FITPED

Frameworks for frontend application development (jQuery, Angular, Vue)

Jozef Kapusta
Wojciech Baran
Ján Skalka

www.fitped.eu

2021

Co-funded by the Erasmus+ Programme of the European Union

Work-Based Learning in Future IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

Frameworks for Frontend Application Development (jQuery, Angular, Vue)

Published on

November 2021

Authors

Jozef Kapusta | Pedagogical University of Cracow, Poland

Wojciech Baran | Pedagogical University of Cracow, Poland

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Reviewers

Martin Drlík | Constantine the Philosopher University in Nitra, Slovakia

Peter Švec | Teacher.sk, Slovakia

Cyril Klimeš | Mendel University in Brno, Czech Republic

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

ISBN 978-80-558-1792-7

Table of Contents

jQuery 6
1 jQuery 7
1.1 What is jQuery? 8
1.2 jQuery introduction 12
1.3 Events 18
1.4 More Selectors 24
2 Playground 31
2.1 Hide vs. show 32
2.2 Classes 36
Angular 47
3 Introduction 48
3.1 What is Angular? 49
3.2 TypeScript 49
3.3 Components 50
3.4 Templates 51
3.5 Dependency injection 51
3.6 Angular CLI 52
3.7 Installation 52
3.8 Application structure 53
3.9 Naming 54
3.10 First-party libraries 55
3.11 Introduction (Exercises) 56
4 Components 58
4.1 Components 59
4.2 Overview 62
4.3 Lifecycle 65
4.4 Encapsulation 68
4.5 Interaction 70
4.6 Styles 77
4.7 Content projection 79
4.8 Dynamic Components 82
4.9 Angular elements 85
4.10 Components (Exercises) 87

5 Templates 90
5.1 Text interpolation 91
5.2 Template statements 95
5.3 Pipes 97
5.4 Property binding 99
5.5 Attribute, class, and style bindings 105
5.6 Event binding 108
5.7 Two-way binding 111
5.8 Template variables 113
5.9 Templates (Exercises) 116
6 Directive 118
6.1 Built-in directives 119
6.2 Attribute directives 126
6.3 Structural Directives 132
6.4 Directives (Exercises) 135
7 Dependency Injection 136
7.1 Dependency injection 137
7.2 DI Providers 138
7.3 Dependency injection (Exercises) 143
8 Forms 145
8.1 Forms 146
8.2 Reactive forms 149
8.3 Validating form input 157
8.4 HTTP Client 165
8.5 Forms (Exercises) 179
9 Testing 181
9.1 Angular testing 182
9.2 Testing (Exercises) 192
10 Animations 193
10.1 Introduction 194
10.2 Transitions and triggers 202
10.3 Complex animation sequences 213
10.4 Reusable Animations 218
10.5 Route transition animations 220
10.6 Animations (Exercises) 227
11 PWA 229

11.1 PWA 230
12 Web Workers 234
12.1 Web workers 235
Vue 237
13 Introduction 238
13.1 What is VueJS 239
13.2 Let's go to start 241
13.3 Application structure 246
13.4 Edit default project 256
14 Simple Application 261
14.1 Greeting - variables and functions 262
14.2 Counter - events 264
14.3 Event object 268
14.4 v-model 272
15 Condition and Loop 277
15.1 v-if 278
15.2 v-for 280
16 Lists 284
16.1 Work with List 285
16.2 Material design 295
16.3 Edit inline 301
16.4 Add new element - other way 304
16.5 Bulk data operations 307

jQuery

jQuery

Chapter 1

1.1 What is jQuery?

🕮 1.1.1

jQuery is a fast and function-rich library of coding language JavaScript. Thanks to a combination of versatility and extensibility jQuery created the change of way how many programmers write their JavaScript.

Everything that can be made in jQuery can be also created in the programming language JavaScript.

Everything that can be made in JavaScript can be created in jQuery, too.

jQuery is designed the way to make your Javascript simpler and shorten the time to develop an application. The motto of JQuery is "write less, do more".

jQuery takes many basic tasks which require many lines of Javascript, inserts them into its' own methods, and you can call them with just one line of the code.

🕮 1.1.2

What can JQuery do?

5 main tasks you can do faster and easier with jQuery :

1. Access to DOM (Document Object Model) elements on the page – access to individual or whole groups of elements
2. Set attributes of DOM for element or group of elements (find something on the page and create something with it)
3. Create, delete, view, hide DOM elements on the page
4. Define events on the page (clicking, moving of a mouse, dynamic styles, animations, dynamic content)
5. Calling AJAX

Probably the best advantage of JQuery is the work with AJAX. Calling and working with AJAX, just like the work with web services is incomparable with JavaScript.

📝 1.1.3

jQuery is:

- a library of JavaScript language
- a library of Java language

- a library of PHP language
- a library for new CSS styles

🕮 1.1.4

As an example of the simplicity of code JQuery, the next source code shows how to find resolutions of the browser window in Javascript:

```
var x,y;
if (self.innerHeight) { // all except Explorer
 x = self.innerWidth;
 y = self.innerHeight;
}
else if (document.documentElement &&
 document.documentElement.clientHeight) {
  // Explorer 6 Strict Mode
 x = document.documentElement.clientWidth;
 y = document.documentElement.clientHeight;
}
else if (document.body) { // other Explorers
 x = document.body.clientWidth;
 y = document.body.clientHeight;
}
```

In JQuery the same problem can be solved with the next two lines of code:

```
var x = $(window).width();
var y = $(window).height();
```

🕮 1.1.5

There are many libraries existing in Javascript, but JQuery is the most popular and flexible.

Just like many libraries, even JQuery is located in an external file/ files, which is necessary to add to the page. If we want to work with JQuery we have two options:

1. To download library JQuery from the page jquery.com
2. To add JQuery to your page with CDN

Our recommendation is the second option. CDN (Content Delivery Network) is a distributed group of servers cooperating with the fast distribution of internet

content. CDN allows a fast transfer of tools needed for loading Internet content, even HTML pages, JavaScript files, styles, pictures and videos.

In our case, the whole library code of JQuery will be inserted from another source located on the internet. The only disadvantage of the method is a permanent internet connection. Library jQuery is necessary with every loading of a web page, furthermore, every time when the web page is loading (even locally), the browser will be connected to CND and download the JQuery library. It is important to realize, every computer does that independently even though the library is on our web server or using CDN (when talking about the "basic" web page)

🕮 1.1.6

For using CDN and inserting JQuery library into our source code, it is important to call the next script into our web pages:

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
```

CDN is often added into the element **<head>**. like in JavaScript, if your pages are larger and you want to improve the speed of loading, you can insert CDN at the end of element **<body>** before ending character **</body>**. JQuery includes the basic method for safe executing of its' functions after the loading of the whole page.

1.1.7

Copy the next CDN loading of a library JQuery and insert it into an HTML code into the element **<head>**.

```
 <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
```

```
<!doctype html>
<html>
  <head><title>First JavaScript</title>_____
  </head>
  <body>
  </body>
</html>
```

1.1.8

CDN is available at ajax.googleapis.com but you can either use another CDN, for example, Microsoft CDN:

```
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.min.js"></script>
```

1.1.9

The whole form of the jQuery syntax is designed with the objective to speed up basic operations with DOM elements. Access to DOM elements is realized by calling:

```
$(selector).action()
```

Character **$** defines access to jQuery, selector is a specification of a name, class, or ID element, to which we want to get access. The action indicates the function, that will be called by the found element.

In the next code, we will show an example of the calling method **hide()** for chosen selectors. The method will hide the chosen element on the page.

```
$(this).hide() //hides actual element
$("img").hide() // hides all pictures, known as elements <img> on the page
$("p").hide() //hides all paragraphs, known as elements <p> on the page
```

1.1.10

Use the correct selector for hiding all elements **<p>** on the webpage.

```
$("_____").hide();
```

1.1.11

Use the correct selector for hiding all elements **<h1>** on the webpage.

```
$("_____").hide();
```

1.1.12

Use the correct selector and JQuery syntax for hiding all elements ** ** on the webpage.

```
_____("_____").hide();
```

1.2 jQuery introduction

1.2.1

From the practical point of view, you can alway create an element **<script>**. Sometimes troubles occur with the slow loading of JQuery. If in any of its function is link to, let's say, picture, which isn't yet loaded , or to not yet loaded button, JQuery won't work correctly in case a function is linked to an unloaded picture or a temporary uncreated button.

To avoid these problems, the so-called programmer of „good habits“ add the whole code into function **$(document).ready**.. It's the function that will execute after the whole page is loaded, what means all the needed elements created in DOM. This function is some kind of a **main()** method of JQuery.

Every basic JQuery code looks as following:

```
<!doctype html>
<html><head>
     <title>First JavaScript</title>
         <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
         <script>
            $(document).ready(function () {
                // Your code goes here
            });
         </script>
  </head>
  <body>
  </body>
</html>
```

1.2.2

jQuery is a library of JavaScript, so all the functions of JavaScript can be called even in jQuery.

Task: Insert call **alert('jQuery works correctly')** into the **$(document).ready**.

```
<!doctype html>
<html>
  <head>
     <title>First JavaScript</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
     <script>
         $(document).ready(function () {
             _____('jQuery works correctly');
         });
     </script>
  </head><body></body>
</html>
```

1.2.3

Essential of every JQuery code is to choose the correct selector and the use of a function, setting features or defining an event for a chosen selector. Selectors allow us to choose and manipulate with HTML elements in JQuery.

For example, if we need access to all the elements in JavaScript ** **, the code would look as follows:

```
document.getElementsByName("div")
```

while in jQuery we can use:

```
$("div")
```

1.2.4

In the next example, we will use the function **hide()**. We already know the method hides a chosen element on the webpage..

Task: Insert the element **<h1>JavaScript</h1>** into an HTML webpage.

```
<!doctype html>
<html><head>
     <title>First JavaScript</title>
     <script>
         $(document).ready(function () {
```

```
        });
    </script>
  </head>
  <body>

    _____
  </body>
</html>
```

1.2.5

Insert second element **<h2>AngularJS</h2>** under element **<h1>JavaScript</h1>**

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
      <script>
           $(document).ready(function () {

           });
    </script>
  </head>
  <body>
       <h1>JavaScript</h1>

       _____
      <h2>JSON</h2>
  </body>
</html>
```

1.2.6

Insert the link for CDN

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

into the element **<head>** for loading thejQuery library.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>_____
     <script>
```

```
            $(document).ready(function () {

            });
    </script>
  </head>
  <body>
       <h1>JavaScript</h1>
              <h2>AngularJS</h2>
              <h2>JSON</h2>
  </body>
</html>
```

1.2.7

Use the correct selector for hiding all the second level titles, such as elements **<h2>** on the webpage.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
<script>
            $(document).ready(function () {
                      $("_____").hide();
            });
    </script>
  </head>
  <body>
       <h1>JavaScript</h1>
              <h2>AngularJS</h2>
              <h2>JSON</h2>
  </body>
</html>
```

1.2.8

Add a correct calling for hiding all the first level titles, such as element **<h1>**.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
<script>
         $(document).ready(function () {
            $("h2").hide();
            $("h1")._____;
         });
    </script>
  </head>
  <body>
       <h1>JavaScript</h1>
       <h2>AngularJS</h2>
       <h2>JSON</h2>
  </body>
</html>
```

1.2.9

Which text will be shown on the webpage after executing the script??

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
                $("span").hide();
                $("p").hide();
          });
     </script>
   </head>
   <body>
        <span>If you please--draw me a sheep!</span>
        <p>What!</p>
        <div>Draw me a sheep!</div>
        <p>I jumped to my feet,completely thunderstruck.</p>
```

```
   </body>
</html>
```

- Draw me a sheep!
- If you please--draw me a sheep!
- I jumped to my feet, completely thunderstruck.
- If you please--draw me a sheep! I jumped to my feet, completely thunderstruck.

1.2.10

Add another call into the script of a function **hide()** so the JQuery hides even remaining element (** **) on the webpage (all the elements will be hidden):

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
<script>
          $(document).ready(function () {
             $("span").hide();
             $("p").hide();
             _____
          });
    </script>
  </head>
  <body>
       <span>If you please--draw me a sheep!</span>
       <p>What!</p>
       <div>Draw me a sheep!</div>
       <p> I jumped to my feet, completely thunderstruck.</p>
  </body>
</html>
```

1.3 Events

🕮 1.3.1

The programmer usually prefers the scripting language for HTML in in case he expects some activity from the visitors of his website. Using jQuery only for setting attributes of elements on the page is wasting of JQuery's potential. Attributes of elements can be set directly in HTML.

The purpose of jQuery, like in other scripting languages for HTML, is programming events and subsequently dynamical editing of pages. All actions from different visitors of our pages, to which the web page can dynamically reply , are called events. Event is a moment when something happens. For example moving the mouse, clicking, pressing the button on a keyboard and also receiving an answer on the required web service, etc.

Most events in HTML elements have their method in jQuery.

🕮 1.3.2

Probably the most used event in the environment of webpages is the event of clicking the mouse in HTML element. If we have a button on the webpage, it means we have an element **<button>** in HTML code, for example:

```
<button type="button">Click Me!</button>
```

Event of clicking the mouse on the button is defined in jQuery as follows:

```
$("button").click();
```

We used the name of the element as a selector, so the event is defined for all buttons which we add to the webpage. In an event **click()** we must define, what should be executed if the button is pressed.

It is typical in jQuery, the event of function is created directly:

```
$("button").click(function(){
    // action goes here!!
});
```

1.3.3

Which event is used for defining the click action with the mouse on an object?

- click()
- mouseclick()
- clickmouse()
- clicked()
- clicking()

1.3.4

Task: Define an event the mouse on a button in the script. .

It is important to mention that even events are called after loading all elements on the webpage. Every event is added into the function **$(document).ready**.

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $("button")._____(function(){
                    // action goes here!!
               });
          });
     </script>
   </head>
   <body>
        <button type="button">Click Me!</button>
   </body>
</html>
```

1.3.5

Task: Add a function for the printing of a user message into the event **click()** . Text of the message will be "You clicked on a button!", then insert the message with the help of the function **alert()**. Don't forget the semicolon at the end of the line.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
```

```
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
<script>
       $(document).ready(function () {
         $("button").click(function(){
            _____
         });
       });
    </script>
  </head>
  <body>
       <button type="button">Click Me!</button>
  </body>
```

1.3.6

Alike the pressing the button, the event **click()** is able to define for all the elements on the HTML page.

Task: Create two sections, i.e. insert the text with the beginning of „This asteroid has only“ until „Grown-ups are like that . . .“ into the element for a section **<p>**. Similiarly, create the second paragraph **<p>** for the text with the beginning of „Fortunately, however“ and ending with „accepted his report.“.

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {

          });
     </script>
   </head>
   <body>                <button>Show paragraph</button>
_____This asteroid has only once been seen through the
telescope. That was by a Turkish astronomer, in 1909.
On making his discovery, the astronomer had presented it to
the International Astronomical Congress, in a great
```

```
demonstration. But he was in Turkish costume, and so nobody
would believe what he said.
Grown-ups are like that . . ._____

_____Fortunately, however, for the reputation of Asteroid B-
612, a Turkish dictator made a law that his subjects, under
pain of death, should change to European costume. So in 1920
the astronomer gave his demonstration all over again, dressed
with impressive style and elegance. And this time everybody
accepted his report._____
   </body>
</html>
```

1.3.7

Task: In jQuery create an event **click** for the element **<button>** and also for the element **<p>**.

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $("button")._____(function(){

               });

               $("_____").click(function(){

               });
          });
     </script>
   </head>
   <body>                <button>Show paragraph</button>
<p>This asteroid has only once been seen through the
telescope. That was by a Turkish astronomer, in 1909.
On making his discovery, the astronomer had presented it to
the International Astronomical Congress, in a great
demonstration. But he was in Turkish costume, and so nobody
would believe what he said.
```

```
Grown-ups are like that . . .</p>

<p>Fortunately, however, for the reputation of Asteroid B-612,
a Turkish dictator made a law that his subjects, under pain of
death, should change to European costume. So in 1920 the
astronomer gave his demonstration all over again, dressed with
impressive style and elegance. And this time everybody
accepted his report.</p>
   </body>
</html>
```

1.3.8

Task: Add an effect of hiding paragraphs with the event **click()** for the element**<p>** . Use the function **.hide()**.

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $("button").click(function(){

               });

               $("p").click(function(){
                   $("p")._____;
               });
          });
     </script>
   </head>
   <body>                <button>Show paragraph</button>
<p>This asteroid has only once been seen through the
telescope. That was by a Turkish astronomer, in 1909.
On making his discovery, the astronomer had presented it to
the International Astronomical Congress, in a great
demonstration. But he was in Turkish costume, and so nobody
would believe what he said.
Grown-ups are like that . . .</p>
```

```
<p>Fortunately, however, for the reputation of Asteroid B-612,
a Turkish dictator made a law that his subjects, under pain of
death, should change to European costume. So in 1920 the
astronomer gave his demonstration all over again, dressed with
impressive style and elegance. And this time everybody
accepted his report.</p>
   </body>
</html>
```

1.3.9

The opposite of the function **.hide()** is the function **.show()**. We can show hidden elements with the use of it. .

Task: Add the effect of showing paragraphs after the button click. Use the function **.show()**.

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $("button").click(function(){
                   $("p")._____;
               });

               $("p").click(function(){
                   $("p").hide();
               });
          });
     </script>
   </head>
   <body>                <button>Show paragraph</button>
<p>This asteroid has only once been seen through the
telescope. That was by a Turkish astronomer, in 1909.
On making his discovery, the astronomer had presented it to
the International Astronomical Congress, in a great
demonstration. But he was in Turkish costume, and so nobody
would believe what he said.
Grown-ups are like that . . .</p>
```

```
<p>Fortunately, however, for the reputation of Asteroid B-612,
a Turkish dictator made a law that his subjects, under pain of
death, should change to European costume. So in 1920 the
astronomer gave his demonstration all over again, dressed with
impressive style and elegance. And this time everybody
accepted his report.</p>
   </body>
</html>
```

1.4 More Selectors

🕮 1.4.1

The use of the correct selector is one of the most important parts in jQuery.JQuery uses CSS syntax for choosing elements with a selector. In case of CSS and also jQuery, there are differences between id selector and a class selector.

Id selector uses a specific attribute **id** for finding the correct element. It is important to mention, that the value of attribute **id** should be unique. . Id selector in jQuery is defined with a character # before the name of the selector

For Example:

```
<!doctype html>
<html>
  <head>
     <title>First JavaScript</title>
         <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
         <script>
            $(document).ready(function () {
                $("#prvy").hide();
            });
         </script>
  </head>
  <body>
        <p id="prvy">JavaScript</p>
        <p>AngularJS</p>
        <p>JSON</p>
  </body>
</html>
```

1.4.2

Task: Edit the script to hide**id** the paragraph with id attribute "**first**".

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $("_____").hide();
          });
     </script>
   </head>
   <body>
      <p id="prvy"> JavaScript </p>
      <p>AngularJS</p>
      <p>JSON</p>
   </body>
</html>
```

1.4.3

Which text will be shown on the webpage after executing the script?

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $("#druhy").hide();
               $("#treti").hide();
          });
     </script>
   </head>
   <body>
      <p id="prvy">JavaScript</p>
```

```
      <p id="druhy">AngularJS</p>
      <p id="treti">JSON</p>
   </body>
</html>
```

- JavaScript
- AngularJS
- JSON

1.4.4

Another often used selector is a class selector. Multiple elements of HTML can be added into the same class with an attribute **class**. Furthermore, these elements can use the same defined function. Selector of class (class selector) is in jQuery defined with a character **.** (dot) before the name of selector.

Example:

```
<!doctype html>
<html>
  <head>
     <title>First JavaScript</title>
         <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
         <script>
            $(document).ready(function () {
                $(".data").hide();
            });
         </script>
  </head>
  <body>
     <p class="app"> JavaScript </p>
     <p class="data">XML</p>
     <p class="data">JSON</p>
  </body>
</html>
```

1.4.5

Task: Edit the script to show the paragraphs with the **class** attribute **“prince“**.

```
<!doctype html>
```

```
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
         $(document).ready(function () {
             $("_____").hide();
         });
    </script>
  </head>
  <body>
       <p class="prince">If you please--draw me a sheep!</p>
       <p class="antoine">What!</p>
       <p class="prince">Draw me a sheep!</p>
       <p> I jumped to my feet, completely thunderstruck.</p>
  </body>
</html>
```

1.4.6

Which text will be shown on the webpage after executing the script?

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $(".druhy").hide();
          });
     </script>
   </head>
   <body>
      <p id="prvy">JavaScript</p>
      <p class="druhy">AngularJS</p>
      <p class="druhy">JSON</p>
   </body>
</html>
```

- JavaScript
- AngularJS
- JSON

1.4.7

Thing about both **id** selectors and **class** selectors. Which text will be shown on the webpage after executing the script?

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $(".simple").hide();
               $("#favorite").hide();
          });
     </script>
   </head>
   <body>
      <p class="simple">HTML</p>
      <p id="favorite"> JavaScript </p>
      <p class="simple">JSON</p>
      <p id="for_data">XML</p>
   </body>
</html>
```

- XML
- JavaScript
- JSON
- HTML

1.4.8

We can use a lot more logs for writing selectors in jQuery jQuery besides **id** and **class** because jQuery uses CSS syntax for selectors.

For example:

```
$("p.my_class")
// Selector for selecting all elements <p> , that have class
class="my_class"

$("p:first")
// Selector for selecting first element <p> , in v case of
page having multiple elements <p>, selector selects first
element.

$("ul li:first")
// Selecting first <li> element, that is inserted into element
<ul>
```

More options and the showcase of selectors can be found on the webpage: https://learn.jquery.com/using-jquery-core/selecting-elements/

1.4.9

When calling events of selectors with the change of html elements, they behave similarly like using **id** and **class** selector. For example **id** selector is one of the most used selectors when operating with buttons.

Task: Add to the script a correct code for calling the method click() for the button. **Define an event with idselector.**

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $("_____n").click(function(){
                    // action goes here!!
               });
          });
     </script>
   </head>
   <body>
      <button id="my_button">Click Me!</button>
   </body>
</html>
```

1.4.10

Task: HTML code consists of two buttons with different atributes of **id.** Insert selectors into code to print messages in a function **alert()** correctly.

```
<!doctype html>
<html>
   <head>
      <title>JavaScript Example</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
      <script>
          $(document).ready(function () {
               $("_____").click(function(){
                    alert("You clicked on B button");
               });

               $("{_____").click(function(){
                    alert("You clicked on A button");
               });
          });
     </script>
   </head>
   <body>
      <button id="buttonA">ButtonA</button>
      <button id="buttonB">ButtonB</button>
    </body>
</html>
```

Playground

Chapter 2

2.1 Hide vs. show

📖 2.1.1

In the previous examples, we have often used the function **.hide()**, which hid the selected element on the page. If we would see more in the HTML page after executing the function **.hide()**, we would find out that it adds the style "**display: none;**" to the HTML element of the page.

The opposite of the function **.hide()** is **.show()**. Obviously, this function can show hidden elements.

Note: You can also display elements that were not only hidden by **.hide()** but they had style set to **style="display: none;"**

📝 2.1.2

Which function is used to display hidden elements on a web page?

- .show()
- .hide()
- .visible()
- .display()
- .reveal()

📝 2.1.3

What text will be displayed on the web page after executing the script?

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
        $(document).ready(function () {
           $("p").hide();
           $(".prince").show();
            $("#first_request").hide();
        });
```

```
    </script>
  </head>
  <body>
       <p class="prince" id="first_request">If you please--
draw me a sheep!</p>
       <p>What!</p>
       <p class="prince" id="second_request">Draw me a
sheep!</p>
       <p> I jumped to my feet, completely thunderstruck.</p>
  </body>
</html>
```

- Draw me a sheep!
- If you please--draw me a sheep!
- What!
- I jumped to my feet, completely thunderstruck.

2.1.4

Add the correct selectors to your code: when you click the button, the paragraph containing the text will be hidden. Use the selector **id** to define the event for the button.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
         $(document).ready(function () {
             $("_____").click(function(){
             $("_____").hide();
             });
         });
    </script>
  </head>
  <body>

     <p> If I have told you these details about the asteroid,
and made a note of its number for you, it is on account of the
grown-ups and their ways. When you tell them that you have
```

```
made a new friend, they never ask you any questions about
essential matters. They never say to you, "What does his voice
sound like? What games does he love best? Does he collect
butterflies?" Instead, they demand: "How old is he? How many
brothers has he? How much does he weigh? How much money does
his father make?"</p>
      <button id="buttonHide">Hide paragraph</button>

  </body>
</html>
```

2.1.5

Insert a new button to the code, **id** of the button will be "**buttonShow**" and in the description (text on the button) will be "**Show paragraph**"

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
         $(document).ready(function () {
             $("#buttonHide").click(function(){
             $("p").hide();
             });
         });
    </script>
  </head>
  <body>

     <p> If I have told you these details about the asteroid,
and made a note of its number for you, it is on account of the
grown-ups and their ways. When you tell them that you have
made a new friend, they never ask you any questions about
essential matters. They never say to you, "What does his voice
sound like? What games does he love best? Does he collect
butterflies?" Instead, they demand: "How old is he? How many
brothers has he? How much does he weigh? How much money does
his father make?"</p>
      <button id="buttonHide">Hide paragraph</button>
 <button id="_____">_____</button>
```

```
  </body>
</html>
```

2.1.6

Insert the event **.click()** for the new button, using the new button id as the selector. In the event, use the function **show()** for showing a hidden paragraph.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
         $(document).ready(function () {
             $("#buttonHide").click(function(){
             $("p").hide();
             });

         $("_____").click(function(){
             $("p")._____;
             });
         });
    </script>
  </head>
  <body>

     <p> If I have told you these details about the asteroid,
and made a note of its number for you, it is on account of the
grown-ups and their ways. When you tell them that you have
made a new friend, they never ask you any questions about
essential matters. They never say to you, "What does his voice
sound like? What games does he love best? Does he collect
butterflies?" Instead, they demand: "How old is he? How many
brothers has he? How much does he weigh? How much money does
his father make?"</p>
      <button id="buttonHide">Hide paragraph</button>
 <button id="buttonShow">Show paragraph</button>

  </body>
</html>
```

2.2 Classes

📖 2.2.1

jQuery has several options for working with CSS, i.e. setting, changing, removing CSS styles on a web page. The basic method for working with styles is the method **css()**. Using this, we can:

- Read the currently set styles for the attribute on the webpage
- Set new styles for the attribute on the webpage.

The**.css()** method can be used with two syntax how to write method with parameters. The basic syntax is

```
css("property_name","value");
```

E.g. to set the paragraph text color, the jQuery script might look like this:

```
$(document).ready(function () {
   $("p").css("color","red");
});
```

This syntax is useful for simple **css()**methods of application. We can use it to set only one css property.

📝 2.2.2

Which method is used to dynamically adjust the css properties of the selected attribute?

- .css()
- .style()
- .casscadin()
- .set_css()
- .set_style()

📝 2.2.3

By means of the jQuery set blue background color for the paragraph, i. element **<p>**. To set it up, use the jQuery**css()** method. Use the css property **background-color**, use **blue** to adjust the blue color.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
        $(document).ready(function () {
             $("p").css("_____", "_____");
        });
    </script>
  </head>
  <body>
       <p> If I have told you these details about the
asteroid, and made a note of its number for you, it is on
account of the grown-ups and their ways. When you tell them
that you have made a new friend, they never ask you any
questions about essential matters. They never say to you,
"What does his voice sound like? What games does he love best?
Does he collect butterflies?" Instead, they demand: "How old
is he? How many brothers has he? How much does he weigh? How
much money does his father make?"</p>
   </body>
</html>
```

2.2.4

It is obvious that the **.css()** method usange in the main jQuery method has no meaningful significance. Quicker and easier is to set styles directly via css on the web page. The most important meaning of the **.css()** method is in the dynamic style change.

Task:

We created a paragraph and two buttons in the source code. We have also defined the **.click()** event definition for both buttons. Use the **.css()** to set the paragraph font color when you click the button with **id="button_orange"**. Set the font color using the .css() property **"color"** with value **"orange"**. Keep in mind that the css property and its value are actually text strings, so they must be included in the quotation marks in the css method.

```
<!doctype html>
<html>
  <head>
```

```
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
        $(document).ready(function () {
        $("#button_orange").click(function(){
                  $("p").css("_____", "_____");
              });

   $("#button_pink").click(function(){

 });
            });
    </script>
  </head>
  <body>
       <button id="button_orange">Set color to orange</button>
       <button id="button_pink">Set color to pink</button>
       <p> If I have told you these details about the
asteroid, and made a note of its number for you, it is on
account of the grown-ups and their ways. When you tell them
that you have made a new friend, they never ask you any
questions about essential matters. They never say to you,
"What does his voice sound like? What games does he love best?
Does he collect butterflies?" Instead, they demand: "How old
is he? How many brothers has he? How much does he weigh? How
much money does his father make?"</p>
   </body>
</html>
```

2.2.5

Task:

Use the ..**css()** method to set the paragraph font color after you click on the second button with **id=“button_pink“**. Set the font color with css property **“color“**value**“pink“**.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
```

```
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
    $(document).ready(function () {
       $("#button_orange").click(function(){
           $("p").css("color", "orange");
             });

       $("#button_pink").click(function(){
          _____
       });
    });
   </script>
  </head>
  <body>
      <button id="button_orange">Set color to orange</button>
      <button id="button_pink">Set color to pink</button>
      <p> If I have told you these details about the
asteroid, and made a note of its number for you, it is on
account of the grown-ups and their ways. When you tell them
that you have made a new friend, they never ask you any
questions about essential matters. They never say to you,
"What does his voice sound like? What games does he love best?
Does he collect butterflies?" Instead, they demand: "How old
is he? How many brothers has he? How much does he weigh? How
much money does his father make?"</p>
   </body>
</html>
```

2.2.6

Task:

Create a button that change the position (move to left), when you click on it. The button changes its horizontal position to the center. Use the left property to change the position of the button. The property value "**50%**"ensures the centering of the horizontal position of the button.

Note: For the "jump" option through the page, we set the position to fixed for the button using the css style.

```
<!doctype html>
<html>
```

```
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
           $(document).ready(function () {
        $("#button_jump").click(function(){
                     $("#button_jump").css("_____", "_____");
                });
            });
    </script>
  </head>
  <body>
       <button id="button_jump">Jump button</button>
   </body>
</html>
```

2.2.7

The **.css** method can be also used to read the set style. In this case, the method has only one paramete, the style. The output is the specified values of the style in the parameter.

In the following illustration, when you click on paragraph, the text color of the paragraph is displayed and it value is displayed by means of the **alert()**function.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
     $(document).ready(function () {
          $("p").click(function(){

                  alert($("p").css("color"));

            });
         });
    </script>
  </head>
```

```
  <body>
       <p> I jumped to my feet, completely thunderstruck.</p>
   </body>
</html>
```

2.2.8

In the previous examples, we used the **.css()** method to set the properties of the html elements. Let´s imagine a situation, where we need to set multiple css properties at the same time. For example, after clicking on a paragraph, we would like this to completely reformatted by changing the font style, font color and font size and so on. Of course the correct solution could be to change every new property in a separated call of the **.css()** method.

Sometimes, it is a faster and more "clever" procedure to set the required properties beforehand by means of the css style, setting the class as a selector. jQuery provides the **addClass()** method. Use this class to assign a selected element to a class that we used previously as a selector to set multiple css styles at the same time. This way we apply multiple ccs styles to the element.

2.2.9

Create a webpage, that will include text with paragraph and a button with label "Change style". After the clicking on the button assign a class to the paragraph, to which is multiple css styles created.

In the example, we prepared css styles and a button event.

Task:

Assign a class to the style that you created. Call the class "**newstyle**". Note that CSS (as well in jQuery) uses a dot before the class name to indicate the class.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
     <style type="text/css">
       _____{
           font-size:20px;
           color:orange;
```

```
            background-color:yellow;
        }
    </style>
    <script>
        $(document).ready(function () {

        });
    </script>
  </head>
  <body>
        <p> I jumped to my feet, completely thunderstruck.</p>
      <button>Zmena štýlu</button>
   </body>
</html>
```

2.2.10

Task: Create a button click event, use the **button** selector.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
     <style type="text/css">
       .newstyle{
          font-size:20px;
          color:orange;
          background-color:yellow;
       }
    </style>
    <script>
       $(document).ready(function () {
          $("_____")._____(function(){


          });
       });
    </script>
  </head>
  <body>
       <p> I jumped to my feet, completely thunderstruck.</p>
```

```
      <button>Change style</button>
   </body>
</html>
```

2.2.11

Task: Supplement a function that add a "**newstyle**" to the **p** selector. This ensures that the "**newstyle**" class is assigned to the paragraph. There are multiple css styles for the given class, which are applied to the paragraph format at the same time.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
     <style type="text/css">
      .newstyle{
       font-size:20px;
       color:orange;
       background-color:yellow;
      }
    </style>
    <script>
      $(document).ready(function () {
          $("button").click(function(){

               $("p")._____;
          });
      });
    </script>
  </head>
  <body>
       <p> I jumped to my feet, completely thunderstruck.</p>
      <button>Change Style</button>
   </body>
</html>
```

2.2.12

The counterpart of the **.addClass()** method is the **.removeClass()**, which removes the class from the selected elements.

Task:

In our example we added new styles. Add for the newly created styles the paragraph selector. For that purpose use selector **p**.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
     <style type="text/css">
       .newstyle{
         font-size:20px;
         color:orange;
         background-color:yellow;
        }

       _____{
         font-size:14px;
         color:black;
         background-color:white;
       }
    </style>
    <script>
       $(document).ready(function () {
            $("button").click(function(){
                                          $("p").addClass("new
style");
            });
        });
    </script>
  </head>
  <body>
       <p> I jumped to my feet, completely thunderstruck.</p>
      <button>Change style</button>
   </body>
</html>
```

2.2.13

Task: We created a mouse click event on a paragraph. Add the **.removeClass()** method with the correct parameter (specifying the name of the style being removed from the paragraph) to remove the style from the paragraph.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
     <style type="text/css">
       .newstyle{
         font-size:20px;
         color:orange;
         background-color:yellow;
       }

       p{
         font-size:14px;
         color:black;
         background-color:white;
       }
     </style>
     <script>
        $(document).ready(function () {
            $("button").click(function(){
                                          $("p").addClass("new
style");
            });

            $("p").click(function(){


            });

        });
      </script>
  </head>
  <body>
       <p> I jumped to my feet, completely thunderstruck.</p>
      <button>Change style</button>
   </body>
```

```
</html>
```

2.2.14

By means of the jQuery set the blue background color for the paragraph, i. element **<p>** To set it, use the jQuery **css()** method. Use the css property **background-color**, use blue parameter to adjust the **blue** color.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
     <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery
.min.js"></script>
 <script>
        $(document).ready(function () {
             $("p").css("_____", "_____");
        });
    </script>
  </head>
  <body>
       <p> If I have told you these details about the
asteroid, and made a note of its number for you, it is on
account of the grown-ups and their ways. When you tell them
that you have made a new friend, they never ask you any
questions about essential matters. They never say to you,
"What does his voice sound like? What games does he love best?
Does he collect butterflies?" Instead, they demand: "How old
is he? How many brothers has he? How much does he weigh? How
much money does his father make?"</p>
   </body>
</html>
```

Angular

Introduction

Chapter 3

3.1 What is Angular?

🕮 3.1.1

Angular is a platform and framework for building single-page client applications using HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

🕮 3.1.2

With basic building blocks framework Angular components which are Angular organized in NgModules. They are collecting NgModules connected code into functional sets; the Angular application is being defined by the NgModules set. The application always has at least main module which the bootstrap enables, and usually has modules far more of function.

📝 3.1.3

What language is Angular written in?

3.2 TypeScript

🕮 3.2.1

TypeScript is an open-source language which builds on JavaScript, one of the world’s most used tools, by adding static type definitions.

🕮 3.2.2

Types provide a way to describe the shape of an object, providing better documentation, and allowing TypeScript to validate that your code is working correctly.

📖 3.2.3

Writing types can be optional in TypeScript, because type inference allows you to get a lot of power without writing additional code.

📝 3.2.4

What type is TypeScript?

3.3 Components

📖 3.3.1

Components are the building blocks that compose an application. A component includes a TypeScript class with a @Component() decorator, an HTML template, and styles. The @Component() decorator specifies the following Angular-specific information:

- A CSS selector that defines how the component is used in a template. HTML elements in your template that match this selector become instances of the component.
- An HTML template that instructs Angular how to render the component.
- An optional set of CSS styles that define the appearance of the template's HTML elements.

📖 3.3.2

Angular's component model offers strong encapsulation and an intuitive application structure. Components also make your application easier to unit test and can improve the overall readability of your code.

📝 3.3.3

What is a component?

3.4 Templates

🕮 3.4.1

Each Angular template in your app is a section of HTML that you can include as a part of the page that the browser displays. An Angular HTML template renders a view, or user interface, in the browser, just like regular HTML, but with a lot more functionality.

🕮 3.4.2

When you generate an Angular app with the Angular CLI, the app.component.html file is the default template containing placeholder HTML.

📝 3.4.3

In which programming language are the templates written?

3.5 Dependency injection

🕮 3.5.1

Dependency injection allows you to declare the dependencies of your TypeScript classes without taking care of their instantiation. Instead, Angular handles the instantiation for you.

🕮 3.5.2

This design pattern allows you to write more testable and flexible code.

📝 3.5.3

What Angular does for us by injecting dependencies?

3.6 Angular CLI

🕮 3.6.1

Angular CLI is the fastest, easiest, and recommended way to develop Angular applications.

- ng build Compiles an Angular app into an output directory.
- ng serve Builds and serves your application, rebuilding on file changes.
- ng generate Generates or modifies files based on a schematic.
- ng test Runs unit tests on a given project.
- ng e2e Builds and serves an Angular application, then runs end-to-end tests.

3.6.2

Builds and serves an Angular application, then runs end-to-end tests.

- ng test
- ng e2e
- ng test

3.7 Installation

🕮 3.7.1

Before the startup from Angularem one should install Node.js along with the manager of npm packages. In order to check whether we have him installed on one's computer one should write down into the terminal:

node --version

If a mistake will appear one should download the installer from the side:

https://nodejs.org/en/download/.

🕮 3.7.2

After node.js installing, with order "npm install - g typscript" we are installing TypeScript, and at the end with order "npm install - g angular/cli" we are proceeding to the Angular installation is imposing a customs duty. Angular is imposing a

customs duty is a tool of the interface of the command line which lets us for full managing the Angular application.

3.7.3

How to check if we have a Node installed?

3.8 Application structure

3.8.1

Introduction to directory structure:

e2e It contains the code related to automated testing purpose.

node_modules It saves all the dev dependencies (used only at development time) and dependencies (used for development as well as needed in production time), any new dependency when added to project it is automatically saved to this folder.

src This directory contains all of our work related to project i.e. creating components, creating services, adding CSS to the respective page, etc.

package.json This file stores the information about the libraries added and used in the project with their specified version installed. Whenever a new library is added to the project it's name and version is added to the dependencies in package.json.

3.8.2

Inside src folder:

index.html This is the entry point for the application, **app-root** tag is the entry point of the application on this single page application, on this page angular will add or remove the content from the DOM or will add new content to the DOM.
Base **href="/"** is important for routing purposes.

style.scss This file is the global stylesheet you can add that CSS classes or selectors which are common to many components, for example, you can import custom fonts, import bootstrap.css, etc.

assets It contains the js images, fonts, icons and many other files for your project.

🕮 3.8.3

Inside app folder:

app.module.ts Defines the root module, named AppModule, that tells Angular how to assemble the application. Initially declares only the AppComponent. As you add more components to the app, they must be declared here.

app.component.html Defines the HTML template associated with the root AppComponent.

app.component.spec.ts Defines a unit test for the root AppComponent.

app.component.ts Defines the logic for the app's root component, named AppComponent. The view associated with this root component becomes the root of the view hierarchy as you add components and services to your application.

3.9 Naming

🕮 3.9.1

All artifacts in your project; folders, files, classes, etc. should be named to convey meaning. The names should give an indication of what the artifact does.

Use names that express your intentions

Use searchable names

Use nouns for classes, folders, and filenames

Use verbs or verb expressions for methods or functions

Avoid abbreviations and notations as they can be confusing

In addition to the general naming guidelines discussed above, Angular mainly uses three case styles for naming artifacts. camelCase, PascalCase, and kebab-case. It is important to know when and where to use each of these case styles.

🕮 3.9.2

Kebab-case is a naming style where all letters in the name are lowercase and uses a hyphen to separate words in the name. Additionally, Angular uses a period to separate the name, type, and extension of filenames.

Including type in file names makes it easier to find a specific file type with a text editor or IDE. In addition, they ensure pattern matching to automated tasks.

Kebab-case is used to name folders, component selectors, files, and the Angular application itself. Common files in an Angular project include component files, service files, template files, module files, etc.

🕮 3.9.3

PascalCase is a style where all the first letters of the words in the name are capitalized or capitalized.

The Pascal case is mainly used for class naming in the Angular project.

e.g. Export class DogsListCompoent {}

🕮 3.9.4

The camelCase naming style is somewhat similar to the PascalCase style, except that the first letter of the name should always be lowercase. All other words in the name will have their first letter capitalized.

Note: CamelCase and kebab-case for single word names will be similar.

CamelCase is used to name methods or functions, properties, fields, directive selectors, and pipe selectors.

3.10 First-party libraries

🕮 3.10.1

Angular provides many proprietary libraries to extend the functionality of the website. You can also create your own libraries or use others.

🕮 3.10.2

Sample libraries:

- Angular Forms Uniform system for form participation and validation.

- Angular HttpClient Robust HTTP client that can power more advanced client-server communication.
- Angular Animations Rich system for driving animations based on application state.
- Angular Schematics Automated scaffolding, refactoring, and update tools that simplify development at large scale.

3.10.3

Uniform system for form participation and validation.

- Angular Forms
- Angular Animations
- Angular Schematics

3.11 Introduction (Exercises)

3.11.1

What language is Angular written in?

3.11.2

What type is TypeScript?

3.11.3

What is a component?

3.11.4

In which programming language are the templates written?

3.11.5

What Angular does for us by injecting dependencies?

3.11.6

Builds and serves an Angular application, then runs end-to-end tests.

3.11.7

How to check if we have a Node installed?

Components

Chapter 4

4.1 Components

📖 4.1.1

A component must belong to an NgModule in order for it to be available to another component or application. To make it a member of an NgModule, list it in the declarations field of the NgModule metadata.

Note that, in addition to these options for configuring a directive, you can control a component's runtime behavior by implementing life-cycle hooks.

📝 4.1.2

What must a component belong to to be available to another component or application?

📖 4.1.3

ChangeDetection

The change-detection strategy to use for this component.

```
changeDetection?: ChangeDetectionStrategy
```

When a component is instantiated, Angular creates a change detector, which is responsible for propagating the component's bindings. The strategy is one of:

ChangeDetectionStrategy#OnPush sets the strategy to CheckOnce (on demand).

ChangeDetectionStrategy#Default sets the strategy to CheckAlways.

📖 4.1.4

viewProviders

Defines the set of injectable objects that are visible to its view DOM children.

```
viewProviders?: Provider[]
```

🕮 4.1.5

Property binding

Property binding in Angular helps you set values for properties of HTML elements or directives. With property binding, you can do things such as toggle button functionality, set paths programmatically, and share values between components.

Property binding moves a value in one direction, from a component's property into a target element property.

To bind to an element's property, enclose it in square brackets, [], which identifies the property as a target property. A target property is the DOM property to which you want to assign a value. For example, the target property in the following code is the image element's src property.

```
<img [src]="itemImageUrl">
```

In this example, src is the name of the <img> element property.

The brackets, [], cause Angular to evaluate the right-hand side of the assignment as a dynamic expression. Without the brackets, Angular treats the right-hand side as a string literal and sets the property to that static value.

```
<app-item-detail childItem="parentItem"></app-item-detail>
```

Omitting the brackets renders the string parentItem, not the value of parentItem.

🕮 4.1.6

Event binding

Event binding allows you to listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.

To bind to an event you use the Angular event binding syntax. This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right. In the following example, the target event name is click and the template statement is onSave().

```
<button (click)="onSave()">Save</button>
```

The event binding listens for the button's click events and calls the component's onSave() method whenever a click occurs.

4.1.7

Event binding allows you to listen for and respond to user actions such as:

4.1.8

NgFor

A structural directive that renders a template for each item in a collection. The directive is placed on an element, which becomes the parent of the cloned templates.

The ngForOf directive is generally used in the shorthand form *ngFor. In this form, the template to be rendered for each iteration is the content of an anchor element containing the directive.

Angular automatically expands the shorthand syntax as it compiles the template. The context for each embedded view is logically merged to the current component context according to its lexical position.

4.1.9

Template variables

Template variables help you use data from one part of a template in another part of the template. With template variables, you can perform tasks such as respond to user input or finely tune your application's forms.

In the template, you use the hash symbol, #, to declare a template variable. The following template variable, #phone, declares a phone variable on an <input> element.

```
<input #phone placeholder="phone number" />
```

You can refer to a template variable anywhere in the component's template. Here, a <button> further down the template refers to the phone variable.

```
<input #phone placeholder="phone number" />
<!-- lots of other elements -->
<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

4.2 Overview

🕮 4.2.1

Prerequisites - To create a component, verify that you have met the following prerequisites:

Install the Angular CLI.

Create an Angular workspace with initial application. If you don't have a project, you can create one using ng new <project-name>, where <project-name> is the name of your Angular application.

🕮 4.2.2

Creating a component using the Angular CLI

To create a component using the Angular CLI:

1. From a terminal window, navigate to the directory containing your application.
2. Run the ng generate component <component-name> command, where <component-name> is the name of your new component.

By default, this command creates the following:

- A folder named after the component
- A component file, <component-name>.component.ts
- A template file, <component-name>.component.html
- A CSS file, <component-name>.component.css
- A testing specification file, <component-name>.component.spec.ts

Where <component-name> is the name of your component.

🕮 4.2.3

Creating a component manually:

Although the Angular CLI is the easiest way to create an Angular component, you can also create a component manually. This section describes how to create the core component file within an existing Angular project.

To create a new component manually:

1. Navigate to your Angular project directory.
2. Create a new file, <component-name>.component.ts.
3. At the top of the file, add the following import statement.
4. After the import statement, add a @Component decorator.
5. Choose a CSS selector for the component.
6. Define the HTML template that the component uses to display information. In most cases, this template is a separate HTML file.
7. Select the styles for the component's template. In most cases, you define the styles for your component's template in a separate file.
8. Add a class statement that includes the code for the component.

🕮 4.2.4

Specifying a component's CSS selector

Every component requires a CSS selector. A selector instructs Angular to instantiate this component wherever it finds the corresponding tag in template HTML. For example, consider a component hello-world.component.ts that defines its selector as app-hello-world. This selector instructs Angular to instantiate this component any time the tag <app-hello-world> appears in a template.

Specify a component's selector by adding a selector statement to the @Component decorator.

🕮 4.2.5

Defining a component's template

A template is a block of HTML that tells Angular how to render the component in your application. You can define a template for your component in one of two ways: by referencing an external file, or directly within the component.

To define a template as an external file, add a templateUrl property to the @Component decorator.

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

To define a template within the component, add a template property to the @Component decorator that contains the HTML you want to use.

```
@Component({
```

```
 selector: 'app-component-overview',
 template: '<h1>Hello World!</h1>',
})
```

If you want your template to span multiple lines, you can use backticks (`). For example:

```
@Component({
 selector: 'app-component-overview',
 template: `
  <h1>Hello World!</h1>
  <p>This template definition spans multiple lines.</p>
 `
})
```

🕮 4.2.6

Declaring a component's styles

You can declare component styles uses for its template in one of two ways: by referencing an external file, or directly within the component.

To declare the styles for a component in a separate file, add a styleUrls property to the @Component decorator.

```
@Component({
 selector: 'app-component-overview',
 templateUrl: './component-overview.component.html',
 styleUrls: ['./component-overview.component.css']
})
```

To declare the styles within the component, add a styles property to the @Component decorator that contains the styles you want to use.

```
@Component({
 selector: 'app-component-overview',
 template: '<h1>Hello World!</h1>',
 styles: ['h1 { font-weight: normal; }']
})
```

The styles property takes an array of strings that contain the CSS rule declarations.

4.3 Lifecycle

🕮 4.3.1

Lifecycle hooks

A component instance has a lifecycle that starts when Angular instantiates the component class and renders the component view along with its child views. The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed. The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM. Directives have a similar lifecycle, as Angular creates, updates, and destroys instances in the course of execution.

Your application can use lifecycle hook methods to tap into key events in the lifecycle of a component or directive in order to initialize new instances, initiate change detection when needed, respond to updates during change detection, and clean up before deletion of instances.

🕮 4.3.2

Responding to lifecycle events

You can respond to events in the lifecycle of a component or directive by implementing one or more of the lifecycle hook interfaces in the Angular core library. The hooks give you the opportunity to act on a component or directive instance at the appropriate moment, as Angular creates, updates, or destroys that instance.

Each interface defines the prototype for a single hook method, whose name is the interface name prefixed with ng. For example, the OnInit interface has a hook method named ngOnInit(). If you implement this method in your component or directive class, Angular calls it shortly after checking the input properties for that component or directive for the first time.

```
@Directive({selector: '[appPeekABoo]'})
export class PeekABooDirective implements OnInit {
 constructor(private logger: LoggerService) { }
 // implement OnInit's `ngOnInit` method
 ngOnInit() {
  this.logIt(`OnInit`);
 }
 logIt(msg: string) {
  this.logger.log(`#${nextId++} ${msg}`);
 }
```

```
}
```

You don't have to implement all (or any) of the lifecycle hooks, just the ones you need.

📖 4.3.3

Lifecycle event sequence

After your application instantiates a component or directive by calling its constructor, Angular calls the hook methods you have implemented at the appropriate point in the lifecycle of that instance.

Angular executes hook methods in the following sequence. You can use them to perform the following kinds of operations.

```
ngOnChanges(), ngOnInit(), ngDoCheck(), ngAfterContentInit(),
ngAfterContentChecked(), ngAfterViewInit(),
ngAfterViewChecked(), ngOnDestroy().
```

📖 4.3.4

Initializing a component or directive

Use the ngOnInit() method to perform the following initialization tasks.

- Perform complex initializations outside of the constructor. Components should be cheap and safe to construct. You should not, for example, fetch data in a component constructor. You shouldn't worry that a new component will try to contact a remote server when created under test or before you decide to display it. An ngOnInit() is a good place for a component to fetch its initial data.
- Set up the component after Angular sets the input properties. Constructors should do no more than set the initial local variables to simple values. Keep in mind that a directive's data-bound input properties are not set until after construction. If you need to initialize the directive based on those properties, set them when ngOnInit() runs.

📖 4.3.5

Cleaning up on instance destruction

Put cleanup logic in ngOnDestroy(), the logic that must run before Angular destroys the directive.

This is the place to free resources that won't be garbage-collected automatically. You risk memory leaks if you neglect to do so.

- Unsubscribe from Observables and DOM events.
- Stop interval timers.
- Unregister all callbacks that the directive registered with global or application services.

The ngOnDestroy() method is also the time to notify another part of the application that the component is going away.

🕮 4.3.6

Using change detection hooks

Angular calls the ngOnChanges() method of a component or directive whenever it detects changes to the input properties. The onChanges example demonstrates this by monitoring the OnChanges() hook.

```
ngOnChanges(changes: SimpleChanges) {
 for (const propName in changes) {
  const chng = changes[propName];
  const cur = JSON.stringify(chng.currentValue);
  const prev = JSON.stringify(chng.previousValue);
  this.changeLog.push(`${propName}: currentValue = ${cur},
previousValue = ${prev}`);
 }
}
```

The ngOnChanges() method takes an object that maps each changed property name to a SimpleChange object holding the current and previous property values. This hook iterates over the changed properties and logs them.

The example component, OnChangesComponent, has two input properties: hero and power.

```
@Input() hero!: Hero;
@Input() power = '';
```

The host OnChangesParentComponent binds to them as follows.

```
<on-changes [hero]="hero" [power]="power"></on-changes>
```

4.4 Encapsulation

📖 4.4.1

View encapsulation

In Angular, component CSS styles are encapsulated into the component's view and don't affect the rest of the application.

To control how this encapsulation happens on a *per component* basis, you can set the *view encapsulation mode* in the component metadata. Choose from the following modes:

- ShadowDom view encapsulation uses the browser's native shadow DOM implementation to attach a shadow DOM to the component's host element, and then puts the component view inside that shadow DOM. The component's styles are included within the shadow DOM.
- Emulated view encapsulation (the default) emulates the behavior of shadow DOM by preprocessing (and renaming) the CSS code to effectively scope the CSS to the component's view.
- None means that Angular does no view encapsulation. Angular adds the CSS to the global styles. The scoping rules, isolations, and protections discussed earlier don't apply. This mode is essentially the same as pasting the component's styles into the HTML.

To set the component's encapsulation mode, use the encapsulation property in the component metadata:

```
// warning: not all browsers support shadow DOM encapsulation
at this time
encapsulation: ViewEncapsulation.ShadowDom
```

📖 4.4.2

Inspecting generated CSS

When using emulated view encapsulation, Angular preprocesses all component styles so that they approximate the standard shadow CSS scoping rules.

In the DOM of a running Angular application with emulated view encapsulation enabled, each DOM element has some extra attributes attached to it:

```
<hero-details _nghost-pmm-5>
<h2 _ngcontent-pmm-5>Mister Fantastic</h2>
```

```
 <hero-team _ngcontent-pmm-5 _nghost-pmm-6>
  <h3 _ngcontent-pmm-6>Team</h3>
 </hero-team>
</hero-detail>
```

There are two kinds of generated attributes:

- An element that would be a shadow DOM host in native encapsulation has a generated _nghost attribute. This is typically the case for component host elements.
- An element within a component's view has a _ngcontent attribute that identifies to which host's emulated shadow DOM this element belongs.

The exact values of these attributes aren't important. They are automatically generated and you should never refer to them in application code. But they are targeted by the generated component styles, which are in the <head> section of the DOM:

```
[_nghost-pmm-5] {
 display: block;
 border: 1px solid black;
}

h3[_ngcontent-pmm-6] {
 background-color: white;
 border: 1px solid #777;
}
```

These styles are post-processed so that each selector is augmented with _nghost or _ngcontent attribute selectors. These extra selectors enable the scoping rules described in this page.

🕮 4.4.3

Mixing encapsulation modes

Avoid mixing components that use different view encapsulation. Where it is necessary, you should be aware of how the component styles will interact.

- The styles of components with ViewEncapsulation.Emulated are added to the <head> of the document, making them available throughout the application, but are "scoped" so they only affect elements within the component's template.

- The styles of components with ViewEncapsulation.None are added to the <head> of the document, making them available throughout the application, and are not "scoped" so they can affect any element in the application.
- The styles of components with ViewEncapsulation.ShadowDom are only added to the shadow DOM host, ensuring that they only affect elements within the component's template.

All the styles for ViewEncapsulation.Emulated and ViewEncapsulation.None components are also added to the shadow DOM host of each ViewEncapsulation.ShadowDom component.

The result is that styling for components with ViewEncapsulation.None will affect matching elements within the shadow DOM.

This approach may seem counter-intuitive at first, but without it a component with ViewEncapsulation.None could not be used within a component with ViewEncapsulation.ShadowDom, since its styles would not be available.

4.5 Interaction

🕮 4.5.1

Pass data from parent to child with input binding

HeroChildComponent has two input properties, typically adorned with @Input() decorator.

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

import { Hero } from './hero';

@Component({
 selector: 'app-hero-child',
 template: `
  <h3>{{hero.name}} says:</h3>
  <p>I, {{hero.name}}, am at your service, {{masterName}}.</p>
 `
})
export class HeroChildComponent {
 @Input() hero!: Hero;
 @Input('master') masterName = ''; // tslint:disable-line: no-
input-rename
}
```

The second @Input aliases the child component property name masterName as 'master'.

The HeroParentComponent nests the child HeroChildComponent inside an *ngFor repeater, binding its master string property to the child's master alias, and each iteration's hero instance to the child's hero property.

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

import { HEROES } from './hero';

@Component({
 selector: 'app-hero-parent',
 template: `
  <h2>{{master}} controls {{heroes.length}} heroes</h2>

  <app-hero-child
   *ngFor="let hero of heroes"
   [hero]="hero"
   [master]="master">
  </app-hero-child>
 `
})
export class HeroParentComponent {
 heroes = HEROES;
 master = 'Master';
}
```

📖 4.5.2

Intercept input property changes with a setter

Use an input property setter to intercept and act upon a value from the parent.

The setter of the name input property in the child NameChildComponent trims the whitespace from a name and replaces an empty value with default text.

```
import { Component, Input } from '@angular/core';
@Component({
 selector: 'app-name-child',
 template: '<h3>"{{name}}"</h3>'
})
export class NameChildComponent {
 @Input()
 get name(): string { return this._name; }
```

```
 set name(name: string) {
  this._name = (name && name.trim()) || '<no name set>';
 }
 private _name = '';
}
Here's the NameParentComponent demonstrating name variations
including a name with all spaces:
import { Component } from '@angular/core';
@Component({
 selector: 'app-name-parent',
 template: `
  <h2>Master controls {{names.length}} names</h2>

  <app-name-child *ngFor="let name of names"
[name]="name"></app-name-child>
 `
})
export class NameParentComponent {
 // Displays 'Dr IQ', '<no name set>', 'Bombasto'
 names = ['Dr IQ', '  ', ' Bombasto '];
}
```

📖 4.5.3

Intercept input property changes with ngOnChanges()

Detect and act upon changes to input property values with the ngOnChanges() method of the OnChanges lifecycle hook interface.

This VersionChildComponent detects changes to the major and minor input properties and composes a log message reporting these changes:

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

@Component({
 selector: 'app-version-child',
 template: `
  <h3>Version {{major}}.{{minor}}</h3>
  <h4>Change log:</h4>
  <ul>
   <li *ngFor="let change of changeLog">{{change}}</li>
  </ul>
 `
```

```
})
export class VersionChildComponent implements OnChanges {
 @Input() major = 0;
 @Input() minor = 0;
 changeLog: string[] = [];

 ngOnChanges(changes: SimpleChanges) {
  const log: string[] = [];
  for (const propName in changes) {
   const changedProp = changes[propName];
   const to = JSON.stringify(changedProp.currentValue);
   if (changedProp.isFirstChange()) {
    log.push(`Initial value of ${propName} set to ${to}`);
   } else {
    const from = JSON.stringify(changedProp.previousValue);
    log.push(`${propName} changed from ${from} to ${to}`);
   }
  }
  this.changeLog.push(log.join(', '));
 }
}
```

The VersionParentComponent supplies the minor and major values and binds buttons to methods that change them.

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

@Component({
 selector: 'app-version-parent',
 template: `
  <h2>Source code version</h2>
  <button (click)="newMinor()">New minor version</button>
  <button (click)="newMajor()">New major version</button>
  <app-version-child [major]="major" [minor]="minor"></app-
version-child>
 `
})
export class VersionParentComponent {
 major = 1;
 minor = 23;

 newMinor() {
  this.minor++;
 }
```

```
 newMajor() {
  this.major++;
  this.minor = 0;
 }
}
```

🕮 4.5.4

Parent listens for child event

The child component exposes an EventEmitter property with which it emits events when something happens. The parent binds to that event property and reacts to those events.

The child's EventEmitter property is an output property, typically adorned with an @Output() decorator as seen in this VoterComponent:

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

@Component({
 selector: 'app-voter',
 template: `
  <h4>{{name}}</h4>
  <button
(click)="vote(true)" [disabled]="didVote">Agree</button>
  <button (click)="vote(false)"
[disabled]="didVote">Disagree</button>
 `
})
export class VoterComponent {
 @Input() name = '';
 @Output() voted = new EventEmitter<boolean>();
 didVote = false;

 vote(agreed: boolean) {
  this.voted.emit(agreed);
  this.didVote = true;
 }
}
```

Clicking a button triggers emission of a true or false, the boolean payload.

The parent VoteTakerComponent binds an event handler called onVoted() that responds to the child event payload $event and updates a counter.

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

@Component({
 selector: 'app-vote-taker',
 template: `
  <h2>Should mankind colonize the Universe?</h2>
  <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>

  <app-voter
   *ngFor="let voter of voters"
   [name]="voter"
   (voted)="onVoted($event)">
  </app-voter>
 `
})
export class VoteTakerComponent {
 agreed = 0;
 disagreed = 0;
 voters = ['Narco', 'Celeritas', 'Bombasto'];

 onVoted(agreed: boolean) {
  agreed ? this.agreed++ : this.disagreed++;
 }
}
```

📖 4.5.5

Parent interacts with child using local variable

A parent component cannot use data binding to read child properties or invoke child methods. You can do both by creating a template reference variable for the child element and then reference that variable within the parent template as seen in the following example.

The following is a child CountdownTimerComponent that repeatedly counts down to zero and launches a rocket. It has start and stop methods that control the clock and it displays a countdown status message in its own template.

```
import { Component, OnDestroy } from '@angular/core';
@Component({
 selector: 'app-countdown-timer',
 template: '<p>{{message}}</p>'
})
export class CountdownTimerComponent implements OnDestroy {
```

```
 intervalId = 0;
 message = '';
 seconds = 11;

 ngOnDestroy() { this.clearTimer(); }

 start() { this.countDown(); }
 stop() {
  this.clearTimer();
  this.message = `Holding at T-${this.seconds} seconds`;
 }

 private clearTimer() { clearInterval(this.intervalId); }

 private countDown() {
  this.clearTimer();
  this.intervalId = window.setInterval(() => {
   this.seconds -= 1;
   if (this.seconds === 0) {
    this.message = 'Blast off!';
   } else {
    if (this.seconds < 0) { this.seconds = 10; } // reset
    this.message = `T-${this.seconds} seconds and counting`;
   }
  }, 1000);
 }
}
```

The CountdownLocalVarParentComponent that hosts the timer component is as follows:

```
import { Component } from '@angular/core';
import { CountdownTimerComponent } from './countdown-
timer.component';

@Component({
 selector: 'app-countdown-parent-lv',
 template: `
  <h3>Countdown to Liftoff (via local variable)</h3>
  <button (click)="timer.start()">Start</button>
  <button (click)="timer.stop()">Stop</button>
  <div class="seconds">{{timer.seconds}}</div>
  <app-countdown-timer #timer></app-countdown-timer>
 `,
 styleUrls: ['../assets/demo.css']
```

```
})
export class CountdownLocalVarParentComponent { }
```

The parent component cannot data bind to the child's start and stop methods nor to its seconds property.

You can place a local variable, #timer, on the tag <app-countdown-timer> representing the child component. That gives you a reference to the child component and the ability to access any of its properties or methods from within the parent template.

4.6 Styles

🕮 4.6.1

Component styles

Angular applications are styled with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

Additionally, Angular can bundle component styles with components, enabling a more modular design than regular stylesheets.

🕮 4.6.2

Using component styles

For every Angular component you write, you may define not only an HTML template, but also the CSS styles that go with that template, specifying any selectors, rules, and media queries that you need.

One way to do this is to set the styles property in the component metadata. The styles property takes an array of strings that contain CSS code. Usually you give it one string, as in the following example:

```
@Component({
 selector: 'app-root',
 template: `
  <h1>Tour of Heroes</h1>
  <app-hero-main [hero]="hero"></app-hero-main>
 `,
 styles: ['h1 { font-weight: normal; }']
```

```
})
export class HeroAppComponent {
/* . . . */
}
```

📖 4.6.3

Style scope

They are not inherited by any components nested within the template nor by any content projected into the component.

In this example, the h1 style applies only to the HeroAppComponent, not to the nested HeroMainComponent nor to <h1> tags anywhere else in the application.

This scoping restriction is a styling modularity feature.

- You can use the CSS class names and selectors that make the most sense in the context of each component.
- Class names and selectors are local to the component and don't collide with classes and selectors used elsewhere in the application.
- Changes to styles elsewhere in the application don't affect the component's styles.
- You can co-locate the CSS code of each component with the TypeScript and HTML code of the component, which leads to a neat and tidy project structure.
- You can change or remove component CSS code without searching through the whole application to find where else the code is used.

📖 4.6.4

Special selectors

Component styles have a few special selectors from the world of shadow DOM style scoping

```
:host
```

Use the :host pseudo-class selector to target styles in the element that hosts the component (as opposed to targeting elements inside the component's template).

The :host selector is the only way to target the host element. You can't reach the host element from inside the component with other selectors because it's not part of the component's own template. The host element is in a parent component's template.

Use the function form to apply host styles conditionally by including another selector inside parentheses after :host.

```
:host-context
```

Sometimes it's useful to apply styles based on some condition outside of a component's view. For example, a CSS theme class could be applied to the document <body> element, and you want to change how your component looks based on that.

Use the :host-context() pseudo-class selector, which works just like the function form of :host(). The :host-context() selector looks for a CSS class in any ancestor of the component host element, up to the document root. The :host-context() selector is useful when combined with another selector.

🕮 4.6.5

Loading component styles

There are several ways to add styles to a component:

- By setting styles or styleUrls metadata.
- Inline in the template HTML.
- With CSS imports.

4.7 Content projection

🕮 4.7.1

Single-slot content projection

The most basic form of content projection is single-slot content projection. Single-slot content projection refers to creating a component into which you can project one component.

To create a component that uses single-slot content projection:

- Create a component.
- In the template for your component, add an <ng-content> element where you want the projected content to appear.

🕮 4.7.2

Multi-slot content projection

A component can have multiple slots. Each slot can specify a CSS selector that determines which content goes into that slot. This pattern is referred to as multi-slot content projection. With this pattern, you must specify where you want the projected content to appear. You accomplish this task by using the select attribute of <ng-content>.

To create a component that uses multi-slot content projection:

1. Create a component.
2. In the template for your component, add an <ng-content> element where you want the projected content to appear.
3. Add a select attribute to the <ng-content> elements. Angular supports selectors for any combination of tag name, attribute, CSS class, and the :not pseudo-class.

🕮 4.7.3

Conditional content projection

If your component needs to conditionally render content, or render content multiple times, you should configure that component to accept an <ng-template> element that contains the content you want to conditionally render.

Using an <ng-content> element in these cases is not recommended, because when the consumer of a component supplies the content, that content is always initialized, even if the component does not define an <ng-content> element or if that <ng-content> element is inside of an ngIf statement.

With an <ng-template> element, you can have your component explicitly render content based on any condition you want, as many times as you want. Angular will not initialize the content of an <ng-template> element until that element is explicitly rendered.

The following steps demonstrate a typical implementation of conditional content projection using <ng-template>.

1.Create a component.

2.In the component that accepts an <ng-template> element, use an <ng-container> element to render that template, such as:

```
<ng-container [ngTemplateOutlet]="content.templateRef"></ng-
container>
```

This example uses the ngTemplateOutlet directive to render a given <ng-template> element, which you will define in a later step. You can apply an ngTemplateOutlet directive to any type of element. This example assigns the directive to an <ng-container> element because the component does not need to render a real DOM element.

3.Wrap the <ng-container> element in another element, such as a div element, and apply your conditional logic.

```
<div *ngIf="expanded" [id]="contentId">
  <ng-container [ngTemplateOutlet]="content.templateRef"></ng-
container>
</div>
```

4.In the template where you want to project content, wrap the projected content in an <ng-template> element, such as:

```
<ng-template appExampleZippyContent>
 It depends on what you do with it.
</ng-template>
```

The <ng-template> element defines a block of content that a component can render based on its own logic. A component can get a reference to this template content, or TemplateRef, by using either the @ContentChild or @ContentChildren decorators. The preceding example creates a custom directive, appExampleZippyContent, as an API to mark the <ng-template> for the component's content. With the TemplateRef, the component can render the referenced content by using either the ngTemplateOutlet directive, or with the ViewContainerRef method createEmbeddedView().

5.Create an attribute directive with a selector that matches the custom attribute for your template. In this directive, inject a TemplateRef instance.

```
@Directive({
 selector: '[appExampleZippyContent]'
})
export class ZippyContentDirective {
 constructor(public templateRef: TemplateRef<unknown>) {}
}
```

In the previous step, you added an <ng-template> element with a custom attribute, appExampleZippyDirective. This code provides the logic that Angular will use when it encounters that custom attribute. In this case, that logic instructs Angular to instantiate a template reference.

6.In the component you want to project content into, use @ContentChild to get the template of the projected conten

```
@ContentChild(ZippyContentDirective) content!:
ZippyContentDirective;
```

Prior to this step, your application has a component that instantiates a template when certain conditions are met. You've also created a directive that provides a reference to that template. In this last step, the @ContentChild decorator instructs Angular to instantiate the template in the designated component.

4.8 Dynamic Components

🕮 4.8.1

The anchor directive

Before you can add components you have to define an anchor point to tell Angular where to insert components.

The ad banner uses a helper directive called AdDirective to mark valid insertion points in the template.

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

@Directive({
 selector: '[adHost]',
})
export class AdDirective {
 constructor(public viewContainerRef: ViewContainerRef) { }
}
```

AdDirective injects ViewContainerRef to gain access to the view container of the element that will host the dynamically added component.

In the @Directive decorator, notice the selector name, adHost; that's what you use to apply the directive to the element. The next section shows you how.

📖 4.8.2

Loading components

Most of the ad banner implementation is in ad-banner.component.ts. To keep things simple in this example, the HTML is in the @Component decorator's template property as a template string.

The <ng-template> element is where you apply the directive you just made. To apply the AdDirective, recall the selector from ad.directive.ts, [adHost]. Apply that to <ng-template> without the square brackets. Now Angular knows where to dynamically load components.

```
template: `
 <div class="ad-banner-example">
  <h3>Advertisements</h3>
  <ng-template adHost></ng-template>
 </div>
`
```

The <ng-template> element is a good choice for dynamic components because it doesn't render any additional output.

📖 4.8.3

Resolving components

Take a closer look at the methods in ad-banner.component.ts.

AdBannerComponent takes an array of AdItem objects as input, which ultimately comes from AdService. AdItem objects specify the type of component to load and any data to bind to the component.AdService returns the actual ads making up the ad campaign.

Passing an array of components to AdBannerComponent allows for a dynamic list of ads without static elements in the template.

With its getAds() method, AdBannerComponent cycles through the array of AdItems and loads a new component every 3 seconds by calling loadComponent().

```
export class AdBannerComponent implements OnInit, OnDestroy {

 @Input() ads: AdItem[] = [];

 currentAdIndex = -1;
```

```
 @ViewChild(AdDirective, {static: true}) adHost!: AdDirective;
 interval: number | undefined;

 constructor(private componentFactoryResolver:
ComponentFactoryResolver) { }

 ngOnInit() {
  this.loadComponent();
  this.getAds();
 }

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

 loadComponent() {
  this.currentAdIndex = (this.currentAdIndex + 1) %
this.ads.length;
  const adItem = this.ads[this.currentAdIndex];

  const componentFactory =
this.componentFactoryResolver.resolveComponentFactory(adItem.c
omponent);

  const viewContainerRef = this.adHost.viewContainerRef;
  viewContainerRef.clear();

  const componentRef =
viewContainerRef.createComponent<AdComponent>(componentFactory
);
  componentRef.instance.data = adItem.data;
 }

 getAds() {
  this.interval = setInterval(() => {
   this.loadComponent();
  }, 3000);
 }
}
```

The loadComponent() method is doing a lot of the heavy lifting here. Take it step by step. First, it picks an ad.

After loadComponent() selects an ad, it uses ComponentFactoryResolver to resolve a ComponentFactory for each specific component. The ComponentFactory then creates an instance of each component.

Next, you're targeting the viewContainerRef that exists on this specific instance of the component. How do you know it's this specific instance? Because it's referring to adHost and adHost is the directive you set up earlier to tell Angular where to insert dynamic components.

As you may recall, AdDirective injects ViewContainerRef into its constructor. This is how the directive accesses the element that you want to use to host the dynamic component.

To add the component to the template, you call createComponent() on ViewContainerRef.

The createComponent() method returns a reference to the loaded component. Use that reference to interact with the component by assigning to its properties or calling its methods.

4.9 Angular elements

🕮 4.9.1

Angular elements overview

Angular elements are Angular components packaged as custom elements (also called Web Components), a web standard for defining new HTML elements in a framework-agnostic way.

Custom elements are a Web Platform feature currently supported by Chrome, Edge (Chromium-based), Firefox, Opera, and Safari, and available in other browsers through polyfills A custom element extends HTML by allowing you to define a tag whose content is created and controlled by JavaScript code. The browser maintains a CustomElementRegistry of defined custom elements, which maps an instantiable JavaScript class to an HTML tag.

The @angular/elements package exports a createCustomElement() API that provides a bridge from Angular's component interface and change detection functionality to the built-in DOM API.

Transforming a component to a custom element makes all of the required Angular infrastructure available to the browser. Creating a custom element is simple and straightforward, and automatically connects your component-defined view with change detection and data binding, mapping Angular functionality to the corresponding native HTML equivalents.

📖 4.9.2

Using custom elements

Custom elements bootstrap themselves - they start automatically when they are added to the DOM, and are automatically destroyed when removed from the DOM. Once a custom element is added to the DOM for any page, it looks and behaves like any other HTML element, and does not require any special knowledge of Angular terms or usage conventions.

Easy dynamic content in an Angular application

Transforming a component to a custom element provides an easy path to creating dynamic HTML content in your Angular application. HTML content that you add directly to the DOM in an Angular application is normally displayed without Angular processing, unless you define a dynamic component, adding your own code to connect the HTML tag to your application data, and participate in change detection. With a custom element, all of that wiring is taken care of automatically.

Content-rich applications

If you have a content-rich application, such as the Angular app that presents this documentation, custom elements let you give your content providers sophisticated Angular functionality without requiring knowledge of Angular. For example, an Angular guide like this one is added directly to the DOM by the Angular navigation tools, but can include special elements like <code-snippet> that perform complex operations. All you need to tell your content provider is the syntax of your custom element. They don't need to know anything about Angular, or anything about your component's data structures or implementation.

📖 4.9.3

Transforming components to custom elements

Angular provides the createCustomElement() function for converting an Angular component, together with its dependencies, to a custom element. The function collects the component's observable properties, along with the Angular functionality the browser needs to create and destroy instances, and to detect and respond to changes.

The conversion process implements the NgElementConstructor interface, and creates a constructor class that is configured to produce a self-bootstrapping instance of your component.

Use the built-in

`customElements.define()`

function to register the configured constructor and its associated custom-element tag with the browser's

`CustomElementRegistry`

When the browser encounters the tag for the registered element, it uses the constructor to create a custom-element instance.

📖 4.9.4

Mapping

A custom element hosts an Angular component, providing a bridge between the data and logic defined in the component and standard DOM APIs. Component properties and logic maps directly into HTML attributes and the browser's event system.

The creation API parses the component looking for input properties, and defines corresponding attributes for the custom element. It transforms the property names to make them compatible with custom elements, which do not recognize case distinctions. The resulting attribute names use dash-separated lowercase. For example, for a component with @Input('myInputProp') inputProp, the corresponding custom element defines an attribute my-input-prop.

Component outputs are dispatched as HTML Custom Events, with the name of the custom event matching the output name. For example, for a component with @Output() valueChanged = new EventEmitter(), the corresponding custom element will dispatch events with the name "valueChanged", and the emitted data will be stored on the event's detail property. If you provide an alias, that value is used; for example, @Output('myClick') clicks = new EventEmitter<string>(); results in dispatch events with the name "myClick".

4.10 Components (Exercises)

📝 4.10.1

Does each component require a CSS selector?

- yes
- no

4.10.2

How can you define a template for your component?

4.10.3

When the life cycle ends?

4.10.4

What the cleanup logic in ngOnDestroy () protects against?

4.10.5

View encapsulation (the default) emulates the behavior of shadow DOM by preprocessing (and renaming) the CSS code to effectively scope the CSS to the component's view

- ShadowDom
- Emulated
- None

4.10.6

The styles of components with ViewEncapsulation.ShadowDom are only added to the shadow DOM host, ensuring that they only affect elements within the component's template.

- True
- False

4.10.7

Don't style changes elsewhere in the application affect the component's styles?

4.10.8

Możesz zmienić lub usunąć komponentowy kod CSS bez przeszukiwania całej aplikacji, aby dowiedzieć się, gdzie jeszcze ten kod jest używany.

- True
- False

4.10.9

Can a component only have one slot?

- No
- Yes

4.10.10

What do I need to add before adding components?

4.10.11

Passing an array of components to AdBannerComponent allows for a dynamic list of ads without static elements in the template.

- True
- False

Templates

Chapter 5

5.1 Text interpolation

🕮 5.1.1

Text interpolation

Text interpolation lets you incorporate dynamic string values into your HTML templates. Use interpolation to dynamically change what appears in an application view, such as displaying a custom greeting that includes the user's name.

🕮 5.1.2

Displaying values with interpolation

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses the double curly braces **{{** and **}}** as delimiters.

To illustrate how interpolation works, consider an Angular component that contains a **currentCustomer** variable:

src/app/app.component.ts

```
currentCustomer = 'Maria';
```

Use interpolation to display the value of this variable in the corresponding component template:

src/app/app.component.html

```
<h3>Current customer: {{ currentCustomer }}</h3>
```

Angular replaces **currentCustomer** with the string value of the corresponding component property. In this case, the value is **Maria**.

In the following example, Angular evaluates the **title** and **itemImageUrl** properties to display some title text and an image.

src/app/app.component.html

```
<p>{{title}}</p>
<div><img src="{{itemImageUrl}}"></div>
```

🕮 5.1.3

Template expressions

A template expression produces a value and appears within double curly braces, **{{ }}**. Angular resolves the expression and assigns it to a property of a binding target. The target could be an HTML element, a component, or a directive.

Resolving expressions with interpolation

More generally, the text between the braces is a template expression that Angular first evaluates and then converts to a string. The following interpolation illustrates the point by adding two numbers:

src/app/app.component.html

```
<!-- "The sum of 1 + 1 is 2" -->
<p>The sum of 1 + 1 is {{1 + 1}}.</p>
```

Expressions can also invoke methods of the host component such as **getVal()** in the following example:

src/app/app.component.html

```
<!-- "The sum of 1 + 1 is not 4" -->
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}.</p>
```

With interpolation, Angular performs the following tasks:

1. Evaluates all expressions in double curly braces.
2. Converts the expression results to strings.
3. Links the results to any adjacent literal strings.
4. Assigns the composite to an element or directive property.

🕮 5.1.4

Syntax

Template expressions are similar to JavaScript. Many JavaScript expressions are legal template expressions, with the following exceptions.

You can't use JavaScript expressions that have or promote side effects, including:

- Assignments (**=**, **+=**, **-=**, **...**)
- Operators such as **new**, **typeof**, or **instanceof**

- Chaining expressions with **;** or **,**
- The increment and decrement operators **++** and **--**
- Some of the ES2015+ operators

Other notable differences from JavaScript syntax include:

- No support for the bitwise operators such as | and **&**
- New template expression operators, such as |, **?.** and **!**

🕮 5.1.5

Expression context

Interpolated expressions have a context—a particular part of the application to which the expression belongs. Typically, this context is the component instance.

In the following snippet, the expression **recommended** and the expression **itemImageUrl2** refer to properties of the **AppComponent**.

src/app/app.component.html

```
<h4>{{recommended}}</h4>
<img [src]="itemImageUrl2">
```

An expression can also refer to properties of the template's context such as a template input variable or a template reference variable.

The following example uses a template input variable of **customer**.

src/app/app.component.html (template input variable)

```
<ul>
  <li *ngFor="let customer of
customers">{{customer.name}}</li>
</ul>
```

This next example features a template reference variable, **#customerInput**.

src/app/app.component.html (template reference variable)

```
<label>Type something:
  <input #customerInput>{{customerInput.value}}
</label>
```

🕮 5.1.6

Preventing name collisions

The context against which an expression evaluates is the union of the template variables, the directive's context object—if it has one—and the component's members. If you reference a name that belongs to more than one of these namespaces, Angular applies the following logic to determine the context:

1. The template variable name.
2. A name in the directive's context.
3. The component's member names.

To avoid variables shadowing variables in another context, keep variable names unique. In the following example, the **AppComponent** template greets the **customer**, Padma.

An ngfor then lists each **customer** in the **customers** array.

src/app/app.component.ts

```
@Component({
  template: `
    <div>
      <!-- Hello, Padma -->
      <h1>Hello, {{customer}}</h1>
      <ul>
        <!-- Ebony and Chiho in a list-->
        <li *ngFor="let customer of customers">{{
customer.value }}</li>
      </ul>
    </div>
  `
})
class AppComponent {
  customers = [{value: 'Ebony'}, {value: 'Chiho'}];
  customer = 'Padma';
}
```

The customer within the ngFor is in the context of an <ng-template> and so refers to the customer in the customers array, in this case Ebony and Chiho. This list does not feature Padma because customer outside of the ngFor is in a different context. Conversely, customer in the <h1> doesn't include Ebony or Chiho because the context for this customer is the class and the class value for customer is Padma.

🕮 5.1.7

Expression best practices

When using template expressions, follow these best practices:

- Use short expressions
- Use property names or method calls whenever possible. Keep application and business logic in the component, where it is accessible to develop and test.
- Quick execution
- Angular executes template expressions after every change detection cycle. Many asynchronous activities trigger change detection cycles, such as promise resolutions, HTTP results, timer events, key presses and mouse moves.
- Expressions should finish quickly to keep the user experience as efficient as possible, especially on slower devices. Consider caching values when their computation requires greater resources.
- No visible side effects
- According to Angular's unidirectional data flow model,, a template expression should not change any application state other than the value of the target property. Reading a component value should not change some other displayed value. The view should be stable throughout a single rendering pass.

5.2 Template statements

🕮 5.2.1

Template statements

Template statements are methods or properties that you can use in your HTML to respond to user events. With template statements, your application can engage users through actions such as displaying dynamic content or submitting forms.

In the following example, the template statement **deleteHero()** appears in quotes to the right of the **=** symbol as in **(event)="statement"**.

src/app/app.component.html

```
<button (click)="deleteHero()">Delete hero</button>
```

When the user clicks the Delete hero button, Angular calls the **deleteHero()** method in the component class.

Use template statements with elements, components, or directives in response to events.

🕮 5.2.2

Syntax

Like template expressions template statements use a language that looks like JavaScript. However, the parser for template statements differs from the parser for template expressions. In addition, the template statements parser specifically supports both basic assignment, **=**, and chaining expressions with semicolons, **;**.

The following JavaScript and template expression syntax is not allowed:

- **new**
- increment and decrement operators, **++** and **--**
- operator assignment, such as **+=** and **-=**
- the bitwise operators, such as | and **&**
- the pipe operator

🕮 5.2.3

Statement context

Statements have a context—a particular part of the application to which the statement belongs.

Statements can refer only to what's in the statement context, which is typically the component instance. For example, **deleteHero()** of **(click)="deleteHero()"** is a method of the component in the following snippet.

src/app/app.component.html

```
<button (click)="deleteHero()">Delete hero</button>
```

The statement context may also refer to properties of the template's own context. In the following example, the component's event handling method, onSave() takes the template's own $event object as an argument. On the next two lines, the deleteHero() method takes a template input variable, hero, and onSubmit() takes a template reference variable, #heroForm.

src/app/app.component.html

```
<button (click)="onSave($event)">Save</button>
```

```
<button *ngFor="let hero of heroes"
(click)="deleteHero(hero)">{{hero.name}}</button>
<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ... </form>
```

In this example, the context of the **$event** object, **hero**, and **#heroForm** is the template.

Template context names take precedence over component context names. In the preceding **deleteHero(hero)**, the **hero** is the template input variable, not the component's **hero** property.

🕮 5.2.4

Statement best practices

- Conciseness
- Use method calls or basic property assignments to keep template statements minimal.
- Work within the context
- The context of a template statement can be the component class instance or the template. Because of this, template statements cannot refer to anything in the global namespace such as **window** or **document**. For example, template statements can't call **console.log()** or **Math.max()**.

5.3 Pipes

🕮 5.3.1

Transforming Data Using Pipes

Use pipesto transform strings, currency amounts, dates, and other data for display. Pipes are simple functions to use in template expressions to accept an input value and return a transformed value. Pipes are useful because you can use them throughout your application, while only declaring each pipe once. For example, you would use a pipe to show a date as April 15, 1988 rather than the raw string format.

Angular provides built-in pipes for typical data transformations, including transformations for internationalization (i18n), which use locale information to format data. The following are commonly used built-in pipes for data formatting:

- **DatePipe**: Formats a date value according to locale rules.
- **UpperCasePipe**: Transforms text to all upper case.
- **LowerCasePipe**: Transforms text to all lower case.

- **CurrencyPipe**: Transforms a number to a currency string, formatted according to locale rules.
- **DecimalPipe**: Transforms a number into a string with a decimal point, formatted according to locale rules.
- **PercentPipe**: Transforms a number to a percentage string, formatted according to locale rules.

🕮 5.3.2

Transforming data with parameters and chained pipes

Use optional parameters to fine-tune a pipe's output. For example, use the **CurrencyPipe** with a country code such as EUR as a parameter. The template expression **{{ amount | currency:'EUR' }}** transforms the **amount** to currency in euros. Follow the pipe name (**currency**) with a colon (**:**) and the parameter value (**'EUR'**).

If the pipe accepts multiple parameters, separate the values with colons. For example, **{{ amount | currency:'EUR':'Euros '}}** adds the second parameter, the string literal **'Euros '**, to the output string. Use any valid template expression as a parameter, such as a string literal or a component property.

Some pipes require at least one parameter and allow more optional parameters, such as **SlicePipe**. For example, **{{ slice:1:5 }}** creates a new array or string containing a subset of the elements starting with element **1** and ending with element **5**.

🕮 5.3.3

Detecting changes with data binding in pipes

You use data binding with a pipe to display values and respond to user actions. If the data is a primitive input value, such as **String** or **Number**, or an object reference as input, such as **Date** or **Array**, Angular executes the pipe whenever it detects a change for the input value or reference.

For example, you could change the previous custom pipe example to use two-way data binding with **ngModel** to input the amount and boost factor, as shown in the following code example.

src/app/power-boost-calculator.component.ts

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

```
@Component({
  selector: 'app-power-boost-calculator',
  template: `
    <h2>Power Boost Calculator</h2>
    <label for="power-input">Normal power: </label>
    <input id="power-input" type="text" [(ngModel)]="power">
    <label for="boost-input">Boost factor: </label>
    <input id="boost-input" type="text" [(ngModel)]="factor">
    <p>
      Super Hero Power: {{power | exponentialStrength:
factor}}
    </p>
  `,
  styles: ['input {margin: .5rem 0;}']
})
export class PowerBoostCalculatorComponent {
  power = 5;
  factor = 1;
}
```

The **exponentialStrength** pipe executes every time the user changes the "normal power" value or the "boost factor".

Angular detects each change and immediately runs the pipe. This is fine for primitive input values. However, if you change something *inside* a composite object (such as the month of a date, an element of an array, or an object property), you need to understand how change detection works, and how to use an **impure** pipe.

5.4 Property binding

🕮 5.4.1

Property binding

Property binding in Angular helps you set values for properties of HTML elements or directives. Use property binding to do things such as toggle button functionality, set paths programmatically, and share values between components.

🕮 5.4.2

Binding to a property

To bind to an element's property, enclose it in square brackets, **[]**, which identifies the property as a target property. A target property is the DOM property to which you want to assign a value. For example, the target property in the following code is the image element's **src** property.

src/app/app.component.html

```
<img [src]="itemImageUrl">
```

In most cases, the target name is the name of a property, even when it appears to be the name of an attribute. In this example, **src** is the name of the **<img>** element property.

The brackets, **[]**, cause Angular to evaluate the right-hand side of the assignment as a dynamic expression. Without the brackets, Angular treats the right-hand side as a string literal and sets the property to that static value.

src/app.component.html

```
<app-item-detail childItem="parentItem"></app-item-detail>
```

🕮 5.4.3

Setting an element property to a component property value

To bind the **src** property of an **<img>** element to a component's property, place the target, **src**, in square brackets followed by an equal sign and then the property. The property here is **itemImageUrl**.

src/app/app.component.html

```
<img [src]="itemImageUrl">
```

Declare the **itemImageUrl** property in the class, in this case **AppComponent**.

src/app/app.component.ts

```
itemImageUrl = '../assets/phone.png';
```

colspan and colSpan

A common point of confusion is between the attribute, **colspan**, and the property, **colSpan**. Notice that these two names differ by only a single letter.

If you wrote something like this:

```
<tr><td colspan="{{1 + 1}}">Three-Four</td></tr>
```

You'd get this error:

```
Template parse errors:
Can't bind to 'colspan' because it isn't a known built-in
property
```

As the message says, the **<td>** element does not have a **colspan** property. This is true because **colspan** is an attribute—**colSpan**, with a capital **S**, is the corresponding property. Interpolation and property binding can set only *properties*, not attributes.

Instead, you'd use property binding and write it like this:

src/app/app.component.html

```
<!-- Notice the colSpan property is camel case -->
<tr><td [colSpan]="1 + 1">Three-Four</td></tr>
```

Another example is disabling a button when the component says that it **isUnchanged**:

src/app/app.component.html

```
<!-- Bind button disabled state to `isUnchanged` property -->
<button [disabled]="isUnchanged">Disabled Button</button>
```

Another is setting a property of a directive:

src/app/app.component.html

```
<p [ngClass]="classes">[ngClass] binding to the classes
property making this blue</p>
```

Yet another is setting the model property of a custom component—a great way for parent and child components to communicate:

src/app/app.component.html

```
<app-item-detail [childItem]="parentItem"></app-item-detail>
```

🕮 5.4.4

Toggling button functionality

To disable a button's functionality depending on a Boolean value, bind the DOM **disabled** property to a property in the class that is **true** or **false**.

src/app/app.component.html

```
<!-- Bind button disabled state to `isUnchanged` property -->
<button [disabled]="isUnchanged">Disabled Button</button>
```

Because the value of the property **isUnchanged** is **true** in the **AppComponent**, Angular disables the button.

src/app/app.component.ts

```
isUnchanged = true;
```

🕮 5.4.5

Setting a directive property

To set a property of a directive, place the directive within square brackets , such as **[ngClass]**, followed by an equal sign and the property. Here, the property is **classes**.

src/app/app.component.html

```
<p [ngClass]="classes">[ngClass] binding to the classes
property making this blue</p>
```

To use the property, you must declare it in the class, which in this example is **AppComponent**. The value of **classes** is **special**.

src/app/app.component.ts

```
classes = 'special';
```

Angular applies the class **special** to the **<p>** element so that you can use **special** to apply CSS styles.

🕮 5.4.6

Bind values between components

To set the model property of a custom component, place the target, here **childItem**, between square brackets **[]** followed by an equal sign and the property. Here, the property is **parentItem**.

src/app/app.component.html

```
<app-item-detail [childItem]="parentItem"></app-item-detail>
```

To use the target and the property, you must declare them in their respective classes.

Declare the target of **childItem** in its component class, in this case **ItemDetailComponent**.

For example, the following code declares the target of **childItem** in its component class, in this case **ItemDetailComponent**.

Then, the code contains an **@Input()** decorator with the **childItem** property so data can flow into it.

src/app/item-detail/item-detail.component.ts

```
@Input() childItem = '';
```

Next, the code declares the property of **parentItem** in its component class, in this case **AppComponent**. In this example the type of **childItem** is **string**, so **parentItem** needs to be a string. Here, **parentItem** has the string value of **lamp**.

src/app/app.component.ts

```
parentItem = 'lamp';
```

With this configuration, the view of **<app-item-detail>** uses the value of **lamp** for **childItem**.

🕮 5.4.7

Property binding and security

Property binding can help keep content secure. For example, consider the following malicious content.

src/app/app.component.ts

```
evilTitle = 'Template <script>alert("evil never
sleeps")</script> Syntax';
```

The component template interpolates the content as follows:

src/app/app.component.html

```
<p><span>"{{evilTitle}}" is the <i>interpolated</i> evil
title.</span></p>
```

The browser doesn't process the HTML and instead displays it raw, as follows.

```
"Template <script>alert("evil never sleeps")</script> Syntax"
is the interpolated evil title.
```

Angular does not allow HTML with **<script>** tags, neither with interpolation nor property binding, which prevents the JavaScript from running.

In the following example, however, Angular sanitizes the values before displaying them.

src/app/app.component.html

```
<!--
 Angular generates a warning for the following line as it
sanitizes them
 WARNING: sanitizing HTML stripped some content (see
https://g.co/ng/security#xss).
-->
 <p>"<span [innerHTML]="evilTitle"></span>" is the <i>property
bound</i> evil title.</p>
```

Interpolation handles the **<script>** tags differently than property binding, but both approaches render the content harmlessly. The following is the browser output of the sanitized **evilTitle** example.

```
"Template Syntax" is the property bound evil title.
```

🕮 5.4.8

Property binding and interpolation

Often interpolation and property binding can achieve the same results. The following binding pairs do the same thing.

src/app/app.component.html

```
<p><img src="{{itemImageUrl}}"> is the <i>interpolated</i>
image.</p>
<p><img [src]="itemImageUrl"> is the <i>property bound</i>
image.</p>
```

```
<p><span>"{{interpolationTitle}}" is the <i>interpolated</i>
title.</span></p>
<p>"<span [innerHTML]="propertyTitle"></span>" is the
<i>property bound</i> title.</p>
```

Use either form when rendering data values as strings, though interpolation is preferable for readability. However, when setting an element property to a non-string data value, you must use property binding.

5.5 Attribute, class, and style bindings

🕮 5.5.1

Attribute, class, and style bindings

Attribute binding in Angular helps you set values for attributes directly. With attribute binding, you can improve accessibility, style your application dynamically, and manage multiple CSS classes or styles simultaneously.

🕮 5.5.2

Binding to an attribute

It is recommended that you set an element property with a property binding whenever possible. However, sometimes you don't have an element property to bind. In those situations, use attribute binding.

For example, ARIA and SVG are purely attributes. Neither ARIA nor SVG correspond to element properties and don't set element properties. In these cases, you must use attribute binding because there are no corresponding property targets.

🕮 5.5.3

Syntax

Attribute binding syntax resembles property binding, but instead of an element property between brackets, you precede the name of the attribute with the prefix **attr**, followed by a dot. Then, you set the attribute value with an expression that resolves to a string.

```
<p [attr.attribute-you-are-targeting]="expression"></p>
```

🕮 5.5.4

Binding ARIA attributes

One of the primary use cases for attribute binding is to set ARIA attributes, as in this example:

src/app/app.component.html

```
<!-- create and set an aria attribute for assistive technology
-->
<button [attr.aria-label]="actionName">{{actionName}} with
Aria</button>
```

🕮 5.5.5

Binding to **colspan**

Another common use case for attribute binding is with the **colspan** attribute in tables. Binding to the **colspan** attribute helps you keep your tables programmatically dynamic. Depending on the amount of data that your application populates a table with, the number of columns that a row spans could change.

To use attribute binding with the **<td>** attribute **colspan**:

1. Specify the **colspan** attribute by using the following syntax: **[attr.colspan]**.
2. Set **[attr.colspan]** equal to an expression.

In the following example, you bind the **colspan** attribute to the expression **1 + 1**.

src/app/app.component.html

```
<!--  expression calculates colspan=2 -->
<tr><td [attr.colspan]="1 + 1">One-Two</td></tr>
```

🕮 5.5.6

Binding to the class attribute

Use class binding to add and remove CSS class names from an element's **class** attribute.

Binding to a single CSS class

To create a single class binding, use the prefix **class** followed by a dot and the name of the CSS class—for example, **[class.sale]="onSale"**. Angular adds the class when the bound expression, **onSale** is truthy, and it removes the class when the expression is falsy—with the exception of **undefined**. See styling delegation for more information.

Binding to multiple CSS classes

To bind to multiple classes, use **[class]** set to an expression—for example, **[class]="classExpression"**. The expression can be one of:

- A space-delimited string of class names.
- An object with class names as the keys and truthy or falsy expressions as the values.
- An array of class names.

With the object format, Angular adds a class only if its associated value is truthy.

🕮 5.5.7

Injecting attribute values

There are cases where you need to differentiate the behavior of a Component or Directive based on a static value set on the host element as an HTML attribute. For example, you might have a directive that needs to know the **type** of a **<button>** or **<input>** element.

The Attribute parameter decorator is great for passing the value of an HTML attribute to a component/directive constructor using dependency injection.

src/app/my-input-with-attribute-decorator.component.ts

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

@Component({
  selector: 'app-my-input-with-attribute-decorator',
  template: '<p>The type of the input is: {{ type }}</p>'
})
export class MyInputWithAttributeDecoratorComponent {
  constructor(@Attribute('type') public type: string) { }
}
```

src/app/app.component.html

```
<app-my-input-with-attribute-decorator type="number"></app-my-input-with-attribute-decorator>
```

In the preceding example, the result of **app.component.html** is The type of the input is: number.

Another example is the RouterOutlet directive, which makes use of the Attribute decorator to retrieve the unique name on each outlet.

@ATTRIBUTE() VS @INPUT()

Remember, use @Input() when you want to keep track of the attribute value and update the associated property. Use @Attribute() when you want to inject the value of an HTML attribute to a component or directive constructor.

5.6 Event binding

🕮 5.6.1

Event binding

Event binding lets you listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.

🕮 5.6.2

Binding to events

To bind to an event you use the Angular event binding syntax. This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right. In the following example, the target event name is **click** and the template statement is **onSave()**.

Event binding syntax

```
<button (click)="onSave()">Save</button>
```

The event binding listens for the button's click events and calls the component's **onSave()** method whenever a click occurs.

🕮 5.6.3

Binding to passive events

Angular also supports passive event listeners. For example, use the following steps to make a scroll event passive.

1. Create a file **zone-flags.ts** under **src** directory.
2. Add the following line into this file.

```
(window as any)['__zone_symbol__PASSIVE_EVENTS'] = ['scroll'];
```

1. In the **src/polyfills.ts** file, before importing zone.js, import the newly created **zone-flags**.

```
import './zone-flags';
import 'zone.js';  // Included with Angular CLI.
```

After those steps, if you add event listeners for the **scroll** event, the listeners will be **passive**.

🕮 5.6.4

Custom events with EventEmitter

Directives typically raise custom events with an Angular EventEmitter as follows.

1. The directive creates an **EventEmitter** and exposes it as a property.
2. The directive then calls **EventEmitter.emit(data)** to emit an event, passing in message data, which can be anything.
3. Parent directives listen for the event by binding to this property and accessing the data through the **$event** object.

Consider an **ItemDetailComponent** that presents item information and responds to user actions. Although the **ItemDetailComponent** has a delete button, it doesn't contain the functionality to delete the hero. It can only raise an event reporting the user's delete request.

src/app/item-detail/item-detail.component.html (template)

```
<img src="{{itemImageUrl}}" [style.display]="displayNone">
<span [style.text-decoration]="lineThrough">{{ item.name }}
</span>
<button (click)="delete()">Delete</button>
```

The component defines a **deleteRequest** property that returns an **EventEmitter**. When the user clicks Delete, the component invokes the **delete()** method, telling the **EventEmitter** to emit an **Item** object.

src/app/item-detail/item-detail.component.ts (deleteRequest)

```
// This component makes a request but it can't actually delete
a hero.
@Output() deleteRequest = new EventEmitter<Item>();

delete() {
  this.deleteRequest.emit(this.item);
  this.displayNone = this.displayNone ? '' : 'none';
  this.lineThrough = this.lineThrough ? '' : 'line-through';
}
```

The hosting parent component binds to the **deleteRequest** event of the **ItemDetailComponent** as follows.

src/app/app.component.html (event-binding-to-component)

```
<app-item-detail (deleteRequest)="deleteItem($event)"
[item]="currentItem"></app-item-detail>
```

When the **deleteRequest** event fires, Angular calls the parent component's **deleteItem()** method with the item.

Determining an event target

To determine an event target, Angular checks if the name of the target event matches an event property of a known directive. In the following example, Angular checks to see if **myClick** is an event on the custom **ClickDirective**.

src/app/app.component.html

```
<h4>myClick is an event on the custom ClickDirective:</h4>
<button (myClick)="clickMessage=$event" clickable>click with
myClick</button>
{{clickMessage}}
```

If the target event name, **myClick** fails to match an element event or an output property of **ClickDirective**, Angular reports an "unknown directive" error.

5.7 Two-way binding

🕮 5.7.1

Two-way binding

Two-way binding gives components in your application a way to share data. Use two-way binding to listen for events and update values simultaneously between parent and child components.

🕮 5.7.2

Adding two-way data binding

Angular's two-way binding syntax is a combination of square brackets and parentheses, **[()]**. The **[()]** syntax combines the brackets of property binding, **[]**, with the parentheses of event binding, **()**, as follows.

src/app/app.component.html

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
```

🕮 5.7.3

How two-way binding works

For two-way data binding to work, the **@Output()** property must use the pattern, **inputChange**, where **input** is the name of the **@Input()** property. For example, if the **@Input()** property is **size**, the **@Output()** property must be **sizeChange**.

The following **sizerComponent** has a **size** value property and a **sizeChange** event. The **size** property is an **@Input()**, so data can flow into the **sizerComponent**. The **sizeChange** event is an **@Output()**, which lets data flow out of the **sizerComponent** to the parent component.

Next, there are two methods, **dec()** to decrease the font size and **inc()** to increase the font size. These two methods use **resize()** to change the value of the **size** property within min/max value constraints, and to emit an event that conveys the new **size** value.

src/app/sizer.component.ts

```
export class SizerComponent {
```

```
  @Input()  size!: number | string;
  @Output() sizeChange = new EventEmitter<number>();

  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }
}
```

The **sizerComponent** template has two buttons that each bind the click event to the **inc()** and **dec()** methods. When the user clicks one of the buttons, the **sizerComponent** calls the corresponding method. Both methods, **inc()** and **dec()**, call the **resize()** method with a **+1** or **-1**, which in turn raises the **sizeChange** event with the new size value.

src/app/sizer.component.html

```
<div>
  <button (click)="dec()" title="smaller">-</button>
  <button (click)="inc()" title="bigger">+</button>
  <label [style.font-size.px]="size">FontSize:
{{size}}px</label>
</div>
```

In the **AppComponent** template, **fontSizePx** is two-way bound to the **SizerComponent**.

src/app/app.component.html

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
<div [style.font-size.px]="fontSizePx">Resizable Text</div>
```

In the **AppComponent**, **fontSizePx** establishes the initial **SizerComponent.size** value by setting the value to **16**.

src/app/app.component.ts

```
fontSizePx = 16;
```

Clicking the buttons updates the **AppComponent.fontSizePx**. The revised **AppComponent.fontSizePx** value updates the style binding, which makes the displayed text bigger or smaller.

The two-way binding syntax is shorthand for a combination of property binding and event binding. The **SizerComponent** binding as separate property binding and event binding is as follows.

src/app/app.component.html (expanded)

```
<app-sizer [size]="fontSizePx"
(sizeChange)="fontSizePx=$event"></app-sizer>
```

The **$event** variable contains the data of the **SizerComponent.sizeChange** event. Angular assigns the **$event** value to the **AppComponent.fontSizePx** when the user clicks the buttons.

TWO-WAY BINDING IN FORMS

Because no built-in HTML element follows the **x** value and **xChange** event pattern, two-way binding with form elements requires **NgModel**.

5.8 Template variables

🕮 5.8.1

Template variablesTemplate variables help you use data from one part of a template in another part of the template. Use template variables to perform tasks such as respond to user input or finely tune your application's forms.

A template variable can refer to the following:

- a DOM element within a template
- a directive
- an element
- TemplateRef
- a web component

🕮 5.8.2

Syntax

In the template, you use the hash symbol, **#**, to declare a template variable. The following template variable, **#phone**, declares a **phone** variable on an **<input>** element.

src/app/app.component.html

```
<input #phone placeholder="phone number" />
```

Refer to a template variable anywhere in the component's template. Here, a **<button>** further down the template refers to the **phone** variable.

src/app/app.component.html

```
<input #phone placeholder="phone number" />

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

📖 5.8.3

How Angular assigns values to template variables

Angular assigns a template variable a value based on where you declare the variable:

- If you declare the variable on a component, the variable refers to the component instance.
- If you declare the variable on a standard HTML tag, the variable refers to the element.
- If you declare the variable on an **<ng-template>** element, the variable refers to a **TemplateRef** instance, which represents the template. For more information on **<ng-template>**, see How Angular uses the asterisk, *****, syntax in Structural directives.
- If the variable specifies a name on the right-hand side, such as **#var="ngModel"**, the variable refers to the directive or component on the element with a matching **exportAs** name.

Using NgForm with template variables

In most cases, Angular sets the template variable's value to the element on which it occurs. In the previous example, **phone** refers to the phone number **<input>**. The button's click handler passes the **<input>** value to the component's **callPhone()** method.

The **NgForm** directive demonstrates getting a reference to a different value by reference a directive's **exportAs** name. In the following example, the template variable, **itemForm**, appears three times separated by HTML.

src/app/hero-form.component.html

```
<form #itemForm="ngForm" (ngSubmit)="onSubmit(itemForm)">
  <label for="name">Name <input class="form-control"
name="name" ngModel required />
  </label>
  <button type="submit">Submit</button>
</form>

<div [hidden]="!itemForm.form.valid">
  <p>{{ submitMessage }}</p>
</div>
```

Without the ngForm attribute value, the reference value of **itemForm** would be the HTMLFormElement, **<form>**. There is, however, a difference between a **Component** and a **Directive** in that Angular references a **Component** without specifying the attribute value, and a **Directive** does not change the implicit reference, or the element.

With **NgForm**, **itemForm** is a reference to the NgForm directive with the ability to track the value and validity of every control in the form.

Unlike the native **<form>** element, the **NgForm** directive has a **form** property. The **NgForm** **form** property lets you disable the submit button if the **itemForm.form.valid** is invalid.

🕮 5.8.4

Template input variable

A *template input variable* is a variable to reference within a single instance of the template. You declare a template input variable using the **let** keyword as in **let hero**.

There are several such variables in this example: **hero**, **i**, and **odd**.

```
<ng-template #hero let-hero let-i="index" let-odd="isOdd">
  <div [class]="{'odd-row': odd}">{{i}}:{{hero.name}}</div>
</ng-template>
```

The variable's scope is limited to a single instance of the repeated template. Use the same variable name again in the definition of other structural directives.

In contrast, you declare a template variable by prefixing the variable name with #, as in **#var**. A template variable refers to its attached element, component, or directive.

Template input variables and template variables names have their own namespaces. The template input variable **hero** in **let hero** is distinct from the template variable **hero** in **#hero**.

5.9 Templates (Exercises)

5.9.1

What are the Best Expression Practices?

5.9.2

Statement best practices

5.9.3

To bind to an element's property, enclose it in ...

5.9.4

What to use when you want to track the value of an attribute and update the related property?

5.9.5

What to use when you want to inject the value of an HTML attribute into the constructor of a component or directive?

5.9.6

Event binding allows you to listen and respond to user actions such as keystrokes, mouse movements, clicks, and touches.

- Yes
- No

5.9.7

What Does Angular Do To Determine The Purpose Of An Event?

- Angular checks if the name of the target event matches an event property of a known directive.
- Angular checks that the name of the target event matches the event property of each directive.

5.9.8

Since no built-in HTML element is compatible with the xvalue and xChangewith event pattern, bidirectional binding to form elements requires ...

5.9.9

If you declare the variable on a standard HTML tag ...

- the variable refers to the component instance.
- the variable refers to the element.
- the variable refers to the directive or component on the element with a matching exportAs name.

Directive

Chapter 6

6.1 Built-in directives

🕮 6.1.1

Built-in directivesDirectives are classes that add additional behavior to elements in your Angular applications. With Angular's built-in directives, you can manage forms, lists, styles, and what users see.

The different types of Angular directives are as follows:

1. Components—directives with a template. This type of directive is the most common directive type.
2. Attribute directives—directives that change the appearance or behavior of an element, component, or another directive.
3. Structural directives—directives that change the DOM layout by adding and removing DOM elements.

🕮 6.1.2

Built-in attribute directives

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components.

Many NgModules such as the **RouterModule** and the **FormsModule** define their own attribute directives. The most common attribute directives are as follows:

- **NgClass**—adds and removes a set of CSS classes.
- **NgStyle**—adds and removes a set of HTML styles.
- **NgModel**—adds two-way data binding to an HTML form element.

🕮 6.1.3

Adding and removing classes with NgClass

You can add or remove multiple CSS classes simultaneously with **ngClass**.

Using NgClass with an expression

On the element you'd like to style, add **[ngClass]** and set it equal to an expression. In this case, **isSpecial** is a boolean set to **true** in **app.component.ts**.
Because **isSpecial** is true, **ngClass** applies the class of **special** to the ** **.

src/app/app.component.html

```
<!-- toggle the "special" class on/off with a property -->
<div [ngClass]="isSpecial ? 'special' : ''">This div is
special</div>
```

Using NgClass with a methodlink

1. To use NgClass with a method, add the method to the component class. In the following example, **setCurrentClasses()** sets the property **currentClasses** with an object that adds or removes three classes based on the **true** or **false** state of three other component properties.
2. Each key of the object is a CSS class name. If a key is **true**, ngClass adds the class. If a key is **false**, ngClass removes the class.
3. src/app/app.component.ts

```
currentClasses: Record<string, boolean> = {};
/* . . . */
  setCurrentClasses() {
    // CSS classes: added/removed per current state of
component properties
    this.currentClasses =  {
      saveable: this.canSave,
      modified: !this.isUnchanged,
      special:  this.isSpecial
    };
  }
```

1. In the template, add the ngClass property binding to **currentClasses** to set the element's classes:
2. src/app/app.component.html

```
<div [ngClass]="currentClasses">This div is initially
saveable, unchanged, and special.</div>
```

For this use case, Angular applies the classes on initialization and in case of changes. The full example calls **setCurrentClasses()** initially with **ngOnInit()** and when the dependent properties change through a button click. These steps are not necessary to implement ngClass.

🕮 6.1.4

Setting inline styles with NgStyle

You can use **NgStyle** to set multiple inline styles simultaneously, based on the state of the component.

1. To use **NgStyle**, add a method to the component class.
2. In the following example, **setCurrentStyles()** sets the property **currentStyles** with an object that defines three styles, based on the state of three other component properties.
3. src/app/app.component.ts

```
currentStyles: Record<string, string> = {};
/* . . . */
  setCurrentStyles() {
    // CSS styles: set per current state of component
properties
    this.currentStyles = {
      'font-style':  this.canSave      ? 'italic' : 'normal',
      'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
      'font-size':   this.isSpecial    ? '24px'   : '12px'
    };
  }
```

1. To set the element's styles, add an **ngStyle** property binding to **currentStyles**.
2. src/app/app.component.html

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large
(24px).
</div>
```

For this use case, Angular applies the styles upon initialization and in case of changes. To do this, the full example calls **setCurrentStyles()** initially with **ngOnInit()** and when the dependent properties change through a button click. However, these steps are not necessary to implement **ngStyle** on its own.

🕮 6.1.5

Displaying and updating properties with ngModel

You can use the **NgModel** directive to display a data property and update that property when the user makes changes.

1. Import **FormsModule** and add it to the NgModule's **imports** list.
2. src/app/app.module.ts (FormsModule import)

```
import { FormsModule } from '@angular/forms'; // <---
JavaScript import from Angular
/* . . . */
@NgModule({
/* . . . */

  imports: [
    BrowserModule,
    FormsModule // <--- import into the NgModule
  ],
/* . . . */
})
export class AppModule { }
```

1. Add an **[(ngModel)]** binding on an HTML **<form>** element and set it equal to the property, here **name**.
2. src/app/app.component.html (NgModel example)

```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

1. This **[(ngModel)]** syntax can only set a data-bound property.

To customize your configuration, you can write the expanded form, which separates the property and event binding. Use property binding to set the property and event binding to respond to changes. The following example changes the **<input>** value to uppercase:

src/app/app.component.html

```
<input [ngModel]="currentItem.name"
(ngModelChange)="setUppercaseName($event)" id="example-
uppercase">
```

NgModel and value accessors

The **NgModel** directive works for an element supported by a ControlValueAccessor. Angular provides *value accessors* for all of the basic HTML form elements. For more information, see Forms.

To apply [(ngModel)] to a non-form native element or a third-party custom component, you have to write a value accessor.

📖 6.1.6

Built-in structural directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

This section introduces the most common built-in structural directives:

- **NgIf**—conditionally creates or disposes of subviews from the template.
- **NgFor**—repeat a node for each item in a list.
- **NgSwitch**—a set of directives that switch among alternative views.

📖 6.1.7

Adding or removing an element with NgIf

You can add or remove an element by applying an **NgIf** directive to a host element.

When **NgIf** is **false**, Angular removes an element and its descendants from the DOM. Angular then disposes of their components, which frees up memory and resources.

To add or remove an element, bind ***ngIf** to a condition expression such as **isActive** in the following example.

src/app/app.component.html

```
<app-item-detail *ngIf="isActive" [item]="item"></app-item-detail>
```

When the **isActive** expression returns a truthy value, **NgIf** adds the **ItemDetailComponent** to the DOM. When the expression is falsy, **NgIf** removes the **ItemDetailComponent** from the DOM and disposes of the component and all of its sub-components.

📖 6.1.8

Listing items with NgFor

You can use the **NgFor** directive to present a list of items.

1. Define a block of HTML that determines how Angular renders a single item.
2. To list your items, assign the short hand **let item of items** to ***ngFor**.

src/app/app.component.html

```
<div *ngFor="let item of items">{{item.name}}</div>
```

The string "**let item of items**" instructs Angular to do the following:

- Store each item in the **items** array in the local **item** looping variable
- Make each item available to the templated HTML for each iteration
- Translate "**let item of items**" into an **<ng-template>** around the host element
- Repeat the **<ng-template>** for each **item** in the list
-

Repeating a component view

To repeat a component element, apply ***ngFor** to the selector. In the following example, the selector is **<app-item-detail>**.

src/app/app.component.html

```
<app-item-detail *ngFor="let item of items"
[item]="item"></app-item-detail>
```

You can reference a template input variable, such as **item**, in the following locations:

- within the **ngFor** host element
- within the host element descendants to access the item's properties

The following example references **item** first in an interpolation and then passes in a binding to the **item** property of the **<app-item-detail>** component.

src/app/app.component.html

```
<div *ngFor="let item of items">{{item.name}}</div>
<!-- . . . -->
  <app-item-detail *ngFor="let item of items"
[item]="item"></app-item-detail>
```

🕮 6.1.9

Switching cases with NgSwitch

Like the JavaScript **switch** statement, NgSwitch displays one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.

NgSwitch is a set of three directives:

- NgSwitch—an attribute directive that changes the behavior of its companion directives.
- NgSwitchCase—structural directive that adds its element to the DOM when its bound value equals the switch value and removes its bound value when it doesn't equal the switch value.
- NgSwitchDefault—structural directive that adds its element to the DOM when there is no selected NgSwitchCase.

1. On an element, such as a ** **, add **[**ngSwitch**]** bound to an expression that returns the switch value, such as **feature**. Though the **feature** value in this example is a string, the switch value can be of any type.
2. Bind to ***ngSwitchCase** and ***ngSwitchDefault** on the elements for the cases.
3. src/app/app.component.html

```
<div [ngSwitch]="currentItem.feature">
  <app-stout-item    *ngSwitchCase="'stout'"
[item]="currentItem"></app-stout-item>
  <app-device-item   *ngSwitchCase="'slim'"
[item]="currentItem"></app-device-item>
  <app-lost-item     *ngSwitchCase="'vintage'"
[item]="currentItem"></app-lost-item>
  <app-best-item     *ngSwitchCase="'bright'"
[item]="currentItem"></app-best-item>
<!-- . . . -->
  <app-unknown-item  *ngSwitchDefault
[item]="currentItem"></app-unknown-item>
</div>
```

1. In the parent component, define **currentItem** so you can use it in the **[**ngSwitch**]** expression.
2. src/app/app.component.ts

```
currentItem!: Item;
```

1. In each child component, add an **item** input property which is bound to the **currentItem** of the parent component. The following two snippets show

the parent component and one of the child components. The other child components are identical to **StoutItemComponent**.
2. In each child component, here StoutItemComponent

```
export class StoutItemComponent {
  @Input() item!: Item;
}
```

Switch directives also work with native HTML elements and web components. For example, you could replace the **<app-best-item>** switch case with a ** ** as follows.

src/app/app.component.html

```
<div *ngSwitchCase="'bright'"> Are you as bright as
{{currentItem.name}}?</div>
```

6.2 Attribute directives

🕮 6.2.1

Attribute directives

With attribute directives, you can change the appearance or behavior of DOM elements and Angular components.

🕮 6.2.2

Building an attribute directive

This section walks you through creating a highlight directive that sets the background color of the host element to yellow.

1. To create a directive, use the CLI command **ng generate directive**.

```
ng generate directive highlight
```

1. The CLI creates **src/app/highlight.directive.ts**, a corresponding test file **src/app/highlight.directive.spec.ts**, and declares the directive class in the **AppModule**.
2. The CLI generates the default **src/app/highlight.directive.ts** as follows:
3.
4.

5. src/app/highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

1. The **@Directive()** decorator's configuration property specifies the directive's CSS attribute selector, **[appHighlight]**.
2. Import **ElementRef** from **@angular/core**. **ElementRef** grants direct access to the host DOM element through its **nativeElement** property.
3. Add **ElementRef** in the directive's **constructor()** to inject a reference to the host DOM element, the element to which you apply **appHighlight**.
4. Add logic to the **HighlightDirective** class that sets the background to yellow.
5. src/app/highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

🕮 6.2.3

Applying an attribute directive

1. To use the **HighlightDirective**, add a **<p>** element to the HTML template with the directive as an attribute.
2. src/app/app.component.html

```
<p appHighlight>Highlight me!</p>
```

Angular creates an instance of the **HighlightDirective** class and injects a reference to the **<p>** element into the directive's constructor, which sets the **<p>** element's background style to yellow.

🕮 6.2.4

Handling user events

This section shows you how to detect when a user mouses into or out of the element and to respond by setting or clearing the highlight color.

1. Import **HostListener** from '@angular/core'.
2. src/app/highlight.directive.ts (imports)

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

1. Add two event handlers that respond when the mouse enters or leaves, each with the **@HostListener()** decorator.
2. src/app/highlight.directive.ts (mouse-methods)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight('');
}

private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```

1. With the **@HostListener()** decorator, you can subscribe to events of the DOM element that hosts an attribute directive, the **<p>** in this case.
2. The handlers delegate to a helper method, **highlight()**, that sets the color on the host DOM element, **el**.

The complete directive is as follows:

src/app/highlight.directive.ts

```
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) { }

  @HostListener('mouseenter') onMouseEnter() {
```

```
    this.highlight('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight('');
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }

}
```

🕮 6.2.5

Passing values into an attribute directive

This section walks you through setting the highlight color while applying the **HighlightDirective**.

1. In **highlight.directive.ts**, import **Input** from **@angular/core**.
2. src/app/highlight.directive.ts (imports)

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

1. Add an **appHighlight** **@Input()** property.
2. src/app/highlight.directive.ts

```
@Input() appHighlight = '';
```

1. The **@Input()** decorator adds metadata to the class that makes the directive's **appHighlight** property available for binding.
2. In **app.component.ts**, add a **color** property to the **AppComponent**.
3. src/app/app.component.ts (class)

```
export class AppComponent {
  color = 'yellow';
}
```

1. To simultaneously apply the directive and the color, use property binding with the **appHighlight** directive selector, setting it equal to **color**.
2. src/app/app.component.html (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

1. The **[appHighlight]** attribute binding performs two tasks:

- applies the highlighting directive to the **<p>** element
- sets the directive's highlight color with a property binding

Setting the value with user input

This section guides you through adding radio buttons to bind your color choice to the **appHighlight** directive.

1. Add markup to **app.component.html** for choosing a color as follows:
2. src/app/app.component.html (v2)

```
<h1>My First Attribute Directive</h1>

<h2>Pick a highlight color</h2>
<div>
  <input type="radio" name="colors"
(click)="color='lightgreen'">Green
  <input type="radio" name="colors"
(click)="color='yellow'">Yellow
  <input type="radio" name="colors"
(click)="color='cyan'">Cyan
</div>
<p [appHighlight]="color">Highlight me!</p>
```

1. Revise the **AppComponent.color** so that it has no initial value.
2. src/app/app.component.ts (class)

```
export class AppComponent {
  color = '';
}
```

🕮 6.2.6

Binding to a second property

This section guides you through configuring your application so the developer can set the default color.

1. Add a second **Input()** property to **HighlightDirective** called **defaultColor**.
2. src/app/highlight.directive.ts (defaultColor)

```
@Input() defaultColor = '';
```

1. Revise the directive's **onMouseEnter** so that it first tries to highlight with the **highlightColor**, then with the **defaultColor**, and falls back to **red** if both properties are **undefined**.
2. src/app/highlight.directive.ts (mouse-enter)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || this.defaultColor ||
'red');
}
```

1. To bind to the **AppComponent.color** and fall back to "violet" as the default color, add the following HTML. In this case, the **defaultColor** binding doesn't use square brackets, **[]**, because it is static.
2. src/app/app.component.html (defaultColor)

```
<p [appHighlight]="color" defaultColor="violet">
  Highlight me too!
</p>
```

1. As with components, you can add multiple directive property bindings to a host element.

🕮 6.2.7

Deactivating Angular processing with NgNonBindable

To prevent expression evaluation in the browser, add **ngNonBindable** to the host element. **ngNonBindable** deactivates interpolation, directives, and binding in templates.

In the following example, the expression **{{ 1 + 1 }}** renders just as it does in your code editor, and does not display **2**.

src/app/app.component.html

```
<p>Use ngNonBindable to stop evaluation.</p>
<p ngNonBindable>This should not evaluate: {{ 1 + 1 }}</p>
```

Applying **ngNonBindable** to an element stops binding for that element's child elements. However, **ngNonBindable** still allows directives to work on the element where you apply **ngNonBindable**. In the following example,
the **appHighlight** directive is still active but Angular does not evaluate the expression **{{ 1 + 1 }}**.

src/app/app.component.html

```
<h3>ngNonBindable with a directive</h3>
<div ngNonBindable [appHighlight]="'yellow'">This should not
evaluate: {{ 1 +1 }}, but will highlight yellow.
</div>
```

If you apply **ngNonBindable** to a parent element, Angular disables interpolation and binding of any sort, such as property binding or event binding, for the element's children.

6.3 Structural Directives

🕮 6.3.1

Creating a structural directive

This section guides you through creating an **UnlessDirective** and how to set **condition** values. The **UnlessDirective** does the opposite of **NgIf**, and **condition** values can be set to **true** or **false**. **NgIf** displays the template content when the condition is **true**. **UnlessDirective** displays the content when the condition is **false**.

Following is the **UnlessDirective** selector, **appUnless**, applied to the paragraph element. When **condition** is **false**, the browser displays the sentence.

src/app/app.component.html (appUnless-1)

```
<p *appUnless="condition">Show this sentence unless the
condition is true.</p>
```

1. Using the Angular CLI, run the following command, where **unless** is the name of the directive:

```
ng generate directive unless
```

1. Angular creates the directive class and specifies the CSS selector, **appUnless**, that identifies the directive in a template.
2. Import **Input**, **TemplateRef**, and **ViewContainerRef**.
3. src/app/unless.directive.ts (skeleton)

```
import { Directive, Input, TemplateRef, ViewContainerRef }
from '@angular/core';

@Directive({ selector: '[appUnless]'})
export class UnlessDirective {
}
```

1. Inject **TemplateRef** and **ViewContainerRef** in the directive constructor as private variables.
2. src/app/unless.directive.ts (ctor)

```
constructor(
  private templateRef: TemplateRef<any>,
  private viewContainer: ViewContainerRef) { }
```

1. The **UnlessDirective** creates an embedded view from the Angular-generated **<ng-template>** and inserts that view in a view container adjacent to the directive's original **<p>** host element.
2. **TemplateRef** helps you get to the **<ng-template>** contents and **ViewContainerRef** accesses the view container.
3. Add an **appUnless** **@Input()** property with a setter.
4. src/app/unless.directive.ts (set)

```
@Input() set appUnless(condition: boolean) {
  if (!condition && !this.hasView) {
    this.viewContainer.createEmbeddedView(this.templateRef);
    this.hasView = true;
  } else if (condition && this.hasView) {
    this.viewContainer.clear();
    this.hasView = false;
  }
}
```

1. Angular sets the **appUnless** property whenever the value of the condition changes.

- If the condition is falsy and Angular hasn't created the view previously, the setter causes the view container to create the embedded view from the template.
- If the condition is truthy and the view is currently displayed, the setter clears the container, which disposes of the view.

The complete directive is as follows:

src/app/unless.directive.ts (excerpt)

```
import { Directive, Input, TemplateRef, ViewContainerRef }
from '@angular/core';

/**
 * Add the template content to the DOM unless the condition is
true.
 */
@Directive({ selector: '[appUnless]'})
```

```
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set appUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}
```

🕮 6.3.2

Testing the directive

In this section, you'll update your application to test the **UnlessDirective**.

1. Add a **condition** set to **false** in the **AppComponent**.
2. src/app/app.component.ts (excerpt)

```
condition = false;
```

1. Update the template to use the directive. Here, ***appUnless** is on two **<p>** tags with opposite **condition** values, one **true** and one **false**.
2. src/app/app.component.html (appUnless)

```
<p *appUnless="condition" class="unless a">
  (A) This paragraph is displayed because the condition is
false.
</p>

<p *appUnless="!condition" class="unless b">
  (B) Although the condition is true,
  this paragraph is displayed because appUnless is set to
false.
</p>
```

1. The asterisk is shorthand that marks **appUnless** as a structural directive. When the **condition** is falsy, the top (A) paragraph appears and the bottom (B) paragraph disappears. When the **condition** is truthy, the top (A) paragraph disappears and the bottom (B) paragraph appears.
2. To change and display the value of **condition** in the browser, add markup that displays the status and a button.
3. src/app/app.component.html

```
<p>
  The condition is currently
  <span [ngClass]="{ 'a': !condition, 'b': condition,
'unless': true }">{{condition}}</span>.
  <button
    (click)="condition = !condition"
    [ngClass] = "{ 'a': condition, 'b': !condition }" >
    Toggle condition to {{condition ? 'false' : 'true'}}
  </button>
</p>
```

6.4 Directives (Exercises)

6.4.1

conditionally creates or disposes of subviews from the template -

6.4.2

repeat a node for each item in a list -

6.4.3

a set of directives that switch among alternative views -

6.4.4

With attribute directives, you can change the ... of DOM elements and Angular components.

6.4.5

Which command to use to create a CLI command directive?

Dependency Injection

Chapter 7

7.1 Dependency injection

🕮 7.1.1

Injecting dependencies in Angular

Dependencies are services or objects that a class needs to perform its function. Dependency injection or DI is a design pattern in which a class requests dependencies from external sources rather than creating them.

Angular's DI framework provides dependencies to a class when an instance is created. You can use Angular DI to increase flexibility and modularity in your applications.

🕮 7.1.2

Creating an injectable service

To generate a new **HeroService** class in the **src/app/heroes** folder use the following Angular CLI command.

```
ng generate service heroes/hero
```

This command creates the following default **HeroService**.

src/app/heroes/hero.service.ts (CLI-generated)

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {
  constructor() { }
}
```

The **@Injectable()** decorator specifies that Angular can use this class in the DI system. The metadata, **providedIn: 'root'**, means that the **HeroService** is visible throughout the application.

Next, to get the hero mock data, add a **getHeroes()** method that returns the heroes from **mock.heroes.ts**.

src/app/heroes/hero.service.ts

```
import { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';

@Injectable({
  // declares that this service should be created
  // by the root application injector.
  providedIn: 'root',
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

For clarity and maintainability, it is recommended that you define components and services in separate files.

If you do combine a component and service in the same file, it is important to define the service first, and then the component. If you define the component before the service, Angular returns a run-time null reference error.

🕮 7.1.3

Injecting services

Injecting services results in making them visible to a component.

To inject a dependency in a component's **constructor()**, supply a constructor argument with the dependency type. The following example specifies the **HeroService** in the **HeroListComponent** constructor. The type of **heroService** is **HeroService**.

src/app/heroes/hero-list.component (constructor signature)

```
constructor(heroService: HeroService)
```

7.2 DI Providers

🕮 7.2.1

Dependency providers

By configuring providers, you can make services available to the parts of your application that need them.

A dependency provider configures an injector with a DI token, which that injector uses to provide the runtime version of a dependency value.

🕮 7.2.2

Specifying a provider token

If you specify the service class as the provider token, the default behavior is for the injector to instantiate that class with **new**.

In the following example, the **Logger** class provides a **Logger** instance.

```
providers: [Logger]
```

You can, however, configure an injector with an alternative provider in order to deliver some other object that provides the needed logging functionality.

You can configure an injector with a service class, you can provide a substitute class, an object, or a factory function.

🕮 7.2.3

Dependency injection tokens

When you configure an injector with a provider, you are associating that provider with a dependency injection token, or DI token. The injector allows Angular to create a map of any internal dependencies. The DI token acts as a key to that map.

The dependency value is an instance, and the class type serves as a lookup key. Here, the injector uses the **HeroService** type as the token for looking up **heroService**.

src/app/injector.component.ts

```
heroService: HeroService;
```

When you define a constructor parameter with the **HeroService** class type, Angular knows to inject the service associated with that **HeroService** class token:

src/app/heroes/hero-list.component.ts

```
constructor(heroService: HeroService)
```

Though classes provide many dependency values, the expanded **provide** object lets you associate different kinds of providers with a DI token.

🕮 7.2.4

Defining providers

The class provider syntax is a shorthand expression that expands into a provider configuration, defined by the **Provider** interface. The following example is the class provider syntax for providing a **Logger** class in the **providers** array.

```
providers: [Logger]
```

Angular expands the **providers** value into a full provider object as follows.

```
[{ provide: Logger, useClass: Logger }]
```

The expanded provider configuration is an object literal with two properties:

- The **provide** property holds the token that serves as the key for both locating a dependency value and configuring the injector.
- The second property is a provider definition object, which tells the injector how to create the dependency value. The provider-definition key can be **useClass**, as in the example. It can also be **useExisting**, **useValue**, or **useFactory**. Each of these keys provides a different type of dependency, as discussed below.

🕮 7.2.5

Specifying an alternative class provider

Different classes can provide the same service. For example, the following code tells the injector to return a **BetterLogger** instance when the component asks for a logger using the **Logger** token.

```
[{ provide: Logger, useClass: BetterLogger }]
```

Configuring class providers with dependencies

If the alternative class providers have their own dependencies, specify both providers in the **providers** metadata property of the parent module or component.

```
[ UserService,
```

```
  { provide: Logger, useClass: EvenBetterLogger }]
```

In this example, **EvenBetterLogger** displays the user name in the log message. This logger gets the user from an injected **UserService** instance.

```
@Injectable()
export class EvenBetterLogger extends Logger {
  constructor(private userService: UserService) { super(); }

  log(message: string) {
    const name = this.userService.user.name;
    super.log(`Message to ${name}: ${message}`);
  }
}
```

The injector needs providers for both this new logging service and its dependent **UserService**.

Aliasing class providers

To alias a class provider, specify the alias and the class provider in the **providers** array with the **useExisting** property.

In the following example, the injector injects the singleton instance of **NewLogger** when the component asks for either the new or the old logger. In this way, **OldLogger** is an alias for **NewLogger**.

```
[ NewLogger,
  // Alias OldLogger w/ reference to NewLogger
  { provide: OldLogger, useExisting: NewLogger}]
```

Be sure you don't alias **OldLogger** to **NewLogger** with **useClass**, as this creates two different **NewLogger** instances.

🕮 7.2.6

Injecting an object

To inject an object, configure the injector with the **useValue** option. The following provider object uses the **useValue** key to associate the variable with the **Logger** token.

```
[{ provide: Logger, useValue: SilentLogger }]
```

In this example, **SilentLogger** is an object that fulfills the logger role.

```
// An object in the shape of the logger service
function silentLoggerFn() {}

export const SilentLogger = {
  logs: ['Silent logger says "Shhhhh!". Provided via
"useValue"'],
  log: silentLoggerFn
};
```

Injecting a configuration object

A common use case for object literals is a configuration object. The following configuration object includes the title of the application and the address of a web API endpoint.

src/app/app.config.ts (excerpt)

```
export const HERO_DI_CONFIG: AppConfig = {
  apiEndpoint: 'api.heroes.com',
  title: 'Dependency Injection'
};
```

To provide and inject the configuration object, specify the object in the **@NgModule() providers** array.

src/app/app.module.ts (providers)

```
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
```

Using an InjectionToken object

You can define and use an **InjectionToken** object for choosing a provider token for non-class dependencies. The following example defines a token, **APP_CONFIG** of the type **InjectionToken**.

src/app/app.config.ts

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

export const APP_CONFIG = new
InjectionToken<AppConfig>('app.config');
```

The optional type parameter, **<AppConfig>**, and the token description, **app.config**, specify the token's purpose.

Next, register the dependency provider in the component using the **InjectionToken** object of **APP_CONFIG**.

src/app/providers.component.ts

```
providers: [{ provide: APP_CONFIG, useValue: HERO_DI_CONFIG }]
```

Now you can inject the configuration object into the constructor with **@Inject()** parameter decorator.

src/app/app.component.ts

```
constructor(@Inject(APP_CONFIG) config: AppConfig) {
  this.title = config.title;
}
```

Interfaces and dependency injection

Though the TypeScript **AppConfig** interface supports typing within the class, the **AppConfig** interface plays no role in dependency injection. In TypeScript, an interface is a design-time artifact, and doesn't have a runtime representation, or token, that the DI framework can use.

When the transpiler changes TypeScript to JavaScript, the interface disappears because JavaScript doesn't have interfaces.

Since there is no interface for Angular to find at runtime, the interface cannot be a token, nor can you inject it.

```
// Can't use interface as provider token
[{ provide: AppConfig, useValue: HERO_DI_CONFIG })]
content_copy
// Can't inject using the interface as the parameter type
constructor(private config: AppConfig){ }
```

7.3 Dependency injection (Exercises)

7.3.1

What is recommended to ensure transparency and ease of maintenance?

7.3.2

If you define the component before the service ...

7.3.3

A dependency provider configures an injector with a DI token, which that injector uses to provide the runtime version of a dependency value.

- True
- False

Forms

Chapter 8

8.1 Forms

🕮 8.1.1

Choosing an approach

Reactive forms and template-driven forms process and manage form data differently. Each approach offers different advantages.

Reactive forms provide direct, explicit access to the underlying forms object model. Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

Template-driven forms rely on directives in the template to create and manipulate the underlying object model. They are useful for adding a simple form to an app, such as an email list signup form. They're easy to add to an app, but they don't scale as well as reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, template-driven forms could be a good fit.

🕮 8.1.2

Scalability

If forms are a central part of your application, scalability is very important. Being able to reuse form models across components is critical.

Reactive forms are more scalable than template-driven forms. They provide direct access to the underlying form API, and use synchronous data flow between the view and the data model, which makes creating large-scale forms easier. Reactive forms require less setup for testing, and testing does not require deep understanding of change detection to properly test form updates and validation.

Template-driven forms focus on simple scenarios and are not as reusable. They abstract away the underlying form API, and use asynchronous data flow between the view and the data model. The abstraction of template-driven forms also affects testing. Tests are deeply reliant on manual change detection execution to run properly, and require more setup.

📖 8.1.3

Setting up the form model

Both reactive and template-driven forms track value changes between the form input elements that users interact with and the form data in your component model. The two approaches share underlying building blocks but differ in how you create and manage the common form-control instances.

Common form foundation classes

Both reactive and template-driven forms are built on the following base classes.

- **FormControl** tracks the value and validation status of an individual form control.
- **FormGroup** tracks the same values and status for a collection of form controls.
- **FormArray** tracks the same values and status for an array of form controls.
- **ControlValueAccessor** creates a bridge between Angular **FormControl** instances and native DOM elements.

Setup in reactive forms

With reactive forms, you define the form model directly in the component class. The **[formControl]** directive links the explicitly created **FormControl** instance to a specific form element in the view, using an internal value accessor.

The following component implements an input field for a single control, using reactive forms. In this example, the form model is the **FormControl** instance.

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

@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text"
[formControl]="favoriteColorControl">
  `
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```

📖 8.1.4

Data flow in forms

When an application contains a form, Angular must keep the view in sync with the component model and the component model in sync with the view. As users change values and make selections through the view, the new values must be reflected in the data model. Similarly, when the program logic changes values in the data model, those values must be reflected in the view.

Reactive and template-driven forms differ in how they handle data flowing from the user or from programmatic changes.

📖 8.1.5

Form validation

Validation is an integral part of managing any set of forms. Whether you're checking for required fields or querying an external API for an existing username, Angular provides a set of built-in validators as well as the ability to create custom validators.

- Reactive forms define custom validators as functions that receive a control to validate.
- Template-driven forms are tied to template directives, and must provide custom validator directives that wrap validation functions.

📖 8.1.6

Testing

Testing plays a large part in complex applications. A simpler testing strategy is useful when validating that your forms function correctly. Reactive forms and template-driven forms have different levels of reliance on rendering the UI to perform assertions based on form control and form field changes.

8.2 Reactive forms

🕮 8.2.1

Overview of reactive forms

Reactive forms use an explicit and immutable approach to managing the state of a form at a given point in time. Each change to the form state returns a new state, which maintains the integrity of the model between changes. Reactive forms are built around observable streams, where form inputs and values are provided as streams of input values, which can be accessed synchronously.

Reactive forms also provide a straightforward path to testing because you are assured that your data is consistent and predictable when requested. Any consumers of the streams have access to manipulate that data safely.

Reactive forms differ from template-driven forms in distinct ways. Reactive forms provide synchronous access to the data model, immutability with observable operators, and change tracking through observable streams.

Template-driven forms allow direct access to modify data in your template, but are less explicit than reactive forms because they rely on directives embedded in the template, along with mutable data to track changes asynchronously.

🕮 8.2.2

Adding a basic form control

There are three steps to using form controls.

1. Register the reactive forms module in your application. This module declares the reactive-form directives that you need to use reactive forms.
2. Generate a new **FormControl** instance and save it in the component.
3. Register the **FormControl** in the template.

You can then display the form by adding the component to the template.

The following examples show how to add a single form control. In the example, the user enters their name into an input field, captures that input value, and displays the current value of the form control element.

Register the reactive forms module

To use reactive form controls, import **ReactiveFormsModule** from the **@angular/forms** package and add it to your NgModule's **imports** array.

src/app/app.module.ts (excerpt)

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Generate a new **FormControl**

Use the CLI command **ng generate** to generate a component in your project to host the control.

```
ng generate component NameEditor
```

To register a single form control, import the **FormControl** class and create a new instance of **FormControl** to save as a class property.

src/app/name-editor/name-editor.component.ts

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

@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css']
})
export class NameEditorComponent {
  name = new FormControl('');
}
```

Use the constructor of **FormControl** to set its initial value, which in this case is an empty string. By creating these controls in your component class, you get immediate access to listen for, update, and validate the state of the form input.

Register the control in the template

After you create the control in the component class, you must associate it with a form control element in the template. Update the template with the form control using the **formControl** binding provided by **FormControlDirective**, which is also included in the **ReactiveFormsModule**.

src/app/name-editor/name-editor.component.html

```
<label for="name">Name: </label>
<input id="name" type="text" [formControl]="name">
```

- For a summary of the classes and directives provided by **ReactiveFormsModule**, see the Reactive forms API section below.
- For complete syntax details of these classes and directives, see the API reference documentation for the Forms package.

Using the template binding syntax, the form control is now registered to the **name** input element in the template. The form control and DOM element communicate with each other: the view reflects changes in the model, and the model reflects changes in the view.

Display the component

The form control assigned to **name** is displayed when the component is added to a template.

src/app/app.component.html (name editor)

```
<app-name-editor></app-name-editor>
```

🕮 8.2.3

Displaying a form control value

You can display the value in the following ways.

- Through the **valueChanges** observable where you can listen for changes in the form's value in the template using **AsyncPipe** or in the component class using the **subscribe()** method.
- With the **value** property, which gives you a snapshot of the current value.

The following example shows you how to display the current value using interpolation in the template.

src/app/name-editor/name-editor.component.html (control value)

```
<p>Value: {{ name.value }}</p>
```

The displayed value changes as you update the form control element.

Reactive forms provide access to information about a given control through properties and methods provided with each instance. These properties and

methods of the underlying AbstractControl class are used to control form state and determine when to display messages when handling input validation.

🕮 8.2.4

Grouping form controls

Forms typically contain several related controls. Reactive forms provide two ways of grouping multiple related controls into a single input form.

- A form *group* defines a form with a fixed set of controls that you can manage together. Form group basics are discussed in this section. You can also nest form groups to create more complex forms.
- A form *array* defines a dynamic form, where you can add and remove controls at run time. You can also nest form arrays to create more complex forms. For more about this option, see Creating dynamic forms below.

Just as a form control instance gives you control over a single input field, a form group instance tracks the form state of a group of form control instances (for example, a form). Each control in a form group instance is tracked by name when creating the form group. The following example shows how to manage multiple form control instances in a single group.

Generate a **ProfileEditor** component and import the **FormGroup** and **FormControl** classes from the **@angular/forms** package.

```
ng generate component ProfileEditor
```

src/app/profile-editor/profile-editor.component.ts (imports)

```
import { FormGroup, FormControl } from '@angular/forms';
```

To add a form group to this component, take the following steps.

1. Create a **FormGroup** instance.
2. Associate the **FormGroup** model and view.
3. Save the form data.

Create a FormGroup instance

Create a property in the component class named **profileForm** and set the property to a new form group instance. To initialize the form group, provide the constructor with an object of named keys mapped to their control.

For the profile form, add two form control instances with the names **firstName** and **lastName**.

src/app/profile-editor/profile-editor.component.ts (form group)

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });
}
```

The individual form controls are now collected within a group.
A **FormGroup** instance provides its model value as an object reduced from the values of each control in the group. A form group instance has the same properties (such as **value** and **untouched**) and methods (such as **setValue()**) as a form control instance.

Associate the FormGroup model and view

A form group tracks the status and changes for each of its controls, so if one of the controls changes, the parent control also emits a new status or value change. The model for the group is maintained from its members. After you define the model, you must update the template to reflect the model in the view.

src/app/profile-editor/profile-editor.component.html (template form group)

```
<form [formGroup]="profileForm">

  <label for="first-name">First Name: </label>
  <input id="first-name" type="text"
formControlName="firstName">

  <label for="last-name">Last Name: </label>
  <input id="last-name" type="text"
formControlName="lastName">

</form>
```

Note that just as a form group contains a group of controls,
the *profileForm* **FormGroup** is bound to the **form** element with

the FormGroup directive, creating a communication layer between the model and the form containing the inputs. The formControlName input provided by the FormControlName directive binds each individual input to the form control defined in FormGroup. The form controls communicate with their respective elements. They also communicate changes to the form group instance, which provides the source of truth for the model value.

Save form data

The **ProfileEditor** component accepts input from the user, but in a real scenario you want to capture the form value and make available for further processing outside the component. The FormGroup directive listens for the **submit** event emitted by the **form** element and emits an **ngSubmit** event that you can bind to a callback function.

Add an **ngSubmit** event listener to the **form** tag with the **onSubmit()** callback method.

src/app/profile-editor/profile-editor.component.html (submit event)

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">
```

The **onSubmit()** method in the **ProfileEditor** component captures the current value of **profileForm**. Use EventEmitter to keep the form encapsulated and to provide the form value outside the component. The following example uses **console.warn** to log a message to the browser console.

src/app/profile-editor/profile-editor.component.ts (submit method)

```
onSubmit() {
  // TODO: Use EventEmitter with form value
  console.warn(this.profileForm.value);
}
```

The **submit** event is emitted by the **form** tag using the native DOM event. You trigger the event by clicking a button with **submit** type. This allows the user to press the Enter key to submit the completed form.

Use a **button** element to add a button to the bottom of the form to trigger the form submission.

src/app/profile-editor/profile-editor.component.html (submit button)

```
<p>Complete the form to enable button.</p>
<button type="submit"
[disabled]="!profileForm.valid">Submit</button>
```

Note: The button in the snippet above also has a **disabled** binding attached to it to disable the button when **profileForm** is invalid. You aren't performing any validation yet, so the button is always enabled. Basic form validation is covered in the Validating form input section.

Display the component

To display the **ProfileEditor** component that contains the form, add it to a component template.

src/app/app.component.html (profile editor)

```
<app-profile-editor></app-profile-editor>
```

ProfileEditor allows you to manage the form control instances for the **firstName** and **lastName** controls within the form group instance.

🕮 8.2.5

Creating nested form groups

Form groups can accept both individual form control instances and other form group instances as children. This makes composing complex form models easier to maintain and logically group together.

When building complex forms, managing the different areas of information is easier in smaller sections. Using a nested form group instance allows you to break large forms groups into smaller, more manageable ones.

To make more complex forms, use the following steps.

1. Create a nested group.
2. Group the nested form in the template.

Some types of information naturally fall into the same group. A name and address are typical examples of such nested groups, and are used in the following examples.

Create a nested group

To create a nested group in **profileForm**, add a nested **address** element to the form group instance.

src/app/profile-editor/profile-editor.component.ts (nested form group)

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

```
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });
}
```

In this example, **address group** combines the current **firstName** and **lastName** controls with the new **street**, **city**, **state**, and **zip** controls. Even though the **address** element in the form group is a child of the overall **profileForm** element in the form group, the same rules apply with value and status changes. Changes in status and value from the nested form group propagate to the parent form group, maintaining consistency with the overall model.

Group the nested form in the template

After you update the model in the component class, update the template to connect the form group instance and its input elements.

Add the **address** form group containing the **street**, **city**, **state**, and **zip** fields to the **ProfileEditor** template.

src/app/profile-editor/profile-editor.component.html (template nested form group)

```
<div formGroupName="address">
  <h2>Address</h2>

  <label for="street">Street: </label>
  <input id="street" type="text" formControlName="street">

  <label for="city">City: </label>
  <input id="city" type="text" formControlName="city">
```

```
  <label for="state">State: </label>
  <input id="state" type="text" formControlName="state">

  <label for="zip">Zip Code: </label>
  <input id="zip" type="text" formControlName="zip">
</div>
```

The **ProfileEditor** form is displayed as one group, but the model is broken down further to represent the logical grouping areas.

8.3 Validating form input

🕮 8.3.1

Validating form input

You can improve overall data quality by validating user input for accuracy and completeness. This page shows how to validate user input from the UI and display useful validation messages, in both reactive and template-driven forms.

🕮 8.3.2

Validating input in template-driven forms

To add validation to a template-driven form, you add the same validation attributes as you would with native HTML form validation. Angular uses directives to match these attributes with validator functions in the framework.

Every time the value of a form control changes, Angular runs validation and generates either a list of validation errors that results in an INVALID status, or null, which results in a VALID status.

You can then inspect the control's state by exporting **ngModel** to a local template variable. The following example exports **NgModel** into a variable called **name**:

template/hero-form-template.component.html (name)

```
<input type="text" id="name" name="name" class="form-control"
      required minlength="4" appForbiddenName="bob"
      [(ngModel)]="hero.name" #name="ngModel">

<div *ngIf="name.invalid && (name.dirty || name.touched)"
```

```
    class="alert">

  <div *ngIf="name.errors?.required">
    Name is required.
  </div>
  <div *ngIf="name.errors?.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors?.forbiddenName">
    Name cannot be Bob.
  </div>

</div>
```

Notice the following features illustrated by the example.

- The **<input>** element carries the HTML validation attributes: **required** and **minlength**. It also carries a custom validator directive, **forbiddenName**. For more information, see the Custom validators section.
- **#name="ngModel"** exports **NgModel** into a local variable called **name**. **NgModel** mirrors many of the properties of its underlying **FormControl** instance, so you can use this in the template to check for control states such as **valid** and **dirty**. For a full list of control properties, see the AbstractControl API reference.
- The ***ngIf** on the ** ** element reveals a set of nested message **divs** but only if the **name** is invalid and the control is either **dirty** or **touched**.
- Each nested ** ** can present a custom message for one of the possible validation errors. There are messages for **required**, **minlength**, and **forbiddenName**.

To prevent the validator from displaying errors before the user has a chance to edit the form, you should check for either the **dirty** or **touched** states in a control.

- When the user changes the value in the watched field, the control is marked as "dirty".
- When the user blurs the form control element, the control is marked as "touched".

🕮 8.3.3

Validating input in reactive forms

In a reactive form, the source of truth is the component class. Instead of adding validators through attributes in the template, you add validator functions directly to

the form control model in the component class. Angular then calls these functions whenever the value of the control changes.

Validator functions

Validator functions can be either synchronous or asynchronous.

- Sync validators: Synchronous functions that take a control instance and immediately return either a set of validation errors or **null**. You can pass these in as the second argument when you instantiate a **FormControl**.
- Async validators: Asynchronous functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or **null**. You can pass these in as the third argument when you instantiate a **FormControl**.

For performance reasons, Angular only runs async validators if all sync validators pass. Each must complete before errors are set.

Built-in validator functions

You can choose to write your own validator functions, or you can use some of Angular's built-in validators.

The same built-in validators that are available as attributes in template-driven forms, such as **required** and **minlength**, are all available to use as functions from the **Validators** class. For a full list of built-in validators, see the Validators API reference.

To update the hero form to be a reactive form, you can use some of the same built-in validators—this time, in function form, as in the following example.

reactive/hero-form-reactive.component.ts (validator functions)

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    name: new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you
pass in the custom validator.
    ]),
    alterEgo: new FormControl(this.hero.alterEgo),
    power: new FormControl(this.hero.power,
Validators.required)
  });

}
```

```
get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }
```

In this example, the **name** control sets up two built-in validators—**Validators.required** and **Validators.minLength(4)**—and one custom validator, **forbiddenNameValidator**.

All of these validators are synchronous, so they are passed as the second argument. Notice that you can support multiple validators by passing the functions in as an array.

This example also adds a few getter methods. In a reactive form, you can always access any form control through the **get** method on its parent group, but sometimes it's useful to define getters as shorthand for the template.

If you look at the template for the **name** input again, it is fairly similar to the template-driven example.

reactive/hero-form-reactive.component.html (name with error msg)

```
<input type="text" id="name" class="form-control"
      formControlName="name" required>

<div *ngIf="name.invalid && (name.dirty || name.touched)"
    class="alert alert-danger">

  <div *ngIf="name.errors?.required">
    Name is required.
  </div>
  <div *ngIf="name.errors?.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors?.forbiddenName">
    Name cannot be Bob.
  </div>
</div>
```

This form differs from the template-driven version in that it no longer exports any directives. Instead, it uses the **name** getter defined in the component class.

Notice that the **required** attribute is still present in the template. Although it's not necessary for validation, it should be retained to for accessibility purposes.

🕮 8.3.4

Defining custom validators

The built-in validators don't always match the exact use case of your application, so you sometimes need to create a custom validator.

Consider the **forbiddenNameValidator** function from previous reactive-form examples. Here's what the definition of that function looks like.

shared/forbidden-name.directive.ts (forbiddenNameValidator)

```
/** A hero's name can't match the given regular expression */
export function forbiddenNameValidator(nameRe: RegExp):
ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null
=> {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {forbiddenName: {value: control.value}}
: null;
  };
}
```

The function is a factory that takes a regular expression to detect a *specific* forbidden name and returns a validator function.

In this sample, the forbidden name is "bob", so the validator will reject any hero name containing "bob". Elsewhere it could reject "alice" or any name that the configuring regular expression matches.

The **forbiddenNameValidator** factory returns the configured validator function. That function takes an Angular control object and returns *either* null if the control value is valid *or* a validation error object. The validation error object typically has a property whose name is the validation key, **'forbiddenName'**, and whose value is an arbitrary dictionary of values that you could insert into an error message, **{name}**.

Custom async validators are similar to sync validators, but they must instead return a Promise or observable that later emits null or a validation error object. In the case of an observable, the observable must complete, at which point the form uses the last value emitted for validation.

🕮 8.3.5

Control status CSS classes

Angular automatically mirrors many control properties onto the form control element as CSS classes. You can use these classes to style form control elements according to the state of the form. The following classes are currently supported.

- **.ng-valid**
- **.ng-invalid**
- **.ng-pending**
- **.ng-pristine**
- **.ng-dirty**
- **.ng-untouched**
- **.ng-touched**
- **.ng-submitted** (enclosing form element only)

In the following example, the hero form uses the **.ng-valid** and **.ng-invalid** classes to set the color of each form control's border.

forms.css (status classes)

```
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}

.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}

.alert div {
  background-color: #fed3d3;
  color: #820000;
  padding: 1rem;
  margin-bottom: 1rem;
}

.form-group {
  margin-bottom: 1rem;
}

label {
  display: block;
  margin-bottom: .5rem;
}
```

```
select {
  width: 100%;
  padding: .5rem;
}
```

📖 8.3.6

Creating asynchronous validators

Asynchronous validators implement
the **AsyncValidatorFn** and **AsyncValidator** interfaces. These are very similar to their synchronous counterparts, with the following differences.

- The **validate()** functions must return a Promise or an observable,
- The observable returned must be finite, meaning it must complete at some point. To convert an infinite observable into a finite one, pipe the observable through a filtering operator such as **first**, **last**, **take**, or **takeUntil**.

Asynchronous validation happens after the synchronous validation, and is performed only if the synchronous validation is successful. This check allows forms to avoid potentially expensive async validation processes (such as an HTTP request) if the more basic validation methods have already found invalid input.

After asynchronous validation begins, the form control enters a **pending** state. You can inspect the control's **pending** property and use it to give visual feedback about the ongoing validation operation.

A common UI pattern is to show a spinner while the async validation is being performed. The following example shows how to achieve this in a template-driven form.

```
<input [(ngModel)]="name" #model="ngModel"
appSomeAsyncValidator>
<app-spinner *ngIf="model.pending"></app-spinner>
```

Implementing a custom async validator

In the following example, an async validator ensures that heroes pick an alter ego that is not already taken. New heroes are constantly enlisting and old heroes are leaving the service, so the list of available alter egos cannot be retrieved ahead of time. To validate the potential alter ego entry, the validator must initiate an asynchronous operation to consult a central database of all currently enlisted heroes.

The following code create the validator class, **UniqueAlterEgoValidator**, which implements the **AsyncValidator** interface.

```
@Injectable({ providedIn: 'root' })
export class UniqueAlterEgoValidator implements AsyncValidator
{
  constructor(private heroesService: HeroesService) {}

  validate(
    ctrl: AbstractControl
  ): Promise<ValidationErrors | null> |
Observable<ValidationErrors | null> {
    return
this.heroesService.isAlterEgoTaken(ctrl.value).pipe(
      map(isTaken => (isTaken ? { uniqueAlterEgo: true } :
null)),
      catchError(() => of(null))
    );
  }
}
```

The constructor injects the **HeroesService**, which defines the following interface.

```
interface HeroesService {
  isAlterEgoTaken: (alterEgo: string) => Observable<boolean>;
}
```

In a real world application, the **HeroesService** would be responsible for making an HTTP request to the hero database to check if the alter ego is available. From the validator's point of view, the actual implementation of the service is not important, so the example can just code against the **HeroesService** interface.

As the validation begins, the **UniqueAlterEgoValidator** delegates to the **HeroesService isAlterEgoTaken()** method with the current control value. At this point the control is marked as **pending** and remains in this state until the observable chain returned from the **validate()** method completes.

The **isAlterEgoTaken()** method dispatches an HTTP request that checks if the alter ego is available, and returns **Observable<boolean>** as the result.
The **validate()** method pipes the response through the **map** operator and transforms it into a validation result.

The method then, like any validator, returns **null** if the form is valid, and ValidationErrors if it is not. This validator handles any potential errors with the **catchError** operator. In this case, the validator treats the **isAlterEgoTaken()** error as a successful validation, because failure to make a validation request does not necessarily mean that the alter ego is invalid. You could handle the error differently and return the **ValidationError** object instead.

After some time passes, the observable chain completes and the asynchronous validation is done. The **pending** flag is set to **false**, and the form validity is updated.

Optimizing performance of async validators

By default, all validators run after every form value change. With synchronous validators, this does not normally have a noticeable impact on application performance. Async validators, however, commonly perform some kind of HTTP request to validate the control. Dispatching an HTTP request after every keystroke could put a strain on the backend API, and should be avoided if possible.

You can delay updating the form validity by changing the **updateOn** property from **change** (default) to **submit** or **blur**.

With template-driven forms, set the property in the template.

```
<input [(ngModel)]="name" [ngModelOptions]="{updateOn:
'blur'}">
```

With reactive forms, set the property in the **FormControl** instance.

```
new FormControl('', {updateOn: 'blur'});
```

🕮 8.3.7

Interaction with native HTML form validation

By default, Angular disables native HTML form validation by adding the **novalidate** attribute on the enclosing **<form>** and uses directives to match these attributes with validator functions in the framework. If you want to use native validation in combination with Angular-based validation, you can re-enable it with the **ngNativeValidate** directive.

8.4 HTTP Client

🕮 8.4.1

Communicating with backend services using HTTP

Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the **HttpClient** service class in **@angular/common/http**.

The HTTP client service offers the following major features.

- The ability to request typed response objects.
- Streamlined error handling.
- Testability features.
- Request and response interception.

🕮 8.4.2

Setup for server communication

Before you can use **HttpClient**, you need to import the Angular **HttpClientModule**. Most apps do so in the root **AppModule**.

app/app.module.ts (excerpt)

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

You can then inject the **HttpClient** service as a dependency of an application class, as shown in the following **ConfigService** example.

app/config/config.service.ts (excerpt)

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
```

```
}
```

The HttpClient service makes use of observables for all transactions. You must import the RxJS observable and operator symbols that appear in the example snippets. These **ConfigService** imports are typical.

app/config/config.service.ts (RxJS imports)

```
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
```

📖 8.4.3

Requesting data from a server

Use the **HttpClient.get()** method to fetch data from a server. The asynchronous method sends an HTTP request, and returns an Observable that emits the requested data when the response is received. The return type varies based on the **observe** and **responseType** values that you pass to the call.

The **get()** method takes two arguments; the endpoint URL from which to fetch, and an *options* object that you can use to configure the request.

```
options: {
    headers?: HttpHeaders | {[header: string]: string |
string[]},
    observe?: 'body' | 'events' | 'response',
    params?: HttpParams|{[param: string]: string | number |
boolean | ReadonlyArray<string | number | boolean>},
    reportProgress?: boolean,
    responseType?: 'arraybuffer'|'blob'|'json'|'text',
    withCredentials?: boolean,
  }
```

Important options include the *observe* and *responseType* properties.

- The *observe* option specifies how much of the response to return.
- The *responseType* option specifies the format in which to return data.

You can use the **options** object to configure various other aspects of an outgoing request. In Adding headers, for example, the service set the default headers using the **headers** option property.

Use the **params** property to configure a request with HTTP URL parameters, and the **reportProgress** option to listen for progress events when transferring large amounts of data.

Applications often request JSON data from a server. In the **ConfigService** example, the app needs a configuration file on the server, **config.json**, that specifies resource URLs.

assets/config.json

```
{
  "heroesUrl": "api/heroes",
  "textfile": "assets/textfile.txt",
  "date": "2020-01-29"
}
```

To fetch this kind of data, the **get()** call needs the following options: **{observe: 'body', responseType: 'json'}**. These are the default values for those options, so the following examples do not pass the options object. Later sections show some of the additional option possibilities.

The example conforms to the best practices for creating scalable solutions by defining a re-usable injectable service to perform the data-handling functionality. In addition to fetching data, the service can post-process the data, add error handling, and add retry logic.

The **ConfigService** fetches this file using the **HttpClient.get()** method.

app/config/config.service.ts (getConfig v.1)

```
configUrl = 'assets/config.json';

getConfig() {
  return this.http.get<Config>(this.configUrl);
}
```

The **ConfigComponent** injects the **ConfigService** and calls the **getConfig** service method.

Because the service method returns an **Observable** of configuration data, the component *subscribes* to the method's return value. The subscription callback performs minimal post-processing. It copies the data fields into the component's **config** object, which is data-bound in the component template for display.

app/config/config.component.ts (showConfig v.1)

```
showConfig() {
  this.configService.getConfig()
    .subscribe((data: Config) => this.config = {
        heroesUrl: data.heroesUrl,
```

```
        textfile:  data.textfile,
        date: data.date,
    });
}
```

📖 8.4.4

Requesting a typed response

You can structure your **HttpClient** request to declare the type of the response object, to make consuming the output easier and more obvious. Specifying the response type acts as a type assertion at compile time.

Specifying the response type is a declaration to TypeScript that it should treat your response as being of the given type. This is a build-time check and doesn't guarantee that the server will actually respond with an object of this type. It is up to the server to ensure that the type specified by the server API is returned.

To specify the response object type, first define an interface with the required properties. Use an interface rather than a class, because the response is a plain object that cannot be automatically converted to an instance of a class.

```
export interface Config {
  heroesUrl: string;
  textfile: string;
  date: any;
}
```

Next, specify that interface as the **HttpClient.get()** call's type parameter in the service.

app/config/config.service.ts (getConfig v.2)

```
getConfig() {
  // now returns an Observable of Config
  return this.http.get<Config>(this.configUrl);
}
```

When you pass an interface as a type parameter to the **HttpClient.get()** method, you can use the RxJS **map** operator to transform the response data as needed by the UI. You can then pass the transformed data to the async pipe.

The callback in the updated component method receives a typed data object, which is easier and safer to consume:

app/config/config.component.ts (showConfig v.2)

```
config: Config | undefined;

showConfig() {
  this.configService.getConfig()
    // clone the data object, using its known Config shape
    .subscribe((data: Config) => this.config = { ...data });
}
```

To access properties that are defined in an interface, you must explicitly convert the plain object you get from the JSON to the required response type. For example, the following **subscribe** callback receives **data** as an Object, and then type-casts it in order to access the properties.

```
.subscribe(data => this.config = {
  heroesUrl: (data as any).heroesUrl,
  textfile:  (data as any).textfile,
});
```

OBSERVE AND *RESPONSE* TYPES

The types of the **observe** and **response** options are *string unions*, rather than plain strings.

```
options: {
    ...
    observe?: 'body' | 'events' | 'response',
    ...
    responseType?: 'arraybuffer'|'blob'|'json'|'text',
    ...
  }
```

This can cause confusion. For example:

```
// this works
client.get('/foo', {responseType: 'text'})

// but this does NOT work
const options = {
  responseType: 'text',
};
client.get('/foo', options)
```

In the second case, TypeScript infers the type of **options** to be **{responseType: string}**. The type is too wide to pass to **HttpClient.get** which is expecting the type of **responseType** to be one of the *specific* strings. **HttpClient** is typed explicitly this

way so that the compiler can report the correct return type based on the options you provided.

Use **as const** to let TypeScript know that you really do mean to use a constant string type:

```
const options = {
  responseType: 'text' as const,
};
client.get('/foo', options);
```

🕮 8.4.5

Handling request errors

If the request fails on the server, **HttpClient** returns an *error* object instead of a successful response.

The same service that performs your server transactions should also perform error inspection, interpretation, and resolution.

When an error occurs, you can obtain details of what failed in order to inform your user. In some cases, you might also automatically retry the request.

Getting error details

An app should give the user useful feedback when data access fails. A raw error object is not particularly useful as feedback. In addition to detecting that an error has occurred, you need to get error details and use those details to compose a user-friendly response.

Two types of errors can occur.

- The server backend might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error *responses*.
- Something could go wrong on the client-side such as a network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors have **status** set to **0** and the **error** property contains a **ProgressEvent** object, whose **type** might provide further information.

HttpClient captures both kinds of errors in its **HttpErrorResponse**. You can inspect that response to identify the error's cause.

The following example defines an error handler in the previously defined ConfigService.

app/config/config.service.ts (handleError)

```
private handleError(error: HttpErrorResponse) {
  if (error.status === 0) {
    // A client-side or network error occurred. Handle it
accordingly.
    console.error('An error occurred:', error.error);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went
wrong.
    console.error(
      `Backend returned code ${error.status}, body was: `,
error.error);
  }
  // Return an observable with a user-facing error message.
  return throwError(
    'Something bad happened; please try again later.');
}
```

The handler returns an RxJS **ErrorObservable** with a user-friendly error message. The following code updates the **getConfig()** method, using a pipe to send all observables returned by the **HttpClient.get()** call to the error handler.

app/config/config.service.ts (getConfig v.3 with error handler)

```
getConfig() {
  return this.http.get<Config>(this.configUrl)
    .pipe(
      catchError(this.handleError)
    );
}
```

Retrying a failed request

Sometimes the error is transient and goes away automatically if you try again. For example, network interruptions are common in mobile scenarios, and trying again can produce a successful result.

The RxJS library offers several *retry* operators. For example, the **retry()** operator automatically re-subscribes to a failed **Observable** a specified number of times. *Re-subscribing* to the result of an **HttpClient** method call has the effect of reissuing the HTTP request.

The following example shows how you can pipe a failed request to the **retry()** operator before passing it to the error handler.

app/config/config.service.ts (getConfig with retry)

```
getConfig() {
  return this.http.get<Config>(this.configUrl)
    .pipe(
      retry(3), // retry a failed request up to 3 times
      catchError(this.handleError) // then handle the error
    );
}
```

📖 8.4.6

Sending data to a server

In addition to fetching data from a server, **HttpClient** supports other HTTP methods such as PUT, POST, and DELETE, which you can use to modify the remote data.

The sample app for this guide includes an abridged version of the "Tour of Heroes" example that fetches heroes and enables users to add, delete, and update them. The following sections show examples of the data-update methods from the sample's **HeroesService**.

Making a POST request

Apps often send data to a server with a POST request when submitting a form. In the following example, the **HeroesService** makes an HTTP POST request when adding a hero to the database.

app/heroes/heroes.service.ts (addHero)

```
/** POST: add a new hero to the database */
addHero(hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero,
httpOptions)
    .pipe(
      catchError(this.handleError('addHero', hero))
    );
}
```

The **HttpClient.post()** method is similar to **get()** in that it has a type parameter, which you can use to specify that you expect the server to return data of a given type. The method takes a resource URL and two additional parameters:

- *body* - The data to POST in the body of the request.
- *options* - An object containing method options which, in this case, specify required headers.

The example catches errors as described above.

The **HeroesComponent** initiates the actual POST operation by subscribing to the **Observable** returned by this service method.

app/heroes/heroes.component.ts (addHero)

```
this.heroesService
  .addHero(newHero)
  .subscribe(hero => this.heroes.push(hero));
```

When the server responds successfully with the newly added hero, the component adds that hero to the displayed **heroes** list.

🕮 8.4.7

Making a DELETE request

This application deletes a hero with the **HttpClient.delete** method by passing the hero's id in the request URL.

app/heroes/heroes.service.ts (deleteHero)

```
/** DELETE: delete the hero from the server */
deleteHero(id: number): Observable<unknown> {
  const url = `${this.heroesUrl}/${id}`; // DELETE
api/heroes/42
  return this.http.delete(url, httpOptions)
    .pipe(
      catchError(this.handleError('deleteHero'))
    );
}
```

The **HeroesComponent** initiates the actual DELETE operation by subscribing to the **Observable** returned by this service method.

app/heroes/heroes.component.ts (deleteHero)

```
this.heroesService
  .deleteHero(hero.id)
  .subscribe();
```

The component isn't expecting a result from the delete operation, so it subscribes without a callback. Even though you are not using the result, you still have to subscribe. Calling the **subscribe()** method *executes* the observable, which is what initiates the DELETE request.

You must call *subscribe()* or nothing happens. Just calling **HeroesService.deleteHero()** does not initiate the DELETE request.

```
// oops ... subscribe() is missing so nothing happens
this.heroesService.deleteHero(hero.id);
```

Always *subscribe*!

An **HttpClient** method does not begin its HTTP request until you call **subscribe()** on the observable returned by that method. This is true for *all* **HttpClient** *methods*.

The **AsyncPipe** subscribes (and unsubscribes) for you automatically.

All observables returned from **HttpClient** methods are *cold* by design. Execution of the HTTP request is *deferred*, allowing you to extend the observable with additional operations such as **tap** and **catchError** before anything actually happens.

Calling **subscribe(...)** triggers execution of the observable and causes **HttpClient** to compose and send the HTTP request to the server.

You can think of these observables as *blueprints* for actual HTTP requests.

In fact, each **subscribe()** initiates a separate, independent execution of the observable. Subscribing twice results in two HTTP requests.

```
const req = http.get<Heroes>('/api/heroes');
// 0 requests made - .subscribe() not called.
req.subscribe();
// 1 request made.
req.subscribe();
// 2 requests made.
```

🕮 8.4.8

Configuring HTTP URL parameters

Use the **HttpParams** class with the **params** request option to add URL query strings in your **HttpRequest**.

The following example, the **searchHeroes()** method queries for heroes whose names contain the search term.

Start by importing **HttpParams** class.

```
import {HttpParams} from "@angular/common/http";
content_copy
/* GET heroes whose name contains search term */
searchHeroes(term: string): Observable<Hero[]> {
  term = term.trim();

  // Add safe, URL encoded search parameter if there is a
search term
  const options = term ?
   { params: new HttpParams().set('name', term) } : {};

  return this.http.get<Hero[]>(this.heroesUrl, options)
    .pipe(
      catchError(this.handleError<Hero[]>('searchHeroes', []))
    );
}
```

If there is a search term, the code constructs an options object with an HTML URL-encoded search parameter. If the term is "cat", for example, the GET request URL would be **api/heroes?name=cat**.

The **HttpParams** object is immutable. If you need to update the options, save the returned value of the **.set()** method.

You can also create HTTP parameters directly from a query string by using the **fromString** variable:

```
const params = new HttpParams({fromString: 'name=foo'});
```

🕮 8.4.9

Intercepting requests and responses

With interception, you declare *interceptors* that inspect and transform HTTP requests from your application to a server. The same interceptors can also inspect and transform a server's responses on their way back to the application. Multiple interceptors form a *forward-and-backward* chain of request/response handlers.

Interceptors can perform a variety of *implicit* tasks, from authentication to logging, in a routine, standard way, for every HTTP request/response.

Without interception, developers would have to implement these tasks *explicitly* for each **HttpClient** method call.

📖 8.4.10

Optimizing server interaction with debouncing

If you need to make an HTTP request in response to user input, it's not efficient to send a request for every keystroke. It's better to wait until the user stops typing and then send a request. This technique is known as debouncing.

Consider the following template, which lets a user enter a search term to find an npm package by name. When the user enters a name in a search-box, the **PackageSearchComponent** sends a search request for a package with that name to the npm web API.

app/package-search/package-search.component.html (search)

```
<input type="text" (keyup)="search(getValue($event))"
id="name" placeholder="Search"/>

<ul>
  <li *ngFor="let package of packages$ | async">
    <b>{{package.name}} v.{{package.version}}</b> -
    <i>{{package.description}}</i>
  </li>
</ul>
```

Here, the **keyup** event binding sends every keystroke to the component's **search()** method.

The type of **$event.target** is only **EventTarget** in the template. In the **getValue()** method, the target is cast to an **HTMLInputElement** to allow type-safe access to its **value** property.

```
getValue(event: Event): string {
  return (event.target as HTMLInputElement).value;
}
```

The following snippet implements debouncing for this input using RxJS operators.

app/package-search/package-search.component.ts (excerpt)

```
withRefresh = false;
packages$!: Observable<NpmPackageInfo[]>;
private searchText$ = new Subject<string>();

search(packageName: string) {
```

```
  this.searchText$.next(packageName);
}

ngOnInit() {
  this.packages$ = this.searchText$.pipe(
    debounceTime(500),
    distinctUntilChanged(),
    switchMap(packageName =>
      this.searchService.search(packageName,
this.withRefresh))
  );
}

constructor(private searchService: PackageSearchService) { }
```

The **searchText$** is the sequence of search-box values coming from the user. It's defined as an RxJS **Subject**, which means it is a multicasting **Observable** that can also emit values for itself by calling **next(value)**, as happens in the **search()** method.

Rather than forward every **searchText** value directly to the injected **PackageSearchService**, the code in **ngOnInit()** pipes search values through three operators, so that a search value reaches the service only if it's a new value and the user has stopped typing.

- **debounceTime(500)**—Wait for the user to stop typing (1/2 second in this case).
- **distinctUntilChanged()**—Wait until the search text changes.
- **switchMap()**—Send the search request to the service.

The code sets **packages$** to this re-composed **Observable** of search results. The template subscribes to **packages$** with the AsyncPipe and displays search results as they arrive.

🕮 8.4.11

Using the *switchMap()* operator

The **switchMap()** operator takes a function argument that returns an **Observable**. In the example, **PackageSearchService.search** returns an **Observable**, as other data service methods do. If a previous search request is still in-flight (as when the network connection is poor), the operator cancels that request and sends a new one.

Note that **switchMap()** returns service responses in their original request order, even if the server returns them out of order.

If you think you'll reuse this debouncing logic, consider moving it to a utility function or into the **PackageSearchService** itself.

🕮 8.4.12

Testing HTTP requests

As for any external dependency, you must mock the HTTP backend so your tests can simulate interaction with a remote server.
The **@angular/common/http/testing** library makes it straightforward to set up such mocking.

Angular's HTTP testing library is designed for a pattern of testing in which the app executes code and makes requests first. The test then expects that certain requests have or have not been made, performs assertions against those requests, and finally provides responses by "flushing" each expected request.

🕮 8.4.13

Passing metadata to interceptors

Many interceptors require or benefit from configuration. Consider an interceptor that retries failed requests. By default, the interceptor might retry a request three times, but you might want to override this retry count for particularly error-prone or sensitive requests.

HttpClient requests contain a *context* that can carry metadata about the request. This context is available for interceptors to read or modify, though it is not transmitted to the backend server when the request is sent. This allows applications or other interceptors to tag requests with configuration parameters, such as how many times to retry a request.

8.5 Forms (Exercises)

📝 8.5.1

define custom validators as functions that receive a control to validate.

8.5.2

are tied to template directives, and must provide custom validator directives that wrap validation functions.

8.5.3

Each change to the form state returns a new state, which maintains the integrity of the model between changes.

8.5.4

Reactive forms provide access to information about a given control through properties and methods provided with each instance

- True
- False

8.5.5

o prevent the validator from displaying errors before the user has a chance to edit the form, you should check for either the ... or ... states in a control.

8.5.6

Functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or null. You can pass these in as the third argument when you instantiate a FormControl.

8.5.7

Angular doesn't provides a client HTTP API for Angular applications, the HttpClient service class in @angular/common/http.

- True
- False

8.5.8

You can use the ... object to configure various other aspects of an outgoing request. In Adding headers, for example, the service set the default headers using the **headers** option property.

8.5.9

If the request fails on the server what does HttpClient return?

Testing

Chapter 9

9.1 Angular testing

🕮 9.1.1

Angular testing is a core feature available in every project set up with the Angular CLI.

To stay synchronized with the JavaScript ecosystem, the Angular team makes a point to release two major Angular version each year. Since its inception through to its most recent release, Angular 11, Angular has been designed with testability in mind.

There are two types of Angular testing:

1. **Unit testing** is the process of testing small, isolated pieces of code. Also known as isolated testing, unit tests do not use external resources, such as the network or a database
2. **Functional testing** refers to testing the functionality and of your Angular app from a user experience perspective — i.e., interacting with your app as it's running in a browser just as a user would

🕮 9.1.2

Unit testing in Angular refers to the process of testing individual units of code.

An Angular unit test aims to uncover issues such as incorrect logic, misbehaving functions, etc. by isolating pieces of code. This is sometimes more difficult than it sounds, especially for complex projects with poor separation of concerns. Angular is designed to help you write code in such a way that enables you to test your app's functions individually in isolation.

Angular unit testing enables you to test your app based on user behavior. While testing each possible behavior would be tedious, inefficient, and ineffective, writing tests for each coupling block in your application can help demonstrate how these blocks behave.

One of the easiest ways to test the strengths of these blocks is to write a test for each one. You don't necessarily need to wait until your users complain about how the input field behaves when the button is clicked. By writing a unit test for your blocks (components, services, etc.), you can easily detect when there is a break.

Our example Angular app has a service, a component, and an async task to simulate data being fetched from the server.

🕮 9.1.3

How do you write an Angular test?

When you create a new project with the Angular CLI (**ng new appName**), a default component and test file are added. Also — if, like me, you're always looking for a shortcut — a test script is always created alongside any component module (service, component) you create using the Angular CLI.

This test script, which ends with **.spec.ts**, is always added. Let's take a look at the initial test script file, which is the **app.component.spec.ts**:

```
import { TestBed, async } from '@angular/core/testing';
import { AppComponent } from './app.component';
describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  }));
  it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  }));
  it(`should have as title 'angular-unit-test'`, async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app.title).toEqual('angular-unit-test');
  }));
  it('should render title in a h1 tag', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;

expect(compiled.querySelector('h1').textContent).toContain('We
lcome to angular-unit-test!');
  }));
});
```

Let's run our first test to make sure nothing has broken yet:

```
ng test
```

You might be wondering, how can we simulate a user behavior by simply writing a test, even though the project is being rendered in a browser? As we proceed, I'll demonstrate how to simulate and Angular app running on a browser.

🕮 9.1.4

What is Karma in Angular?

Karma is a JavaScript test runner that runs the unit test snippet in Angular. Karma also ensures the result of the test is printed out either in the console or in the file log.

By default, Angular runs on Karma. Other test runners include Mocha and Jasmine. Karma provides tools that make it easier to call Jasmine tests while writing code in Angular.

🕮 9.1.5

How to write a unit test in Angular

The Angular testing package includes two utilities
called **TestBed** and **async**. **TestBed** is the main Angular utility package.

The describe container contains different blocks (it, beforeEach, xit, etc.). beforeEach runs before any other block. Other blocks do not depend on each other to run.

From the app.component.spec.ts file, the first block is the beforeEach inside the container (describe). This is the only block that runs before any other block (it). The declaration of the app module in app.module.ts file is simulated (declared) in the beforeEach block. The component (AppComponent) declared in the beforeEach block is the main component we want to have in this testing environment. The same logic applies to other test declaration.

The compileComponents object is called to compile your component's resources like the template, styles etc. You might not necessarily compile your component if you are using webpack:

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
   declarations: [
     AppComponent
   ],
  }).compileComponents();
}));
```

Now that the component has been declared in the beforeEach block, let's check if the component is created.

Thefixture.debugElement.componentInstance creates an instance of the class (AppComponent) . We will test to see if the instance of the class is truly created or not using toBeTruthy :

```
it('should create the app', async(() => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  expect(app).toBeTruthy();
}));
```

The third block demonstrates how you can have access to the properties of the created component (AppComponent). The only property added by default is the title. You can easily check if the title you set has changed or not from the instance of the component (AppComponent) created:

```
it(`should have as title 'angular-unit-test'`, async(() => {
   const fixture = TestBed.createComponent(AppComponent);
   const app = fixture.debugElement.componentInstance;
   expect(app.title).toEqual('angular-unit-test');
}));
```

The fourth block demonstrates how the test behaves in the browser environment. After creating the component, an instance of the created component (detectChanges) to simulate running on the browser environment is called. Now that the component has been rendered, you can have access to its child element by accessing the nativeElelment object of the rendered component (fixture.debugElement.nativeElement):

```
it('should render title in a h1 tag', async(() => {
  const fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement;
 expect(compiled.querySelector('h1').textContent).toContain('W
elcome to angular-unit-test!');
}));
```

Now that you have familiarized yourself with the basics of testing a component, let's test our Angular example application.

🕮 9.1.6

How to test an Angular service

The **describe** container contains different blocks (**it**, **beforeEach**, **xit**, etc.). **beforeEach** runs before any other block. Other blocks do not depend on each other to run.

From the **app.component.spec.ts** file, the first block is the **beforeEach** inside the container (**describe**). This is the only block that runs before any other block (**it**). The declaration of the app module in **app.module.ts** file is simulated (declared) in the **beforeEach** block. The component (**AppComponent**) declared in the **beforeEach** block is the main component we want to have in this testing environment. The same logic applies to other test declaration.

The **compileComponents** object is called to compile your component's resources like the template, styles etc. You might not necessarily compile your component if you are using webpack:

```
beforeEach(async(() => {
   TestBed.configureTestingModule({
      declarations: [
         AppComponent
      ],
   }).compileComponents();
}));
```

Now that the component has been declared in the **beforeEach** block, let's check if the component is created.

The**fixture.debugElement.componentInstance** creates an instance of the class (**AppComponent**) . We will test to see if the instance of the class is truly created or not using **toBeTruthy** :

```
it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
}));
```

The third block demonstrates how you can have access to the properties of the created component (**AppComponent**). The only property added by default is the

title. You can easily check if the title you set has changed or not from the instance of the component (**AppComponent**) created:

```
it(`should have as title 'angular-unit-test'`, async(() => {
     const fixture = TestBed.createComponent(AppComponent);
     const app = fixture.debugElement.componentInstance;
     expect(app.title).toEqual('angular-unit-test');
}));
```

The fourth block demonstrates how the test behaves in the browser environment. After creating the component, an instance of the created component (**detectChanges**) to simulate running on the browser environment is called. Now that the component has been rendered, you can have access to its child element by accessing the **nativeElelment** object of the rendered component (**fixture.debugElement.nativeElement**):

```
it('should render title in a h1 tag', async(() => {
   const fixture = TestBed.createComponent(AppComponent);
   fixture.detectChanges();
   const compiled = fixture.debugElement.nativeElement;

expect(compiled.querySelector('h1').textContent).toContain('We
lcome to angular-unit-test!');
}));
```

Now that you have familiarized yourself with the basics of testing a component, let's test our Angular example application.

🕮 9.1.7

How to test an Angular component

In our Angular unit testing example app, the **service** is injected into the **QuoteComponent** to access its properties, which will be needed by the view:

```
import { Component, OnInit } from '@angular/core';
import { QuoteService } from '../service/Quote.service';
import { QuoteModel } from '../model/QuoteModel';

@Component({
  selector: 'app-Quotes',
  templateUrl: './Quotes.component.html',
  styleUrls: ['./Quotes.component.css']
})
```

```
export class QuotesComponent implements OnInit {

  public quoteList: QuoteModel[];
  public quoteText: String = null;

  constructor(private service: QuoteService) { }

  ngOnInit() {
    this.quoteList = this.service.getQuote();
  }

  createNewQuote() {
    this.service.addNewQuote(this.quoteText);
    this.quoteText = null;
  }

  removeQuote(index) {
    this.service.removeQuote(index);
  }
}
<div class="container-fluid">
  <div class="row">
    <div class="col-8 col-sm-8 mb-3 offset-2">
      <div class="card">
        <div class="card-header">
          <h5>What Quote is on your mind ?</h5>
        </div>
        <div class="card-body">
          <div role="form">
            <div class="form-group col-8 offset-2">
              <textarea #quote class="form-control" rows="3"
cols="8" [(ngModel)]="quoteText" name="quoteText"></textarea>
            </div>
            <div class="form-group text-center">
              <button class="btn btn-primary"
(click)="createNewQuote()" [disabled]="quoteText ==
null">Create a new
                quote</button>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
```

```
  <div class="row">
    <div class="card mb-3 col-5 list-card" id="quote-cards"
style="max-width: 18rem;" *ngFor="let quote of quoteList; let
i = index"
      (click)="removeQuote(i)">
      <div class="card-body">
        <h6>{{ quote.text }}</h6>
      </div>
      <div class="card-footer text-muted">
        <small>Created on {{ quote.timeCreated }}</small>
      </div>
    </div>
  </div>
</div>
```

The first two blocks in the **describe** container run consecutively. In the first block, the **FormsModule** is imported into the configure test. This ensures the form's related directives, such as **ngModel**, can be used.

Also, the **QuotesComponent** is declared in the **configTestMod** similar to how the components are declared in **ngModule** residing in the **appModule** file. The second block **creates** a **QuoteComponent** and its **instance**, which will be used by the other blocks:

```
let component: QuotesComponent;
  let fixture: ComponentFixture<QuotesComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [FormsModule],
      declarations: [QuotesComponent]
    });
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(QuotesComponent);
    component = fixture.debugElement.componentInstance;
  });
```

This block tests if the instance of the component that is created is defined:

```
it("should create Quote component", () => {
    expect(component).toBeTruthy();
  });
```

The injected service handles the manipulation of all operations (**add**, **remove**, **fetch**). The **quoteService** variable holds the injected service (**QuoteService**). At this point, the component is yet to be rendered until the **detectChanges**method is called:

```
it("should use the quoteList from the service", () => {
    const quoteService =
fixture.debugElement.injector.get(QuoteService);
    fixture.detectChanges();

expect(quoteService.getQuote()).toEqual(component.quoteList);
  });
```

Now let's test whether we can successfully create a post. The properties of the component can be accessed upon instantiation, so the component rendered detects the new changes when a value is passed into the **quoteText** model. The **nativeElement** object gives access to the HTML element rendered which makes it easier to check if the **quote** added is part of the texts rendered:

```
it("should create a new post", () => {
    component.quoteText = "I love this test";
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.innerHTML).toContain("I love this test");
  });
```

Apart from having access to the HTML contents, you can also get an element by its CSS property. When the **quoteText** model is empty or null, the button is expected to be disabled:

```
it("should disable the button when textArea is empty", () => {
    fixture.detectChanges();
    const button =
fixture.debugElement.query(By.css("button"));
    expect(button.nativeElement.disabled).toBeTruthy();
  });
it("should enable button when textArea is not empty", () => {
    component.quoteText = "I love this test";
    fixture.detectChanges();
    const button =
fixture.debugElement.query(By.css("button"));
    expect(button.nativeElement.disabled).toBeFalsy();
  });
```

Just like the way we access an element with its CSS property, we can also access an element by its class name. Multiple classes can be accessed at the same time using **By e.g By.css('.className.className')** .

The button clicks are simulated by calling the **triggerEventHandler** . The **event** type must be specified which ,in this case, is click. A quote displayed is expected to be deleted from the **quoteList** when clicked on:

```
it("should remove post upon card click", () => {
    component.quoteText = "This is a fresh post";
    fixture.detectChanges();

    fixture.debugElement
      .query(By.css(".row"))
      .query(By.css(".card"))
      .triggerEventHandler("click", null);
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.innerHTML).toContain("This is a fresh
post");
  });
```

🕮 9.1.8

How to test an async operation in Angular

It's inevitable that you'll eventually need to fetch data remotely. This operation is best treated as an asynchronous task.

fetchQoutesFromServer represents an async task that returns an array of quotes after two seconds:

```
fetchQuotesFromServer() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve([new QuoteModel("I love unit testing", "Mon 4,
2018")]);
      }, 2000);
    });
  }
```

spyOn objects simulate how **fetchQuotesFromServer** method works. It accepts two argument **quoteService** which is injected into the component and the **fetchQuotesFromServer** method. **fetchQuotesFromServer** is expected to return a promise. **spyOn** chains the method using **and** with a fake promise call, which is returned using **returnValue**. Since we want to emulate how

the **fetchQuotesFromServer** works, we need to pass a **promise** that will resolve with a list of quotes.

Just as we did before, we'll call the **detectChanges** method to get the updated changes. **whenStable** allows access to results of all **async** tasks when they are done:

```
it("should fetch data asynchronously", async () => {
    const fakedFetchedList = [
      new QuoteModel("I love unit testing", "Mon 4, 2018")
    ];
    const quoteService =
fixture.debugElement.injector.get(QuoteService);
    let spy = spyOn(quoteService,
"fetchQuotesFromServer").and.returnValue(
      Promise.resolve(fakedFetchedList)
    );
    fixture.detectChanges();
    fixture.whenStable().then(() => {
      expect(component.fetchedList).toBe(fakedFetchedList);
    });
  });
```

9.2 Testing (Exercises)

9.2.1

is the process of testing small, isolated pieces of code. Also known as isolated testing -

9.2.2

refers to testing the functionality and of your Angular app from a user experience perspective

9.2.3

By default, Angular runs on Karma. Other test runners include ... and .... Karma provides tools that make it easier to call Jasmine tests while writing code in Angular.

Animations

Chapter 10

10.1 Introduction

🕮 10.1.1

Animation provides the illusion of motion: HTML elements change styling over time. Well-designed animations can make your application more fun and easier to use, but they aren't just cosmetic. Animations can improve your application and user experience in a number of ways:

- Without animations, web page transitions can seem abrupt and jarring.
- Motion greatly enhances the user experience, so animations give users a chance to detect the application's response to their actions.
- Good animations intuitively call the user's attention to where it is needed.

Typically, animations involve multiple style *transformations* over time. An HTML element can move, change color, grow or shrink, fade, or slide off the page. These changes can occur simultaneously or sequentially. You can control the timing of each transformation.

Angular's animation system is built on CSS functionality, which means you can animate any property that the browser considers animatable. This includes positions, sizes, transforms, colors, borders, and more. The W3C maintains a list of animatable properties on its CSS Transitions page.

🕮 10.1.2

Getting started

The main Angular modules for animations are **@angular/animations** and **@angular/platform-browser**. When you create a new project using the CLI, these dependencies are automatically added to your project.

To get started with adding Angular animations to your project, import the animation-specific modules along with standard Angular functionality.

Step 1: Enabling the animations module

Import **BrowserAnimationsModule**, which introduces the animation capabilities into your Angular root application module.

src/app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

```
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';

@NgModule({
  imports: [
    BrowserModule,
    BrowserAnimationsModule
  ],
  declarations: [ ],
  bootstrap: [ ]
})
export class AppModule { }
```

Note: When you use the CLI to create your app, the root application module **app.module.ts** is placed in the **src/app** folder.

Step 2: Importing animation functions into component files

If you plan to use specific animation functions in component files, import those functions from **@angular/animations**.

src/app/app.component.ts

```
import { Component, HostBinding } from '@angular/core';
import {
  trigger,
  state,
  style,
  animate,
  transition,
  // ...
} from '@angular/animations';
```

Note: See a summary of available animation functions at the end of this guide.

Step 3: Adding the animation metadata property

In the component file, add a metadata property called **animations:** within the **@Component()** decorator. You put the trigger that defines an animation within the **animations** metadata property.

src/app/app.component.ts

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

```
  styleUrls: ['app.component.css'],
  animations: [
    // animation triggers go here
  ]
})
```

🕮 10.1.3

Animating a transition

Let's animate a transition that changes a single HTML element from one state to another. For example, you can specify that a button displays either Open or Closed based on the user's last action. When the button is in the **open** state, it's visible and yellow. When it's the **closed** state, it's translucent and blue.

In HTML, these attributes are set using ordinary CSS styles such as color and opacity. In Angular, use the **style()** function to specify a set of CSS styles for use with animations. You can collect a set of styles in an animation state, and give the state a name, such as **open** or **closed**.

Let's create a new **open-close** component to animate with simple transitions.

Run the following command in terminal to generate the component:

ng g component open-close

This will create the component at **src/app/open-close.component.ts**.

Animation state and styles

Use Angular's **state()** function to define different states to call at the end of each transition. This function takes two arguments: a unique name like **open** or **closed** and a **style()** function.

Use the **style()** function to define a set of styles to associate with a given state name. Note that the style attributes must be in *camelCase*.

Let's see how Angular's **state()** function works with the **style()** function to set CSS style attributes. In this code snippet, multiple style attributes are set at the same time for the state. In the **open** state, the button has a height of 200 pixels, an opacity of 1, and a background color of yellow.

src/app/open-close.component.ts

```
// ...
state('open', style({
```

```
  height: '200px',
  opacity: 1,
  backgroundColor: 'yellow'
})),
```

In the **closed** state, shown below, the button has a height of 100 pixels, an opacity of 0.7, and a background color of blue.

src/app/open-close.component.ts

```
state('closed', style({
  height: '100px',
  opacity: 0.8,
  backgroundColor: '#c6ecff'
})),
```

🕮 10.1.4

Transitions and timing

In Angular, you can set multiple styles without any animation. However, without further refinement, the button instantly transforms with no fade, no shrinkage, or other visible indicator that a change is occurring.

To make the change less abrupt, we need to define an animation transition to specify the changes that occur between one state and another over a period of time. The transition() function accepts two arguments: the first argument accepts an expression that defines the direction between two transition states, and the second argument accepts one or a series of animate() steps.

Use the animate() function to define the length, delay, and easing of a transition, and to designate the style function for defining styles while transitions are taking place. You can also use the animate() function to define the keyframes() function for multi-step animations. These definitions are placed in the second argument of the animate() function.

🕮 10.1.5

Animation metadata: duration, delay, and easing

The **animate()** function (second argument of the transition function) accepts the **timings** and **styles** input parameters.

The **timings** parameter takes a string defined in three parts.

animate ('duration delay easing')

The first part, **duration**, is required. The duration can be expressed in milliseconds as a number without quotes, or in seconds with quotes and a time specifier. For example, a duration of a tenth of a second can be expressed as follows:

- As a plain number, in milliseconds: **100**
- In a string, as milliseconds: **'100ms'**
- In a string, as seconds: **'0.1s'**

The second argument, **delay**, has the same syntax as **duration**. For example:

- Wait for 100ms and then run for 200ms: **'0.2s 100ms'**

The third argument, **easing**, controls how the animation accelerates and decelerates during its runtime. For example, **ease-in** causes the animation to begin slowly, and to pick up speed as it progresses.

- Wait for 100ms, run for 200ms. Use a deceleration curve to start out fast and slowly decelerate to a resting point: **'0.2s 100ms ease-out'**
- Run for 200ms, with no delay. Use a standard curve to start slow, accelerate in the middle, and then decelerate slowly at the end: **'0.2s ease-in-out'**
- Start immediately, run for 200ms. Use an acceleration curve to start slow and end at full velocity: **'0.2s ease-in'**

Note: See the Material Design website's topic on Natural easing curves for general information on easing curves.

This example provides a state transition from **open** to **closed** with a one second transition between states.

src/app/open-close.component.ts

```
transition('open => closed', [
  animate('1s')
]),
```

In the code snippet above, the **=>** operator indicates unidirectional transitions, and **<=>** is bidirectional. Within the transition, **animate()** specifies how long the transition takes. In this case, the state change from **open** to **closed** takes one second, expressed here as **1s**.

This example adds a state transition from the **closed** state to the **open** state with a 0.5 second transition animation arc.

src/app/open-close.component.ts

```
transition('closed => open', [
```

```
  animate('0.5s')
]),
```

Note: Some additional notes on using styles within **state** and **transition** functions.

- Use **state()** to define styles that are applied at the end of each transition, they persist after the animation has completed.
- Use **transition()** to define intermediate styles, which create the illusion of motion during the animation.
- When animations are disabled, **transition()** styles can be skipped, but **state()** styles can't.
- You can include multiple state pairs within the same **transition()** argument:
- **transition('on => off, off => void')**.

🕮 10.1.6

Triggering the animation

An animation requires a *trigger*, so that it knows when to start.
The **trigger()** function collects the states and transitions, and gives the animation a name, so that you can attach it to the triggering element in the HTML template.

The **trigger()** function describes the property name to watch for changes. When a change occurs, the trigger initiates the actions included in its definition. These actions can be transitions or other functions, as we'll see later on.

In this example, we'll name the trigger **openClose**, and attach it to the **button** element. The trigger describes the open and closed states, and the timings for the two transitions.

Note: Within each **trigger()** function call, an element can only be in one state at any given time. However, it's possible for multiple triggers to be active at once.

🕮 10.1.7

Defining animations and attaching them to the HTML template

Animations are defined in the metadata of the component that controls the HTML element to be animated. Put the code that defines your animations under the **animations:** property within the **@Component()** decorator.

src/app/open-close.component.ts

```
@Component({
```

```
  selector: 'app-open-close',
  animations: [
    trigger('openClose', [
      // ...
      state('open', style({
        height: '200px',
        opacity: 1,
        backgroundColor: 'yellow'
      })),
      state('closed', style({
        height: '100px',
        opacity: 0.8,
        backgroundColor: '#c6ecff'
      })),
      transition('open => closed', [
        animate('1s')
      ]),
      transition('closed => open', [
        animate('0.5s')
      ]),
    ]),
  ],
  templateUrl: 'open-close.component.html',
  styleUrls: ['open-close.component.css']
})
export class OpenCloseComponent {
  isOpen = true;

  toggle() {
    this.isOpen = !this.isOpen;
  }

}
```

When you've defined an animation trigger for a component, you can attach it to an element in that component's template by wrapping the trigger name in brackets and preceding it with an @ symbol. Then, you can bind the trigger to a template expression using standard Angular property binding syntax as shown below, where **triggerName** is the name of the trigger, and **expression** evaluates to a defined animation state.

```
<div [@triggerName]="expression">...</div>;
```

The animation is executed or triggered when the expression value changes to a new state.

The following code snippet binds the trigger to the value of the **isOpen** property.

src/app/open-close.component.html

```
<nav>
  <button (click)="toggle()">Toggle Open/Close</button>
</nav>

<div [@openClose]="isOpen ? 'open' : 'closed'" class="open-close-container">
  <p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>
</div>
```

In this example, when the **isOpen** expression evaluates to a defined state of **open** or **closed**, it notifies the trigger **openClose** of a state change. Then it's up to the **openClose** code to handle the state change and kick off a state change animation.

For elements entering or leaving a page (inserted or removed from the DOM), you can make the animations conditional. For example, use ***ngIf** with the animation trigger in the HTML template.

Note: In the component file, set the trigger that defines the animations as the value of the **animations:** property in the **@Component()** decorator.

In the HTML template file, use the trigger name to attach the defined animations to the HTML element to be animated.

🕮 10.1.8

trigger() Starts the animation and serves as a container for all other animation function calls. The HTML template is associated with triggerName. Use the first argument to declare a unique trigger name. Uses array syntax.

style() Defines one or more CSS styles for use in animations. Controls the visual appearance of HTML elements during animation. Uses object-oriented syntax.

state() Creates a named set of CSS styles to be applied after a successful transition to a state. The state can then be referenced by name within other animation functions.

animate() Specifies the timing information for the transition. Optional values for delay easing. May contain style()calls within.

transition() Defines an animation sequence between two named states. Uses array syntax.

keyframes() Allows you to sequentially change styles over a specified time interval. Use in animate() string. Can contain multiple style()calls in each keyframe(). Uses array syntax.

group() Specifies a group of animation steps (internal animations) to be run in parallel. The animation continues only after all internal animation steps have been completed. Used in sequence() or transition().

query() Finds at least one internal HTML element in the current element.

sequence() Specifies a list of animation steps that are run sequentially, one after the other.

stagger() Extends the animation start time for multiple elements.

animation() Creates a reusable animation that can be called from elsewhere. Used in conjunction with useAnimation().

useAnimation() Activates a reusable animation. Used with animation().

animateChild() Allows animations to run on slave components at the same time as the parent.

10.2 Transitions and triggers

🕮 10.2.1

Predefined states and wildcard matching

In Angular, transition states can be defined explicitly through the **state()** function or using the predefined * (wildcard) and **void** states.

Wildcard state

An asterisk * or *wildcard* matches any animation state. This is useful for defining transitions that apply regardless of the HTML element's start or end state.

For example, a transition of **open => *** applies when the element's state changes from open to anything else.

The following is another code sample using the wildcard state together with the previous example using the **open** and **closed** states. Instead of defining each state-to-state transition pair, any transition to **closed** takes 1 second, and any transition to **open** takes 0.5 seconds.

This allows us to add new states without having to include separate transitions for each one.

src/app/open-close.component.ts

```
animations: [
  trigger('openClose', [
    // ...
    state('open', style({
      height: '200px',
      opacity: 1,
      backgroundColor: 'yellow'
    })),
    state('closed', style({
      height: '100px',
      opacity: 0.8,
      backgroundColor: '#c6ecff'
    })),
    transition('* => closed', [
      animate('1s')
    ]),
    transition('* => open', [
      animate('0.5s')
    ]),
  ]),
],
```

Use a double arrow syntax to specify state-to-state transitions in both directions.

src/app/open-close.component.ts

```
transition('open <=> closed', [
  animate('0.5s')
]),
```

📖 10.2.2

Using wildcard state with multiple transition states

In the two-state button example, the wildcard isn't that useful because there are only two possible states, **open** and **closed**. Wildcard states are better when an element in one particular state has multiple potential states that it can change to. If the button can change from **open** to either **closed** or something like **inProgress**, using a wildcard state could reduce the amount of coding needed.

src/app/open-close.component.ts

```
animations: [
  trigger('openClose', [
    // ...
    state('open', style({
      height: '200px',
      opacity: 1,
      backgroundColor: 'yellow'
    })),
    state('closed', style({
      height: '100px',
      opacity: 0.8,
      backgroundColor: '#c6ecff'
    })),
    transition('open => closed', [
      animate('1s')
    ]),
    transition('closed => open', [
      animate('0.5s')
    ]),
    transition('* => closed', [
      animate('1s')
    ]),
    transition('* => open', [
      animate('0.5s')
    ]),
    transition('open <=> closed', [
      animate('0.5s')
    ]),
    transition ('* => open', [
      animate ('1s',
        style ({ opacity: '*' }),
      ),
    ]),
    transition('* => *', [
      animate('1s')
    ]),
```

The * **=>** * transition applies when any change between two states takes place.

Transitions are matched in the order in which they are defined. Thus, you can apply other transitions on top of the * **=>** * (any-to-any) transition. For example, define style changes or animations that would apply just to **open => closed**, or just

to **closed => open**, and then use *** => *** as a fallback for state pairings that aren't otherwise called out.

To do this, list the more specific transitions *before* *** => ***.

📖 10.2.3

Using wildcards with styles

Use the wildcard * with a style to tell the animation to use whatever the current style value is, and animate with that. Wildcard is a fallback value that's used if the state being animated isn't declared within the trigger.

src/app/open-close.component.ts

```
content_copy
transition ('* => open', [
  animate ('1s',
    style ({ opacity: '*' }),
  ),
]),
```

Void state

You can use the **void** state to configure transitions for an element that is entering or leaving a page. See Animating entering and leaving a view.

Combining wildcard and void states

You can combine wildcard and void states in a transition to trigger animations that enter and leave the page:

- A transition of *** => void** applies when the element leaves a view, regardless of what state it was in before it left.
- A transition of **void => *** applies when the element enters a view, regardless of what state it assumes when entering.
- The wildcard state * matches to *any* state, including **void**.

📖 10.2.4

:enter and :leave aliases

:enter and **:leave** are aliases for the **void => *** and *** => void** transitions. These aliases are used by several animation functions.

```
transition ( ':enter', [ ... ] );  // alias for void => *
transition ( ':leave', [ ... ] );  // alias for * => void
```

It's harder to target an element that is entering a view because it isn't in the DOM yet. So, use the aliases **:enter** and **:leave** to target HTML elements that are inserted or removed from a view.

Use of *ngIf and *ngFor with :enter and :leave

The **:enter** transition runs when any ***ngIf** or ***ngFor** views are placed on the page, and **:leave** runs when those views are removed from the page.

This example has a special trigger for the enter and leave animation called **myInsertRemoveTrigger**. The HTML template contains the following code.

src/app/insert-remove.component.html

```
<div @myInsertRemoveTrigger *ngIf="isShown" class="insert-
remove-container">
  <p>The box is inserted</p>
</div>
```

In the component file, the **:enter** transition sets an initial opacity of 0, and then animates it to change that opacity to 1 as the element is inserted into the view.

src/app/insert-remove.component.ts

```
trigger('myInsertRemoveTrigger', [
  transition(':enter', [
    style({ opacity: 0 }),
    animate('100ms', style({ opacity: 1 })),
  ]),
  transition(':leave', [
    animate('100ms', style({ opacity: 0 }))
  ])
]),
```

Note that this example doesn't need to use **state()**.

:increment and :decrement in transitions

The **transition()** function takes additional selector values, **:increment** and **:decrement**. Use these to kick off a transition when a numeric value has increased or decreased in value.

Note: The following example uses **query()** and **stagger()** methods, which is discussed in the complex sequences page.

src/app/hero-list-page.component.ts

```
trigger('filterAnimation', [
  transition(':enter, * => 0, * => -1', []),
  transition(':increment', [
    query(':enter', [
      style({ opacity: 0, width: '0px' }),
      stagger(50, [
        animate('300ms ease-out', style({ opacity: 1, width:
'*' })),
      ]),
    ], { optional: true })
  ]),
  transition(':decrement', [
    query(':leave', [
      stagger(50, [
        animate('300ms ease-out', style({ opacity: 0, width:
'0px' })),
      ]),
    ])
  ]),
]),
```

🕮 10.2.5

Boolean values in transitions

If a trigger contains a boolean value as a binding value, then this value can be matched using a **transition()** expression that compares **true** and **false**, or **1** and **0**.

src/app/open-close.component.html

```
<div [@openClose]="isOpen ? true : false" class="open-close-
container">
</div>
```

In the code snippet above, the HTML template binds a ** ** element to a trigger named **openClose** with a status expression of **isOpen**, and with possible values of **true** and **false**. This is an alternative to the practice of creating two named states of **open** and **close**.

In the component code, in the **@Component** metadata under the **animations:** property, when the state evaluates to **true** (meaning "open" here), the associated HTML element's height is a wildcard style or default. In this case, use whatever height the element already had before the animation started. When

the element is "closed," the element animates to a height of 0, which makes it invisible.

src/app/open-close.component.ts

```
animations: [
  trigger('openClose', [
    state('true', style({ height: '*' })),
    state('false', style({ height: '0px' })),
    transition('false <=> true', animate(500))
  ])
],
```

🕮 10.2.6

Animation callbacks

The animation **trigger()** function emits *callbacks* when it starts and when it finishes. The example below features a component that contains an **openClose** trigger.

src/app/open-close.component.ts

```
@Component({
  selector: 'app-open-close',
  animations: [
    trigger('openClose', [
      // ...
    ]),
  ],
  templateUrl: 'open-close.component.html',
  styleUrls: ['open-close.component.css']
})
export class OpenCloseComponent {
  onAnimationEvent( event: AnimationEvent ) {
  }
}
```

In the HTML template, the animation event is passed back via **$event**, as **@trigger.start** and **@trigger.done**, where **trigger** is the name of the trigger being used. In this example, the trigger **openClose** appears as follows.

src/app/open-close.component.html

```
<div [@openClose]="isOpen ? 'open' : 'closed'"
    (@openClose.start)="onAnimationEvent($event)"
```

```
    (@openClose.done)="onAnimationEvent($event)"
    class="open-close-container">
</div>
```

A potential use for animation callbacks could be to cover for a slow API call, such as a database lookup. For example, you could set up the InProgress button to have its own looping animation where it pulsates or does some other visual motion while the backend system operation finishes.

Then, another animation can be called when the current animation finishes. For example, the button goes from the **inProgress** state to the **closed** state when the API call is completed.

An animation can influence an end user to *perceive* the operation as faster, even when it isn't. Thus, a simple animation can be a cost-effective way to keep users happy, rather than seeking to improve the speed of a server call and having to compensate for circumstances beyond your control, such as an unreliable network connection.

Callbacks can serve as a debugging tool, for example in conjunction with **console.warn()** to view the application's progress in a browser's Developer JavaScript Console. The following code snippet creates console log output for the original example, a button with the two states of **open** and **closed**.

src/app/open-close.component.ts

```
export class OpenCloseComponent {
  onAnimationEvent( event: AnimationEvent ) {
    // openClose is trigger name in this example
    console.warn(`Animation Trigger: ${event.triggerName}`);

    // phaseName is start or done
    console.warn(`Phase: ${event.phaseName}`);

    // in our example, totalTime is 1000 or 1 second
    console.warn(`Total time: ${event.totalTime}`);

    // in our example, fromState is either open or closed
    console.warn(`From: ${event.fromState}`);

    // in our example, toState either open or closed
    console.warn(`To: ${event.toState}`);

    // the HTML element itself, the button in this case
    console.warn(`Element: ${event.element}`);
  }
```

```
}
```

📖 10.2.7

Keyframes

The previous section features a simple two-state transition. Now create an animation with multiple steps run in sequence using *keyframes*.

Angular's **keyframe()** function is similar to keyframes in CSS. Keyframes allow several style changes within a single timing segment. For example, the button, instead of fading, could change color several times over a single 2-second timespan.

The code for this color change might look like this.

src/app/status-slider.component.ts

```
transition('* => active', [
  animate('2s', keyframes([
    style({ backgroundColor: 'blue' }),
    style({ backgroundColor: 'red' }),
    style({ backgroundColor: 'orange' })
  ]))
```

Offset

Keyframes include an *offset* that defines the point in the animation where each style change occurs. Offsets are relative measures from zero to one, marking the beginning and end of the animation, respectively and should be applied to each of the keyframe's steps if used at least once.

Defining offsets for keyframes is optional. If you omit them, evenly spaced offsets are automatically assigned. For example, three keyframes without predefined offsets receive offsets of 0, 0.5, and 1. Specifying an offset of 0.8 for the middle transition in the above example might look like this.

The code with offsets specified would be as follows.

src/app/status-slider.component.ts

```
transition('* => active', [
  animate('2s', keyframes([
    style({ backgroundColor: 'blue', offset: 0}),
    style({ backgroundColor: 'red', offset: 0.8}),
    style({ backgroundColor: '#754600', offset: 1.0})
```

```
  ])),
]),
transition('* => inactive', [
  animate('2s', keyframes([
    style({ backgroundColor: '#754600', offset: 0}),
    style({ backgroundColor: 'red', offset: 0.2}),
    style({ backgroundColor: 'blue', offset: 1.0})
  ]))
]),
```

You can combine keyframes with **duration**, **delay**, and **easing** within a single animation.

🕮 10.2.8

Keyframes with a pulsation

Use keyframes to create a pulse effect in your animations by defining styles at specific offset throughout the animation.

Here's an example of using keyframes to create a pulse effect:

- The original **open** and **closed** states, with the original changes in height, color, and opacity, occurring over a timeframe of 1 second.
- A keyframes sequence inserted in the middle that causes the button to appear to pulsate irregularly over the course of that same 1-second timeframe.

The code snippet for this animation might look like this.

src/app/open-close.component.ts

```
trigger('openClose', [
  state('open', style({
    height: '200px',
    opacity: 1,
    backgroundColor: 'yellow'
  })),
  state('close', style({
    height: '100px',
    opacity: 0.5,
    backgroundColor: 'green'
  })),
  // ...
```

```
  transition('* => *', [
    animate('1s', keyframes ( [
      style({ opacity: 0.1, offset: 0.1 }),
      style({ opacity: 0.6, offset: 0.2 }),
      style({ opacity: 1,   offset: 0.5 }),
      style({ opacity: 0.2, offset: 0.7 })
    ]))
  ])
])
```

📖 10.2.9

Animatable properties and units

Angular's animation support builds on top of web animations, so you can animate any property that the browser considers animatable. This includes positions, sizes, transforms, colors, borders, and more. The W3C maintains a list of animatable properties on its CSS Transitions page.

For positional properties with a numeric value, define a unit by providing the value as a string, in quotes, with the appropriate suffix:

- 50 pixels: **'50px'**
- Relative font size: **'3em'**
- Percentage: **'100%'**

If you don't provide a unit when specifying dimension, Angular assumes a default unit of pixels, or px. Expressing 50 pixels as **50** is the same as saying **'50px'**.

Automatic property calculation with wildcards

Sometimes you don't know the value of a dimensional style property until runtime. For example, elements often have widths and heights that depend on their content and the screen size. These properties are often challenging to animate using CSS.

In these cases, you can use a special wildcard * property value under **style()**, so that the value of that particular style property is computed at runtime and then plugged into the animation.

The following example has a trigger called **shrinkOut**, used when an HTML element leaves the page. The animation takes whatever height the element has before it leaves, and animates from that height to zero.

src/app/hero-list-auto.component.ts

```
content_copy
```

```
animations: [
  trigger('shrinkOut', [
    state('in', style({ height: '*' })),
    transition('* => void', [
      style({ height: '*' }),
      animate(250, style({ height: 0 }))
    ])
  ])
]
```

Keyframes summary

The **keyframes()** function in Angular allows you to specify multiple interim styles within a single transition, with an optional offset to define the point in the animation where each style change occurs.

10.3 Complex animation sequences

🕮 10.3.1

Animate multiple elements using query() and stagger() functions

The **query()** function allows you to find inner elements within the element that is being animated. This function targets specific HTML elements within a parent component and applies animations to each element individually. Angular intelligently handles setup, teardown, and cleanup as it coordinates the elements across the page.

The **stagger()** function allows you to define a timing gap between each queried item that is animated and thus animates elements with a delay between them.

The Filter/Stagger tab in the live example shows a list of heroes with an introductory sequence. The entire list of heroes cascades in, with a slight delay from top to bottom.

The following example demonstrates how to use **query()** and **stagger()** functions on the entry of an animated element.

- Use **query()** to look for an element entering the page that meets certain criteria.
- For each of these elements, use **style()** to set the same initial style for the element. Make it invisible and use **transform** to move it out of position so that it can slide into place.
- Use **stagger()** to delay each animation by 30 milliseconds.

- Animate each element on screen for 0.5 seconds using a custom-defined easing curve, simultaneously fading it in and un-transforming it.

src/app/hero-list-page.component.ts

```
animations: [
    trigger('pageAnimations', [
      transition(':enter', [
        query('.hero, form', [
          style({opacity: 0, transform: 'translateY(-
100px)'}),
          stagger(-30, [
            animate('500ms cubic-bezier(0.35, 0, 0.25, 1)',
style({ opacity: 1, transform: 'none' }))
          ])
        ])
      ])
    ]),
  ]
})
export class HeroListPageComponent implements OnInit {
  @HostBinding('@pageAnimations')
  public animatePage = true;

  heroTotal = -1;
  get heroes() { return this._heroes; }
  private _heroes: Hero[] = [];

  ngOnInit() {
    this._heroes = HEROES;
  }

  updateCriteria(criteria: string) {
    criteria = criteria ? criteria.trim() : '';

    this._heroes = HEROES.filter(hero =>
hero.name.toLowerCase().includes(criteria.toLowerCase()));
    const newTotal = this.heroes.length;

    if (this.heroTotal !== newTotal) {
      this.heroTotal = newTotal;
    } else if (!criteria) {
      this.heroTotal = -1;
    }
  }
```

```
}
```

📖 10.3.2

Parallel animation using group() function

You've seen how to add a delay between each successive animation. But you may also want to configure animations that happen in parallel. For example, you may want to animate two CSS properties of the same element but use a different **easing** function for each one. For this, you can use the animation **group()** function.

Note: The **group()** function is used to group animation *steps*, rather than animated elements.

In the following example, using groups on both **:enter** and **:leave** allow for two different timing configurations. They're applied to the same element in parallel, but run independently.

src/app/hero-list-groups.component.ts (excerpt)

```
animations: [
  trigger('flyInOut', [
    state('in', style({
      width: 120,
      transform: 'translateX(0)', opacity: 1
    })),
    transition('void => *', [
      style({ width: 10, transform: 'translateX(50px)',
opacity: 0 }),
      group([
        animate('0.3s 0.1s ease', style({
          transform: 'translateX(0)',
          width: 120
        })),
        animate('0.3s ease', style({
          opacity: 1
        }))
      ])
    ]),
    transition('* => void', [
      group([
        animate('0.3s ease', style({
          transform: 'translateX(50px)',
          width: 10
```

```
        })),
        animate('0.3s 0.2s ease', style({
          opacity: 0
        }))
      ])
    ])
  ])
]
```

📖 10.3.3

Sequential vs. parallel animations

Complex animations can have many things happening at once. But what if you want to create an animation involving several animations happening one after the other? Earlier we used **group()** to run multiple animations all at the same time, in parallel.

A second function called **sequence()** lets you run those same animations one after the other. Within **sequence()**, the animation steps consist of either **style()** or **animate()** function calls.

- Use **style()** to apply the provided styling data immediately.
- Use **animate()** to apply styling data over a given time interval.

📖 10.3.4

Filter animation example

Let's take a look at another animation on the live example page. Under the Filter/Stagger tab, enter some text into the Search Heroes text box, such as **Magnet** or **tornado**.

The filter works in real time as you type. Elements leave the page as you type each new letter and the filter gets progressively stricter. The heroes list gradually re-enters the page as you delete each letter in the filter box.

The HTML template contains a trigger called **filterAnimation**.

src/app/hero-list-page.component.html

```
<ul class="heroes" [@filterAnimation]="heroTotal">
</ul>
```

The component file contains three transitions.

src/app/hero-list-page.component.ts

```
@Component({
  animations: [
    trigger('filterAnimation', [
      transition(':enter, * => 0, * => -1', []),
      transition(':increment', [
        query(':enter', [
          style({ opacity: 0, width: '0px' }),
          stagger(50, [
            animate('300ms ease-out', style({ opacity: 1,
width: '*' })),
          ]),
        ], { optional: true })
      ]),
      transition(':decrement', [
        query(':leave', [
          stagger(50, [
            animate('300ms ease-out', style({ opacity: 0,
width: '0px' })),
          ]),
        ])
      ]),
    ]),
  ]
})
export class HeroListPageComponent implements OnInit {
  heroTotal = -1;
}
```

The animation does the following:

- Ignores any animations that are performed when the user first opens or navigates to this page. The filter narrows what is already there, so it assumes that any HTML elements to be animated already exist in the DOM.
- Performs a filter match for matches.

For each match:

- Hides the element by making it completely transparent and infinitely narrow, by setting its opacity and width to 0.
- Animates in the element over 300 milliseconds. During the animation, the element assumes its default width and opacity.
- If there are multiple matching elements, staggers in each element starting at the top of the page, with a 50-millisecond delay between each element.

🕮 10.3.5

Animation sequence summary

Angular functions for animating multiple elements start with **query()** to find inner elements, for example gathering all images within a ** **. The remaining functions, **stagger()**, **group()**, and **sequence()**, apply cascades or allow you to control how multiple animation steps are applied.

10.4 Reusable Animations

🕮 10.4.1

Creating reusable animations

To create a reusable animation, use the **animation()** method to define an animation in a separate **.ts** file and declare this animation definition as a **const** export variable. You can then import and reuse this animation in any of your application components using the **useAnimation()** API.

src/app/animations.ts

```
import { animation, style, animate, trigger, transition,
useAnimation } from '@angular/animations';

export const transitionAnimation = animation([
  style({
    height: '{{ height }}',
    opacity: '{{ opacity }}',
    backgroundColor: '{{ backgroundColor }}'
  }),
  animate('{{ time }}')
]);
```

In the above code snippet, **transAnimation** is made reusable by declaring it as an export variable.

Note: The **height**, **opacity**, **backgroundColor**, and **time** inputs are replaced during runtime.

You can also export a part of an animation. For example, the following snippet exports the animation **trigger**.

src/app/animations.1.ts

```
import { animation, style, animate, trigger, transition,
useAnimation } from '@angular/animations';
export const triggerAnimation = trigger('openClose', [
  transition('open => closed', [
    useAnimation(transitionAnimation, {
      params: {
        height: 0,
        opacity: 1,
        backgroundColor: 'red',
        time: '1s'
      }
    })
  ])
]);
```

From this point, you can import resuable animation variables in your component class. For example, the following code snippet imports the **transAnimation** variable for use in the **useAnimation()** method.

src/app/open-close.component.ts

```
import { Component } from '@angular/core';
import { transition, trigger, useAnimation } from
'@angular/animations';
import { transAnimation } from './animations';

@Component({
  selector: 'app-open-close-reusable',
  animations: [
    trigger('openClose', [
      transition('open => closed', [
        useAnimation(transAnimation, {
          params: {
            height: 0,
            opacity: 1,
            backgroundColor: 'red',
            time: '1s'
          }
        })
      ])
    ])
  ],
  templateUrl: 'open-close.component.html',
  styleUrls: ['open-close.component.css']
})
```

10.5 Route transition animations

🕮 10.5.1

Routing enables users to navigate between different routes in an application. When a user navigates from one route to another, the Angular router maps the URL path to a relevant component and displays its view. Animating this route transition can greatly enhance the user experience.

The Angular router comes with high-level animation functions that let you animate the transitions between views when a route changes. To produce an animation sequence when switching between routes, you need to define nested animation sequences. Start with the top-level component that hosts the view, and nest additional animations in the components that host the embedded views.

To enable routing transition animation, do the following:

1. Import the routing module into the application and create a routing configuration that defines the possible routes.
2. Add a router outlet to tell the Angular router where to place the activated components in the DOM.
3. Define the animation.

Let's illustrate a router transition animation by navigating between two routes, *Home* and *About* associated with the **HomeComponent** and **AboutComponent** views respectively. Both of these component views are children of the top-most view, hosted by **AppComponent**. We'll implement a router transition animation that slides in the new view to the right and slides out the old view when the user navigates between the two routes.

🕮 10.5.2

Route configuration

To begin, configure a set of routes using methods available in the **RouterModule** class. This route configuration tells the router how to navigate.

Use the **RouterModule.forRoot** method to define a set of routes. Also, import this **RouterModule** to the **imports** array of the main module, **AppModule**.

Note: Use the **RouterModule.forRoot** method in the root module, **AppModule**, to register top-level application routes and providers. For feature modules, call the **RouterModule.forChild** method to register additional routes.

The following configuration defines the possible routes for the application.

src/app/app.module.ts

```
content_copy
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';
import { RouterModule } from '@angular/router';
import { AppComponent } from './app.component';
import { OpenCloseComponent } from './open-close.component';
import { OpenClosePageComponent } from './open-close-
page.component';
import { OpenCloseChildComponent } from './open-
close.component.4';
import { ToggleAnimationsPageComponent } from './toggle-
animations-page.component';
import { StatusSliderComponent } from './status-
slider.component';
import { StatusSliderPageComponent } from './status-slider-
page.component';
import { HeroListPageComponent } from './hero-list-
page.component';
import { HeroListGroupPageComponent } from './hero-list-group-
page.component';
import { HeroListGroupsComponent } from './hero-list-
groups.component';
import { HeroListEnterLeavePageComponent } from './hero-list-
enter-leave-page.component';
import { HeroListEnterLeaveComponent } from './hero-list-
enter-leave.component';
import { HeroListAutoCalcPageComponent } from './hero-list-
auto-page.component';
import { HeroListAutoComponent } from './hero-list-
auto.component';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';
import { InsertRemoveComponent } from './insert-
remove.component';


@NgModule({
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    RouterModule.forRoot([
```

```
      { path: '', pathMatch: 'full', redirectTo: '/enter-
leave' },
      { path: 'open-close', component: OpenClosePageComponent
},
      { path: 'status', component: StatusSliderPageComponent
},
      { path: 'toggle', component:
ToggleAnimationsPageComponent },
      { path: 'heroes', component: HeroListPageComponent,
data: {animation: 'FilterPage'} },
      { path: 'hero-groups', component:
HeroListGroupPageComponent },
      { path: 'enter-leave', component:
HeroListEnterLeavePageComponent },
      { path: 'auto', component: HeroListAutoCalcPageComponent
},
      { path: 'insert-remove', component:
InsertRemoveComponent},
      { path: 'home', component: HomeComponent, data:
{animation: 'HomePage'} },
      { path: 'about', component: AboutComponent, data:
{animation: 'AboutPage'} },

    ])
  ],
```

The **home** and **about** paths are associated with the **HomeComponent** and **AboutComponent** views. The route configuration tells the Angular router to instantiate the **HomeComponent** and **AboutComponent** views when the navigation matches the corresponding path.

In addition to **path** and **component**, the **data** property of each route defines the key animation-specific configuration associated with a route. The **data** property value is passed into **AppComponent** when the route changes. You can also pass additional data in route configuration that is consumed within the animation. The data property value has to match the transitions defined in the **routeAnimation** trigger, which we'll define later.

Note: The **data** property names that you use can be arbitrary. For example, the name *animation* used in the example above is an arbitrary choice.

🕮 10.5.3

Router outlet

After configuring the routes, tell the Angular router where to render the views when matched with a route. You can set a router outlet by inserting a **<router-outlet>** container inside the root **AppComponent** template.

The **<router-outlet>** container has an attribute directive that contains data about active routes and their states, based on the **data** property that we set in the route configuration.

src/app/app.component.html

```
content_copy
<div [@routeAnimations]="prepareRoute(outlet)">
  <router-outlet #outlet="outlet"></router-outlet>
</div>
```

AppComponent defines a method that can detect when a view changes. The method assigns an animation state value to the animation trigger (**@routeAnimation**) based on the route configuration **data** property value. Here's an example of an **AppComponent** method that detects when a route change happens.

src/app/app.component.ts

```
content_copy
prepareRoute(outlet: RouterOutlet) {
  return outlet && outlet.activatedRouteData &&
outlet.activatedRouteData.animation;
}
```

Here, the **prepareRoute()** method takes the value of the outlet directive (established through **#outlet="outlet"**) and returns a string value representing the state of the animation based on the custom data of the current active route. You can use this data to control which transition to execute for each route.

🕮 10.5.4

Animation definition

Animations can be defined directly inside your components. For this example we are defining the animations in a separate file, which allows us to re-use the animations.

The following code snippet defines a reusable animation named **slideInAnimation**.

src/app/animations.ts

```
export const slideInAnimation =
  trigger('routeAnimations', [
    transition('HomePage <=> AboutPage', [
      style({ position: 'relative' }),
      query(':enter, :leave', [
        style({
          position: 'absolute',
          top: 0,
          left: 0,
          width: '100%'
        })
      ]),
      query(':enter', [
        style({ left: '-100%' })
      ]),
      query(':leave', animateChild()),
      group([
        query(':leave', [
          animate('300ms ease-out', style({ left: '100%' }))
        ]),
        query(':enter', [
          animate('300ms ease-out', style({ left: '0%' }))
        ])
      ]),
      query(':enter', animateChild()),
    ]),
    transition('* <=> FilterPage', [
      style({ position: 'relative' }),
      query(':enter, :leave', [
        style({
          position: 'absolute',
          top: 0,
          left: 0,
          width: '100%'
        })
      ]),
      query(':enter', [
        style({ left: '-100%' })
      ]),
      query(':leave', animateChild()),
      group([
        query(':leave', [
          animate('200ms ease-out', style({ left: '100%' }))
        ]),
```

```
        query(':enter', [
          animate('300ms ease-out', style({ left: '0%' }))
        ])
      ]),
      query(':enter', animateChild()),
    ])
  ]);
```

The animation definition does several things:

- Defines two transitions. A single trigger can define multiple states and transitions.
- Adjusts the styles of the host and child views to control their relative positions during the transition.
- Uses **query()** to determine which child view is entering and which is leaving the host view.

A route change activates the animation trigger, and a transition matching the state change is applied.

Note: The transition states must match the **data** property value defined in the route configuration.

Make the animation definition available in your application by adding the reusable animation (**slideInAnimation**) to the **animations** metadata of the **AppComponent**.

src/app/app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css'],
  animations: [
    slideInAnimation
    // animation triggers go here
  ]
})
```

🕮 10.5.5

Styling the host and child components

During a transition, a new view is inserted directly after the old one and both elements appear on screen at the same time. To prevent this, apply additional styling to the host view, and to the removed and inserted child views. The host view must use relative positioning, and the child views must use absolute positioning.

Adding styling to the views animates the containers in place, without the DOM moving things around.

src/app/animations.ts

```
trigger('routeAnimations', [
  transition('HomePage <=> AboutPage', [
    style({ position: 'relative' }),
    query(':enter, :leave', [
      style({
        position: 'absolute',
        top: 0,
        left: 0,
        width: '100%'
      })
    ]),
```

Querying the view containers

Use the **query()** method to find and animate elements within the current host component. The **query(":enter")** statement returns the view that is being inserted, and **query(":leave")** returns the view that is being removed.

Let's assume that we are routing from the *Home => About*.

src/app/animations.ts (Continuation from above)

```
query(':enter', [
    style({ left: '-100%' })
  ]),
  query(':leave', animateChild()),
  group([
    query(':leave', [
      animate('300ms ease-out', style({ left: '100%' }))
    ]),
    query(':enter', [
      animate('300ms ease-out', style({ left: '0%' }))
    ])
  ]),
  query(':enter', animateChild()),
]),
transition('* <=> FilterPage', [
  style({ position: 'relative' }),
  query(':enter, :leave', [
    style({
      position: 'absolute',
```

```
      top: 0,
      left: 0,
      width: '100%'
    })
  ]),
  query(':enter', [
    style({ left: '-100%' })
  ]),
  query(':leave', animateChild()),
  group([
    query(':leave', [
      animate('200ms ease-out', style({ left: '100%' }))
    ]),
    query(':enter', [
      animate('300ms ease-out', style({ left: '0%' }))
    ])
  ]),
  query(':enter', animateChild()),
])
```

The animation code does the following after styling the views:

- **query(':enter', style({ left: '-100%' }))** matches the view that is added and hides the newly added view by positioning it to the far left.
- Calls **animateChild()** on the view that is leaving, to run its child animations.
- Uses **group()** function to make the inner animations run in parallel.
- Within the **group()** function:
- Queries the view that is removed and animates it to slide far to the right.
- Slides in the new view by animating the view with an easing function and duration.
- This animation results in the **about** view sliding from the left to right.
- Calls the **animateChild()** method on the new view to run its child animations after the main animation completes.

10.6 Animations (Exercises)

10.6.1

Finds one or more inner HTML elements within the current element.

10.6.2

Creates a named set of CSS styles that should be applied on successful transition to a given state. The state can then be referenced by name within other animation functions.

10.6.3

Kicks off the animation and serves as a container for all other animation function calls. HTML template binds to **triggerName**. Use the first argument to declare a unique trigger name. Uses array syntax.

10.6.4

The wildcard state ... matches any state, including **void**.

10.6.5

function in Angular allows you to specify multiple interim styles within a single transition, with an optional offset to define the point in the animation where each style change occurs.

10.6.6

Use ... to delay each animation by 30 milliseconds.

10.6.7

To enable routing transition animation, do the following:

1. import the routing module into your application and create a routing configuration that defines possible routes.
2. add a router socket to tell the Angular router where to place the activated components in the DOM.
3. define the animation.

- True
- False

PWA

Chapter 11

11.1 PWA

🕮 11.1.1

The two main requirements of a PWA are a Service Worker and a Web Manifest. While it's possible to add both of these to an app manually, the Angular team has an **@angular/pwa** package that can be used to automate this.

The **@angular/pwa** package will automatically add a service worker and an app manifest to the app. To add this package to the app, run:

```
$ ng add @angular/pwa
```

Once this package has been added run **ionic build --prod** and the **www** directory will be ready to deploy as a PWA.

> By default, the **@angular/pwa** package comes with the Angular logo for the app icons. Be sure to update the manifest to use the correct app name and also replace the icons.
>
> Note: Features like Service Workers and many JavaScript APIs (such as geolocation) require the app be hosted in a secure context. When deploying an app through a hosting service, be aware that HTTPS will be required to take full advantage of Service Workers.

Service Worker configuration

After **@angular/pwa** has been added, a new **ngsw-config.json** file will be created at the root of the project. This file is responsible for configuring how Angular's service worker mechanism will handle caching assets. By default, the following will be provided:

```
{
  "$schema": "./node_modules/@angular/service-
worker/config/schema.json",
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.html",
          "/*.css",
          "/*.js"
        ]
```

```
      }
    }, {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "resources": {
        "files": [
          "/assets/**",

"/*.(eot|svg|cur|jpg|png|webp|gif|otf|ttf|woff|woff2|ani)"
        ]
      }
    }
  ]
}
```

There are two sections in here, one for app specific resources (JS, CSS, HTML) and assets the app will load on demand. Depending on your app, these options can be customized. For a more detailed guide, read the official guide from the Angular Team.

Deploying

Firebase

Firebase hosting provides many benefits for Progressive Web Apps, including fast response times thanks to CDNs, HTTPS enabled by default, and support for HTTP2 push.

First, if not already available, create the project in Firebase.

Next, in a Terminal, install the Firebase CLI:

```
$ npm install -g firebase-tools
```

> Note: If it's the first time you use firebase-tools, login to your Google account with **firebase login** command.

With the Firebase CLI installed, run **firebase init** within your Ionic project. The CLI prompts:

"**Which Firebase CLI features do you want to set up for this folder?**" Choose "Hosting: Configure and deploy Firebase Hosting sites."

"**Select a default Firebase project for this directory:**" Choose the project you created on the Firebase website.

"**What do you want to use as your public directory?**" Enter "www".

> Note: Answering these next two questions will ensure that routing, hard reload, and deep linking work in the app:

Configure as a single-page app (rewrite all urls to /index.html)?" Enter "Yes".

"**File www/index.html already exists. Overwrite?**" Enter "No".

A **firebase.json** config file is generated, configuring the app for deployment.

The last thing needed is to make sure caching headers are being set correctly. To do this, add a **headers** snippet to the **firebase.json** file. The complete **firebase.json** looks like:

```
{
  "hosting": {
    "public": "www",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ],
    "rewrites": [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ],
    "headers": [
      {
        "source": "/build/app/**",
        "headers": [
          {
            "key": "Cache-Control",
            "value": "public, max-age=31536000"
          }
        ]
      },
      {
        "source": "ngsw-worker.js",
        "headers": [
          {
            "key": "Cache-Control",
            "value": "no-cache"
          }
```

```
          ]
        }
      ]
    }
}
```

For more information about the **firebase.json** properties, see the Firebase documentation.

Next, build an optimized version of the app by running:

```
$ ionic build --prod
```

Last, deploy the app by running:

```
$ firebase deploy
```

After this completes, the app will be live.

Web Workers

Chapter 12

12.1 Web workers

🕮 12.1.1

Background processing using web workersWeb workers allow you to run CPU-intensive computations in a background thread, freeing the main thread to update the user interface. If you find your application performs a lot of computations, such as generating CAD drawings or doing heavy geometrical calculations, using web workers can help increase your application's performance.

The CLI does not support running Angular itself in a web worker.

Adding a web worker

To add a web worker to an existing project, use the Angular CLI **ng generate** command.

```
ng generate web-worker <location>
```

You can add a web worker anywhere in your application. For example, to add a web worker to the root component, **src/app/app.component.ts**, run the following command.

```
ng generate web-worker app
```

The command performs the following actions.

- Configures your project to use web workers, if it isn't already.
- Adds the following scaffold code to **src/app/app.worker.ts** to receive messages.
- src/app/app.worker.ts

```
addEventListener('message', ({ data }) => {
  const response = `worker response to ${data}`;
  postMessage(response);
});
```

- Adds the following scaffold code to **src/app/app.component.ts** to use the worker.
- src/app/app.component.ts

```
if (typeof Worker !== 'undefined') {
  // Create a new
  const worker = new Worker(new URL('./app.worker',
import.meta.url));
  worker.onmessage = ({ data }) => {
    console.log(`page got message: ${data}`);
```

```
  };
  worker.postMessage('hello');
} else {
  // Web workers are not supported in this environment.
  // You should add a fallback so that your program still
executes correctly.
}
```

After you generate this initial scaffold, you must refactor your code to use the web worker by sending messages to and from the worker.

Some environments or platforms, such as **@angular/platform-server** used in Server-side Rendering, don't support web workers. To ensure that your application will work in these environments, you must provide a fallback mechanism to perform the computations that the worker would otherwise perform.

Vue

Introduction

Chapter 13

13.1 What is VueJS

🕮 13.1.1

We live in a golden era of *JavaScript* libraries and frameworks. Many of web systems are based on modern frameworks and bring functions from web and mobile application to web browsers.

Vue (pronounced /vjuː/, like **view**) is a framework for building user interfaces based on *JavaScript*. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable.

The core library is focused on the view layer only and is easy to pick up and integrate with other libraries or existing projects. Vue is also capable of powering sophisticated Single-Page Applications.

🕮 13.1.2

Vue is only one of several frontend frameworks. The biggest competition are:

- **React;** React and Vue share many similarities. They both utilize a virtual DOM, provide reactive and composable view components. React and Vue are exceptionally and similarly fast. For large applications, both Vue and React offer robust routing solutions. Important difference is that Vue's companion libraries for state management and routing are all officially supported and kept up to date with the core library. Positive for React is its "extension" React Native that enables writing native-rendered apps for iOS and Android using the same React component model.
- **AngularJS** - Vue is much simpler than AngularJS, both are in terms of API and design. Learning enough to build non-trivial applications typically takes less than a day, which is not true for AngularJS. Some of Vue's syntax will look very similar to AngularJS. It is because Angular was an inspiration in early development phase. Vue is much simpler than AngularJS. Learning enough to build non-trivial applications typically takes less than a day, which is not true for AngularJS. AngularJS has strong opinions about how applications should be structured, while Vue is a more flexible.

Compared to other frameworks, Vue is more approachable and has not as steep of a learning curve.

🕮 13.1.3

The installation of VueJS is simple, and beginners can easily understand and start building their own user interfaces. VueJS is created by Evan You, an ex-employee from Google. The first version of VueJS was released in Feb 2014.

🕮 13.1.4

VueJS uses a virtual DOM, with code-based changes not being made directly in the DOM, but in its replica, which is mirrored in the *JavaScript* data structures.

Whenever any changes are made in the JavaScript data structures at first. Then the changes are compared to their content in the original data structure. The real changes are then projected to the real DOM, which changes are displayed to the user. This approach brings optimization, it's more efficient and changes (both visually and application) are made faster.

🕮 13.1.5

The data binding is a feature which helps manipulate with:

- variables,
- HTML attributes,
- styles/CSS styles,
- using commands like for, if etc. in html parts,
- and many features that will be presented later

Special event handling is defined for DOM elements to listen to the events in VueJS.

🕮 13.1.6

Computed properties help to listen to the changes made using UI elements, performs the necessary calculations and applied changes of variables automatically to DOM elements.

Watchers are usually applied in reverse form. When data are changed in GUI, the watchers make changes in data model. Watcher takes care of handling any data changes making the code simple and fast.

🕮 13.1.7

The connection between DOM and the Vue instance data is based on templates. Vue compiles the templates into virtual DOM model.

If we use more than one page in our application, we can use routing which heps to create URI and navigation between them.

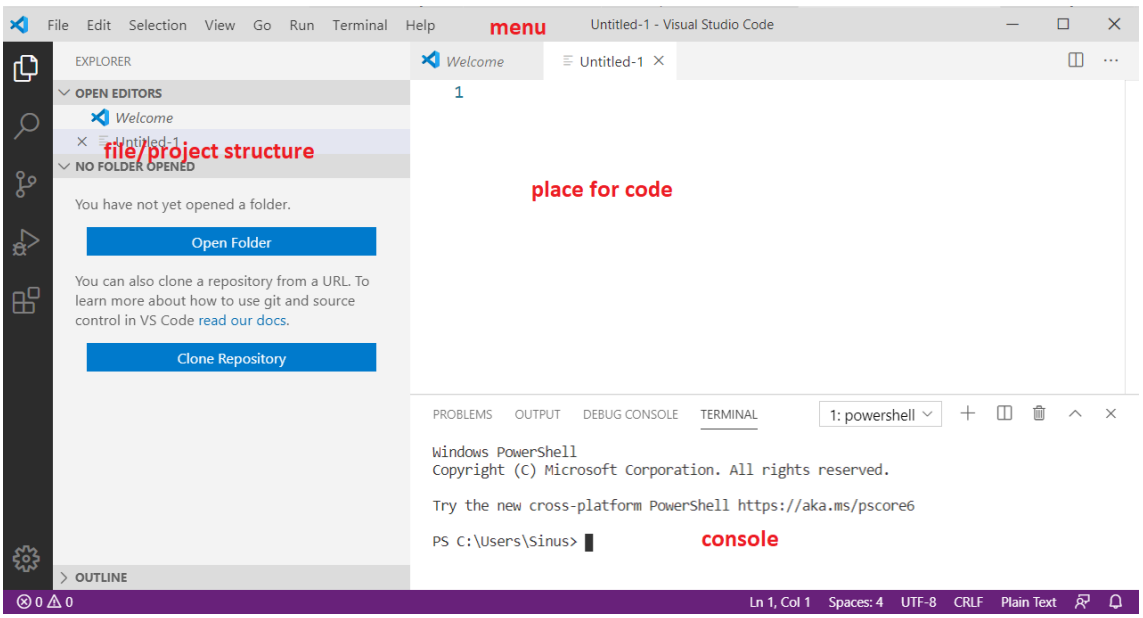
13.2 Let's go to start

🕮 13.2.1

We need some development environment for building app based on combination of JavaScript, HTML and CSS. The best offer is Visual Studio Code, what is open-source and allow to use command for building, configuring, adding plugins and components, building app. etc.

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Java, Python, PHP, Go) and runtimes (such as .NET and Unity).

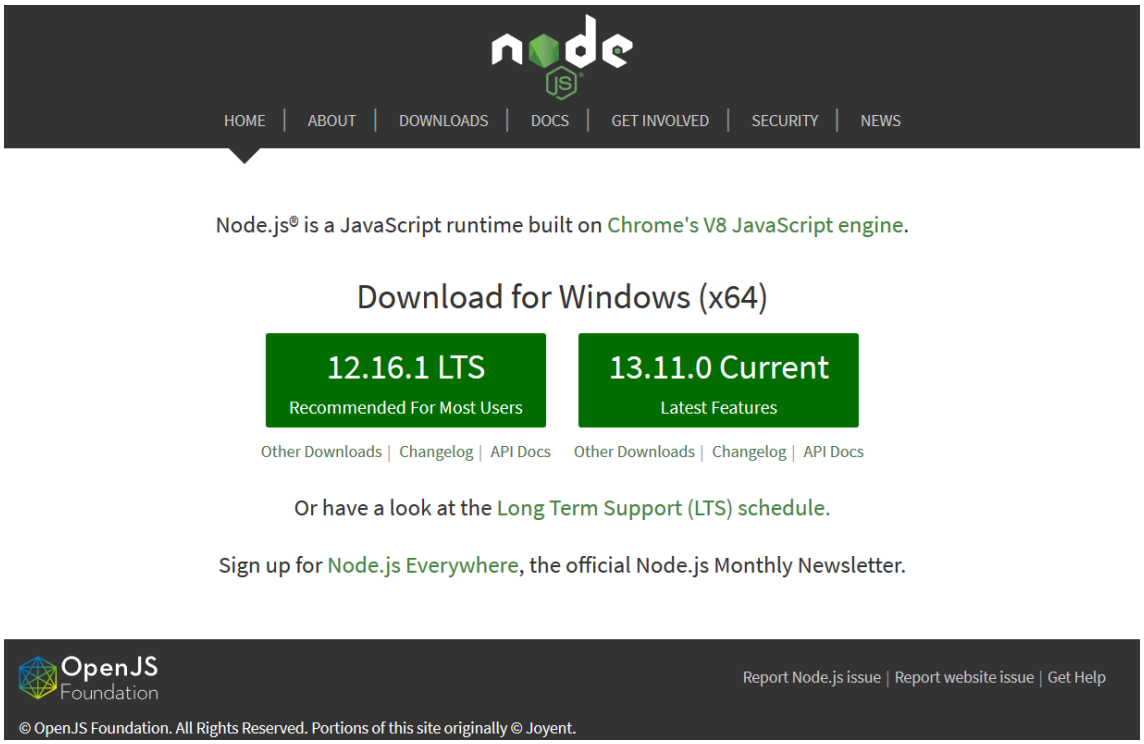
The download link: https://code.visualstudio.com/



🕮 13.2.2

NodeJS is a second inevitable part of development platform. You can create Vue application without NodeJS too, but it is complicated, because thhe best way for download libraries for you application and hold them update is node package manager (npm). To use **npm** for adding and updating libraries, components and files is common - many of component has defined path for npm instalation.

To install NodeJS you can use: https://nodejs.org/en/



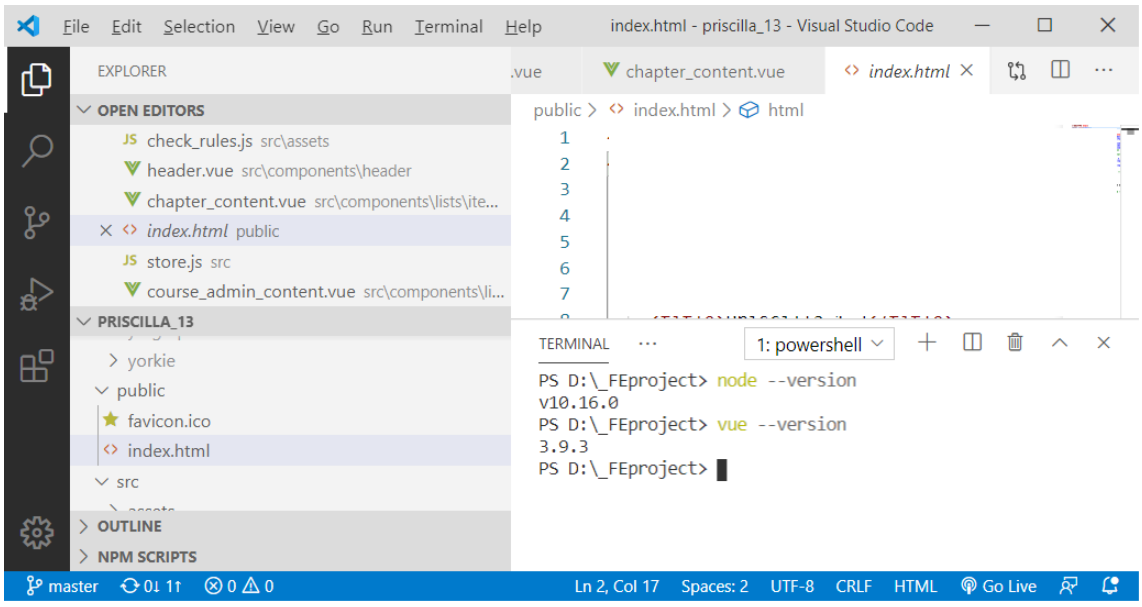
🕮 13.2.3

After NodeJS installation and *Visual Studio Code* running, we have empty environment.

We need to show terminal for application structure creating. We can show it using **Ctrl +** ` or *View -> Terminal*.

We can check our version of npm and vesion of Vue using command:

```
node --version
vue --version
```



13.2.4

Our goal is to use Vue in modern form with many features that makes programming easier. We want to use Vue CLI (command line interface) 3 what is a way for developers to get their Vue applications up and running as fast as possible, without thinking about configuration.

For using Vue CLI3 we need Node version 8.9 above. If you have lower version, you must upgrade (install latest version of NodeJS): https://nodejs.org

We can now install the new CLI version:

```
npm install -g @vue/cli
```

Now you ready to create project.

13.2.5

We have prepared tools for creating projects. To create project structure, we can use some templates.

We will initialize new project with name *my_first*. The directory for project will be created in directory where we are set in system:

```
PS D:\_FEproject> vue create my_first
```

The system started process of building after run.

We can set or change the default settings of project plugins:

```
? Please pick a preset: default (babel, eslint)?
```

We let default values and press *Enter*.

```
✨  Creating project in D:\_FEproject\my_first.
????  Initializing git repository...
⚙  Installing CLI plugins. This might take a while...
...
🎉  Successfully created project my_first.
🎉  Get started with the following commands:

 $ cd my_first
 $ npm run serve

PS D:\_FEproject>
```

After finishing project, we have ready application in new directory - **my_first**.

📖 13.2.6

Project structure was created, and the system writes instructions for its run:

```
🎉  Get started with the following commands:

 $ cd my_first
 $ npm run serve

PS D:\_FEproject>
```

We have to go inside of directory of project:

```
PS D:\_FEproject>cd my_first
PS D:\_FEproject\my_first>
```

To run procect we should write command:

```
PS D:\_FEproject\my_first> npm run serve
```

🕮 13.2.7

After installing we can start our project.

The old procedure used npm run dev.

In vueCLI3 we use:

```
nmp run serve
```

🕮 13.2.8

Command

```
npm run serve
```

started process of starting development server and preparing application to run.

The valuable information how to run application are added to the end of output:

```
 App running at:
 - Local:  http://localhost:8080/
 - Network: http://10.10.10.22:8080/
```

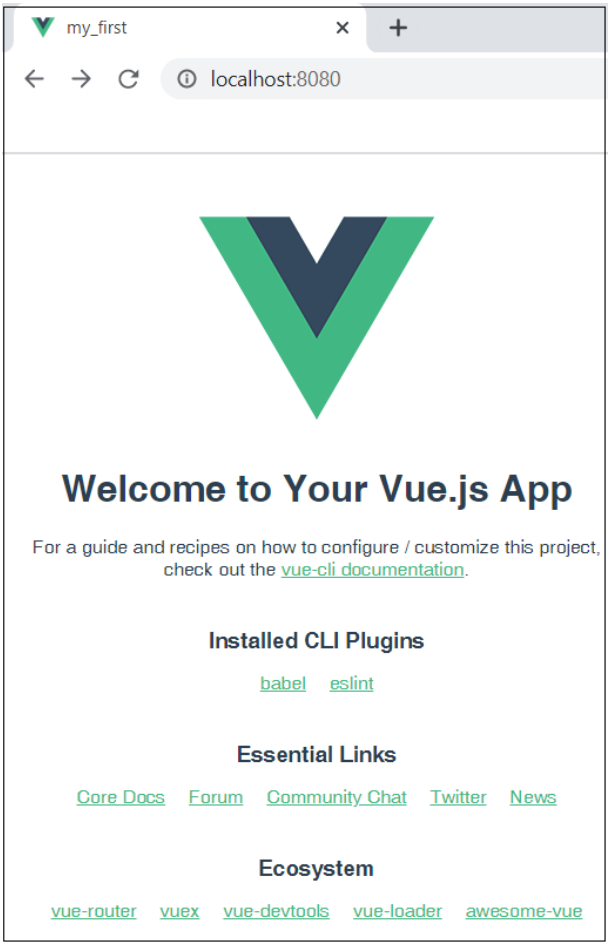
The port is usually 8080, but if this one is occupied, system found first next free port (e.g. 8081, 8082 etc.)

🕮 13.2.9

To start our application, we have to write to the address field in browser

```
http://localhost:8080/
```

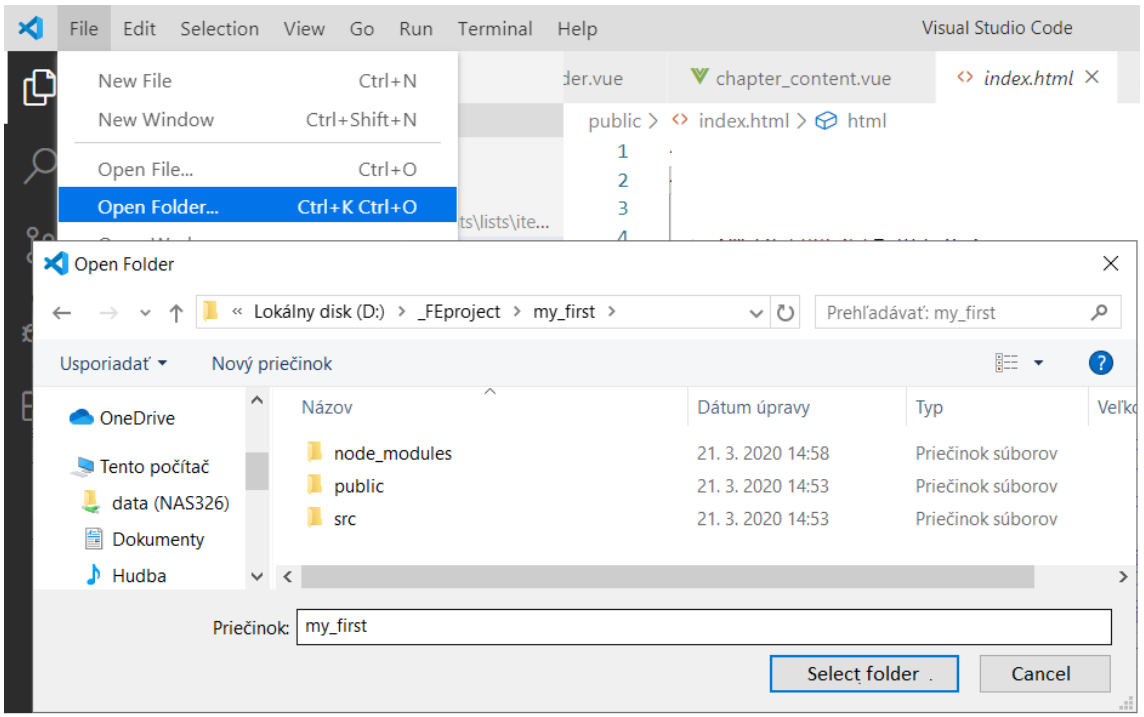
and we have default content placed into default application in vue:



13.3 Application structure

🕮 13.3.1

To inspect structure of created project we can open directory with source code. If we use Visual Code Studio, we can open folder using **File - Open folder** and directory with created project:

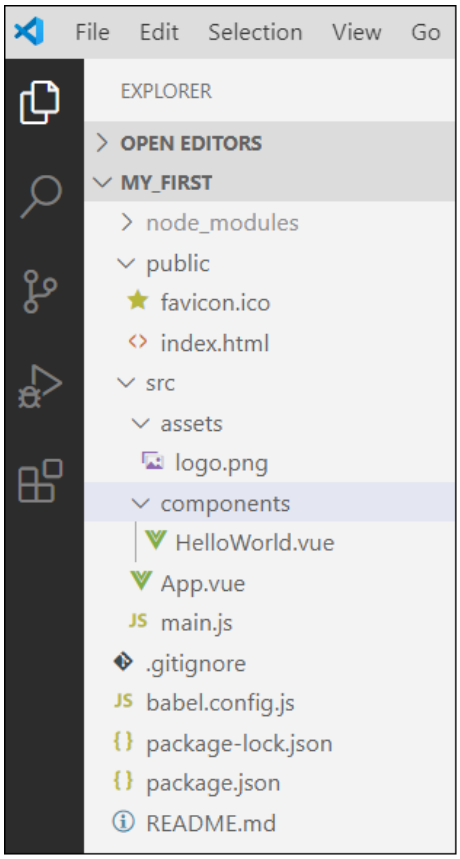


The project consists of three parts:

- node_modules - contains all used and downloaded modules used in application
- public - contains starting files (usually *index.html*) and application icon
- src - contains files with vue source code

🕮 13.3.2

The more precise structure is showed after the project is opened:



13.3.3

The starting file of project is **index.html**.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-
scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.ti
tle %> doesn't work properly without JavaScript enabled. Pleas
e enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
```

```
    <!-- built files will be auto injected -->
  </body>
</html>
```

The content defines necessity of *JavaScript* enabling and place the application defined as **app** into our content (content of *index.html*).

🕮 13.3.4

Second important file of project is **main.js**.

```
import Vue from 'vue'
import App from './App.vue'

new Vue({
  render: h => h(App),
}).$mount('#app')
```

This is the start file of our application - it is an initialisation file of Vue.

The line

```
import Vue from 'vue'
```

imports Vue from node_modules part of project.

The line

```
import App from './App.vue'
```

imports file App.vue with our definition of project (parts, components, design, etc.)

The definition

```
new Vue({
  render: h => h(App),
}).$mount('#app')
```

is the short description for rendering content defined in App.vue file in "app" element.

🕮 13.3.5

The definition of application is placed in **App.vue** file in root of project.

This is the template of application that defines it design and structure.

```
<template>
  <div id="app">
    <img alt="Vue logo" src="./assets/logo.png">
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

The line

```
<div id="app">
```

defines the name of application - the reference to "#app" means this part - this **div**.

This app consists of image

```
<img alt="Vue logo" src="./assets/logo.png">
```

and component defined in other file. The component is named *Helloworld*

```
 <HelloWorld msg="Welcome to Your Vue.js App"/>
```

and has one parameter defined as msg.

We can use HelloWorld component, because:

- we import file with its definition:

```
import HelloWorld from './components/HelloWorld.vue'
```

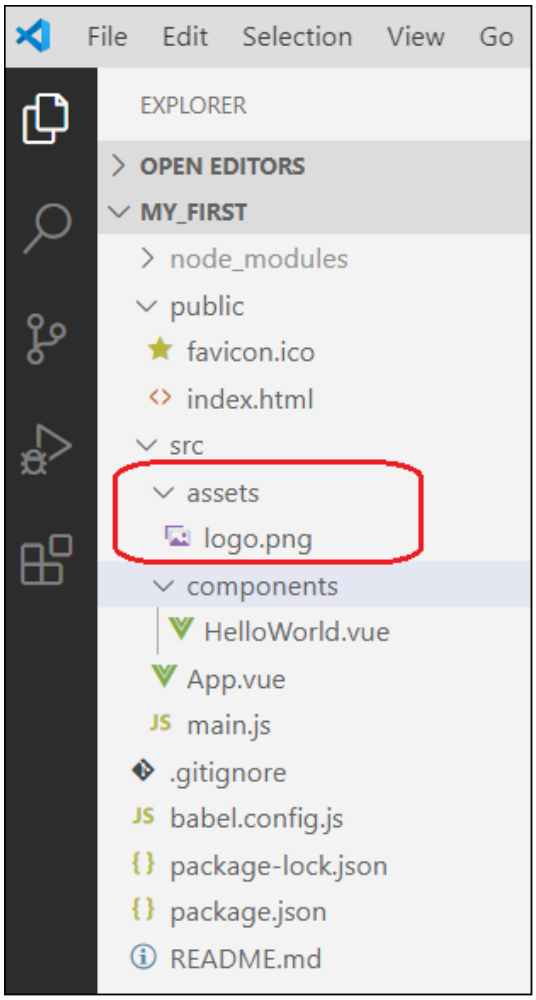
- we define it as a component as part of app:

```
export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
```

Last part (**style**) defines design, and it is not important for application functions.

🕮 13.3.6

Part **assets** in project structure consists of file like picture, documents etc. and we can use it in projects.

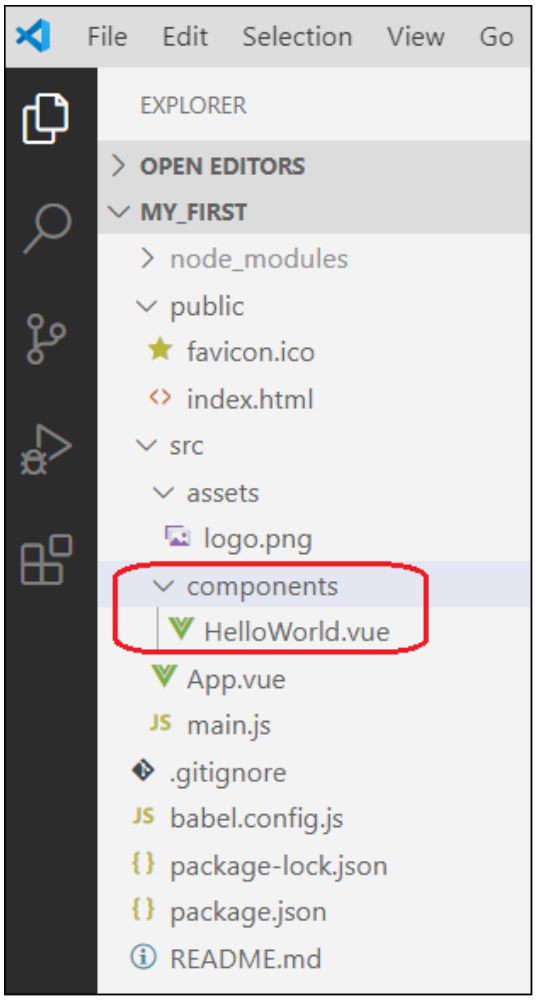


We used link to image in our App.vue definition.

🕮 13.3.7

The part **components** is a part of application where are components and its logic defined. We used one component *HelloWorld* and we placed it to the application structure.

The application usually constits of many components used in different components (not only in main application file).



13.3.8

The **HelloWorld.vue** is file with content which we really see in browser:

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <p>
      For a guide and recipes on how to configure / customize
this project,<br>
      check out the
      <a href="https://cli.vuejs.org" target="_blank" rel="noo
pener">vue-cli documentation</a>.
    </p>
    <h3>Installed CLI Plugins</h3>
    <ul>
```

```
      <li><a href="https://github.com/vuejs/vue-
cli/tree/dev/packages/%40vue/cli-plugin-
babel" target="_blank" rel="noopener">babel</a></li>
      <li><a href="https://github.com/vuejs/vue-
cli/tree/dev/packages/%40vue/cli-plugin-
eslint" target="_blank" rel="noopener">eslint</a></li>
    </ul>
    <h3>Essential Links</h3>
    <ul>
      <li><a href="https://vuejs.org" target="_blank" rel="noo
pener">Core Docs</a></li>
      <li><a href="https://forum.vuejs.org" target="_blank" re
l="noopener">Forum</a></li>
      <li><a href="https://chat.vuejs.org" target="_blank" rel
="noopener">Community Chat</a></li>
      <li><a href="https://twitter.com/vuejs" target="_blank"
rel="noopener">Twitter</a></li>
      <li><a href="https://news.vuejs.org" target="_blank" rel
="noopener">News</a></li>
    </ul>
    <h3>Ecosystem</h3>
    <ul>
      <li><a href="https://router.vuejs.org" target="_blank" r
el="noopener">vue-router</a></li>
      <li><a href="https://vuex.vuejs.org" target="_blank" rel
="noopener">vuex</a></li>
      <li><a href="https://github.com/vuejs/vue-devtools#vue-
devtools" target="_blank" rel="noopener">vue-devtools</a></li>
      <li><a href="https://vue-
loader.vuejs.org" target="_blank" rel="noopener">vue-
loader</a></li>
      <li><a href="https://github.com/vuejs/awesome-
vue" target="_blank" rel="noopener">awesome-vue</a></li>
    </ul>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
```

```
</script>

<!-
- Add "scoped" attribute to limit CSS to this component only -
->
<style scoped>
h3 {
  margin: 40px 0 0;
}
ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  margin: 0 10px;
}
a {
  color: #42b983;
}
</style>
```

The interesting part is

```
<h1>{{ msg }}</h1>
```

where we show the message defined in **App.vue** file. The connection between parameter and its content is defined using **props**:

```
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
```

the component with **name** HelloWorld has defined properties (props) named **msg**...

... and this propety we used in App.vue:

```
<HelloWorld msg="Welcome to Your Vue.js App"/>
```

Other lines in html parts defines only list for our web page.

13.4 Edit default project

🕮 13.4.1

Although we have already created an application with the Hello world component, it is too complicated and consists of many useless commands.

Let's go to make it simpler.

🕮 13.4.2

The most important information for our next activity is, that every vue file consists of three parts:

- HTML part - defines the content of page, HTML elements and its setting and content
- JavaScript part - defines variables, methods and properties used in HTML part
- CSS part - defines styles used in HTML part

The parts are defined by tags as follow:

```
<template>
   HTML PART
</template>

<script>
  JAVASCRIPT PART
</script>

<style>
  CSS PART
</style>
```

Programmer can let some parts empty.

🕮 13.4.3

HTML part is defined by tags <template> and </template>.

The content between these tag must by only one root. If you want to use two *div*-s you have to join it into one div. It means:

this is bad:

```
<template>
   <div id="first">... </div>
   <div id="second">... </div>
</template>
```

this is the solution:

```
<template>
   <div id="main">
      <div id="first">... </div>
      <div id="second">... </div>
   </div>
</template>
```

🕮 13.4.4

JavaScript part usually consists of imports and export:

```
<script>
import SomeComponent from './components/SomeComponent.vue'

export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>
```

The part export defaults consist of many parts defined by Vue and used in template part. We will talk about them later.

🕮 13.4.5

CSS part defines styles in two ways:

- styles are defined in part styles; we can use defined style in all parts or components in all Vue application. The definition has a form:

```
<style>
 // defined styles for all parts of app
</style>
```

- styles as defined only for this file (this component) and other files (components) don't see it:

```
<style scoped>
 // defined styles only for this component
</style>
```

📖 13.4.6

Let's go to edit our project.

First changes will be realised in **HelloWorld.vue**:

- we don't need to set styles - delete content of styles
- we don't need the content used in templates with links to vue pages

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>

<script>

export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>

<style>

</style>
```

📖 13.4.7

We can replace the **prop msg** via direct text in template

We don't need **props** definition, and we don't need the **name** variable of this component too.

The final form of **HelloWord.vue** is:

```
<template>
  <div>
    <h1>Hello world</h1>
  </div>
</template>

<script>
 export default {

 }
</script>

<style>

</style>
```

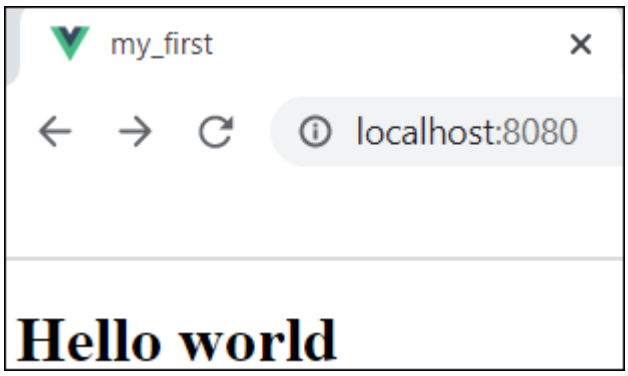
The call of **HelloWorld** component in **App.vue** must be changed - we don't have prop definition in **HelloWorld.vue**.

... and we don't need a picture of Vue.

The final form of Vue.app is:

```
<template>
  <div id="app">
    <HelloWorld />
  </div>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  components: {
    HelloWorld
  }
}
</script>

<style>
</style>
```

The output on **localhost:8080** is:



Simple Application

Chapter 14

14.1 Greeting - variables and functions

🕮 14.1.1

Presented example used simple component for rendering web page. We didn't reach any new features, we used complicated structure to achieve simple goal. The significance of Vue we can see if we work with variables and functions.

We will change the component *HelloWorld* and we use the variable with some greeting. The greeting will be set to variable and its content will be shown on the web page.

```
<script>
export default {
  data() {
    return {
      greeting: "Hello"
    }
  }
}
</script>
```

At first, we changed script-part.

If we want to define variables, we need place them to the method named **data()**. This method returns list of variables defined as a part of returned object (passed into {}). We can set the default value for every variable.

The access from HTML part to value of defined variables in script part is using {{}}.

If we change value of variable, the change will show in content - variables are **reactive**.

```
<template>
  <div>
    <h1>{{ greeting }}</h1>
  </div>
</template>
```

🕮 14.1.2

Now we can add the name and surname of the person we want to say hello.

Script part - we define more variable, one for name and one for surname:

```
<script>

export default {
  data() {
    return {
      greeting: "Hello",
      name: "Jozef",
      surname: "Carrot"
    }
  }
}
</script>
```

HTML part - we add the representation of values of new variables. Notice the comma - if we write some text outside of {{}} it is shown in written form (like in essential HTML code):

```
<template>
  <div>
    <h1>{{ greeting }}, {{ name }} {{ surname }}</h1>
  </div>
</template>
```

🕮 14.1.3

We can use a different approach too. We can create the content of variables and combine name and surname in method.

Vue used separated group for methods in the definition. The group is named **methods** and consists of all methods used in component.

The list of methods is **placed between {** and **}**.

Methods are **separated by commas**.

Every method has parameters defined in the **parentheses** or, if it has none parameters, the parentheses are empty - it is distinguishing sign between variables and methods.

```
<script>

export default {
  data() {
    return {
      greeting: "Hello",
```

```
      name: "Jozef",
      surname: "Carrot"
    }
  },
  methods: {
    getFullName() {
      return this.name + " " + this.surname
    }
  }
}
</script>
```

If we want to use some variable (or to call method in method) we have to use prefix **this**.

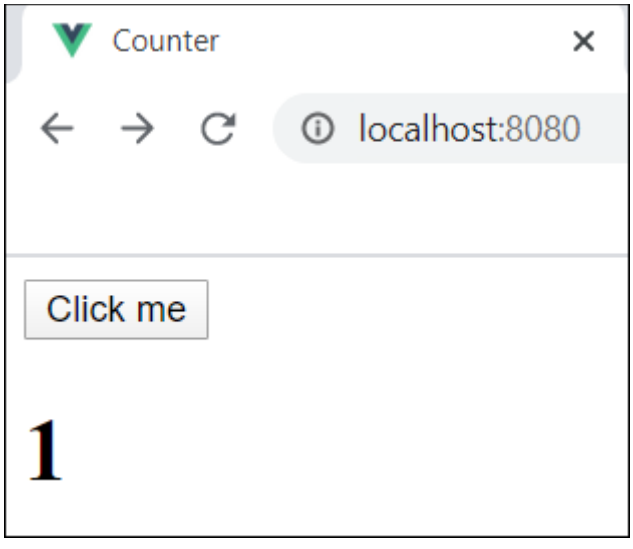
The use of method in HTML part is the same as the use of variables - we use only parentheses (for the identification, that method is called).

```
<template>
  <div>
    <h1>{{ greeting }}, {{ getFullName() }}</h1>
  </div>
</template>
```

14.2 Counter - events

📖 14.2.1

Create Counter - application that will increase ans show value after click to button.



Create new project "counter" using

```
> vue create counter
```

and prepare it for changes in **HelloWorld.vue** file (delete useless parts)

We will create application that will be able to interact with data model using button.

The default template is:

```
<template>
  <div>
    <button>Click me</button>
    <h1>{{ counter }}</h1>
  </div>
</template>
```

and the script consists of one definition of variable **counter**.

```
<script>

export default {
  data() {
    return {
      counter: 1
    }
  }
}

</script>
```

📖 14.2.2

We can use the **v-on** directive to listen to DOM events and run some *JavaScript* code when they're triggered. The event type is defined as the argument separated by dot. The expresion for precessing this event can be a method, like this:

```
<template>
  <div>
    <button v-on:click="increaseCounter()">Click me</button>
    <h1>{{ counter }}</h1>
  </div>
</template>
```

The method for increase counter should be defined in methods part, like this:

```
<script>

export default {
  data() {
    return {
      counter: 1
    }
  },
  methods: {
    increaseCounter() {
      this.counter++
    }
  }
}

</script>
```

The user's click on the button causes event *click* and the command for processing is set to call to method **increaseCounter()**.

This method changes value in **counter** and Vue automatically render changed part of web (=content where counter value is placed).

🕮 14.2.3

We can use different approach too. If the procedure for processing event is short (a few commands or simple change of variable value), we can write code directly into expression defined for event:

```
<template>
  <div>
    <button v-on:click="counter++">Click me</button>
    <h1>{{ counter }}</h1>
  </div>
</template>

<script>

export default {
  data() {
    return {
      counter: 1
    }
  },
```

```
}

</script>
```

In the HTML part we don't need to use prefix this - we use only name of variable. If there in expression are more commands, they are divided by semicolon.

📖 14.2.4

The more experienced programmers shorten notation and omit **v-on** directive. Vue allows shortening of entries and replace **v-on:** by @. The final form of this program will be:

```
<template>
  <div>
    <button @click="counter++">Click me</button>
    <h1>{{ counter }}</h1>
  </div>
</template>

<script>

  export default {
    data() {
      return {
        counter: 1
      }
    },
  }

</script>
```

Every save of code in Visual Studio Code will cause automatic changes to the appropriate web pages.

⌨ 14.2.5 Increase a decrease buttons

Add a new button to decrease counter value. Use as short code as possible.

14.3 Event object

🕮 14.3.1

Information about event are hold in event-object. The easiest way to show all parameters is to write it to console.

Create application to show event-object content to console.

HTML part consists of button and processing event click. The method called after click is called without parentheses and the info about event is automatically packed to its call:

```
<template>
  <div>
    <button @click="showInfo">Click me</button>
  </div>
</template>
```

The processing of event-object is placed into *showInfo()* method - object event is passed to parameter *myEvt*.

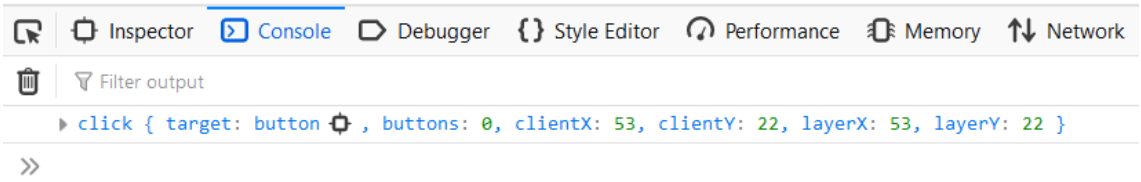
```
<script>

export default {
  data() {
    return {
    }
  },
  methods: {
    showInfo(myEvt) {
      // eslint-disable-next-line no-console
      console.log(myEvt)
    }
  }
}

</script>
```

The console can be used in Vue application but it's use depends on eslint settings. Sometimes we need to disable setting no-console (prohibits data output to the console). We can disable this restriction using

```
// eslint-disable-next-line no-console
```

yes, it is after note setting "//".

The result of this application is placed to console:



The whole event object is a little bit huge (it contains all informatyin about event):

```
click
  altKey: false
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
  cancelable: true
  clientX: 53
  clientY: 22
  composed: true
  ctrlKey: false
  currentTarget: null
  defaultPrevented: false
  detail: 1
  eventPhase: 0
  explicitOriginalTarget: <button>
  isTrusted: true
  layerX: 53
  layerY: 22
  metaKey: false
  movementX: 0
  movementY: 0
  mozInputSource: 1
  mozPressure: 0
  offsetX: 0
  offsetY: 0
  originalTarget: <button>
  pageX: 53
  pageY: 22
  rangeOffset: 0
  rangeParent: null
  region: ""
  relatedTarget: null
  returnValue: true
  screenX: 53
```

```
  screenY: 146
  shiftKey: false
  srcElement: <button>
  target: <button>
  timeStamp: 519694
  type: "click"
  view: Window http://localhost:8080/
  which: 1
  x: 53
  y: 22
```

🕮 14.3.2

We usually use only a part of information stored in event object.

Let's go to read position of mouse cursor when we click to the button.

We need a variable for storing **x** and **y** coords and a place to write it.

The HTML part looks like this:

```
<template>
  <div>
    <button @click="showInfo">Click me</button>
    <p>Coords: [{{ coord.x }},{{ coord.y }}]</p>
  </div>
</template>
```

The text **Coords: [ ]** contains values from object **coord** defined in script part.

```
<script>

export default {
  data() {
    return {
      coord: {
        x : 0,
        y : 0
      }
    }
  },
  methods: {
    showInfo(myEvt) {
      this.coord.x = myEvt.x
```

```
      this.coord.y = myEvt.y
    }
  }
}

</script>
```

The object used in Vue is the same as in JavaScript. It can consists of many variables or of many different objects placed between { and } and separated by commas.

The presented values contains information about cursor position on button (not on the screen).

🕮 14.3.3

Processing of keyboard typing events is usually connected to input fields (user use the keyboard while filling the text fields).

Let's go to create application which will be show the mirror of text written in text field.

We need textfield and method for reading content placed in textfield. The best event for processing is **keyup** - because when the key is released the character is written in field.

The HTML parts consists of:

```
<template>
  <div>
     <p>Write some text</p>
     <input type="text" @keyup="processText"/>
     <p>{{ mirror }}</p>
  </div>
</template>
```

The parameter of the method **processText** is event-object and we can read from it information about:

- target - where the event happens
- value - value of target = text in textfield

... and, at the end, we can process mirroring of written text.

```
<script>

export default {
  data() {
    return {
        mirror : ''
    }
  },
  methods: {
    processText(myEvt) {
      let content = myEvt.target.value
      this.mirror = '';
      for(let i = 0; i < content.length; i++)
        this.mirror = content.substring(i,i+1) + this.mirror
    }
  }
}

</script>
```

The constant **content** is created in **processText** method. The value from text field (text from target of event) is passed into **content**.

The mirror variable defined as string in data part is used form displaying in HTML part and we fill it in loop. A loop in *Vue* is identical with loop in *JavaScript*.

The changes of variable **mirror** are rendered to HTML part.

⌨ 14.3.4 Keypress event

Change the event of processing keyboard typing to **keypress** and follow the changes.

14.4 v-model

🕮 14.4.1

The mechanisms of reading content on change and setting the variable in a dedicated method is little bit complicated. You can use the **v-model** directive to create two-way data bindings on form input, textarea, select elements, etc.

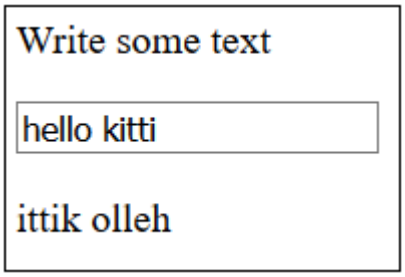
It automatically picks the correct way to update the element based on the input type. The use of **v-model** ignore the initial value of used DOM elements and set it by content of variables set to **v-model**. It will always treat the *Vue* instance data as the source of truth. You should declare the initial value on the *JavaScript* side, inside the data option of your component.

v-model internally uses different properties and emits different events for different input elements:

- text and textarea elements use **value** property and **input** event;
- checkboxes and radiobuttons use **checked** property and **change** event;
- select fields use **value** as a prop and **change** as an event.

🕮 14.4.2

Let's modify our program to show mirrored content of text field.



We begin with the script part. We need:

- variable for value connected to textfield content - **myContent**
- method for mirroring **myContent**; this method can return mirrored value

```
<script>

export default {
  data() {
    return {
        myContent : ''
    }
  },
  methods: {
    getMirror() {
      var mirror = '';
      for(let i = 0; i < this.myContent.length; i++)
        mirror = this.myContent.substring(i,i+1) + mirror
      return mirror
    }
```

```
  }
}

</script>
```

Variable **myContent** is declared and set to empty value.

Variable **mirror** in **getMirror()** method is declared as variable and set to empty at first. The loop iterate value of **myContent** and creates mirror of it. The mirrored content is returned as string value.

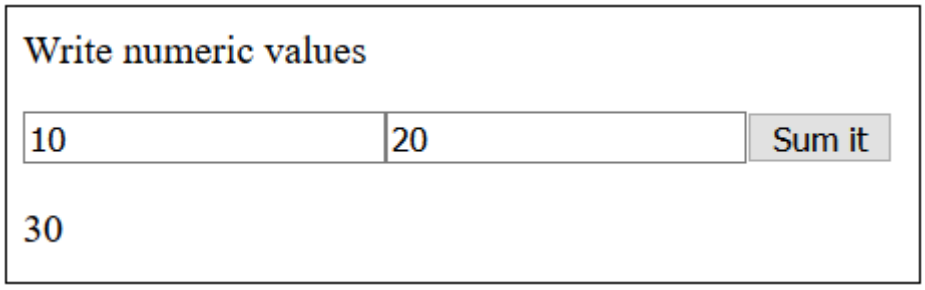
The HTML part consists of:

- texfield connected to **myContent** variable
- output of **getMirror()** method. The returned value is rendered everytime when the **myContent** value (content of texfield) is changed.

```
<template>
  <div>
    <p>Write some text</p>
    <input type="text" v-model="myContent"/>
    <p>{{ getMirror() }}</p>
  </div>
</template>
```

📖 14.4.3

Write program to sum two values written in text fields.



We need to create:

- two text fields connected to two variables
- button with event for reading values and realising numeric operation
- place for result

The definition of HTML part is simple:

```
<template>
```

```
  <div>
    <p>Write numeric values</p>
    <input type="text" v-model="val_a"/>
    <input type="text" v-model="val_b"/>
    <button @click="countSum()">Sum it</button>
    <p>{{ sum }}</p>
  </div>
</template>
```

We need to set default values for variables **var_a** and **var_b**. It should be 0, but set it to empty string is visually better.

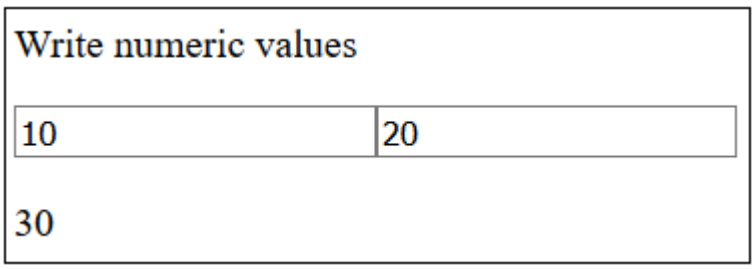
In a method **countSum()** we need to transform string written in text fields to the number.

```
<script>

export default {
  data() {
    return {
        val_a: '',
        val_b: '',
        sum : 0
    }
  },
  methods: {
    countSum() {
      this.sum = parseInt(this.val_a) + parseInt(this.val_b)
    }
  }
}

</script>
```

📖 14.4.4

Change the program to sum values as you type them into text fields.

We don't need button, and we use the method for processing for getting result:



The HTML part will be:

```
<template>
  <div>
    <p>Write numeric values</p>
    <input type="text" v-model="val_a"/>
    <input type="text" v-model="val_b"/>
    <p>{{ countSum() }}</p>
  </div>
</template>
```

And the script part consists of two variable and one getter:

```
<script>

export default {
  data() {
    return {
        val_a: '',
        val_b: ''
    }
  },
  methods: {
    countSum() {
      return parseInt(this.val_a) + parseInt(this.val_b)
    }
  }
}

</script>
```

This form is "more Vue" than previous.

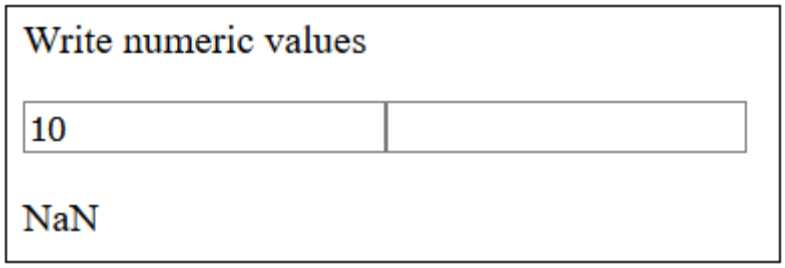
Condition and Loop

Chapter 15

15.1 v-if

🕮 15.1.1

The output of previous program was sometimes confusing:



We should add information why the result is weird.

The NaN is result of numeric operations if they don't finish good. The reason is now empty second text field. We can treat this situation in a method **countSum()** ...

```
    countSum() {
      return parseInt(this.val_a) + parseInt(this.val_b)
    }
```

...or we can use Vue mechanisms and show some information in HTML part.

The directive **v-if** is used to conditionally render a block. The block will only be rendered if the directive's expression returns **true**.

🕮 15.1.2

Extend the application to information, what says that some of used text fields is empty. Use rendering part of application.

We usually place code to script part of Vue application if that code is part of application logic. Many notes and information dedicated only for user to achieve better communication could make the application logic confusing and overcrowded. The solution is to move them to the HTML part.

We solve our task using directive **v-if**.

```
<template>
  <div>
    <p>Write numeric values</p>
    <input type="text" v-model="val_a"/>
    <input type="text" v-model="val_b"/>
```

```
    <p>{{ countSum() }}</p>
    <p v-if="val_a == ''">first field is empty</p>
    <p v-if="val_b == ''">second field is empty</p>
  </div>
</template>
```

The third paragraph is visible only if **val_a** (content of first text field) is empty. If we write some value into first text field, the condition (**val_a ==** '') return **false** and the paragraph is invisible.

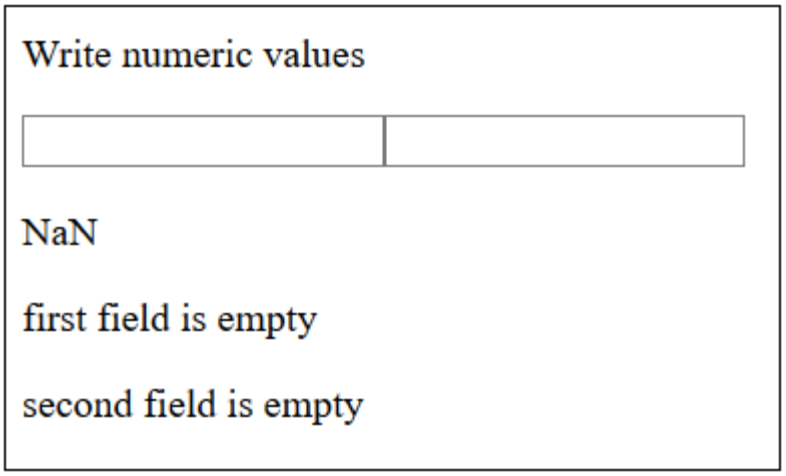
The same rule is set to the last paragraph.

The script part stay not changed:

```
<script>

export default {
  data() {
    return {
        val_a: '',
        val_b: ''
    }
  },
  methods: {
    countSum() {
      return parseInt(this.val_a) + parseInt(this.val_b)
    }
  }
}

</script>
```

You can try application:

- if first text field is empty - we can see this information
- if second text field is empty - we can see this information
- if both text fields are empty - we can see information in two paragraphs



15.2 v-for

🕮 15.2.1

Directive **v-for** is used for loops in HTML level of Vue application. We can use it to render a **list of items** based on an array.

The v-for directive requires a special syntax in the form of **item in items**, where items is the source data **array** and item is an alias for the array element being iterated on:

```
My list:
<ul>
  <li v-for="item in items" :key="item.id">
    {{ item.value}}
  </li>
</ul>
```

The loop renders content of array **items**. The content of array is rendered as <li> elements. The loop defines every elements using part **value**. Id is used because the loop need some unique parameter of item.

Two conditions have to be respected:

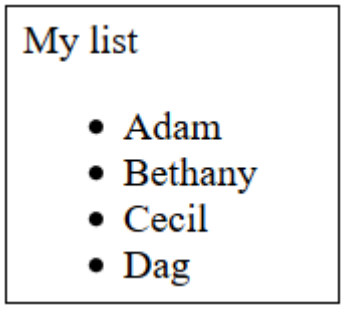
- items is array
- every item has unique value (in example it is id) defined as **key** of loop

The items in example above is defined as:

```
items: [
  {id:1, value: 'Adam'},
  {id:2, value: 'Bethany'},
```

```
  {id:3, value: 'Cecil'},
  {id:4, value: 'Dag'},
]
```

The output is:



🕮 15.2.2

If we use array, which does not contain keys or unique value, we can use new variable as numeric index of processed values. We have to place it as second argument in iterator definition.

```
My list
<ul>
   <li v-for="(item, index) in items" :key="index">
      {{ item }}
   </li>
</ul>
```

The variable index and iterated object are placed into parentheses, and the **:key** is defined as this **index**.

The array **items** in now defined as simple array of string objects:

```
<script>

export default {
  data() {
    return {
        items: ['Adam',
                'Bethany',
                'Cecil',
                'Dag'
               ]
    }
```

```
  }
}

</script>
```

15.2.3

Vue don't support cycles without data processing - we can't write number from 1 to 5. It is not error it is behaviour. If we need some counter, we can use index as it.

```
My list
<ul>
   <li v-for="(item, index) in items" :key="index">
      {{index + 1}}. {{ item }}
   </li>
</ul>
```

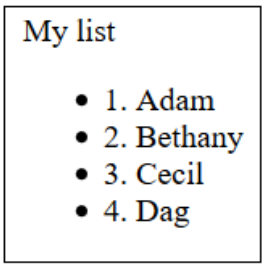
Index started at 0 and if we need counter, the first object need number 1 - we increment output of index.

After variable placed between {{ and }} we render "." and place value of item - i-th item of array.

The script part is defined as follow...

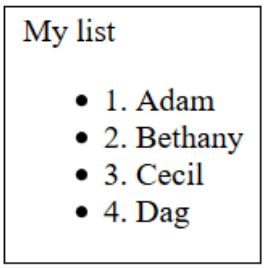
```
<script>

export default {
  data() {
    return {
        items: ['Adam',
                'Bethany',
                'Cecil',
                'Dag'
               ]
    }
  }
}

</script>
```

... and output has the form



📖 15.2.4

The output of previous example is a little bit confusing

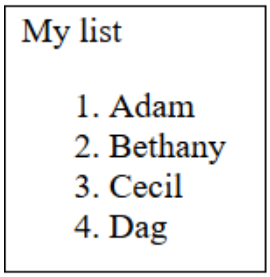


and we need to remove bullets and use ordered list instead of number written by code.

The solution can be prepared as follow:

```
<template>
  <div>
    My list
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
      </li>
    </ol>
  </div>
</template>
```

The cycle creates as many items as there are elements in the array. The items of list are items of ordered list and we have output in this form:



Lists

Chapter 16

16.1 Work with List

🕮 16.1.1

We are able to work with lists.

Let's go to prepare application to add and remove employees in our list. The list will contain surnames only.

HTML part is dedicated to show data. We will read data from array used for storing surnames of our employees. We can add index before surname.

Under the list we add a text field to add new employee and button to confirm adding.

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
      </li>
    </ol>
    <input type="text"/>
    <button>Add</button>
  </div>
</template>
```

We don't use variable **index,** but we need it. The array with our values (bellow in script part) has unique values and we can set as key the item. But if we will have the same surname more times the key value won't be unique and the content couldn't be rendered.

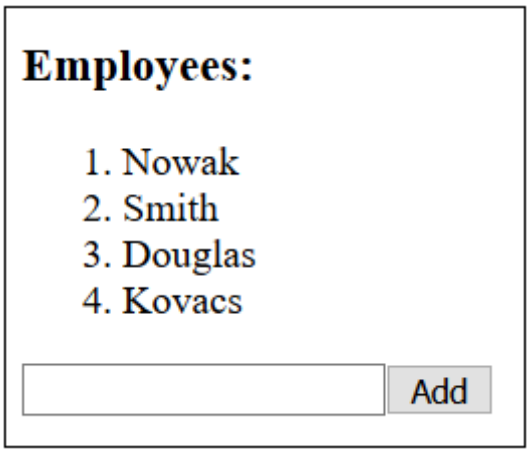
We add text field and button in one line.

Script part is simple:

```
<script>

export default {
  data() {
    return {
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
```

```
                ]
    }
  },
  methods: {
  }
}

</script>
```

And render web content has this form:



16.1.2

First functionality adds new value into existing list of surnames.

We showed how to use data placed in text fields. The best and usual way is to use **v-model** and read data from text field by using variable connected to field. We define variable and set it to v-model in text field.

To run method for add value we use button. We need to set event clicks in button and define method for processing this event.

HTML part had extended text field and button:

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
      </li>
    </ol>
```

```
    <input type="text" v-model="new_empl" />
    <button @click="addEmployee()">Add</button>
  </div>
</template>
```

We don't need parameter in method **addEmployee()** because we read value from variable **new_empl** (connected via **v-model** to the text field).

In script part we need the definition of variable and new method:

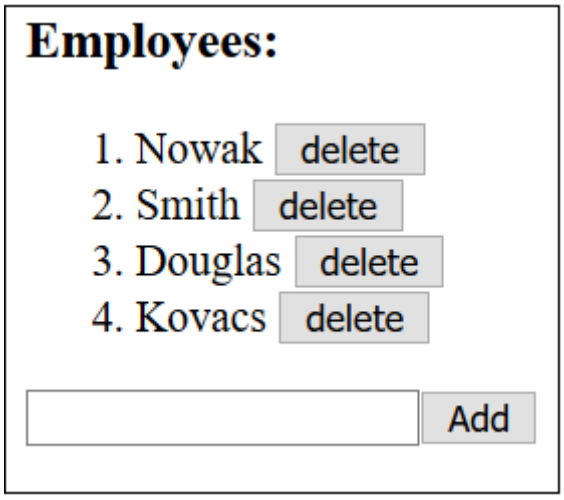
```
<script>

export default {
  data() {
    return {
        new_empl: '',
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ]
    }
  },
  methods: {
    addEmployee() {
      this.items.push(this.new_empl)
      this.new_empl = ''
    }
  }
}

</script>
```

To add new item into array we used method **push** and to empty the text field we assign variable **new_empl** to an empty string. We used prefix **this**, because we work with variable of this Vue object.

The rendering of values is automatic.

🕮 16.1.3

To delete employee from list we need to select him and call a function for delete. One of the ways to select a record is to add button to every record, like this:



The identification of record must be contained in call of function for delete. We add it as parameter to function **deleteEmployee()**.

The HTML part contains new button with **onclick** event:

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }} <button @click="deleteEmployee(index)">de
lete</button>
      </li>
    </ol>
    <input type="text" v-model="new_empl" />
    <button @click="addEmployee()">Add</button>
  </div>
</template>
```

The button for delete is generated in look together with information about employee.

The function for delete look like below. The method **splice** removes items from array. The first parameter defines index of first item to remove, second parameter defines how many items should be deleted.

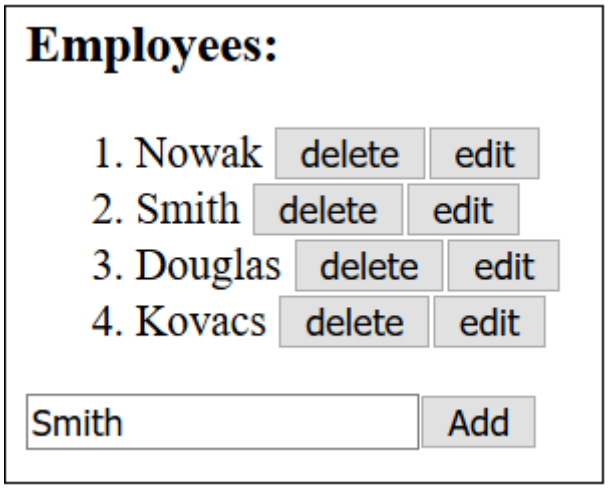
```
<script>

export default {
  data() {
    return {
        new_empl: '',
        items: ['Nowak',
```

```
                'Smith',
                'Douglas',
                'Kovacs'
               ]
    }
  },
  methods: {
    addEmployee() {
      this.items.push(this.new_empl)
      this.new_empl = ''
    },
    deleteEmployee(index) {
      this.items.splice(index,1)
    }
  }
}

</script>
```

16.1.4

Edit the item of a list is common request in work with lists. We can use the same approach what was used in delete function:

- add a button "edit" next to every item
- call the function edit with parameter defined index of item



But, what to do when we change item?

- we can move item from list to edit field and after change add it to the end of list

- we can remember the position of edited item and after change overwrite item on primary position

🕮 16.1.5

Let's go to move item from list to edit field and after change add it to the end of list

The HTML part is similar in both cases:

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
          <button @click="deleteEmployee(index)">delete</butto
n>
          <button @click="editEmployee(index)">edit</button>
      </li>
    </ol>
    <input type="text" v-model="new_empl" />
    <button @click="addEmployee()">Add</button>
  </div>
</template>
```

We can place the buttons to new lines because the position of tags is not important for web rendering.

The **editEmployee()** method will set the selected item into text field and delete it in list.

After change the button **Add** adds the text from text field to the end of list.

```
export default {
  data() {
    return {
        new_empl: '',
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ]
    }
  },
```

```
  methods: {
    addEmployee() {
      this.items.push(this.new_empl)
      this.new_empl = ''
    },
    deleteEmployee(index) {
      this.items.splice(index,1)
    },
    editEmployee(index) {
      this.new_empl = this.items[index]
      this.items.splice(index,1)
    }
  }
}
```

16.1.6

The application works correctly but there is one small user unfriendly behaviour. We wait that after read and set value to text field we can write immediately. But we can't.

We should set the focus to text field after click to edit button. How to do it?

Vue has procedure to set focus to chosen DOM element, but we need:

- to set the ref parameter to the element what will be focused
- to use group of references ($refs) in Vue and select the item for focusing

We will add tag **ref** to edit field. We can name the text field as **add_emp**.

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
          <button @click="deleteEmployee(index)">delete</butto
n>
          <button @click="editEmployee(index)">edit</button>
      </li>
    </ol>
    <input type="text" v-model="new_empl" ref="add_emp" />
    <button @click="addEmployee()">Add</button>
  </div>
</template>
```

We adapt code and add the call for set focus in method **editEmployee()**.

```
export default {
  data() {
    return {
        new_empl: '',
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ]
    }
  },
  methods: {
    addEmployee() {
      this.items.push(this.new_empl)
      this.new_empl = ''
    },
    deleteEmployee(index) {
      this.items.splice(index,1)
    },
    editEmployee(index) {
      this.new_empl = this.items[index]
      this.items.splice(index,1)
      this.$refs.add_emp.focus()
    }
  }
}
```

We used the group of references used by Vue, then we select a specific reference pointed to edit field and at last we set there a focus.

🕮 16.1.7

Let's go to try second approach: remember the position of edited item and after change overwrite item on primary position.

We need:

- to remember the index of edited item
- to change the text on the "Add" button
- to overwrite the item on primary position

These steps can't be realised in one method. We need to prepare data in first method and to save data in second method. Data save is programmed in method **addEmployee()** - we will modify it to method **saveEmployee()** and we will differentiate the code for add and for save edited item.

We can show and hide one button for add and one button for save, but we choose first approach.

In HTML part we use the variable **buttonText** for set changing text for button next to text field.

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
          <button @click="deleteEmployee(index)">delete</butto
n>
          <button @click="editEmployee(index)">edit</button>
      </li>
    </ol>
    <input type="text" v-model="new_empl" ref="add_emp" />
    <button @click="saveEmployee()">{{buttonText}}</button>
  </div>
</template>
```

The changes in script part are larger. We need:

- Variable for text on button (**buttonText**). The variable is set to "Add" after run web page. We change it when we press button edit (in method **editEmployee()**.
- Variable for storing index of changed item (**changeIndex**). If we are changing the item, the variable contains index this one. If we don't have processed changing value, or if we don't start changing, the variable contains -1. This value decide if we replace or add value by click to button next to text field.
- Method **editEmployee()** is starting method of change item value. It changes text on button, stored index of items that we want to edit and stored value of that item into variable **new_empl** what is connected to text field via v-model. The last step is focus setting to text field.
- Method **saveEmployee()** stored new or edited item into our list. The decision what to do depends on value of the variable **changeIndex**. If value is set to -1 (we don't change any item), the text placed in text field is added. If value is greater than -1 (it mean we set there position of edited item), we will overwrite existing value of existing item.

The script part has finally this form:

```
export default {
  data() {
    return {
        new_empl: '',
        buttonText: 'Add',
        changeIndex: -1,
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ]
    }
  },
  methods: {
    saveEmployee() {
      if (this.changeIndex > -1) {
        this.items[this.changeIndex] = this.new_empl
        this.changeIndex = -1
      } else {
        this.items.push(this.new_empl)
      }
      this.new_empl = ''
      this.buttonText = "Add"
    },
    deleteEmployee(index) {
      this.items.splice(index,1)
    },
    editEmployee(index) {
      this.buttonText = "Save"
      this.changeIndex = index
      this.new_empl = this.items[index]
      this.$refs.add_emp.focus()
    }
  }
}
```

📖 16.1.8

The final function for manipulation with values of list is sort. If we have many items, the sort function is necessary because of better looking for.

To add this function to our application is very easy:

- we add new button for call function responsible for items order

- we implement this function using only one line

HTML part (the button i separated in new line using **
**):

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
          <button @click="deleteEmployee(index)">delete</butto
n>
          <button @click="editEmployee(index)">edit</button>
      </li>
    </ol>
    <input type="text" v-model="new_empl" ref="add_emp" />
    <button @click="saveEmployee()">{{buttonText}}</button>
    <br>
    <button @click="sortEmployee()">Sort</button>
  </div>
</template>
```

The script part contains only one new method:

```
sortEmployee() {
    this.items.sort()
}
```

The changes are rendered immediately.

16.2 Material design

🕮 16.2.1

Material Design is a visual language that synthesizes the classic principles of good design with the innovation of technology and science. The principles of material design are defined as:

- Create - create a visual language that synthesizes the classic principles of good design with the innovation and possibility of technology and science.
- Unify - develop a single underlying system that unifies the user experience across platforms, devices, and input methods.

- Customize - expand Material's visual language and provide a flexible foundation for innovation and brand expression.

More information is on a web page: https://material.io/design/

We can use the parts of this system for building nice and modern application with positive user experience and fresh look.

🕮 16.2.2

The team of students from Carnegie Mellon University adapt Material Design to use in Vue. It is one of many alternatives for implementation of Material Design to your application. The description "how to" is usually sufficiently detailed on its web.

We use https://materializecss.com as a typical example with a typical step.

1. We copy a path to minified version of css and google icon font into our **index.html** file from https://materializecss.com/getting-started.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-
scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">

    <!-- Compiled and minified CSS -->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/
ajax/libs/materialize/1.0.0/css/materialize.min.css">
    <!--Import Google Icon Font-->
    <link href="https://fonts.googleapis.com/icon?family=Mater
ial+Icons" rel="stylesheet">


    <title>My Application</title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.ti
tle %> doesn't work properly without JavaScript enabled. Pleas
e enable it to continue.</strong>
    </noscript>
```

```
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

2. We can use Material design component (not only icons) in our HTML parts

🕮 16.2.3

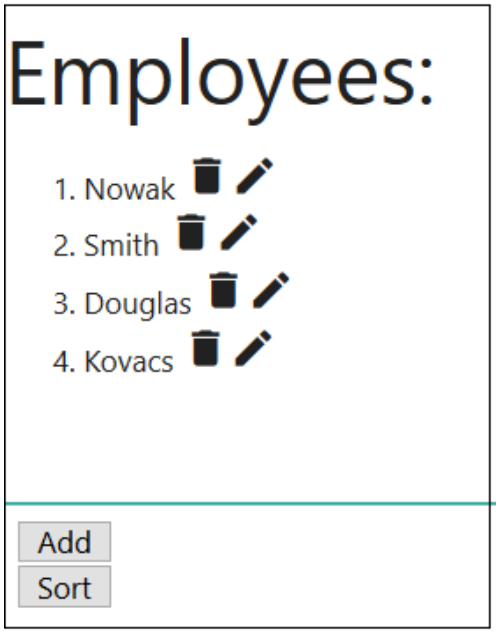
Let's go to adapt design of our application to Material design. We can start by replacing the buttons with icons.

To add icon into HTML part we need to use class **material-icons**, e.g.:

```
<i class = "material-icons">delete</i>
```

Our buttons should be replaced with icons - we change design part and we don't change event-part.

The result should have this form



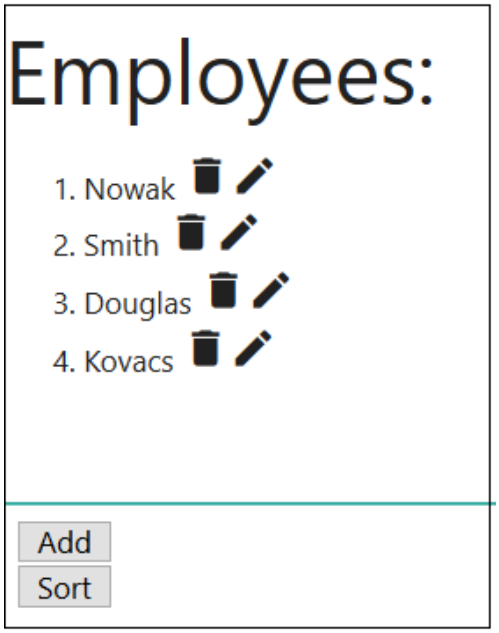
using this code:

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
```

```
          {{ item }}
          <i class = "material-
icons" @click="deleteEmployee(index)">delete</i>
          <i class = "material-
icons" @click="editEmployee(index)">edit</i>
      </li>
    </ol>
    <input type="text" v-model="new_empl" ref="add_emp" />
    <button @click="saveEmployee()">{{buttonText}}</button>
    <br>
    <button @click="sortEmployee()">Sort</button>
  </div>
</template>
```

16.2.4

Our "more beautiful" design has a few shortcomings:



- the icons are too big
- the text field is stretched to full width of page

Reading information about used styles we can get information how to make icons smaller: we can use setting **tiny**, **small**, **medium** or **large**. If we choose **tiny** the size of icon is comparable to the text. We can change their logically order - to place edit in front of delete.

To change size of text field is a little bit complicated. We need to know that:

- the elements on the web page are placed into row
- bootstrap brings useful division of web width to 12 columns and this feature has been taken over to other systems
- the class for text field definition is in used material design template defined as an **input-field**.

According to this behaviour we need to define row in our template an place there into row a text field width specific size (part of 12 columns).

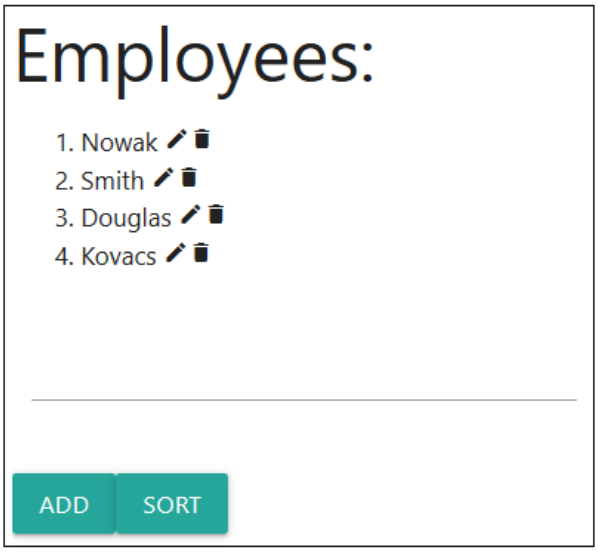
```
    <div class="row">
      <div class="input-field col s3">
          <input type="text" v-model="new_empl" ref="add_emp">
      </div>
    </div>
```

At the and we can change the design of our buttons and set its class to **class="btn"**.

The final code...

```
<template>
  <div>
    <h3>Employees:</h3>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
          <i class = "material-
icons tiny" @click="deleteEmployee(index)">delete</i>
          <i class = "material-
icons tiny" @click="editEmployee(index)">edit</i>
      </li>
    </ol>
    <div class="row">
      <div class="input-field col s3">
          <input type="text" v-model="new_empl" ref="add_emp">
      </div>
    </div>
    <button class="btn" @click="saveEmployee()">{{buttonText}}
</button>
    <button class="btn" @click="sortEmployee()">Sort</button>
  </div>
</template>
```

... generates the final design:



16.2.5

Using Material design libraries, we can improve design not only in elements of web page but we can make more beautiful design in web page as a whole.

We can add a navbar as the element of HTML5 web page structure:

```
    <nav>
      <span class="brand-logo"> Employees </span>
    </nav>
```

We used class **brand-logo** for make the name of page bigger.

Whole HTML part look like below:

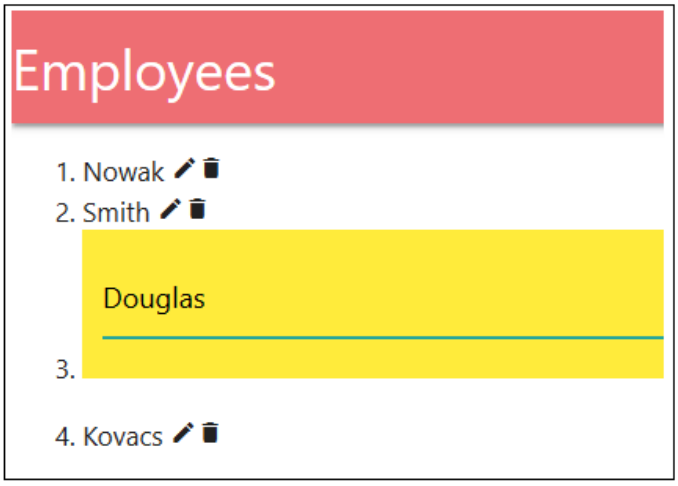
```
<template>
  <div>
    <nav>
      <span class="brand-logo"> Employees </span>
    </nav>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          {{ item }}
          <i class = "material-
icons tiny" @click="editEmployee(index)">edit</i>
          <i class = "material-
icons tiny" @click="deleteEmployee(index)">delete</i>
      </li>
    </ol>
```

```
    <div class="row">
      <div class="input-field col s3">
          <input type="text" v-model="new_empl" ref="add_emp">
      </div>
    </div>
    <button class="btn" @click="saveEmployee()">{{buttonText}}
</button>
    <button class="btn" @click="sortEmployee()">Sort</button>
  </div>
</template>
```

16.3 Edit inline

🕮 16.3.1

The modern approach to build GUI brings possibility to edit values a the place where they are. What does it means? In our example we can change the surname of employees on the place where it is written. We don't copy it to text field below list, but we can change on its position.



If we click to edit, we get a text field where we can change value of selected item.

We don't need to change script part. We have to differentiate a situation in rendering where the item is selected to change and when isn't.

The item selected to change will be rendered in text field - with a v-model set to selected item in items collection. Information about selected item is set to variable **changeIndex** used in our previous activities.

The item which is not selected will be rendered as before.

To achieve this behaviour we add **v-if** into **li** elements rendering.

```
<template>
  <div>
    <nav>
      <span class="brand-logo"> Employees </span>
    </nav>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          <div v-if = "index != changeIndex">
            {{ item }}
            <i class = "material-
icons tiny" @click="editEmployee(index)">edit</i>
            <i class = "material-
icons tiny" @click="deleteEmployee(index)">delete</i>
          </div>
          <div v-else class="row yellow">
            <div class="input-field col s3">
              <input type="text" v-model="items[index]">
            </div>
          </div>
      </li>
    </ol>
  </div>
</template>
```

The item and icons are rendered for items which are not edited.

The **input** and yellow **div** are rendered if item should be edited.

We clear a script part - we don't need function for Add (we will create new one) and function for edit is simpler:

```
export default {
  data() {
    return {
        changeIndex: -1,
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ]
    }
  },
  methods: {
    deleteEmployee(index) {
      this.items.splice(index,1)
```

```
    },
    editEmployee(index) {
      this.changeIndex = index
    }
  }
}
```

16.3.2

We didn't display button to confirm data changed in text field. To use button to confirm changed data is a little-bit older approach. Smartphones and new webpages change data immediately while you type.

We set the v-model to **items[index]**...

```
<div v-else class="row yellow">
   <div class="input-field col s3">
        <input type="text" v-model="items[index]">
   </div>
</div>
```

... and that means - if we change content of text field, content of items is changed too.

This is new situation for us - data is changed immediately and we can only stop editing - hide text field and replace it by **li** element.

The common way is to confirm end of writing by *Enter*. When listening for keyboard events, we often need to check for specific keys. Vue allows adding key modifiers when listening for key events. We add even-listener to **keydown** and process key *Enter*. This situation finish editing and set all elements of **items** to not edited.

```
<input
  type="text"
  v-model="items[index]"
  @keydown.enter="changeIndex = -1"
>
```

Whole code has this form:

```
<template>
  <div>
    <nav>
      <span class="brand-logo"> Employees </span>
    </nav>
```

```
    <ol>
      <li v-for="(item, index) in items" :key="index">
          <div v-if = "index != changeIndex">
            {{ item }}
            <i class = "material-
icons tiny" @click="editEmployee(index)">edit</i>
            <i class = "material-
icons tiny" @click="deleteEmployee(index)">delete</i>
          </div>
          <div v-else class="row yellow">
            <div class="input-field col s3">
              <input
                type="text"
                v-model="items[index]"
                @keydown.enter="changeIndex = -1"
              >
            </div>
          </div>
      </li>
    </ol>
  </div>
</template>
```

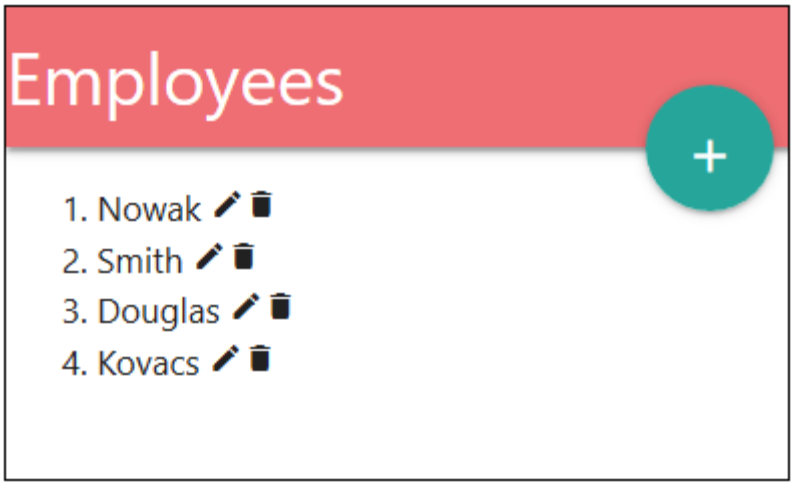
To set back previous value into the text field, you can use Ctrl+Z. This shortcut is implemented in system and helps you to go back and write old value to variable connected via v-model.

16.4 Add new element - other way

🕮 16.4.1

We lost function for add new values, but we can use new information and prepare elements using new way.

The element to add new items should be icon because we use icons not button to run command. Where to place it? We need new employee, the best position is in the navigation bar. We modify **nav** to:



By code:

```
<nav>
 <div class="nav-wrapper">
  <span class="brand-logo">Employees</span>
  <ul class="right">
   <li><i class="material-icons btn-floating btn-
large halfway-fab">add</i></li>
  </ul>
 </div>
</nav>
```

The **ul** provide space for definition of main function. The setting of **i** define position and design of button for add new items to our list.

🕮 16.4.2

We need to show the text field for add new item. We can decide to place it before or after list of elements - the code for add will be the same.

The process for add new element consists of following steps:

1. Click to icon for add element: we start process - we inform render system that we want to show text field. We use boolean variable **addNew** and set it to true.
2. If render system found this variable set to true, it shows text field.
3. The text field have to be joined with string variable **newValue** by **v-model**.
4. After typing new element content user press Enter (it is necessary to use the same behaviour in whole application).
5. The new value will be add to the end of list, the string variable should be set to empty (due to reuse) and the **addNew** will be set to false, because of hidding text field

According to these requirements the form of HTML part is:

```
<template>
  <div>
    <nav>
      <div class="nav-wrapper">
        <span class="brand-logo">Employees</span>
        <ul class="right">
          <li><i
            class="material-icons btn-floating btn-
large halfway-fab"
            @click = "addNew = true"
            >
            add
          </i></li>
        </ul>
      </div>
    </nav>
    <div v-if="addNew" class="input-field col s3">
      <input
        type="text"
        v-model="newValue"
        @keydown.enter="items.push(newValue);
                        newValue='';
                        addNew = false"
      >
    </div>
    <ol>
      <li v-for="(item, index) in items" :key="index">
          <div v-if = "index != changeIndex">
            {{ item }}
            <i class = "material-
icons tiny" @click="editEmployee(index)">edit</i>
            <i class = "material-
icons tiny" @click="deleteEmployee(index)">delete</i>
          </div>
          <div v-else class="row yellow">
            <div class="input-field col s3">
              <input
                type="text"
                v-model="items[index]"
                @keydown.enter="changeIndex = -1"
              >
```

```
            </div>
          </div>
      </li>
    </ol>
  </div>
</template>
```

All parts of the code we add to event methods directly into HTML part. We recommend to think over if it is good or bad - it depends on specific situation.

However, we need to define the variables and their default values in a script part:

```
export default {
  data() {
    return {
        changeIndex: -1,
        addNew: false,
        newValue: '',
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ]
    }
  },
  methods: {
    deleteEmployee(index) {
      this.items.splice(index,1)
    },
    editEmployee(index) {
      this.changeIndex = index
    }
  }
}
```

16.5 Bulk data operations

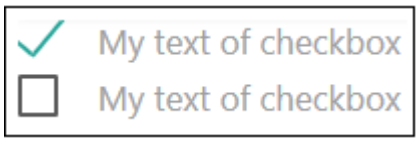
🕮 16.5.1

If we develop application for work with lists, we need procedure and DOM elements for operation supported work with multiple records. We usually use the checkboxes before or after content of record.

Checkbox in Vue requires definition using **label**:

```
<label >
  <input type="checkbox"/>
  <span>My Text of checkbox </span>
</label>
```

with result:



This "nice" design is a outcome of material design styles.

🕮 16.5.2

To remember which checkbox is selected and which one is not, we can use independent array with boolean values (e.g. named **selection**). The value on **index** position in **selection** express if the checkbox associated with name in array items at position **index** is checked or not.

We can use this approach using this HTML part:

```
<template>
  <div>
    <nav>
      <div class="nav-wrapper">
        <span class="brand-logo">Employees</span>
        <ul class="right">
          <li><i
            class="material-icons btn-floating btn-
large halfway-fab"
            @click = "addNew = true"
            >
            add
          </i></li>
        </ul>
      </div>
    </nav>
    <div v-if="addNew" class="input-field col s3">
      <input
        type="text"
        v-model="newValue"
        @keydown.enter="items.push(newValue);
```

```
                        newValue='';
                        addNew = false"
      >
    </div>
    <div v-for="(item, index) in items" :key="index">
        <div v-
if = "index != changeIndex">
          <label >
            <input type="checkbox" v-
model="selection[index]"/>
            <span>{{ item }} </span>
          </label>
          <i class = "material-
icons tiny" @click="editEmployee(index)">edit</i>
          <i class = "material-
icons tiny" @click="deleteEmployee(index)">delete</i>
        </div>
        <div v-else class="row yellow">
          <div class="input-field col s3">
            <input
              type="text"
              v-model="items[index]"
              @keydown.enter="changeIndex = -1"
            >
          </div>
        </div>
    </div>
  </div>
</template>
```

The array **selected** is filled by **false** values, after start this page. The number of values is the same as number of employees.

```
export default {
  data() {
    return {
        changeIndex: -1,
        addNew: false,
        newValue: '',
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ],
        selection: [false,false,false,false]
```

```
    }
  },
  methods: {
    deleteEmployee(index) {
      this.items.splice(index,1)
    },
    editEmployee(index) {
      this.changeIndex = index
    }
  }
}
```

16.5.3

Presented approach works, but we can use different approach too.

At first we prepare empty array at start of page running (the name could be **selection** again). This array is used to collect the selected items. If none item is selected, array is empty. If five elements are selected, the array contains five elements sorted by the time they were added.

The form of code that allows us to look like this:

```
<div v-for="(item, index) in items" :key="index">
  <label>
    <input type="checkbox" v-
model="selection" :value="index" />
       <span>{{ item }}</span>
  </label>
</div>
```

v-model of checkbox is set to array (**selection**) where the values of selected items are stored.

The values stored into array are defined in **:value**. We decided to store indexes, but we can replace it by **item** too (we will store surname of employee).

The array **selection** is defined as empty array in data part:

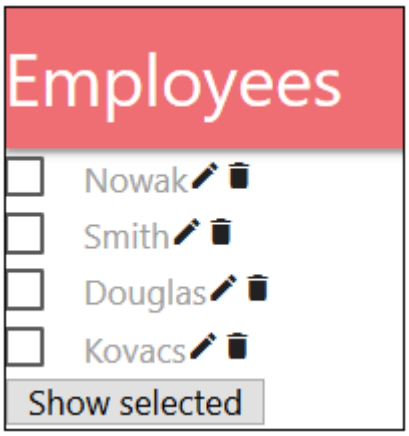
```
export default {
  data() {
    return {
        changeIndex: -1,
        addNew: false,
        newValue: '',
```

```
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ],
        selection: []
    }
  }, ...
```

16.5.4

Let's go to check the values inserted into **selection** array. We can print it to console.

To run printing, we can add new button below our list:



The complete HTML part has this form:

```
<template>
  <div>
    <nav>
      <div class="nav-wrapper">
        <span class="brand-logo">Employees</span>
        <ul class="right">
          <li><i
            class="material-icons btn-floating btn-
large halfway-fab"
            @click = "addNew = true"
            >
            add
          </i></li>
        </ul>
      </div>
    </nav>
```

```
    <div v-if="addNew" class="input-field col s3">
      <input
        type="text"
        v-model="newValue"
        @keydown.enter="items.push(newValue);
                        newValue='';
                        addNew = false"
      >
    </div>
    <div v-for="(item, index) in items" :key="index">
       <div v-
if = "index != changeIndex">
         <label >
           <input type="checkbox" v-
model="selection" :value="index" />
           <span>{{ item }}</span>
         </label>
         <i class = "material-
icons tiny" @click="editEmployee(index)">edit</i>
         <i class = "material-
icons tiny" @click="deleteEmployee(index)">delete</i>
      </div>
      <div v-else class="row yellow">
        <div class="input-field col s3">
          <input
            type="text"
            v-model="items[index]"
            @keydown.enter="changeIndex = -1"
          >
        </div>
      </div>
    </div>
    <button @click="showSelection()">Show selection</button>
  </div>
</template>
```

We add method **showSelection** to write content of array.

```
export default {
  data() {
    return {
        changeIndex: -1,
        addNew: false,
        newValue: '',
        items: ['Nowak',
```

```
                'Smith',
                'Douglas',
                'Kovacs'
              ],
        selection: []
    }
  },
  methods: {
    deleteEmployee(index) {
      this.items.splice(index,1)
    },
    editEmployee(index) {
      this.changeIndex = index
    },
    showSelection() {
      console.log(this.selection)
    }
  }
}
```

If we click to checkbox elements in order Douglas, Nowak, Smith, the output is:

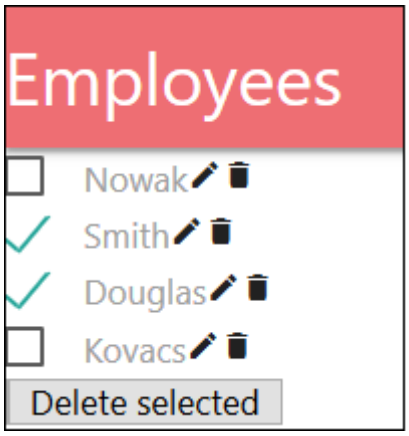
```
Array(3) [ 2, 0, 1 ]
```

Array contains elements ordered in order of insertion.

🕮 16.5.5

The selection is often used to prepare the elements for removal.

Let's go to adapt our code for remove selected elements.



We change the text in button and the method called after button press.

```
<template>
  <div>
    <nav>
      <div class="nav-wrapper">
        <span class="brand-logo">Employees</span>
        <ul class="right">
          <li><i
            class="material-icons btn-floating btn-
large halfway-fab"
            @click = "addNew = true"
            >
            add
          </i></li>
        </ul>
      </div>
    </nav>
    <div v-if="addNew" class="input-field col s3">
      <input
        type="text"
        v-model="newValue"
        @keydown.enter="items.push(newValue);
                        newValue='';
                        addNew = false"
      >
    </div>
    <div v-for="(item, index) in items" :key="index">
        <div v-
if = "index != changeIndex">
          <label >
            <input type="checkbox" v-
model="selection" :value="index" />
            <span>{{ item }}</span>
          </label>
          <i class = "material-
icons tiny" @click="editEmployee(index)">edit</i>
          <i class = "material-
icons tiny" @click="deleteEmployee(index)">delete</i>
        </div>
        <div v-else class="row yellow">
          <div class="input-field col s3">
            <input
              type="text"
              v-model="items[index]"
```

```
              @keydown.enter="changeIndex = -1"
            >
          </div>
        </div>
    </div>
    <button @click="deleteSelected()">Delete selected</button>
  </div>
</template>
```

The method for delete element should work as follow:

- order indexes of elements, because we need to start deletion by the last element in list. If we delete other element, the indexes stored in array will change,
- in **for** loop from last to first element in array we delete element on the stored index
- to empty list of selected element, because they were removed.

The script part will be changed to:

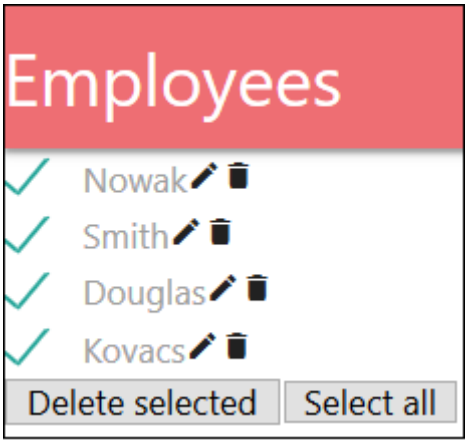
```
export default {
  data() {
    return {
        changeIndex: -1,
        addNew: false,
        newValue: '',
        items: ['Nowak',
                'Smith',
                'Douglas',
                'Kovacs'
               ],
        selection: []
    }
  },
  methods: {
    deleteEmployee(index) {
      this.items.splice(index,1)
    },
    editEmployee(index) {
      this.changeIndex = index
    },
    deleteSelected() {
      this.selection.sort();
      for(let i = this.selection.length - 1; i>= 0; i--) {
         this.items.splice(this.selection[i], 1)
```

```
            this.selection.splice(i, 1)
        }
      }
    }
}
```

16.5.6

The useful function in work with list is selection of all elements.

The solution is simple, we can prepare function to set all elements of data:



The method for selection has this form:

```
selectAll() {
   this.selection = []
   for(let i = 0; i < this.items.length; i++ ){
     this.selection.push(i)
   }
}
```

PRISCILLA

priscilla.fitped.eu