges` interface, and it declares the `ngOnChanges` method with this signature:

code/advanced-components/src/app/lifecycle/on-changes/on-changes.component.ts

```
17    ngOnChanges(changes: {[propName: string]: SimpleChange}): void {
18      console.log('Changes', changes);
19    }
```

This method will be triggered whenever the values of either the *name* or *comment* properties change. When that happens, we receive an object that maps changed fields to `SimpleChange` objects.

Each `SimpleChange` instance has two fields: `currentValue` and `previousValue`. If both name and comment properties change for our component, we expect the value of `changes` in our method to be something like:

```
1  {
2    name: {
3      currentValue: 'new name value',
4      previousValue: 'old name value'
5    },
6    comment: {
7      currentValue: 'new comment value',
8      previousValue: 'old comment value'
9    }
10 }
```

Now, let's change the app component to use our component and also add a little form where we can play with the name and comment properties of our component:

code/advanced-components/src/app/lifecycle/on-changes/on-changes-demo.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-on-changes-demo',
5    templateUrl: './on-changes-demo.component.html',
6    styles: []
7  })
8  export class OnChangesDemoComponent implements OnInit {
9    display: boolean;
10   name: string;
11   comment: string;
12
13   constructor() { }
14
```

```
  ngOnInit() {
    this.display = true;
    this.name = 'Felipe Coury';
    this.comment = 'I am learning so much!';
  }

  setValues(namefld, commentfld): void {
    this.name = namefld.value;
    this.comment = commentfld.value;
  }

  toggle(): void {
    this.display = !this.display;
  }

  }
```

and template:

code/advanced-components/src/app/lifecycle/on-changes/on-changes-demo.component.html

```
<h4 class="ui horizontal divider header">
  OnChanges
</h4>

<div class="ui form">
  <div class="field">
    <label>Name</label>
    <input
      type="text"
      #namefld
      value="{{name}}"
     (keyup)="setValues(namefld, commentfld)">
  </div>

  <div class="field">
    <label>Comment</label>
    <textarea
      #commentfld
      (keyup)="setValues(namefld, commentfld)"
      rows="2">{{comment}}</textarea>
  </div>
</div>
```

```
23
24 <app-on-changes
25   [name]="name"
26   [comment]="comment"
27   ></app-on-changes>
```

Note that in the template important pieces that we added here where the template areas where we declare a new form with name and comment fields.

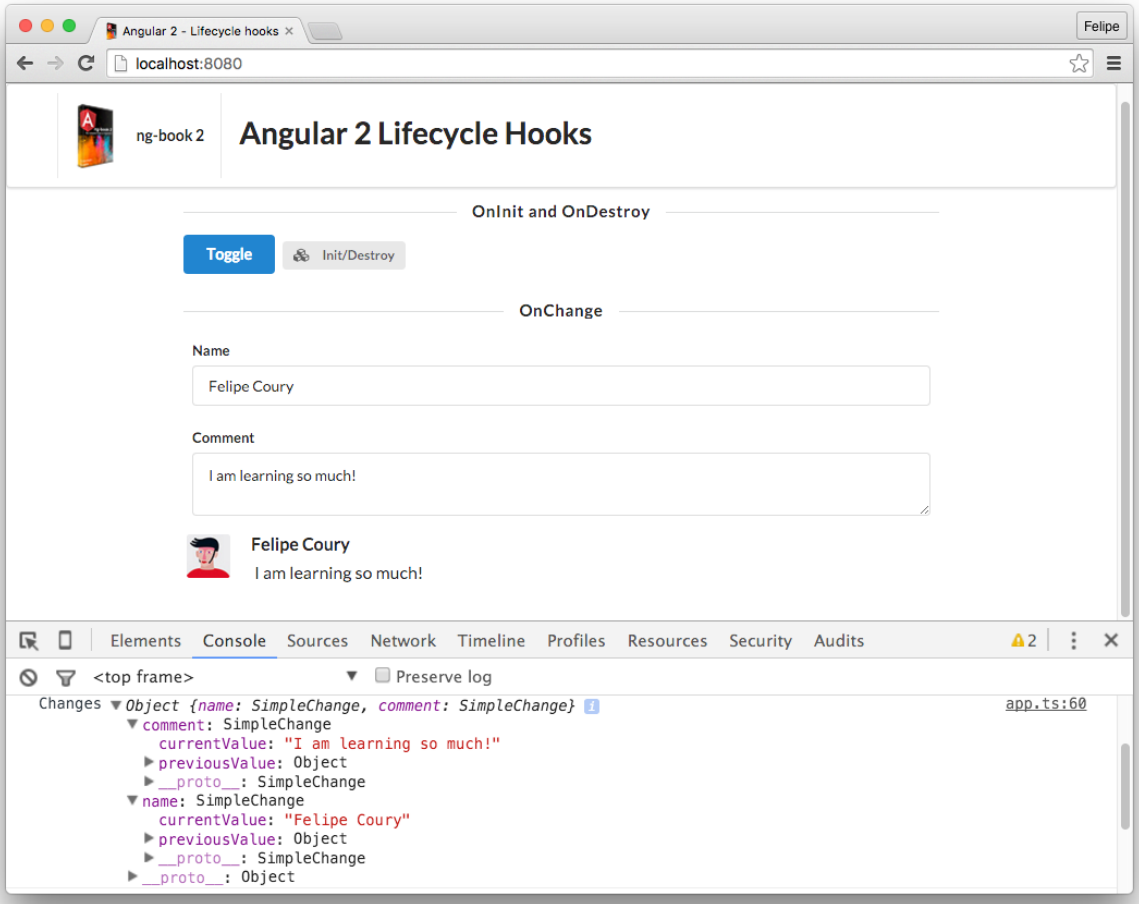
Here, when the `keyup` event is fired for either the name or comment fields, we are calling `setValues` with the template references `namefld` and `commentfld` that represent the `input` and `textarea`.

This method just takes the value from those fields and updates the name and comment properties accordingly:

code/advanced-components/src/app/lifecycle/on-changes/on-changes-demo.component.ts

```
21   setValues(namefld, commentfld): void {
22     this.name = namefld.value;
23     this.comment = commentfld.value;
24   }
```

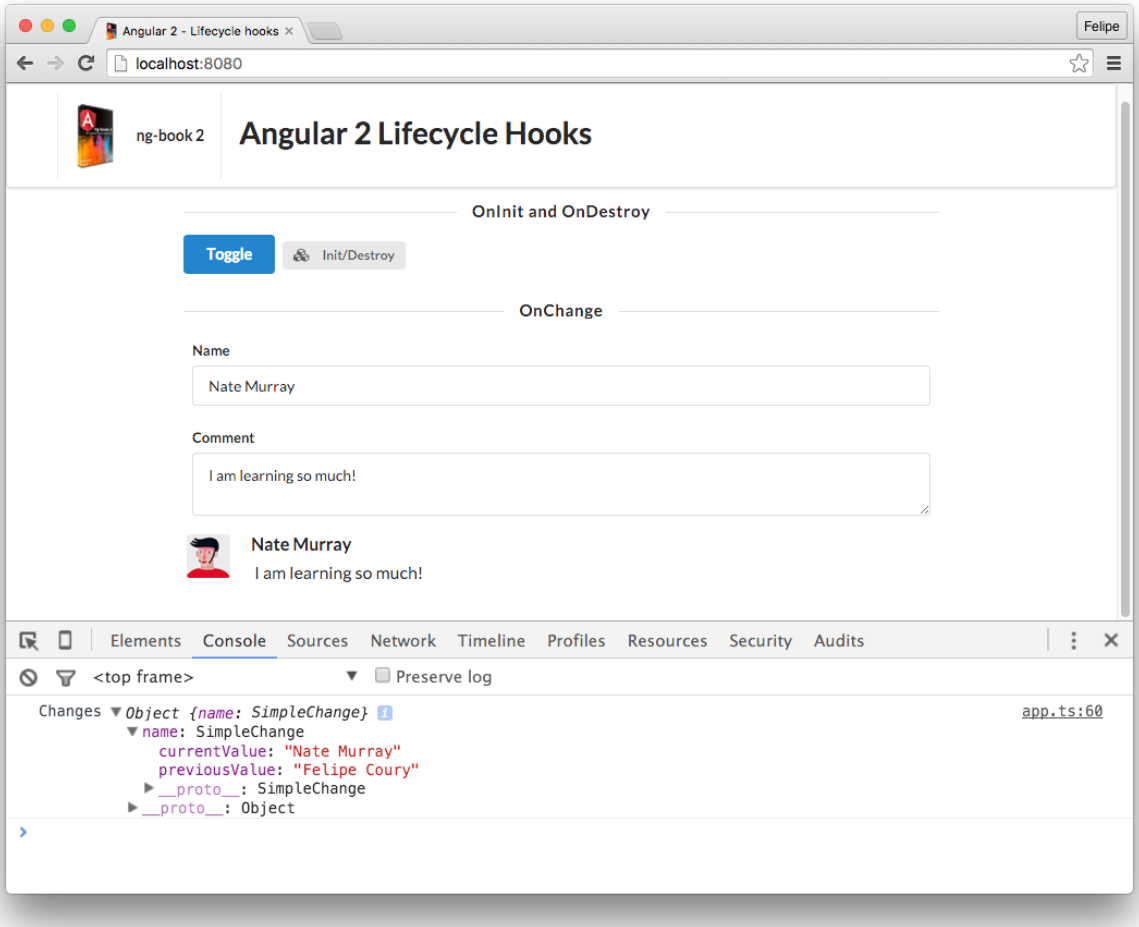
So now, the first time we open the app, we can see that our `OnChanges` hook is called:



OnChanges

This happens when the initial values are set, on the constructor of the `LifecycleSampleApp` component.

Now if we play with the name, we can see that the hook is called repeatedly. In the case below, we pasted the name *Nate Murray* on top of the previous name, and the values for the changes are displayed as expected:



OnChanges

DoCheck

The default notification system implemented by `OnChanges` is triggered every time the Angular change detection mechanism notices there was a change on any of the directive properties.

However, there may be times when the overhead added by this change notification may be too much, specially if performance is a concern.

There may be times when we just want to do something in case an item was removed or added, or if only a particular property changed, for instance.

If we run into one of these scenarios, we can use the `DoCheck` hook.

It's important to note that the `OnChanges` hook gets overriden by `DoCheck` so if we implement both, `OnChanges` will be ignored.

Checking for changes

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* and *key-value differs*.

Iterable differs

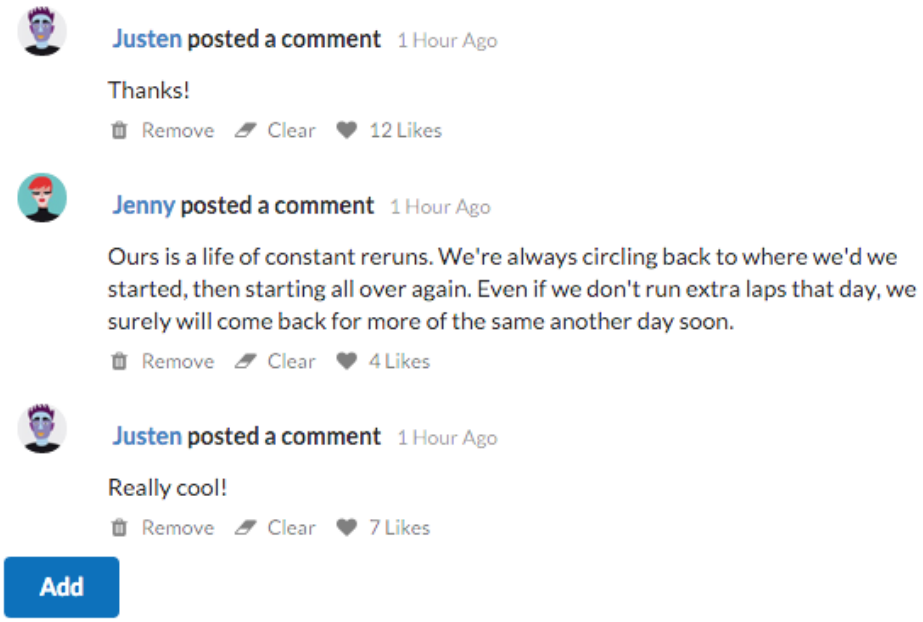
Iterable differs should be used when we have a list-like structure and we're only interested on knowing things that were added or removed from that list.

Key-value differs

Key-value differs should be used for dictionary-like structures, and work at the key level. This differ will identify changes when a new key is added, when a key removed and when the value of a key changed.

Rendering a comment with DoCheck

To illustrate these concepts, let's build a component that renders a stream of comments, like below:



DoCheck example

Let's write a component that will render one individual comment. First, the template:

code/advanced-components/src/app/lifecycle/differs/comment.component.html

```
<div class="ui feed">
  <div class="event">
    <div class="label" *ngIf="comment.author">
      <img src="assets/images/avatars/{{comment.author.toLowerCase()}}.jpg">
    </div>
    <div class="content">
      <div class="summary">
        <a class="user">
          {{comment.author}}
        </a> posted a comment
        <div class="date">
          1 Hour Ago
        </div>
      </div>
      <div class="extra text">
        {{comment.comment}}
      </div>
      <div class="meta">
        <a class="trash" (click)="remove()">
          <i class="trash icon"></i> Remove
        </a>
        <a class="trash" (click)="clear()">
          <i class="eraser icon"></i> Clear
        </a>
        <a class="like" (click)="like()">
          <i class="like icon"></i> {{comment.likes}} Likes
        </a>
      </div>
    </div>
  </div>
</div>
```

and in the component:

code/advanced-components/src/app/lifecycle/differs/comment.component.ts

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

@Component({
  selector: 'app-comment',
  templateUrl: './comment.component.html'
})
export class CommentComponent implements DoCheck {
  @Input() comment: any;
  @Output() onRemove: EventEmitter<any>;
  differ: any;
```

Here we are declaring the component metadata. Our component will receive the comment that should be rendered and it will emit an event with the remove button icon clicked.

On the class declaration we indicate we're implementing the `DoCheck` interface. We then declare the input property `comment`, and the output event `onRemove`. We also declare a `differ` property.

code/advanced-components/src/app/lifecycle/differs/comment.component.ts

```
  constructor(differs: KeyValueDiffers) {
    this.differ = differs.find([]).create(null);
    this.onRemove = new EventEmitter();
  }
```

On the constructor we're receiving a `KeyValueDiffers` instance on the `differs` variable. We then use this variable to create an instance of the key value differ using this syntax `differs.find([]).create(null)`. We're also initializing our event emitter `onRemove`.

Next, let's implement the `ngDoCheck` method, required by the interface:

code/advanced-components/src/app/lifecycle/differs/comment.component.ts

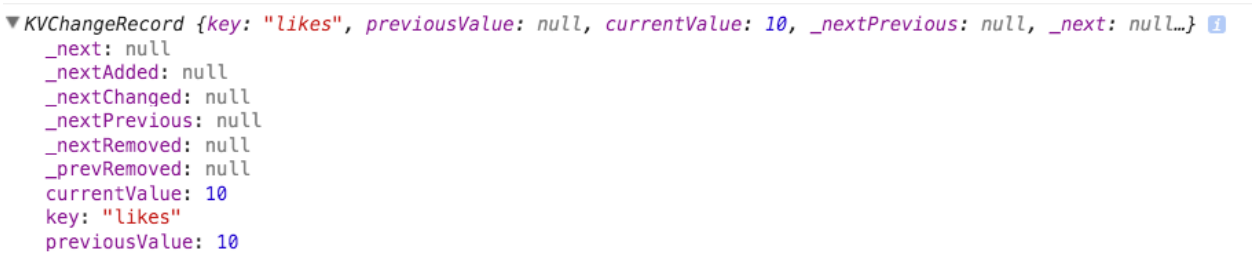
```
24  ngDoCheck(): void {
25    const changes = this.differ.diff(this.comment);
26
27    if (changes) {
28      changes.forEachAddedItem(r => this.logChange('added', r));
29      changes.forEachRemovedItem(r => this.logChange('removed', r));
30      changes.forEachChangedItem(r => this.logChange('changed', r));
31    }
32  }
```

This is how you check for changes, if you're using a key-value differ. You call the `diff` method, providing the property you want to check. In our case, we want to know if there were changes to the `comment` property.

When no changes are detected, the returned value will be `null`. Now, if there are changes, we can call three different iterable methods on the differ:

- `forEachAddedItem`, for *keys* that were added
- `forEachRemovedItem`, for *keys* that were removed
- `forEachChangedItem`, for *keys* that were changed

Each method will call the provided callback with a *record*. For the key-value differ, this record will be an instance of the `KVChangeRecord` class.



Example of a KVChangeRecord instance

The important fields for understanding what changed are *key*, *previousValue* and *currentValue*.

Next, let's write a method that will log to the console a nice sentence about what changed:

code/advanced-components/src/app/lifecycle/differs/comment.component.ts

```
34   logChange(action, r) {
35     if (action === 'changed') {
36       console.log(r.key, action, 'from', r.previousValue, 'to', r.currentValue);
37     }
38     if (action === 'added') {
39       console.log(action, r.key, 'with', r.currentValue);
40     }
41     if (action === 'removed') {
42       console.log(action, r.key, '(was ' + r.previousValue + ')');
43     }
44   }
```

Finally, let's write the methods that will help us change things on our component, to trigger our `DoCheck` hook:

code/advanced-components/src/app/lifecycle/differs/comment.component.ts

```
46   remove(): void {
47     this.onRemove.emit(this.comment);
48   }
49
50   clear(): void {
51     delete this.comment.comment;
52   }
53
54   like(): void {
55     this.comment.likes += 1;
56   }
```

The `remove()` method will emit the event indicating that the user asked for this comment to be removed, the `clear()` method will remove the comment text from the comment object, and the `like()` method will increase to the like counter for the comment.

Rendering a list of comments with `CommentsListComponent`

Now that we have written a component for one individual comment, let's write a second component that will be responsible for rendering the list of comments. First the template:

code/advanced-components/src/app/lifecycle/differs/comments-list.component.html

```
<app-comment
  *ngFor="let comment of comments"
  [comment]="comment"
  (onRemove)="removeComment($event)">
</app-comment>

<button
  class="ui primary button"
  (click)="addComment()">
  Add
</button>
```

The component template is straightforward: we're using the component we created above, and then using `ngFor` to iterate through a list of comments, rendering them. We also have a button that will allow the user to add more comments to the list.

Now let's implement our comment list class `CommentsListComponent`:

code/advanced-components/src/app/lifecycle/differs/comments-list.component.ts

```
/* tslint:disable:max-line-length,quotemark */
import {
  Component,
  IterableDiffers,
  DoCheck
} from '@angular/core';

@Component({
  selector: 'app-comments-list',
  templateUrl: './comments-list.component.html'
})
export class CommentsListComponent implements DoCheck {
  comments: any[];
  iterable: boolean;
  authors: string[];
  texts: string[];
  differ: any;
```

Here we declare the variables we'll use: `comments`, `iterable`, `authors`, and `texts`.

code/advanced-components/src/app/lifecycle/differs/comments-list.component.ts

```
19   constructor(differs: IterableDiffers) {
20     this.differ = differs.find([]).create(null);
21     this.comments = [];
22
23     this.authors = ['Elliot', 'Helen', 'Jenny', 'Joe', 'Justen', 'Matt'];
24     this.texts = [
25       "Ours is a life of constant reruns. We're always circling back to where we\
26 'd we started, then starting all over again. Even if we don't run extra laps tha\
27 t day, we surely will come back for more of the same another day soon.",
28       'Really cool!',
29       'Thanks!'
30     ];
31
32     this.addComment();
33   }
```

For this component, we'll be using an iterable differ. We can see that the class we're using to create the differ is now `IterableDiffers`. However, the way we create a differ remains the same.

On the constructor we also initialize a list of authors and a list of comment texts to be used when adding new comments.

Finally, we call the `addComment()` method so we don't initialize the app with an empty list of comments.

The next three methods are used to add a new comment:

code/advanced-components/src/app/lifecycle/differs/comments-list.component.ts

```
33   getRandomInt(max: number): number {
34     return Math.floor(Math.random() * (max + 1));
35   }
36
37   getRandomItem(array: string[]): string {
38     const pos: number = this.getRandomInt(array.length - 1);
39     return array[pos];
40   }
41
42   addComment(): void {
43     this.comments.push({
44       author: this.getRandomItem(this.authors),
45       comment: this.getRandomItem(this.texts),
46       likes: this.getRandomInt(20)
```

```
47      });
48    }
49
50    removeComment(comment) {
51      const pos = this.comments.indexOf(comment);
52      this.comments.splice(pos, 1);
53    }
```

We are declaring two methods that will return a random integer and a random item from an array, respectively.

Finally, the `addComment()` method will push a new comment to the list, with a random author, random text and a random number of likes.

Next, we have the `removeComment()` method, that will be used to remove one comment from the list:

code/advanced-components/src/app/lifecycle/differs/comments-list.component.ts

```
50    removeComment(comment) {
51      const pos = this.comments.indexOf(comment);
52      this.comments.splice(pos, 1);
53    }
```

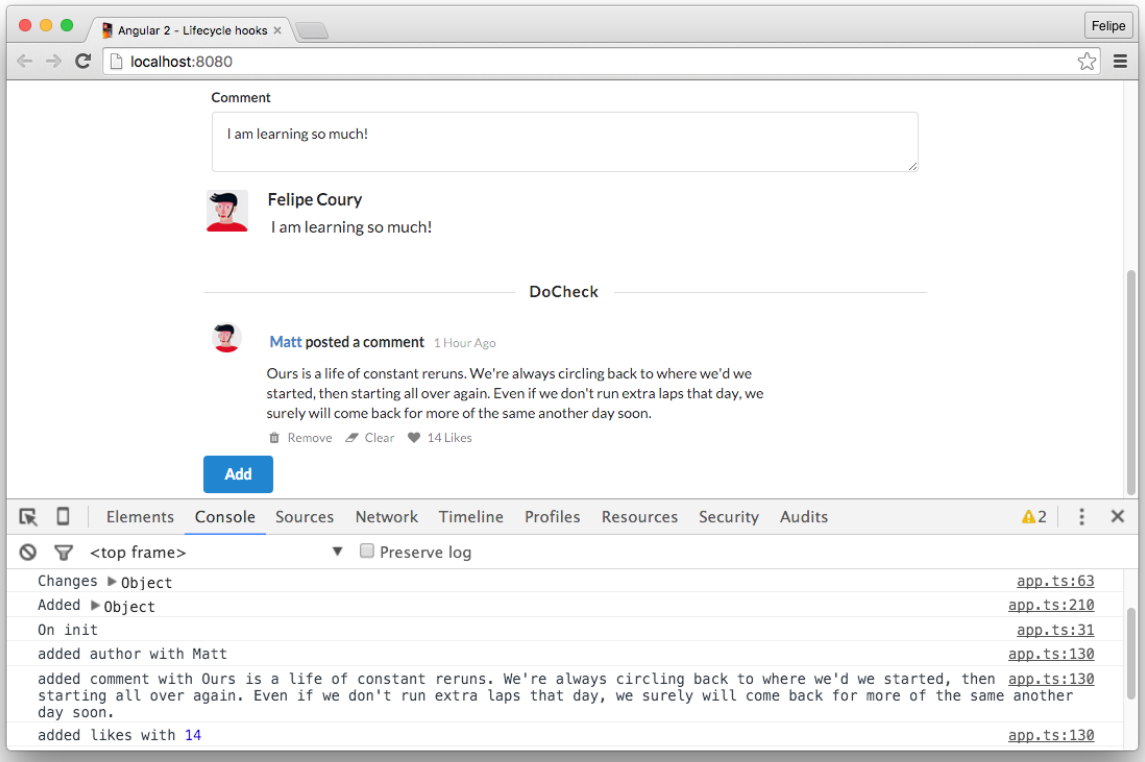
And finally we declare our change detection method `ngDoCheck()`:

code/advanced-components/src/app/lifecycle/differs/comments-list.component.ts

```
55    ngDoCheck(): void {
56      const changes = this.differ.diff(this.comments);
57
58      if (changes) {
59        changes.forEachAddedItem(r => console.log('Added', r.item));
60        changes.forEachRemovedItem(r => console.log('Removed', r.item));
61      }
62    }
```

The iterable differ behaves the same way as the key-value differ but it only provides methods for items that were added or removed.

When we run the app now, we get the list of comments with one comment:

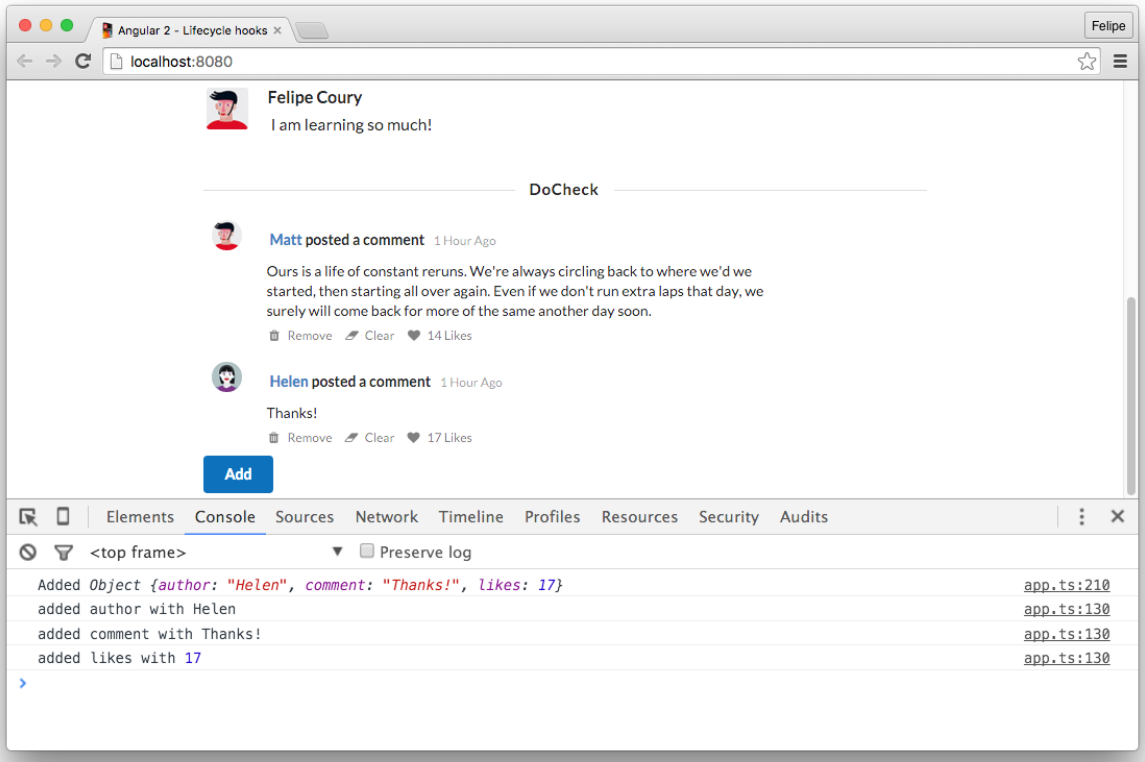


Initial state

We can also see that a few things were logged to the console, like:

```
added author with Matt
...
added likes with 14
```

Let's see what happens when we add a new comment to the list by clicking the `Add` button:



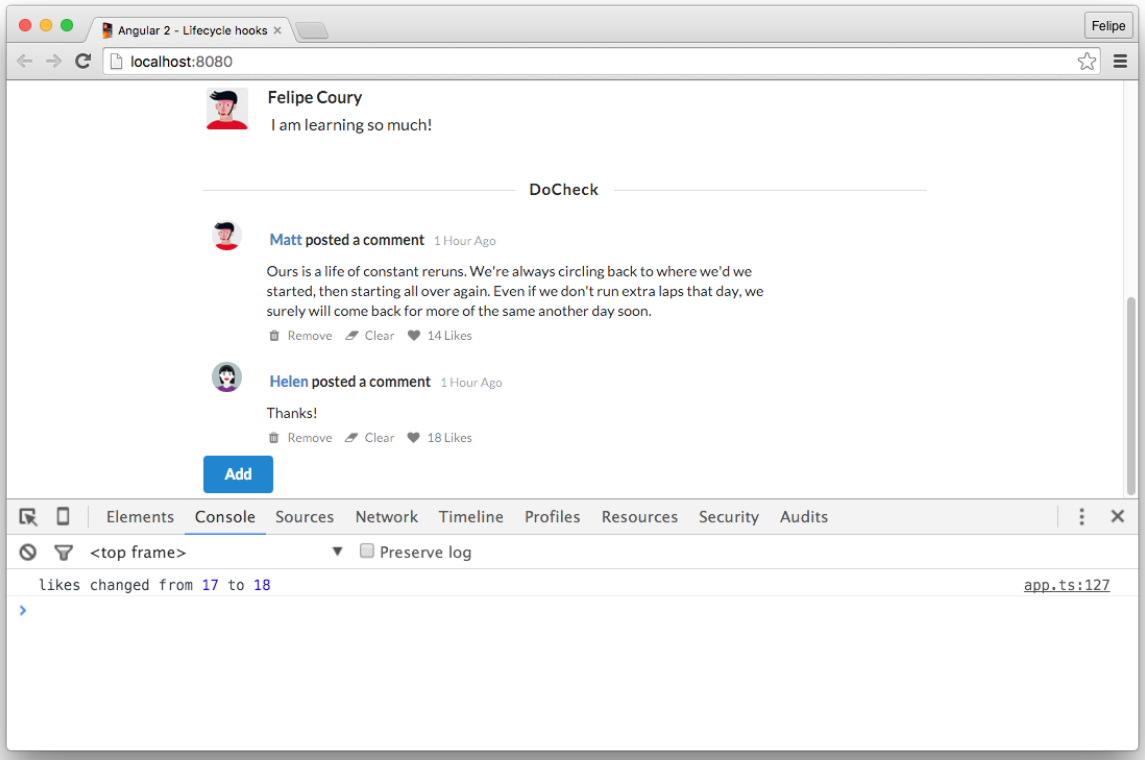
Comment added

We can see that the iterable differs identified that we added a new object to the list `{author: "Hellen", comment: "Thanks!", likes: 17}`.

We also got individual changes to the comment object logged, as detected by the key-value differ:

```
added author with Helen
added comment with Thanks!
added likes with 17
```

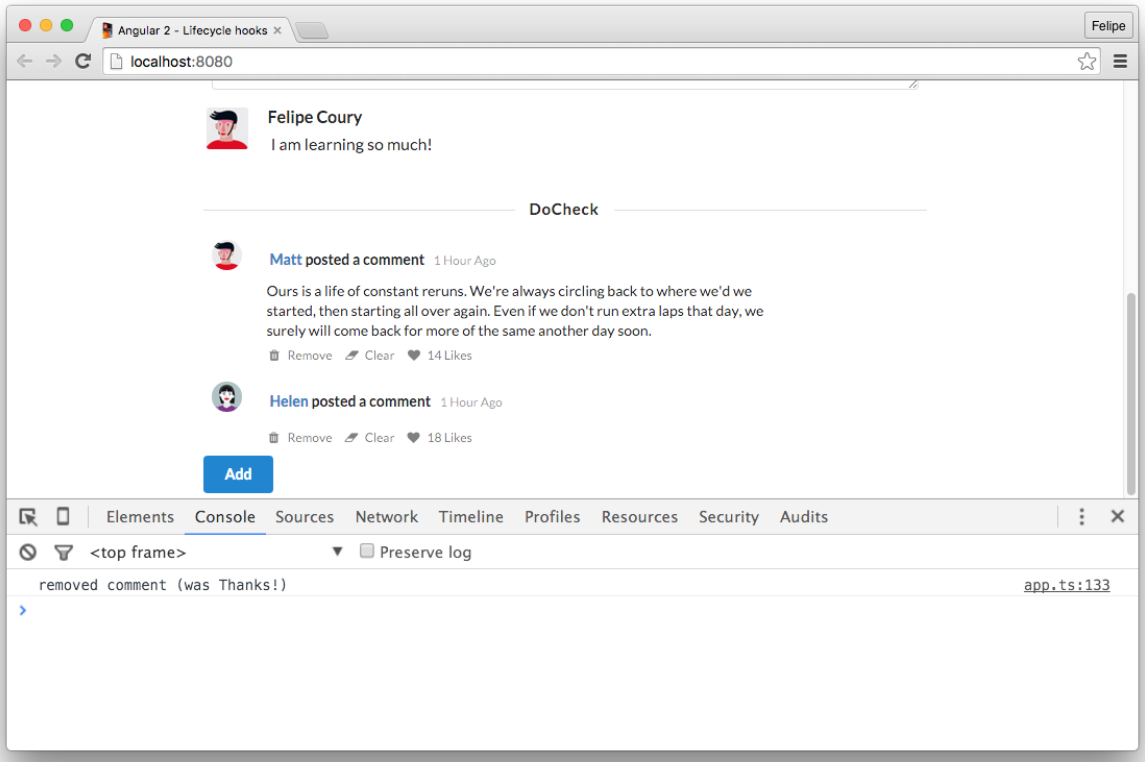
Now we can click the like button for this new comment:



Number of likes changed

And now only the like change was detected.

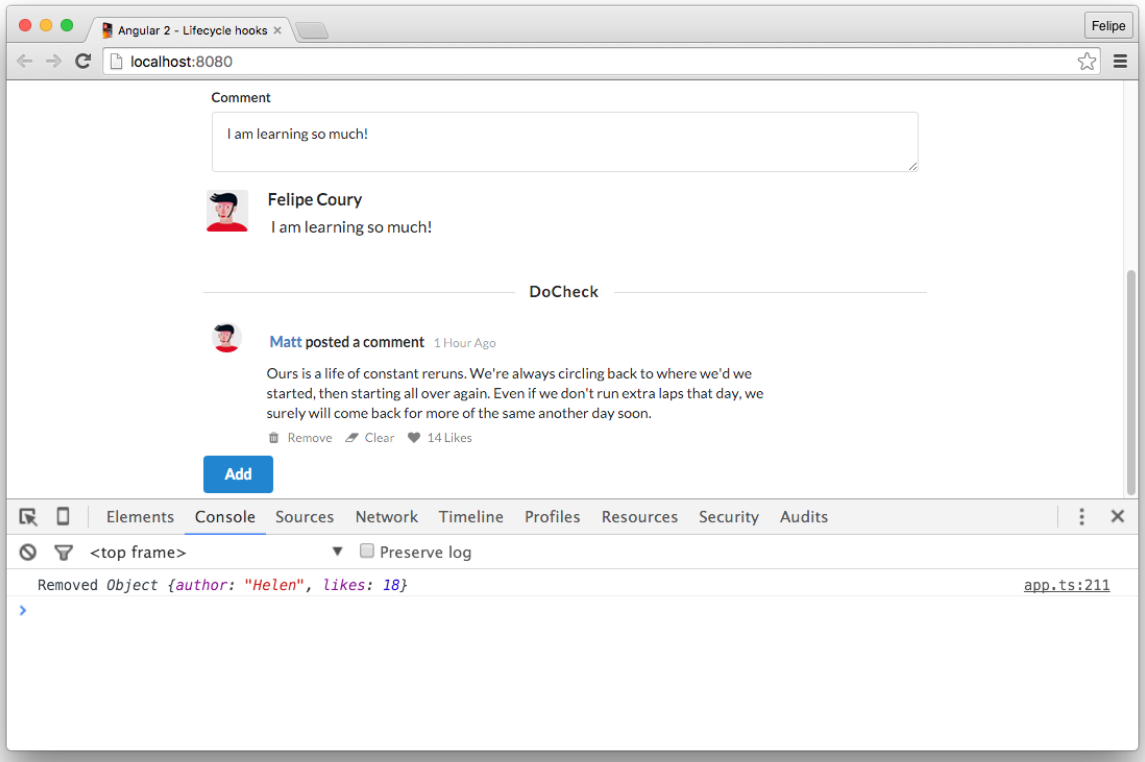
If we click the *Clear* icon, it will remove the `comment` key from the comment object:



Comment text cleared

And the log confirms that we removed that key.

Finally, let's remove the last comment, by clicking the *Remove* icon:



Comment removed

And as expected, we get a removed object log.

AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked

The `AfterContentInit` hook is called after `OnInit`, right after the initialization of the content of the component or directive has finished.

The `AfterContentChecked` works similarly, but it's called after the directive check has finished. The check, in this context, is the change detection system check.

The other two hooks: `AfterViewInit` and `AfterViewChecked` are triggered right after the content ones above, right after the view has been fully initialized. Those two hooks are only applicable to components, and not to directives.

Also, the `AfterXXXInit` hooks are only called once during the directive lifecycle, while the `AfterXXXChecked` hooks are called after every change detection cycle.

To better understand this, let's write another component that logs to the console during each lifecycle hook. It will also have a counter that we can increment by clicking a button:

code/advanced-components/src/app/lifecycle/all-hooks/all-hooks.component.ts

```
import {
  Component,
  OnInit,
  OnDestroy,
  DoCheck,
  OnChanges,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked
} from '@angular/core';

@Component({
  selector: 'app-all-hooks',
  templateUrl: './all-hooks.component.html'
})
export class AllHooksComponent implements OnInit,
  OnDestroy, DoCheck,
  OnChanges, AfterContentInit,
  AfterContentChecked, AfterViewInit,
  AfterViewChecked {
  counter: number;

  constructor() {
    console.log('AllHooksComponent --------- [constructor]');
    this.counter = 1;
  }
  inc() {
    console.log('AllHooksComponent --------- [counter]');
    this.counter += 1;
  }
  ngOnInit() {
    console.log('AllHooksComponent - OnInit');
  }
  ngOnDestroy() {
    console.log('AllHooksComponent - OnDestroy');
  }
  ngDoCheck() {
    console.log('AllHooksComponent - DoCheck');
  }
  ngOnChanges() {
```

```
42      console.log('AllHooksComponent - OnChanges');
43    }
44    ngAfterContentInit() {
45      console.log('AllHooksComponent - AfterContentInit');
46    }
47    ngAfterContentChecked() {
48      console.log('AllHooksComponent - AfterContentChecked');
49    }
50    ngAfterViewInit() {
51      console.log('AllHooksComponent - AfterViewInit');
52    }
53    ngAfterViewChecked() {
54      console.log('AllHooksComponent - AfterViewChecked');
55    }
56
57  }
```

Now let's add it to the app component, along with a `Toggle` button, like the one we used for the `OnDestroy` hook:

code/advanced-components/src/app/lifecycle/all-hooks/all-hooks-demo.component.html

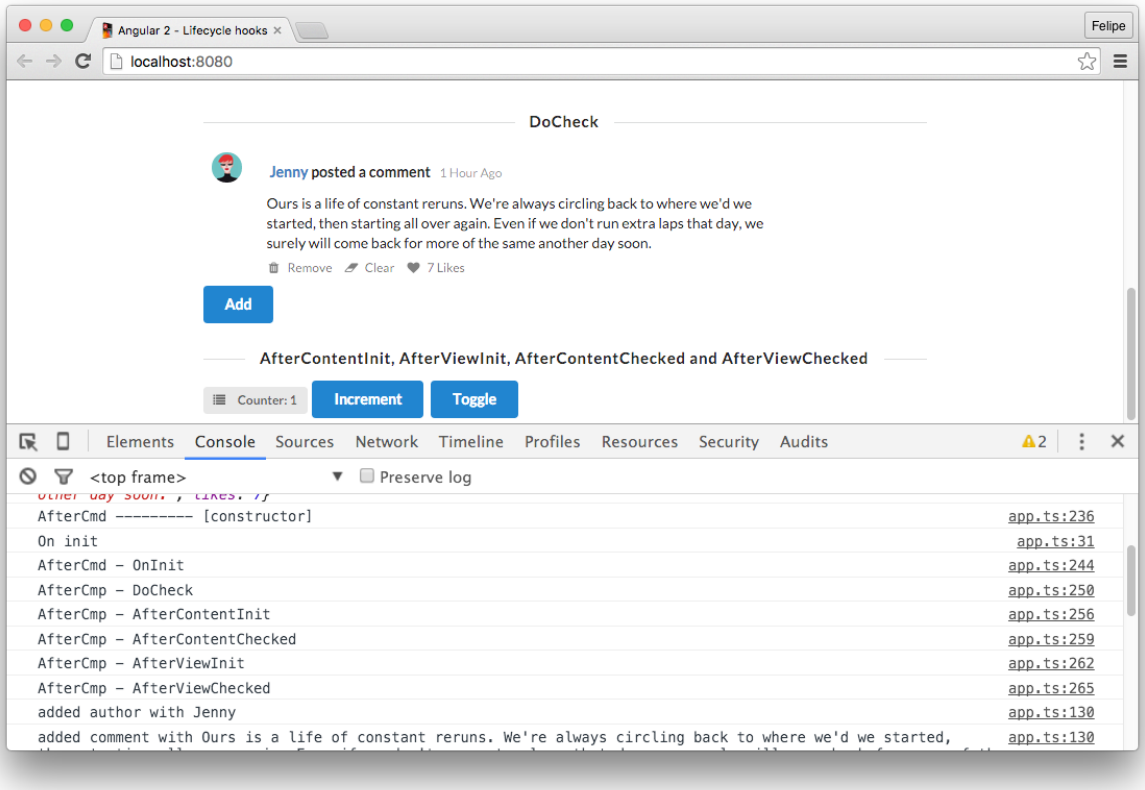
```
1  <h4 class="ui horizontal divider header">
2    AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked
3  </h4>
4
5  <app-all-hooks
6    *ngIf="displayAfters"
7    ></app-all-hooks>
8
9  <button class="ui primary button" (click)="toggleAfters()">
10   Toggle
11 </button>
```

The final implementation for the app demo component now will look like this:

code/advanced-components/src/app/lifecycle/all-hooks/all-hooks-demo.component.ts

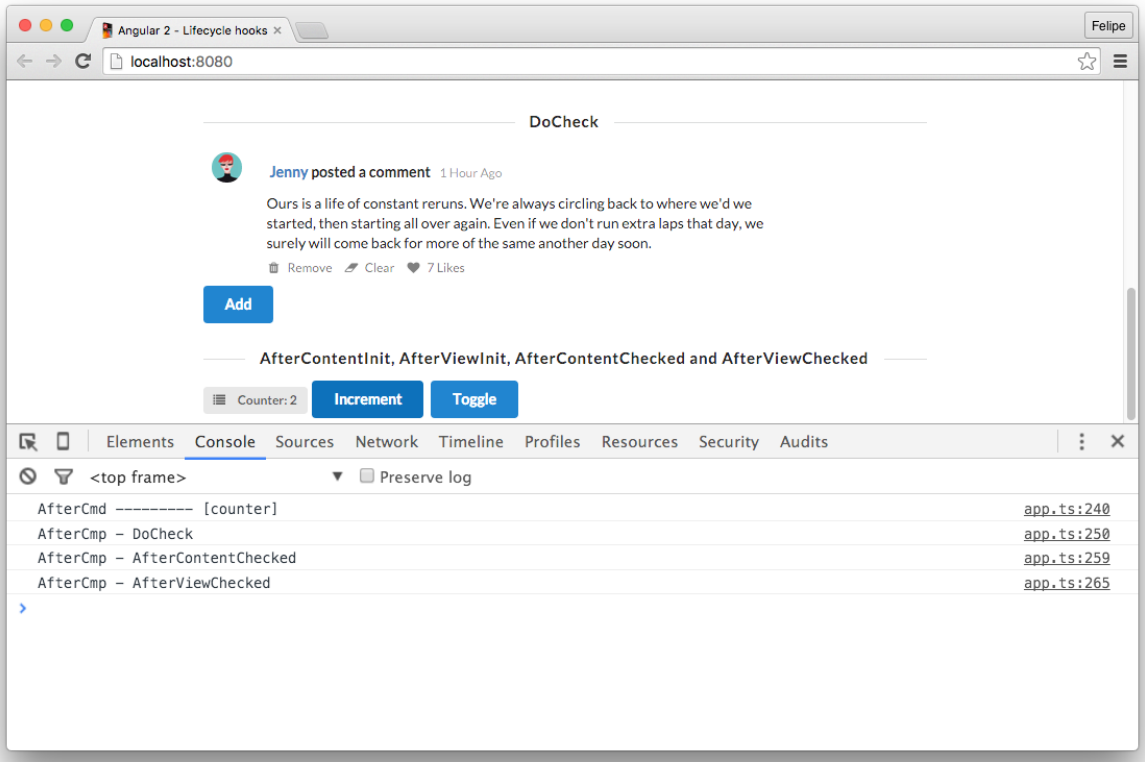
```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-all-hooks-demo',
  templateUrl: './all-hooks-demo.component.html',
  styles: []
})
export class AllHooksDemoComponent implements OnInit {
  displayAfters = true;

  constructor() { }

  ngOnInit() { }

  toggleAfters(): void {
    this.displayAfters = !this.displayAfters;
  }
}
```

When the application starts, we can see each hook is logged:



App started

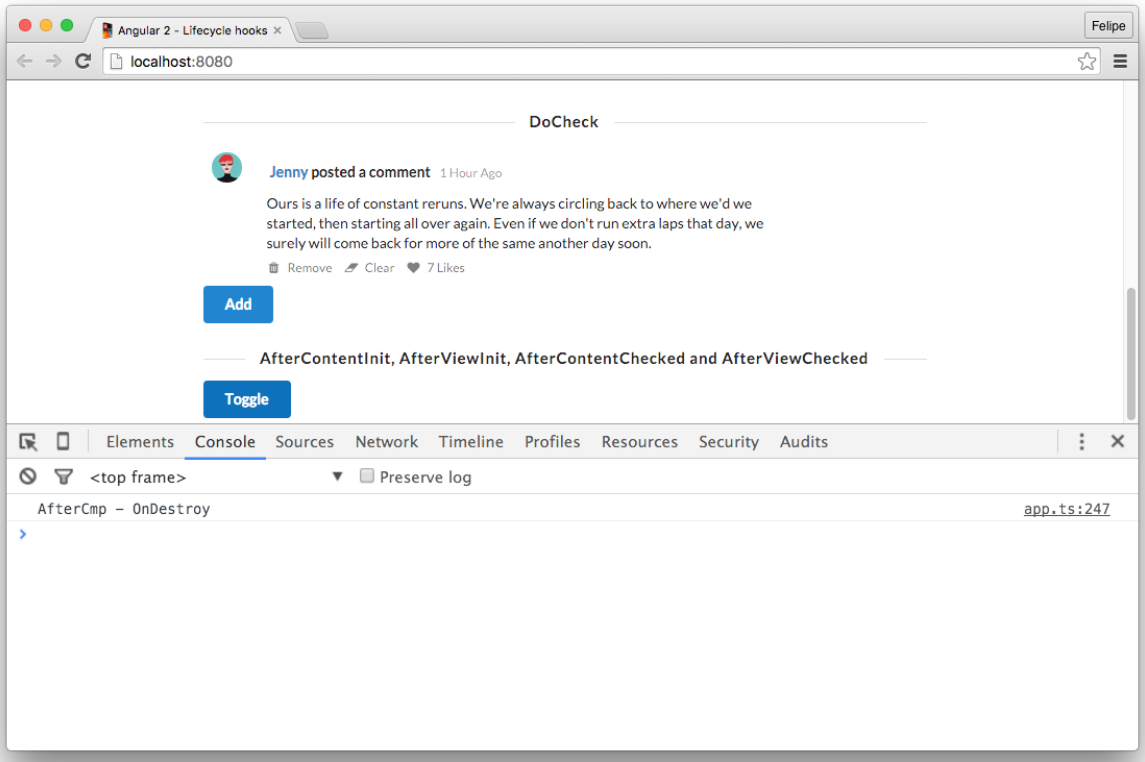
Now let's clear the console and click the `Increment` button:



After counter increment

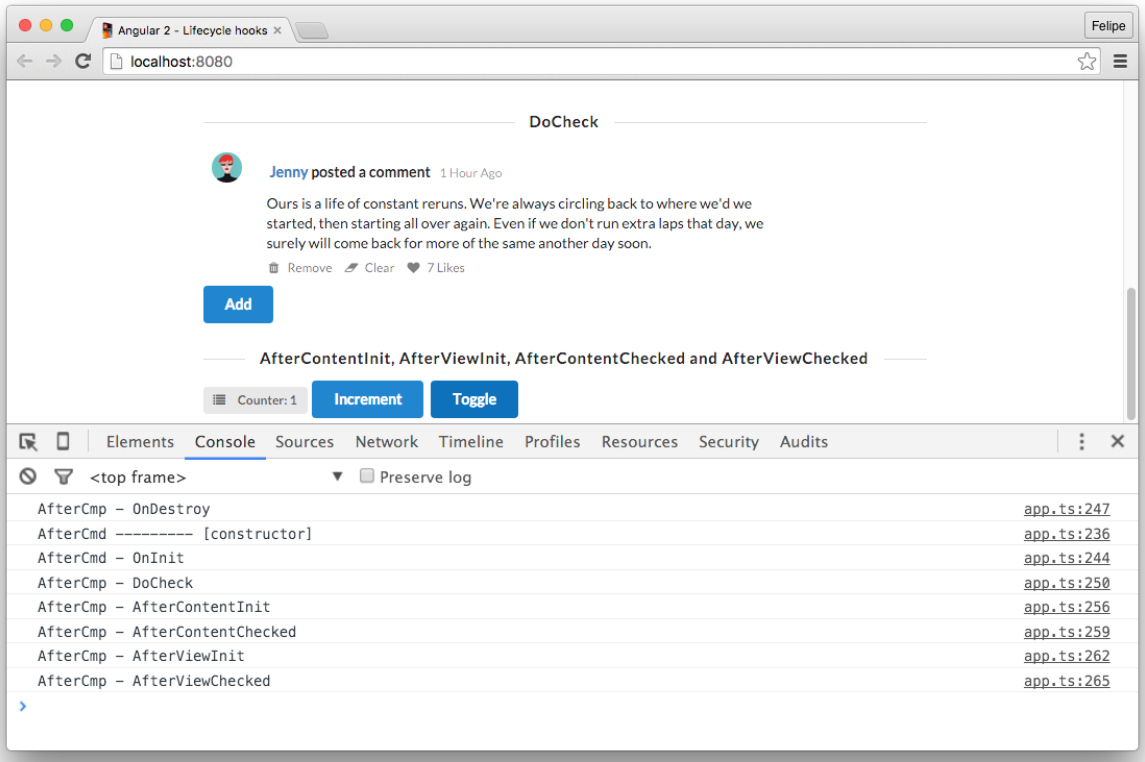
You can see that now only the `DoCheck`, `AfterContentCheck` and `AfterViewCheck` hooks were triggered.

Sure enough, if we click the `Toggle` button:



App started

And click it again:



App started

All the hooks are triggered.

Advanced Templates

Template elements are special elements used to create views that can be dynamically manipulated.

In order to make working with templates simpler, Angular provides some syntatic sugar to create templates, so we often don't create them by hand.

For instance, when we write:

```
<app-comment
  *ngFor="let comment of comments"
  [comment]="comment"
  (onRemove)="removeComment($event)">
</app-comment>
```

This gets converted into:

```
<app-comment
  template="ngFor let comment of comments; #i=index"
  [comment]="comment"
  (onRemove)="removeComment($event)">
</app-comment>
```

Which then gets converted into:

```
<template
  ngFor
  [ngForOf]="comments"
  let-comment="$implicit"
  let-index="i">
  <app-comment
    [comment]="comment"
    (onRemove)="removeComment($event)">
  </app-comment>
</template>
```

It's important that we understand this underlying concept so we can build our own directives.

Rewriting `ngIf` - `ngBookIf`

Let's create a directive that does exactly what `ngIf` does. Let's call it `ngBookIf`.

`ngBookIf` `@Directive`

We start by declaring the `@Directive` decorator for our class:

```
@Directive({
  selector: '[ngBookIf]'
})
```

We're using `[ngBookIf]` as the selector because, as we learned above, when we use `*ngBookIf="condition"`, it will be converted to:

```
<template ngBookIf [ngBookIf]="condition">
```

Since `ngBookIf` is also an attribute we need to indicate that we're expecting to receive it as an input.

The job of this directive should be to add the directive template contents when the condition is true and remove it when it's false.

So when the condition is true, we will use a *view container*. The view container is used to attach one or more views to the directive.

We will use the view container to either:

- create a new view with our directive template embedded or
- clear the view container contents.

Before we do that, we need to inject the `ViewContainerRef` and the `TemplateRef`. They will be injected with the directive's view container and template.

Here's the code we'll need:

code/advanced-components/src/app/templates/ng-book-if/ng-book-if.directive.ts

```
11 export class NgBookIfDirective {
12   constructor(private viewContainer: ViewContainerRef,
13               private template: TemplateRef<any>) {}
```

Now that we have references to both the view container and the template, we will use a TypeScript property setter construct and also specify that this is an `input` using the `Input()` decorator:

code/advanced-components/src/app/templates/ng-book-if/ng-book-if.directive.ts

```
15   @Input() set ngBookIf(condition) {
16     if (condition) {
17       this.viewContainer.createEmbeddedView(this.template);
18     } else {
19       this.viewContainer.clear();
20     }
21   }
22 }
```

This method will be called every time we set a value on the `ngBookIf` property of our class. That is, this method will be called anytime the `condition` in `ngBookIf="condition"` changes.

Now we use the view container's `createEmbeddedView` method to attach the directive's template if the condition is true, or the `clear` method to remove everything from the view container.

Using `ngBookIf`

In order to use our directive, we can write the following demo component:

code/advanced-components/src/app/templates/ng-book-if/ng-book-if-demo.component.ts

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

@Component({
  selector: 'app-ng-book-if',
  templateUrl: './ng-book-if-demo.component.html',
})
export class NgBookIfDemoComponent {
  display: boolean;

  constructor() {
    this.display = true;
  }

  toggle() {
    this.display = !this.display;
  }
}
```

and template:

code/advanced-components/src/app/templates/ng-book-if/ng-book-if-demo.component.html

```
<button class="ui primary button" (click)="toggle()">
  Toggle
</button>

<div *ngBookIf="display">
  The message is displayed
</div>
```

When we run the application, we can see that the directive works as expected: when we click the **Toggle** button the message *This message is displayed* is toggled on and off the page.

Rewriting `ngFor` - `NgBookFor`

Now let's write a simplified version of the `ngFor` directive that Angular provides to handle repetition of templates for a given collection.

`NgBookFor` template deconstruction

This directive will be used with the `*NgBookFor="let var of collection"` notation.

Like we did for the previous directive, we need to declare the selector as being `[NgBookFor]`. However the input parameter in this case won't be `NgBookFor` only.

If we look back at how Angular converts the `*something="let var in collection"` notation, we can see that the final form of the element is the equivalent of:

```
<template something [somethingOf]="collection" let-var="$implicit">
  <!-- ... -->
</template>
```

As we can see, the attribute that's being passed isn't `something` but `somethingOf` instead. That's where our directive receives the collection we're iterating on.

In template that is generated, we're going to have a local view variable `#var`, that will receive the value from the `$implicit` local variable. That's the name of the local variable that Angular uses when "de-sugaring" the syntax into a `template`.

`NgBookFor @Directive`

Time to write the directive.

code/advanced-components/src/app/templates/ng-book-for/ng-book-for.directive.ts

```
import {
  Directive,
  IterableDiffer,
  IterableDiffers,
  ViewRef,
  ViewContainerRef,
  TemplateRef,
  ChangeDetectorRef,
  DoCheck,
  Input
} from '@angular/core';

@Directive({
  selector: '[ngBookFor]'
})
export class NgBookForDirective implements DoCheck {
  private items: any;
  private differ: IterableDiffer<any>;
```

```
19    private views: Map<any, ViewRef> = new Map<any, ViewRef>();
20
21
22    constructor(private viewContainer: ViewContainerRef,
23                private template: TemplateRef<any>,
24                private changeDetector: ChangeDetectorRef,
25                private differs: IterableDiffers) {}
```

We are declaring some properties for our class:

- `items` holds the collection we're iterating on
- `differ` is an `IterableDiffer` (which we learned about in the Lifecycle Hooks section above) that will be used for change detection purposes
- `views` is a `Map` that will link a given item on the collection with the view that contains it

The constructor will receive the `viewContainer`, the template and an `IterableDiffers` instance (we discussed each of these things earlier in this chapter above).

Now, the next thing that's being injected is a change detector. We will have a deep dive in change detection in the next section. For now, let's say that this is the class that Angular creates to trigger the detection when properties of our directive change.

The next step is to write code that will trigger when we set the `ngBookForOf` input:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for.directive.ts

```
27    @Input() set ngBookForOf(items) {
28      this.items = items;
29      if (this.items && !this.differ) {
30        this.differ = this.differs.find(items).create(this.changeDetector);
31      }
32    }
```

When we set this attribute, we're keeping the collection on the directive's `item` property and if the collection is valid and we don't have a differ yet, we create one.

To do that, we're creating an instance of `IterableDiffer` that reuses the directive's change detector (the one we injected in the constructor).

Now it's time to write the code that will react to a change on the collection. For this, we're going to use the **DoCheck** lifecycle hook by implementing the `ngDoCheck` method as follows:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for.directive.ts

```
34  ngDoCheck(): void {
35    if (this.differ) {
36      const changes = this.differ.diff(this.items);
37      if (changes) {
38
39        changes.forEachAddedItem((change) => {
40          const view = this.viewContainer.createEmbeddedView(
41            this.template,
42            {'$implicit': change.item});
43          this.views.set(change.item, view);
44        });
45        changes.forEachRemovedItem((change) => {
46          const view = this.views.get(change.item);
47          const idx = this.viewContainer.indexOf(view);
48          this.viewContainer.remove(idx);
49          this.views.delete(change.item);
50        });
51      }
52    }
53  }
```

Let's break this down a bit. First thing we do in this method is make sure we already instantiated the differ. If not, we do nothing.

Next, we ask the differ what changed. If there are changes, we first iterate through the itmes that were added using `changes.forEachAddedItem`. This method will receive a `CollectionChangeRecord` object for every element that was added.

Then for each element, we create a new embedded view using the view container's `createEmbeddedView` method.

```
let view = this.viewContainer.createEmbeddedView(this.template, {'$implicit': ch\
ange.item});
```

The second argument to `createEmbeddedView` is the *view context*. In this case, we're setting the `$implicit` local variable to `change.item`. This will allow us to reference the variable we declared back on the `*NgBookFor="let var of collection"` as `var` on that view. That is, the `var` in `let var` is the `$implicit` variable. We use `$implicit` because we don't know what name the user will assign to it when we're writing this component.

The final thing we need to do is to connect the item with the collection to its view. The reason behind this is that, if an item gets removed from the collection, we need to get rid of the correct view, as we do next.

Now for each item that was removed from the collection, we use the item-to-view map we keep to find the view. Then we ask the view container for the index of that view. We need that because the view container's `remove` method needs an index. Finally, we also remove the view from the item-to-view map.

Trying out our directive

To test our new directive, let's write the following component:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for-demo.component.ts

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

@Component({
  selector: 'app-ng-book-for-demo',
  templateUrl: './ng-book-for-demo.component.html'
})
export class NgBookForDemoComponent implements OnInit {
  people: any[];

  constructor() { }

  ngOnInit() {
    this.people = [
      {name: 'Joe', age: 10},
      {name: 'Patrick', age: 21},
      {name: 'Melissa', age: 12},
      {name: 'Kate', age: 19}
    ];
  }

  remove(p) {
    const idx: number = this.people.indexOf(p);
    this.people.splice(idx, 1);
    return false;
  }

  add(name, age) {
    this.people.push({name: name.value, age: age.value});
    name.value = '';
    age.value = '';
  }
}
```

and template:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for-demo.component.html

```
<ul>
  <li *ngBookFor="let p of people">
    {{ p.name }} is {{ p.age }}
    <a href (click)="remove(p)">Remove</a>
  </li>
</ul>

<div class="ui form">
  <div class="fields">
    <div class="field">
      <label>Name</label>
      <input type="text" #name placeholder="Name">
    </div>
    <div class="field">
      <label>Age</label>
      <input type="text" #age placeholder="Age">
    </div>
  </div>
</div>
<div class="ui submit button"
     (click)="add(name, age)">
  Add
</div>
```

We're using our directive to iterate through a list of people:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for-demo.component.html

```
<ul>
  <li *ngBookFor="let p of people">
    {{ p.name }} is {{ p.age }}
    <a href (click)="remove(p)">Remove</a>
  </li>
</ul>
```

When we click **Remove** we remove the item from the collection, triggering the change detection.

We also provide a form that allows adding items to the collection:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for-demo.component.html

```
<div class="ui form">
  <div class="fields">
    <div class="field">
      <label>Name</label>
      <input type="text" #name placeholder="Name">
    </div>
    <div class="field">
      <label>Age</label>
      <input type="text" #age placeholder="Age">
    </div>
  </div>
</div>
<div class="ui submit button"
     (click)="add(name, age)">
  Add
</div>
```

Change Detection

As a user interacts with our app, data (state) changes and our app needs to respond accordingly.

One of the big problems any modern JavaScript framework needs to solve is how to figure out when changes have happened and re-render components accordingly.

In order to make the view react to changes to components state, Angular uses *change detection*.

What are the things that can trigger changes in a component's state? The most obvious thing is user interaction. For instance, if we have a component:

```
@Component({
  selector: 'my-component',
  template: `
  Name: {{name}}
  <button (click)="changeName()">Change!</button>
  `
})
class MyComponent {
  name: string;
  constructor() {
    this.name = 'Felipe';
  }
```

```
13
14    changeName() {
15      this.name = 'Nate';
16    }
17  }
```

We can see that when the user *clicks* on the **Change!** button, the component's *name* property will change.

Another source of change could be, for instance, a HTTP request:

```
1   @Component({
2     selector: 'my-component',
3     template: `
4     Name: {{name}}
5     `
6   })
7   class MyComponent {
8     name: string;
9     constructor(private http: Http) {
10      this.http.get('/names/1')
11        .map(res => res.json())
12        .subscribe(data => this.name = data.name);
13    }
14  }
```

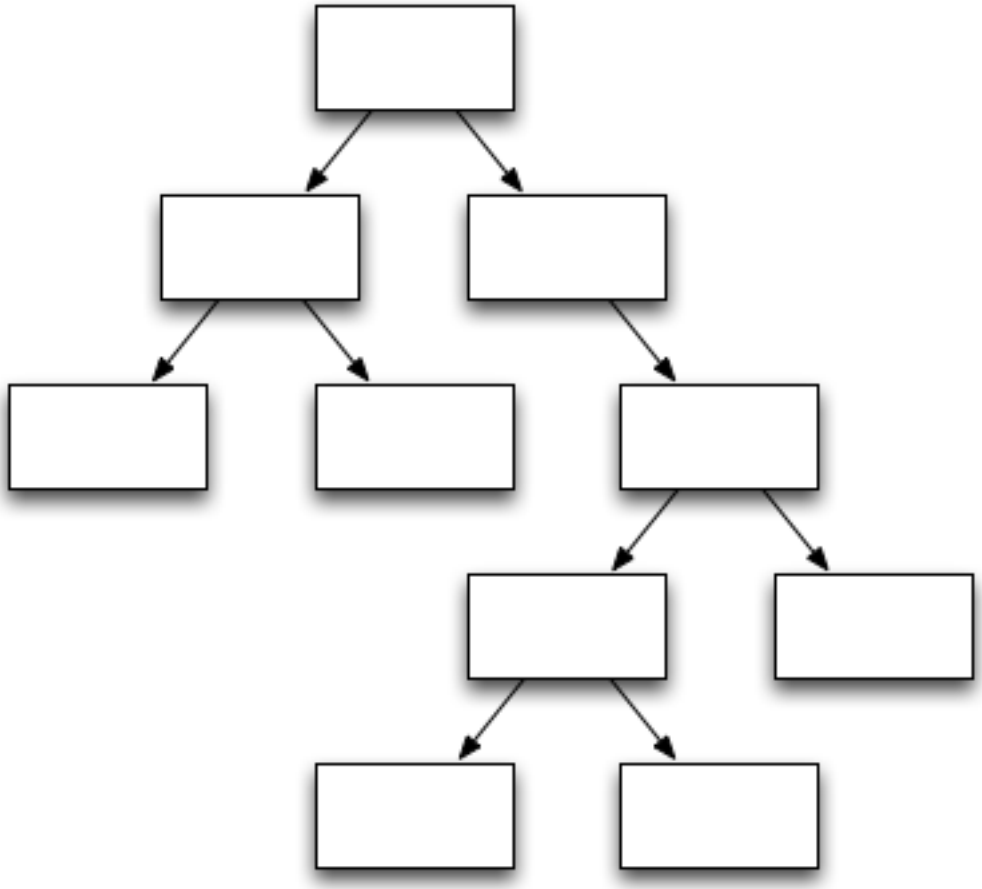
And finally, we could have a timer that would trigger the change:

```
1   @Component({
2     selector: 'my-component',
3     template: `
4     Name: {{name}}
5     `
6   })
7   class MyComponent {
8     name: string;
9     constructor() {
10      setTimeout(() => this.name = 'Felipe', 2000);
11    }
12  }
```

But how does Angular become aware of these changes?

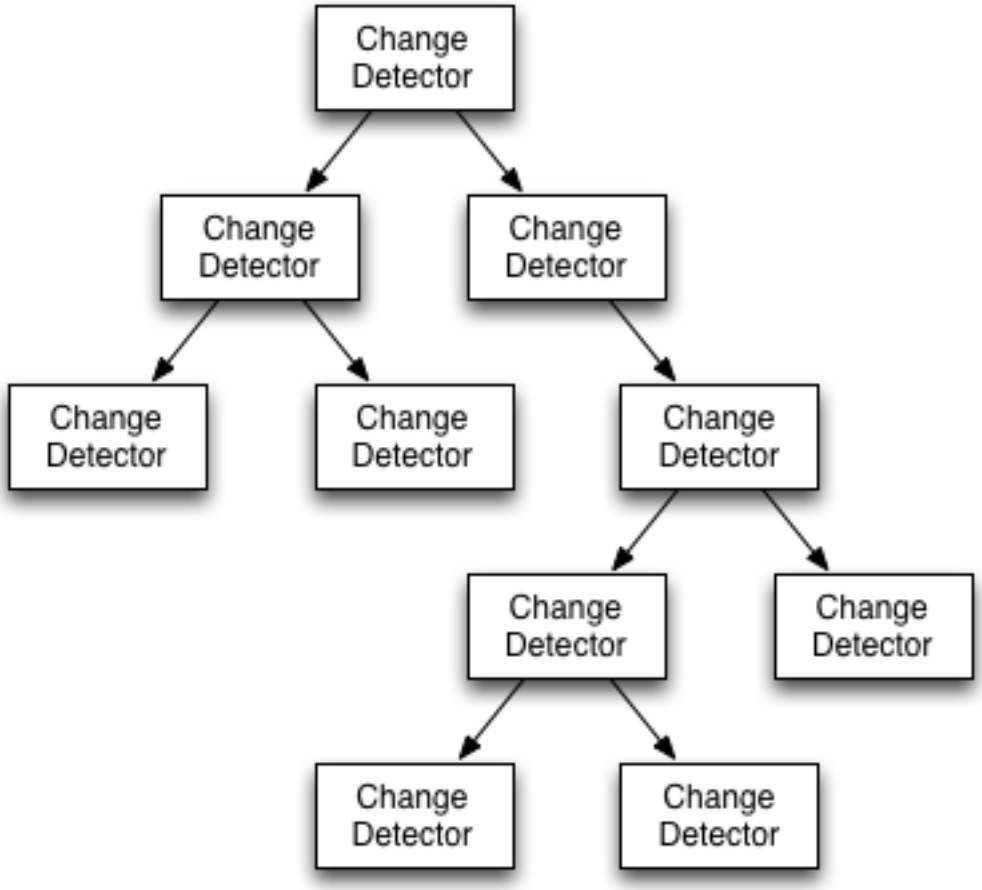
The first thing to know is that each component gets a change detector.

Like we've seen before, a typical application will have a number of components that will interact with each other, creating a dependency tree like below:



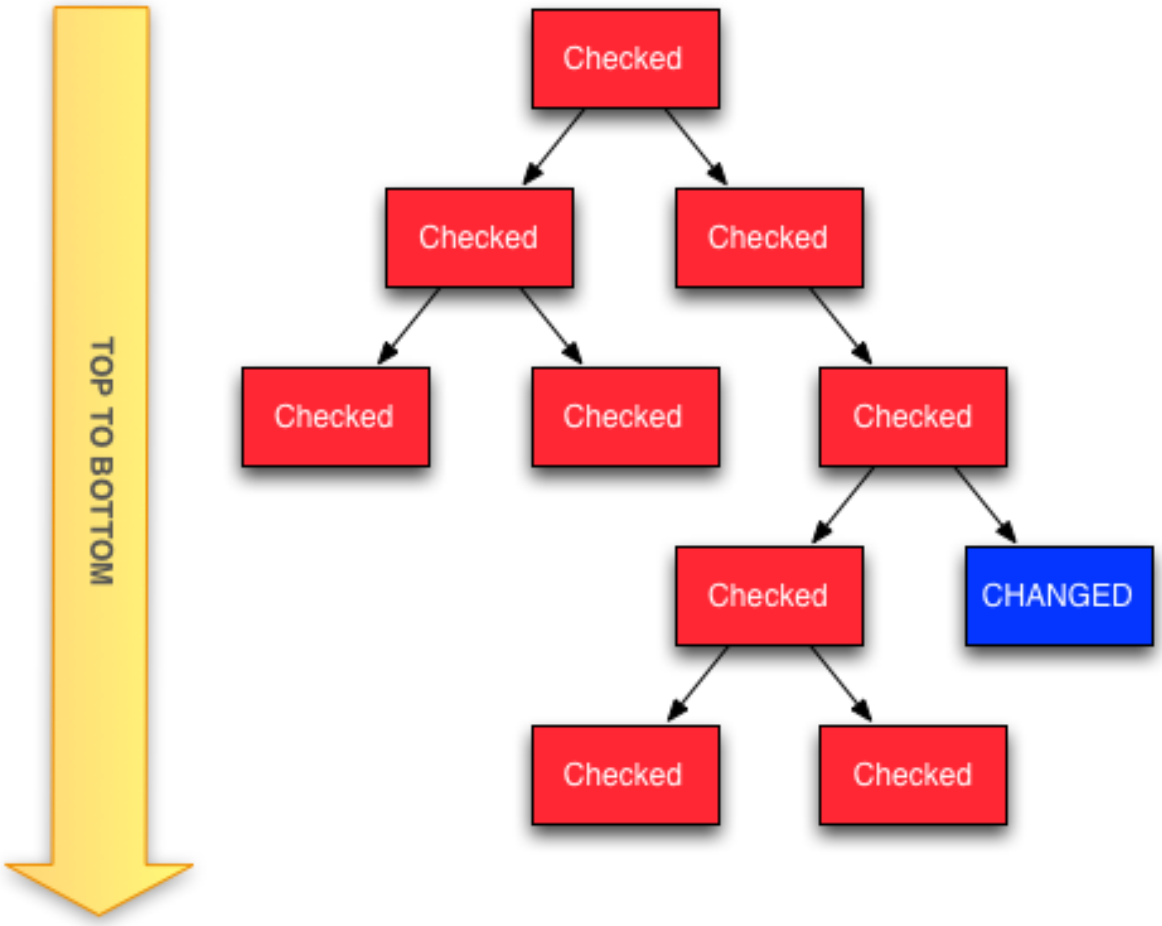
Component tree

For each component on our tree, a change detector is created and so we end up with a tree of change detectors:



Change detector tree

When one of the the components change, no matter where it is in the tree, a change detection pass is triggered for the whole tree. This happens because Angular scans for changes from the top component node, all the way to the bottom leaves of the tree.



Default change detection

In our diagram above, the component in blue changed, but as we can see, it triggered checks for the whole component tree. Objects that were checked are indicated in red (note that the component itself was also checked).

It is natural to think that this check may be a very expensive operation. However, due to a number of optimizations (that make Angular code eligible for further optimization by the JavaScript engine), it's actually surprisingly fast.

Customizing Change Detection

There are times that the built-in or default change detection mechanism may be overkill. One example is if you're using immutable objects or if your application architecture relies on observables. In these cases, Angular provides mechanisms for configuring the change detection system so that you get very fast performance.

The first way to change the change detector behavior is by telling a component that it should only be checked if one of its *input values* change.

To recap, an input value is an attribute your component receives from the outside world. For instance, in this code:

```
class Person {
  constructor(public name: string, public age: string) {}
}

@Component({
  selector: 'mycomp',
  template: `
    <div>
      <span class="name">{{ person.name }}</span>
      is {{ person.age }} years old.
    </div>
  `
})
class MyComp {
  @Input() person: Person;
}
```

We have `person` as an input attribute. Now, if we want to make this component change only when its input attribute changes, we just need to change the change detection strategy, by setting its `changeDetection` attribute to `ChangeDetectionStrategy.OnPush`.

By the way, if you're curious, the default value for `changeDetection` is `ChangeDetectionStrategy.Default`.

Let's write a small experiment with two components. The first one will use the default change detection behavior and the other will use the `OnPush` strategy:

code/advanced-components/src/app/change-detection/on-push-demo/profile.model.ts

```
/**
 * User Profile object, stores the first and
 * last name as well as a function that gives the time
 **/
export class Profile {
  constructor(public first: string, public last: string) {}

  lastChanged() {
```

```
    return new Date();
  }
}
```

So we start with some imports and we declare a `Profile` class that will be used as the input in both of our components. Notice that we also created a method called `lastChange()` on the `Profile` class. It will help us determine when the change detection is triggered. When a given component is marked as needing to be checked, this method will be called, since it's present on the template. So this method will reliably indicate the last time the component was checked for changes.

Next, we declare the `DefaultChangeDetectionComponent` that will use the default change detection strategy:

code/advanced-components/src/app/change-detection/on-push-demo/default-change-detection.component.ts

```
import {
  Component,
  Input
} from '@angular/core';
import { Profile } from './profile.model';

@Component({
  selector: 'app-default-change-detection',
  templateUrl: './default-change-detection.component.html'
})
export class DefaultChangeDetectionComponent {
  @Input() profile: Profile;
}
```

and template:

code/advanced-components/src/app/change-detection/on-push-demo/default-change-detection.component.html

```
<h4 class="ui horizontal divider header">
  Default Strategy
</h4>

<form class="ui form">
  <div class="field">
    <label>First Name</label>
    <input
    type="text"
    [(ngModel)]="profile.first"
```

```
    name="first"
    placeholder="First Name">
  </div>
  <div class="field">
    <label>Last Name</label>
    <input
    type="text"
    [(ngModel)]="profile.last"
    name="last"
    placeholder="Last Name">
  </div>
</form>

<h5><em>Updates if either changes (e.g. more often)</em></h5>
<div>
  {{profile.lastChanged() | date:'medium'}}
</div>
```

And a second component using `OnPush` strategy:

code/advanced-components/src/app/change-detection/on-push-demo/on-push-change-detection.component.ts

```
import {
  Component,
  Input,
  ChangeDetectionStrategy
} from '@angular/core';
import { Profile } from './profile.model';

@Component({
  selector: 'app-on-push-change-detection',
  changeDetection: ChangeDetectionStrategy.OnPush,
  templateUrl: './on-push-change-detection.component.html'
})
export class OnPushChangeDetectionComponent {
  @Input() profile: Profile;
}
```

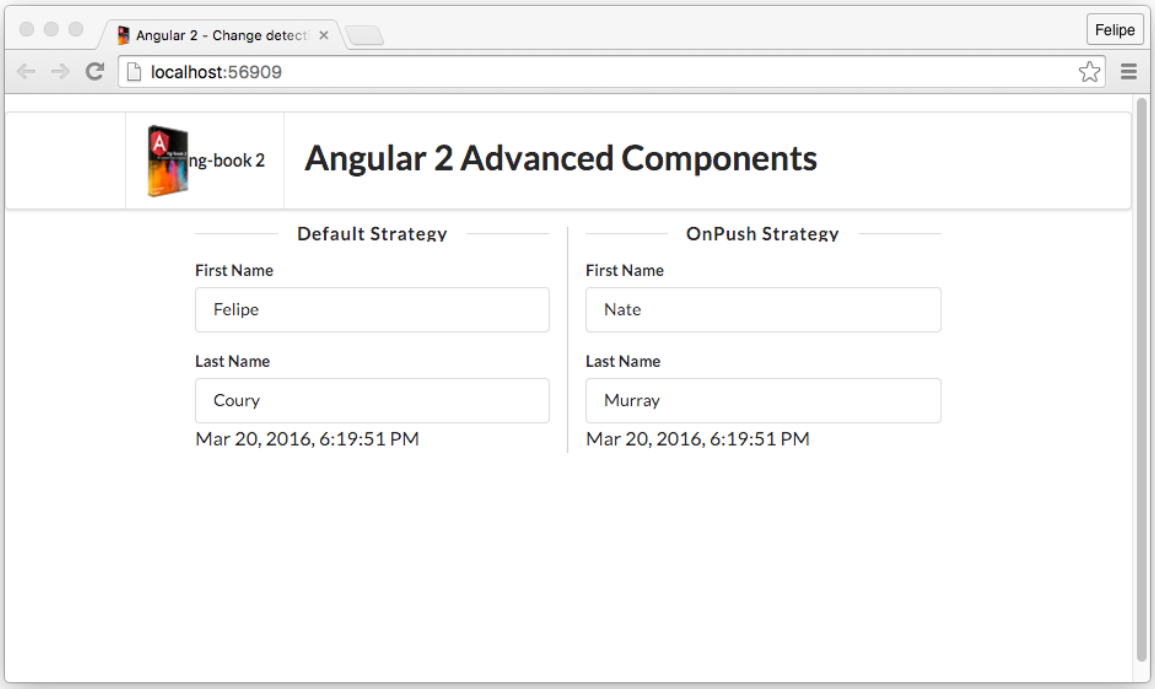
As we can see, both components use the same template. The only thing that is different is the header.

Finally, let's add the component that will render both components side by side:

code/advanced-components/src/app/change-detection/on-push-demo/on-push-demo.component.ts

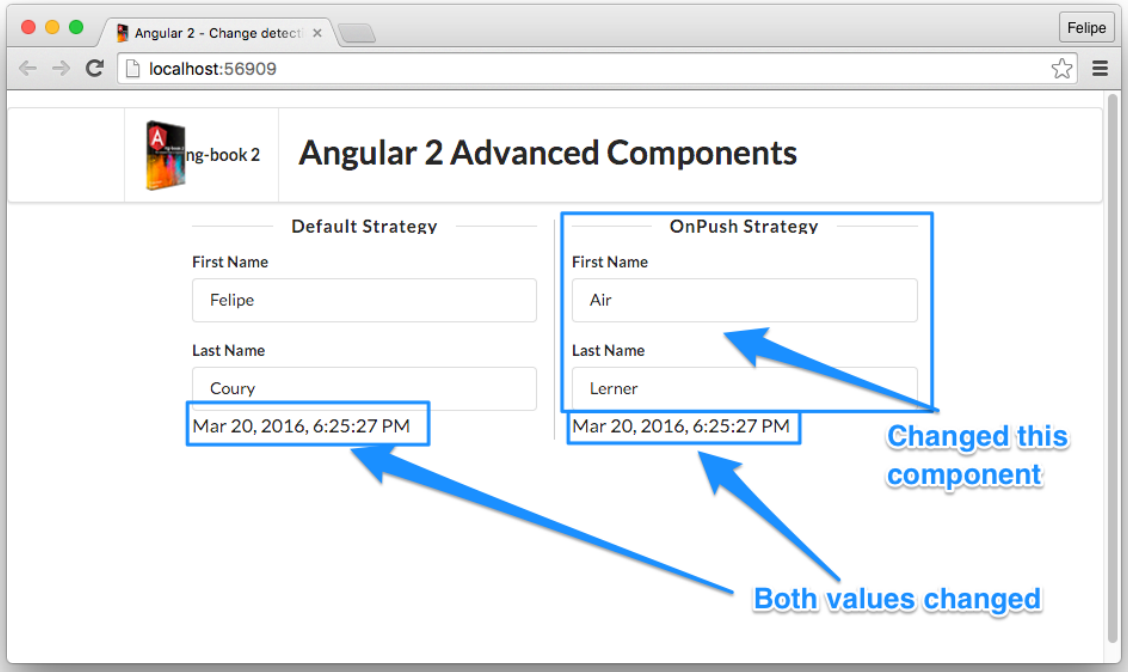
```
import { Component } from '@angular/core';
import { Profile } from './profile.model';

@Component({
  selector: 'app-on-push-demo',
  template: `
  <div class="ui page grid">
    <div class="two column row">
      <div class="column area">
        <app-default-change-detection
          [profile]="profile1">
        </app-default-change-detection>
      </div>
      <div class="column area">
        <app-on-push-change-detection
          [profile]="profile2">
        </app-on-push-change-detection>
      </div>
    </div>
  </div>
  `
})
export class OnPushDemoComponent {
  profile1: Profile = new Profile('Felipe', 'Coury');
  profile2: Profile = new Profile('Nate', 'Murray');
}
```

When we run this application, we should see both components rendered like below:



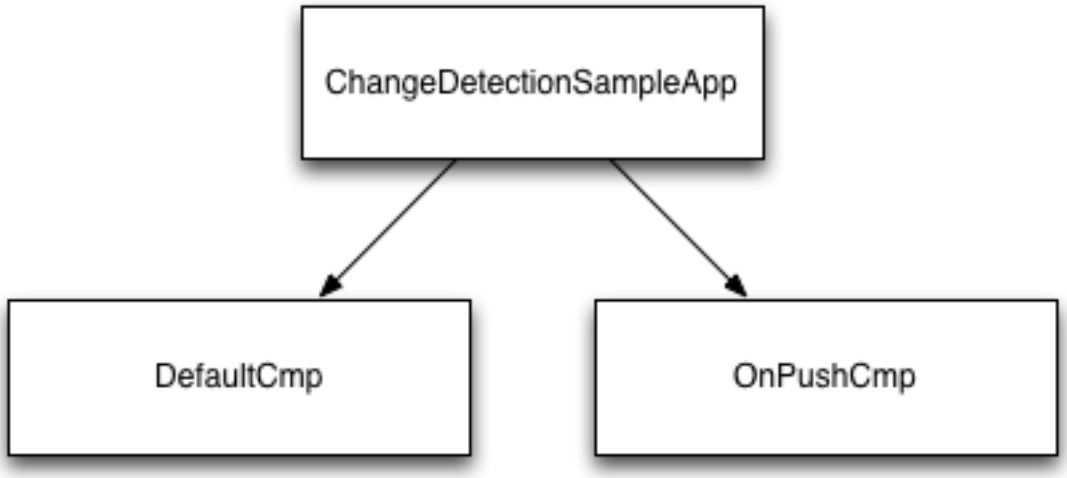
Default vs. OnPush strategies

When we change something on the component on the left, with the default strategy, we notice that the timestamp for the component on the right doesn't change:



OnPush changed, default got checked

To understand why this happened, let's check this new tree of components:

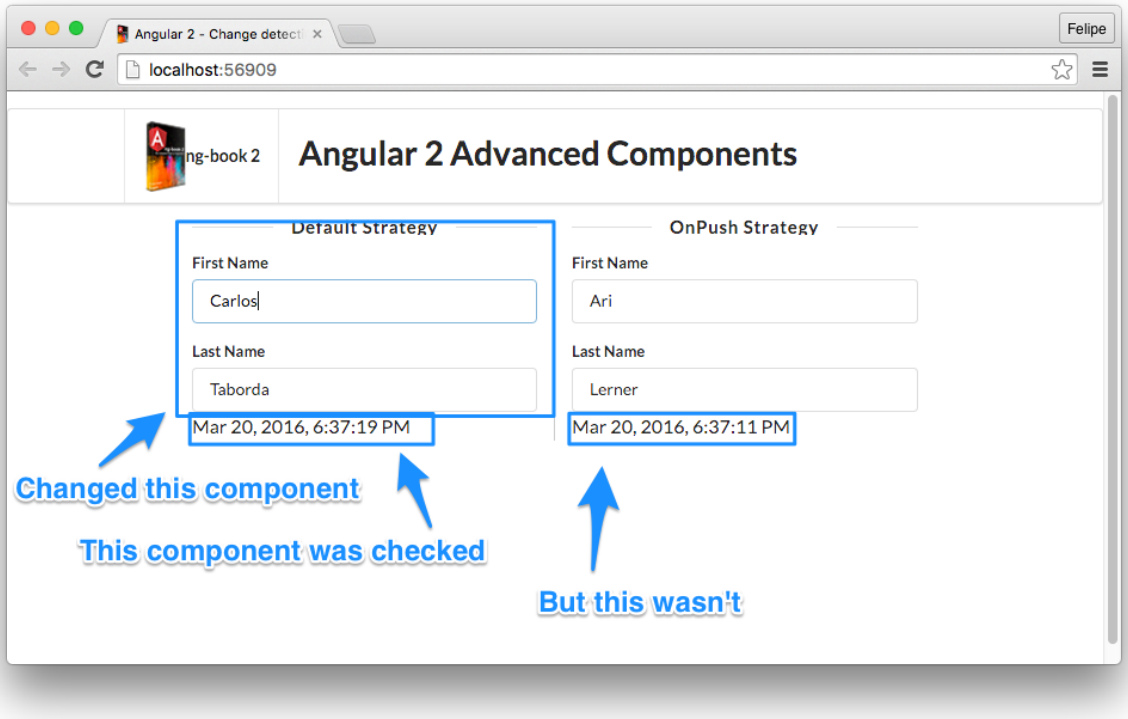


Tree of components

Angular checks for changes from the top to the bottom, so it queried first `OnPushDemoComponent`, then `DefaultChangeDetectionComponent` and finally `OnPushChangeDetectionComponent`. When it

inferred that `OnPushChangeDetectionComponent` changed, it updates all the components of the tree, from top to bottom, making the `DefaultChangeDetectionComponent` to be rendered again.

Now when we change the value of the component on the right:



Default changed, OnPush didn't get checked

So now the change detection engine kicked in, the `DefaultChangeDetectionComponent` component was checked but `OnPushChangeDetectionComponent` wasn't. This happened because when we set the `OnPush` strategy for this component, it made the change detection kick in for this component *only* when one of its input attributes change. Changing other components of the tree doesn't trigger this component's change detector.

Zones

Under the hood, Angular uses a library called Zones to automatically detect changes and trigger the change detection mechanism. Zones will automatically tell Angular that something changed under the most common scenarios:

- when a DOM Event occurs (like *click*, *change*, etc.)
- when an HTTP request is resolved
- when a Timer is trigger (*setTimeout* or *setInterval*)

However, there are scenarios where Zones won't be able to automatically identify that something changed. That's another scenario where the **OnPush** strategy can be very useful.

A few examples of things that is out of the Zones control, would be:

- using a third party library that runs asynchronously
- immutable data
- Observables

these are perfect candidates for using **OnPush** along with a technique to manually hint Angular that something changed.

Observables and OnPush

Let's write a component that receives an **Observable** as a parameter. Every time we receive a value from this observable, we will increment a counter that is a property of the component.

If we were using the regular change detection strategy, any time we incremented the counter, we would get change detection triggered by Angular. However, we will have this component use the **OnPush** strategy and, instead of letting the change detector kick in for each increment, we'll only kick it when the number is a multiple of 5 or when the observable completes.

In order to do that, let's write our component:

code/advanced-components/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
import {
  Component,
  OnInit,
  Input,
  ChangeDetectionStrategy,
  ChangeDetectorRef
 } from '@angular/core';
import { Observable } from 'rxjs/Rx';

@Component({
  selector: 'app-observable-change-detection',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
  <div>
    <div>Total items: {{counter}}</div>
  </div>
  `
})
```

```
export class ObservableChangeDetectionComponent implements OnInit {
  @Input() items: Observable<number>;
  counter = 0;

  constructor(private changeDetector: ChangeDetectorRef) {
  }

  ngOnInit() {
    this.items.subscribe((v) => {
      console.log('got value', v);
      this.counter++;
      if (this.counter % 5 === 0) {
        this.changeDetector.markForCheck();
      }
    },
    null,
    () => {
      this.changeDetector.markForCheck();
    });
  }
}
```

Let's break down the code a bit so we can make sure we understand. First, we're declaring the component to take `items` as the input attribute and to use the `OnPush` detection strategy:

code/advanced-components/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
@Component({
  selector: 'app-observable-change-detection',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
  <div>
    <div>Total items: {{counter}}</div>
  </div>
  `
})
```

Next, we're storing our input attribute on the `items` property of the component class, and setting another property, called `counter`, to `0`.

code/advanced-components/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
19 export class ObservableChangeDetectionComponent implements OnInit {
20   @Input() items: Observable<number>;
21   counter = 0;
```

Then we use the constructor to get hold of the component's change detector:

code/advanced-components/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
23   constructor(private changeDetector: ChangeDetectorRef) {
24   }
```

Then, during the component initialization, on the `ngOnInit` hook:

code/advanced-components/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
26   ngOnInit() {
27     this.items.subscribe((v) => {
28       console.log('got value', v);
29       this.counter++;
30       if (this.counter % 5 === 0) {
31         this.changeDetector.markForCheck();
32       }
33     },
34     null,
35     () => {
36       this.changeDetector.markForCheck();
37     });
38   }
```

We're subscribing to the Observable. The `subscribe` method takes three callbacks as arguments: `onNext`, `onError` and `onCompleted`.

Our `onNext` callback will print out the value we got, then increment the counter. Finally, if the current counter value is a multiple of 5, we call the change detector's `markForCheck` method. That's the method we use whenever we want to tell Angular that a change has been made, so the change detector should kick in.

Then for the `onError` callback, we're using `null`, indicating we don't want to handle this scenario.

Finally, for the `onComplete` callback, we're also triggering the change detector, so the final counter can be displayed.

Now, on to the application component code, that will create the subscriber:

code/advanced-components/src/app/change-detection/observables-demo/observables-demo.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Rx';

@Component({
  selector: 'app-observables-demo',
  template: `
  <app-observable-change-detection
    [items]="itemObservable">
    </app-observable-change-detection>
  `
})
export class ObservablesDemoComponent implements OnInit {
  itemObservable: Observable<number>;

  constructor() { }

  ngOnInit() {
    this.itemObservable = Observable.timer(100, 100).take(101);
  }

}
```

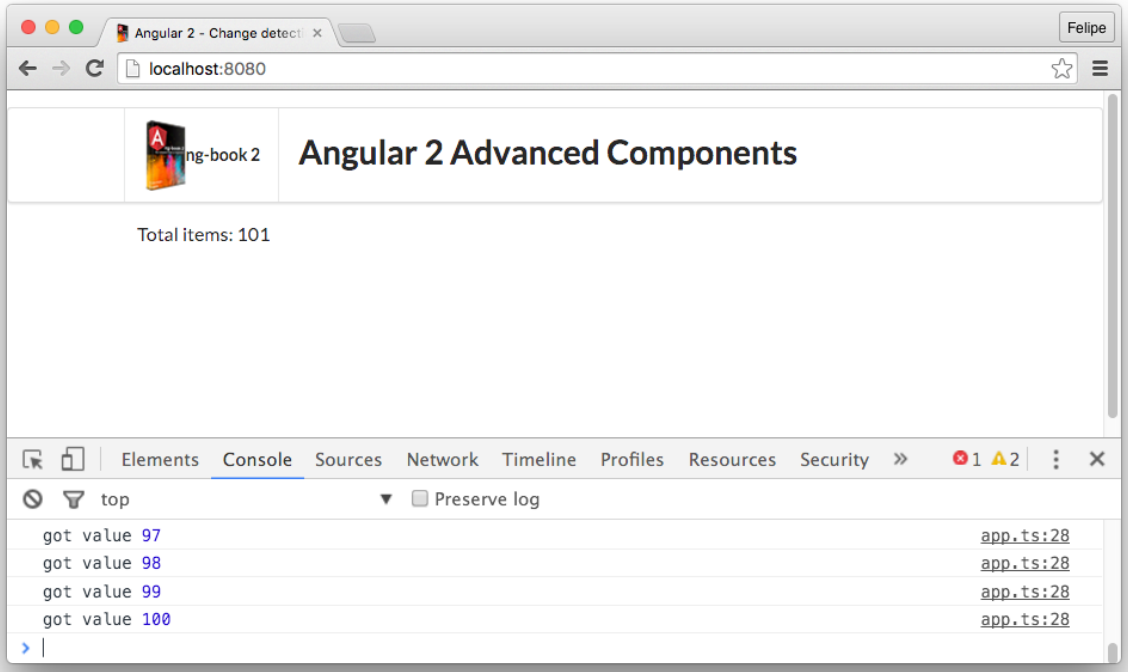
The important line here is the following:

```
this.itemObservable = Observable.timer(100, 100).take(101);
```

This line creates the Observable we're passing to the component on the `items` input attribute. We're passing two parameters to the `timer` method: the first is the number of milliseconds to wait before producing the first value and the second is the milliseconds to wait between values. So this observable will generate sequential values every 100 values forever.

Since we don't want the observable to run forever, we use the `take` method, to take only the first 101 values.

When we run this code, we'll see that the counter will only be updated for each 5 values obtained from the observer and also when the observable completes, generating a final value of 101:



Manually triggering change detection

Summary

Angular provides us with many tools we can use for writing advanced components. Using the techniques in this chapter you will be able to write nearly any component functionality you wish.

However, there's one important concept that you'll use in many advanced components that we haven't talked about yet: Dependency Injection.

With dependency injection we can hook our components into many other parts of the system. In the next chapter we'll talk about what DI is, how you can use it in your apps, and common patterns for injecting services.