

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

Создание веб-приложений с использованием Angular и React

Урок № 4

Angular:
директивы, data
binding, сервисы,
dependency injection,
навигация

Содержание

Pipes	3
Структурные директивы	16
Стили	22
Привязка свойств	25
Привязка событий	28

Pipes

Pipes (каналы или фильтры) это специальные классы, которые позволяют преобразовывать данные в желаемый вид при выводе в HTML-шаблоне.

Например, в шаблоне главного компонента **AppComponent** воспользуемся каналом для преобразования заголовка в верхний регистр:

```
<h1> {{title | uppercase}} </h1>
<app-to-do-items></app-to-do-items>
```

Синтаксис применения канала:

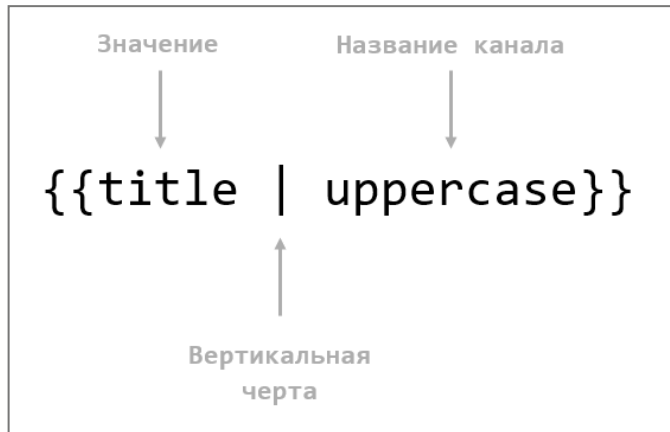


Рисунок 1

В Angular имеется ряд встроенных каналов, все они доступны для использования в любом шаблоне (табл. 1).

Таблица 1

Команда	Описание
number	Форматирует числовые значения
currency	Форматирует числовые значения, представляющие денежные величины
percent	Форматирует числовые значения, представляющие процентную величину
date	Преобразует дату к указанному формату
uppercase	Преобразует все символы в строке к верхнему регистру
lowercase	Преобразует все символы в строке к нижнему регистру
slice	Преобразует строку или массив, копируя заданное подмножество символов или элементов

Для изучения работы каналов давайте добавим в проект новый компонент.

```
C:\AngularWorkspaces\ToDoList>ng g component PipesTest
```

Рисунок 2

В шаблон главного компонента временно добавим вывод **PipesTestComponent**:

```
<app-pipes-test></app-pipes-test>
<h1>{{title | uppercase}}</h1>
<app-to-do-items></app-to-do-items>
```

А теперь в новом компоненте напишем пример использования каналов:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-pipes-test',
  template: `
```

```

    <div>uppercase: {{str | uppercase}}</div>
    <div>lowercase: {{str | lowercase}}</div>
    <div>number: {{num | number}}</div>
    <div>percent: {{num | percent}}</div>
    <div>currency: {{num | currency}}</div>`
  })

export class PipesTestComponent{
  str = 'Hello World!';
  num = 0.14;
}

```

Вывод:

```

uppercase: HELLO WORLD!
lowercase: hello world!
number: 0.14
percent: 14%
currency: $0.14

```

Рисунок 3

Некоторые каналы могут получать параметры. У каждого канала свой набор параметров.

Канал **slice** берет массив или строку и возвращает подмножество элементов (или символов).

У канала **slice** следующий синтаксис:

Параметры канала

```

{{ value_expression | slice : start [ : end ] }}

```

Рисунок 4

- **start** — Обязательный параметр. Если его значение положительно, то начальный индекс элементов, включаемых в подмножество, отсчитывается от первого элемента массива. Если значение отрицательно, то канал отсчитывает индекс от конца массива.
- **end** — Необязательный аргумент. Указывает, индекс конечного элемента, который должен быть включен в результат. Если значение не указано, то будут включены все элементы после индекса **start**.

Пример:

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

@Component({
  selector: 'app-pipes-test',
  template: `<div>{{str | slice:25}}</div>
<div>{{str | slice:0:6}}</div>
<div>{{str | slice:-8}}</div>`
})
export class PipesTestComponent {

  str = 'London is the capital of Great Britain.';
}
```

Вывод:

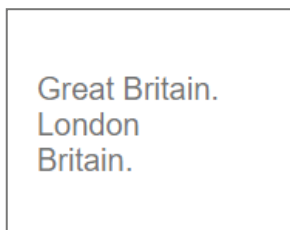


Рисунок 5

Канал **number** преобразует число в строку, форматируя в соответствии с правилами локали, определяющими символ десятичной точки и другие конфигурации, специфичные для локали.

```
{{ value_expression | number [:digitsInfo [:locale]] }}
```

Рисунок 6

- **digitsInfo** — строка в формате:

```
"minIntegerDigits.minFractionDigits-maxFractionDigits"
```

- ▷ **minIntegerDigits** — минимальное количество цифр в целой части;
- ▷ **minFractionDigits** — минимальное количество цифр в дробной части;
- ▷ **maxFractionDigits** — максимальное количество цифр в дробной части.

- **locale** — код применяемой культуры.

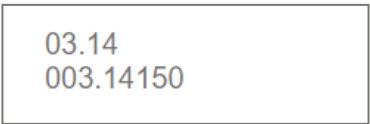
Пример:

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

@Component({
  selector: 'app-pipes-test',
  template: `<div>{{pi | number:'2.1-2'}}</div>
<div>{{pi | number:'3.5-5'}}</div>`
})
export class PipesTestComponent {

  pi = 3.1415;
}
```

Вывод:



```
03.14
003.14150
```

Рисунок 7

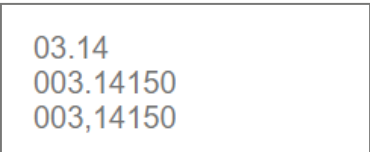
По умолчанию приложения Angular используют локаль **en-US**, но мы можем передать **local** как параметр в канал **number**. Перед тем как указывать локаль, ее нужно зарегистрировать, как показано в примере:

```
import { Component } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localeRu from '@angular/common/locales/ru';
registerLocaleData(localeRu, 'ru');

@Component({
  selector: 'app-pipes-test',
  template: `<div>{{pi | number:'2.1-2'}}</div>
<div>{{pi | number:'3.5-5'}}</div>
<div>{{pi | number:'3.5-5':'ru'}}</div>`
})

export class PipesTestComponent {
  pi = 3.1415;
}
```

Вывод:



```
03.14
003.14150
003,14150
```

Рисунок 8

Канал **currency** преобразует число, представляющее денежную величину, в строку, добавляя знак валюты.

```
{{ value_expression | currency [:currencyCode [:display [:digitsInfo [:locale]]]] }}
```

Рисунок 9

- **currencyCode** — код валюты согласно спецификации ISO 4217. По умолчанию применяется USD.
- **display** — указывает, как отображать символ валюты. Может принимать следующие значения:
 - ▷ **code**: показать код (например, USD);
 - ▷ **symbol** (по умолчанию): Показать символ (например, \$);
 - ▷ **symbol-narrow**: Используйте узкий символ для локалей, которые имеют два символа для своей валюты. Например, канадский доллар CAD имеет символ CA\$ и символ-узкий \$. Если в локале нет узкого символа, используется стандартный символ для локали;
 - ▷ **String**: использовать заданное строковое значение вместо кода или символа. Например, пустая строка будет подавлять валюту и символ.
- **digitsInfo** — формат числа, который применяется в канале **number**.
- **locale** — код используемой локали.

Пример:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-pipes-test',
```

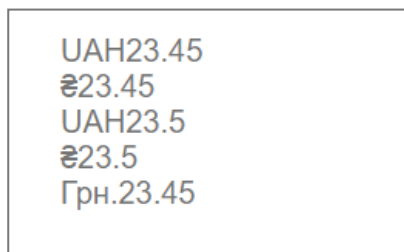
```

template: `
  <div>{{money | currency:'UAH':'code'}}</div>
  <div>{{money | currency:'UAH':
    'symbol-narrow'}} </div>
  <div>{{money | currency:'UAH':
    'symbol':'1.1-1'}} </div>
  <div>{{money | currency:'UAH':
    'symbol-narrow':'1.1-1'}} </div>
  <div>{{money | currency:'UAH':'Грн.'}}</div>`
  })

export class PipesTestComponent {
  money = 23.45;
}

```

Вывод:



```

UAH23.45
€23.45
UAH23.5
€23.5
Грн.23.45

```

Рисунок 10

Канал `percent` преобразует число в строку с процентами с учетом указанной локали. Значения в диапазоне от 0 до 1 форматируются так, чтобы они представляли от 0 до 100 процентов.

```

{{ value_expression | percent [ : digitsInfo [ : locale ] ] }}

```

Рисунок 11

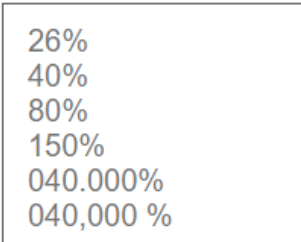
- **digitsInfo** — формат числа, который применяется в **DecimalPipe**.
- **locale** — код используемой локали.

Пример:

```
import { Component } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localeRu from '@angular/common/locales/ru';
registerLocaleData(localeRu, 'ru');

@Component({
  selector: 'app-pipes-test',
  template: `
    <div>{{a | percent}}</div>
    <div>{{b | percent}}</div>
    <div>{{c | percent}}</div>
    <div>{{d | percent}}</div>
    <div>{{b | percent:'3.3-5'}}</div>
    <div>{{b | percent:'3.3-5':'ru'}}</div>`
})
export class PipesTestComponent {
  a = 0.259;
  b = 0.4;
  c = 0.8;
  d = 1.5;
}
```

Вывод:



```
26%
40%
80%
150%
040.000%
040,000 %
```

Рисунок 12

Канал `date` форматирует даты с учетом локального контекста. Даты могут быть представлены:

- объектами JavaScript `Date`;
- в виде числовых значений, представляющих количество миллисекунд с начала 1970 года;
- в виде строки со строго определенным форматом.

```
{{ value_expression | date [ : format [ : timezone [ : locale ] ] ] }}
```

Рисунок 13

`format` — строка, которая задает формат даты.

Для указания пользовательского формата используются следующие обозначения:

Таблица 2

у, уу	Год
M, MMM, MMMM	Месяц
d, dd	День (в числовом представлении)
E, EE, EEEE	День (в текстовом виде)
j, jj	Час
h, hh, H, HH	Час в 12- и 24-часовом формате
m, mm	Минуты
s, ss	Секунды
Z	Часовой пояс

Так же в параметр `format` можно передать predefined значения:

Таблица 3

Команда	Описание
short	Эквивалентно строке 'M/d/yy, h:mm a'
medium	Эквивалентно строке 'MMM d, y, h:mm:ss a'

Команда	Описание
long	Эквивалентно строке 'MMMM d, y, h:mm:ss a z'
full	Эквивалентно строке 'EEEE, MMMM d, y, h:mm:ss a zzzz'
shortDate	Эквивалентно строке 'M/d/yy'
mediumDate	Эквивалентно строке 'MMM d, y'
longDate	Эквивалентно строке 'MMMM d, y'
fullDate	Эквивалентно строке 'EEEE, MMMM d, y'
shortTime	Эквивалентно строке 'h:mm a'
mediumTime	Эквивалентно строке 'h:mm:ss a'
longTime	Эквивалентно строке 'h:mm:ss a z'
fullTime	Эквивалентно строке 'h:mm:ss a zzzz'

- **timezone** — позволяет указать смещение часового пояса, которое может указываться следующим образом:
 - ▷ смещением часового пояса (например, '+0430');
 - ▷ стандартное UTC / GMT;
 - ▷ сокращенное обозначение часового пояса континентальной части США.

Если не указан, используется часовой пояс локальной системы конечного пользователя.

Пример:

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

@Component({
  selector: 'app-pipes-test',
  template: `
    <div>From object: {{ dateObject | date }}</div>
    <div>From string: {{ dateString | date }}</div>
    <div>From number: {{ dateNumber | date }}</div>
    <div>ShortDate: {{ dateObject |
                        date:"shortDate" }}</div>
```

```

    <div>MediumDate: {{ dateObject |
                        date:"mediumDate" }}</div>
    <div>LongDate: {{ dateObject |
                        date:"longDate" }}</div>
    <div>Custom: {{ dateObject |
                    date:"dd-MM-yyyy" }}</div>
    <div>Timezone UTC: {{ dateObject |
                        date:"dd-MM-yyyy": "UTC" }}</div>`
  ))

export class PipesTestComponent {
  dateObject: Date = new Date(2020, 1, 20);
  dateString = "2020-02-20T00:00:00.000Z";
  dateNumber = 1582156800000;
}

```

Вывод:

```

From object: Feb 20, 2020
From string: Feb 20, 2020
From number: Feb 20, 2020
ShortDate: 2/20/20
MediumDate: Feb 20, 2020
LongDate: February 20, 2020
Custom: 20-02-2020
Timezone UTC: 19-02-2020

```

Рисунок 14

Каналы можно соединять в цепочку, т.е. для вывода одного значения использовать сразу несколько каналов. Давайте для вывода сообщения применим сразу два канала: канал [slice](#), для того чтобы обрезать текст сообщения и канал [uppercase](#), для того чтобы преобразовать текст в верхний регистр.

Пример:

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

@Component({
  selector: 'app-pipes-test',
  template: `<div>{{message | slice:6:11 |
                                     uppercase}}</div>`
})

export class PipesTestComponent {
  message = "Hello World!";
}
```

Вывод:



Рисунок 15

После изучения работы каналов, компонент **PipesTest-Component** в нашем проекте не нужен, т.к. не связан с предметной областью приложения. Давайте прокомментируем или уберем его вывод из шаблона главного компонента.

```
<!--<app-pipes-test></app-pipes-test>-->
<h1>{{title | uppercase}}</h1>
<app-to-do-items></app-to-do-items>
```

Структурные директивы

Добавим в наше приложение вывод списка дел, позволим пользователю выбирать элемент из списка и просматривать информацию о нем.

Нам понадобится получать информацию о списке дел. В будущем вы будете получать эту информацию с удаленного сервера, а пока создадим *mock*-объект, хранящий объекты элементов списка дел. **Mock-объект** — это «заглушка», т.е. объект, который имитирует поведение реального объекта. Часто используется в unit тестах, для симулирования работы с хранилищами данных (СУБД), внешними API и другими классами приложения, реализующими бизнес-логику.

Создайте файл с именем *mock-todo-items.ts* в папке *src / app /*. Определите константу **ITEMS** как массив из десяти объектов **ToDoItem** и экспортируйте ее. Файл должен выглядеть следующим образом.

```
import { ToDoItem } from './to-do-item';
export const ITEMS: ToDoItem[] = [
  { id: 1, name: 'Call Joe', isComplete:false },
  { id: 2, name: 'Do my homework', isComplete:false },
  { id: 3, name: 'Prepare presentation',
    isComplete:false },
  { id: 4, name: 'Send email', isComplete:false },
  { id: 5, name: 'Buy products', isComplete:false },
  { id: 6, name: 'Lunch with Mom', isComplete:false },
  { id: 7, name: 'Start learning Angular',
    isComplete:false },
```



```
{ id: 8, name: 'Write article', isComplete:false },
{ id: 9, name: 'feed the dog', isComplete:false },
{ id: 10, name: 'go to the walk', isComplete:false }
];
```

В файле класса `ToDoItemsComponent` импортируем наш список дел и определим свойство компонента с именем `items`, чтобы предоставить массив `ITEMS` для привязки.

```
import { Component, OnInit } from '@angular/core';
import { ToDoItem } from '../to-do-item';
import { ITEMS } from '../mock-todo-items';

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

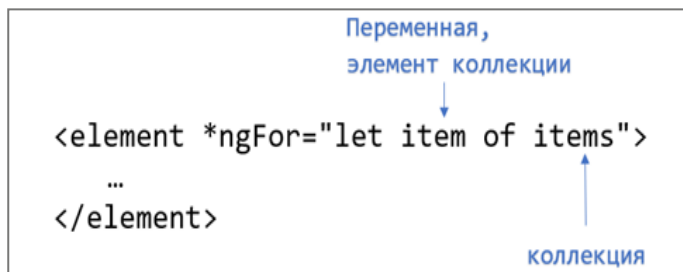
export class ToDoItemsComponent implements OnInit {
  items = ITEMS;
  toDoItem: ToDoItem = {
    id: 1,
    name: "Call Joe",
    isComplete: false
  };
  constructor() { }
  ngOnInit() { }
}
```

Для вывода массива с элементами списка в шаблоне компонента, необходима структурная директива `*ngFor`.

Директива `*ngFor` повторяет блок элементов для каждого объекта в коллекции. Таким образом реализуется

шаблонный аналог цикла `foreach` и предоставляется механизм перебора коллекции объектов.

Синтаксис:



Эта директива повторяет элемент в котором она размещена (host-элемент), для каждого элемента в коллекции. На каждой итерации цикла в переменную `item` копируется значение следующего элемента коллекции, и генерируется host-элемент со всем его содержимым

Добавим в шаблон компонента следующий код:

```

<h2>My List</h2>
<ul class="items">
  <li *ngFor="let item of items">
    {{item.name}}
  </li>
</ul>
  
```

В этом примере:

- `` это `host` элемент, он будет генерироваться для каждого элемента массива.
- `items` — это список из класса `ToDoItemsComponent`.
- `item` — содержит текущий объект для каждой итерации в списке.

После того как браузер перезагрузит страницу, вы увидите список элементов (рис. 16).

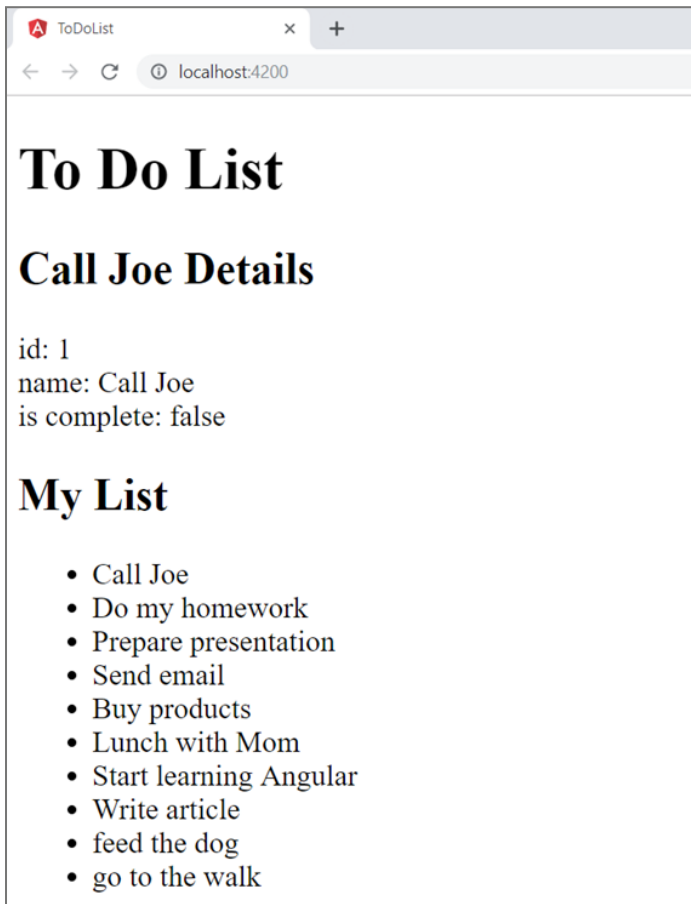


Рисунок 16

Внизу списка отобразим количество элементов, если элементы в списке есть. Для того чтобы **добавлять** или **удалять** элемент из DOM-дерева в зависимости от истинности условия, используется структурная директива `*ngIf` (**true** — добавление, **false** — удаление).

Синтаксис:

```
*ngIf="condition"
```

Добавим в шаблон компонента, под списком, следующий код:

```
<div *ngIf="items.length > 0">
  {{items.length}} {{items.length == 1 ? 'item' :
    'items'}}
</div>
```

Элемент `div` будет отображаться, только в том случае, если количество элементов в массиве `items` будет больше 0.

Так же есть структурная директива `ngSwitch`, она используется для выбора элементов, которые будут включены в документ HTML, на основании сравнения выражения с результатами отдельных выражений, определяемых в директивах `*ngSwitchCase`. Если ни одно из значений `*ngSwitchCase` не совпадает, используется элемент, к которому применена директива `*ngSwitchDefault`.

Синтаксис:

```
<container_element [ngSwitch]="switch_expression">
  <inner_element *ngSwitchCase="match_expresson_1">...
  </inner_element>
  <inner_element *ngSwitchCase="match_expresson_2">...
  </inner_element>
  <inner_element *ngSwitchCase="match_expresson_3">...
  </inner_element>
  <inner_element *ngSwitchDefault>...</element>
</container_element>
```

Рассмотрим пример, не связанный с нашим проектом, для того чтобы познакомиться с применением данной директивы. Предположим, что у нас есть переменная `car` и в зависимости от ее значения нужно вывести разные сообщения:

```
<div [ngSwitch]="car">
  <p *ngSwitchCase="'Audi'">This is Audi</p>
  <p *ngSwitchCase="'BMW'">This is BMW</p>
  <p *ngSwitchCase="'Mercedes'">This is Mercedes</p>
  <p *ngSwitchDefault>Car is undefined</p>
</div>
```

Директива `*ngSwitchDefault` работает в том случае, если нет ни одного `*ngSwitchCase` удовлетворяющего условию.

Также, обратите внимание, что директива `[ngSwitch]` пишется в квадратных скобках, а `*ngSwitchCase` и `*ngSwitchDefault` просто со звездочкой.

Добавим стили к нашему списку дел, чтобы он стал более привлекательным и при наведении курсора элементы выделялись визуально.

Стили всего приложения находятся в файле `src/styles.css`. В этом файле обычно размещают общие стили, применимые ко всему приложению. Стили применимые только для одного конкретного компонента лучше размещать в этом компоненте.

Декоратор `@Component` содержит свойства `styles` или `styleUrls`, позволяющие определить набор стилей.

Свойство `styles` должно содержать массив строк, где каждая строка определяет некоторый `CSS`.

```
@Component({
  selector: 'app-to-do-items',
  templateUrl: './to-do-items.component.html',
  styles: [
    '.items { list-style-type: none;}',
    '.items li {background-color: #EEE;}'
  ]
})

export class ToDoItemsComponent implements OnInit {
  ...
}
```

Если стилей много, то лучше их определить в отдельном файле и путь к нему указать в свойстве `styleUrls`. При создании компонента с помощью Angular CLI команды `ng`

`generate component`, по умолчанию создается файл стилей и путь к нему прописывается в декораторе.

Стили компонента `ToDoItemsComponent`, должны размещаться в файле `to-do-items.component.css`

```
.items {
  list-style-type: none;
  padding: 0;
  width: 15em;
}
.items li {
  cursor: pointer;
  background-color: #EEE;
  margin: .5em 0;
  padding: .3em;
  height: 1.6em;
  border-radius: 4px;
}
.items li:hover {
  color: #607D8B;
  background-color: #DDD;
}
.todo-count{
  font-size: 0.8rem;
}
```

Также в файл `style.css` можно добавить глобальные стили, такие как шрифт, стили для `body` и заголовков.

```
h1 {
  color: rgb(56, 129, 93);
  font-size: 250%;
}
h2, h3 {
  color: rgb(65, 100, 65);
  font-weight: lighter;
}
```

```
body {  
    margin: 2em;  
}  
* {  
    font-family: Arial, Helvetica, sans-serif;  
    color: dimgray;  
}
```

После указания стилей наша страница будет выглядеть так:

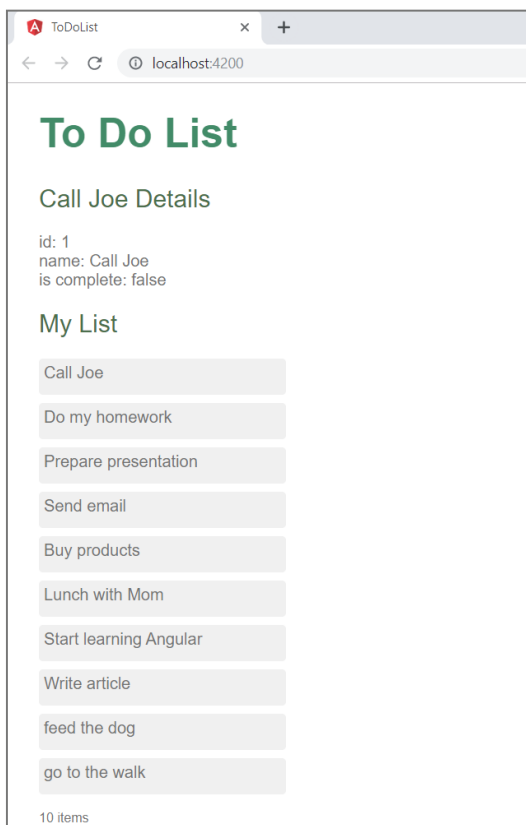


Рисунок 17

Привязка свойств

Привязка свойств — это еще один способ односторонней привязки данных. До этого мы рассматривали интерполяцию строк.

Привязка свойств передает значение из свойства компонента в DOM свойство целевого элемента.

Общий синтаксис привязки свойств выглядит следующим образом:

```
[цель] = 'выражение'.
```

Например:

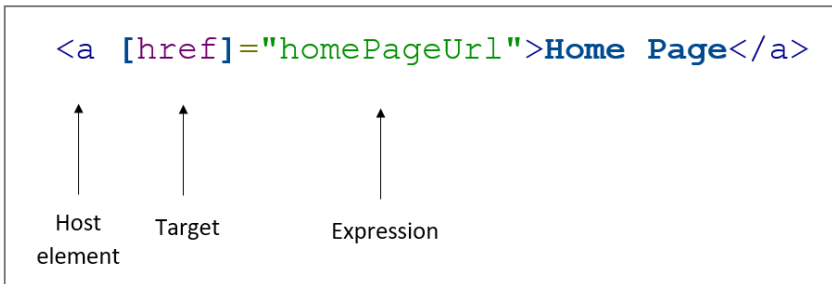


Рисунок 18

В данном случае значение из свойства `homePageUrl` компонента передается свойству `DOM href`.

Управляющий элемент (Host element) — элемент HTML, к которому будет применена привязка (с изменением его внешнего вида, контента или поведения).

Выражение (Expression) — представляет собой фрагмент JavaScript, который вычисляется с использованием

компонента шаблона для получения значения. Обычно в выражении используются свойства и методы, определяемые компонентом.

Цель (Target) — определяет DOM свойство или атрибут, к которому применяется значение выражения.

Есть несколько разных способов задать цель для привязок свойств:

Таблица 4

Способ	Описание
[свойство]	Используется для задания DOM свойства объекта JavaScript, который представляет управляющий элемент
[attr.имя]	Привязка атрибута, используемая для задания значений атрибутов управляющих элементов HTML, для которых не существует свойств DOM
[class.имя]	Специальная привязка для настройки принадлежности класса управляющего элемента
[style.имя]	Специальная привязка для настройки стиля управляющего элемента

Давайте теперь визуально выделим выбранный элемент в списке.

Добавим стили в *to-do-items.component.css*:

```
.items li.selected {
  background-color: rgb(143, 187, 130);
  color: white;
}

.items li.selected:hover {
  background-color: rgb(118, 155, 106);
}
```

И используя одностороннюю привязку мы можем элементу `li` назначить класс `selected`, в том случае, если текущий `item` идентичен `selectedItem`. Добавим привязку в файл *to-do-items.component.html*:

```
<h2>My List</h2>
<ul class="items">
  <li *ngFor="let item of items"
    (click)="onSelect(item)"
      [class.selected]="item === selectedItem">
    {{item.id}}: {{item.name}}
  </li>
</ul>
```

Привязка событий

Когда пользователь взаимодействует с интерфейсом приложения: наводит курсор на элементы, кликает мышкой по элементам web-страницы, нажимает клавиши — он инициирует возникновение соответствующих событий. Чтобы обработать эти события используются привязки.

Общий синтаксис для привязки событий:

```
(eventName) = "functionName($event)".
```

- **eventName** — событие, на которое мы хотим повесить обработчик.
- **functionName** — функция обработчик (тут может быть не только функция, а и другой JS-код, например присвоение переменной и т.д.).
- **\$event** (необязательный параметр) — объект, содержащий данные о событии.

А теперь добавим в наше приложение реакцию на действия пользователя. Когда пользователь будет щелкать по элементу в списке дел, компонент должен отображать сведения о выбранном элементе в верхней части страницы.

Для начала добавим привязку на событие **click** в элементе **li**, в обработчик будем передавать объект **item**, на котором производится щелчок:

```
to-do-items.component.html  
<h2>My List</h2>  
<ul class="items">
```

```

<li *ngFor="let item of items" (click)=
                                "onSelect(item)">
    {{item.id}}: {{item.name}}
</li>
</ul>

```

Теперь определим метод `onSelect` в классе компонента. Также зададим свойство `selectedItem`, в которое будем записывать выбранный элемент. Свойство `toDoItem` удалим, оно нам больше не понадобится.

```

to-do-items.component.ts
export class ToDoItemsComponent implements OnInit {
  items = ITEMS;
  selectedItem: ToDoItem;
  onSelect(item: ToDoItem): void {
    this.selectedItem = item;
  }

  constructor() { }
  ngOnInit() {
  }
}

```

Также в шаблоне необходимо изменить секцию с детальной информацией, так чтобы элементы в ней привязывались не к свойству `toDoItem`, а к `selectedItem`.

```

<h2>{{selectedItem.name}} details</h2>
<div><span>id: </span>{{selectedItem.id}}</div>
<div>
  <label>name:
    <input [(ngModel)]="selectedItem.name" placeholder=
              "name"/>

```

```

    </label>
  </div>

  <div>
    <label>is complete:
      <input type="checkbox" [(ngModel)] =
        "selectedItem.isComplete"/>
    </label>
  </div>

```

Но после перезагрузки страницы, мы получим ошибку. Ее можно просмотреть в консоли инструментов разработчика в браузере.

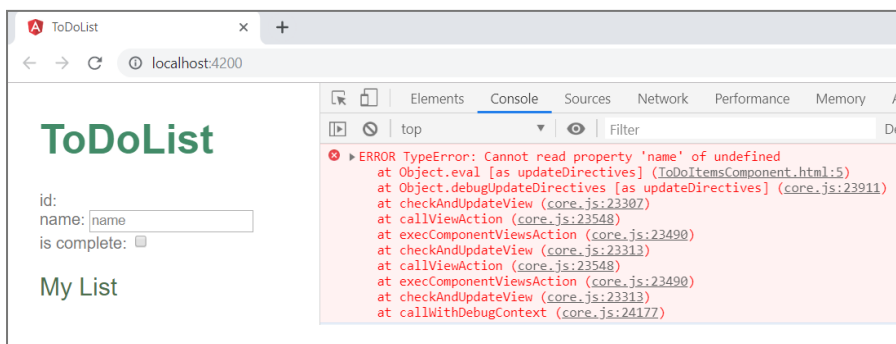


Рисунок 19

Дело в том, что при первом запуске страницы свойство `selectedItem` является `undefined`, т.к. значение ему будет присвоено только при клике на элементе списка.

Для решения этой проблемы используем в шаблоне директиву `*ngIf`.

`*ngIf` добавляет или удаляет элемент из DOM-дерева в зависимости от истинности переданного выражения (`true` — добавляет, `false` — удаляет).

```

<div *ngIf="selectedItem">
  <h2>{{selectedItem.name}} details</h2>
  <div><span>id: </span>{{selectedItem.id}}</div>
  <div>
    <label>name:
      <input [(ngModel)] = "selectedItem.name"
                placeholder = "name"/>
    </label>
  </div>

  <div>
    <label>is complete:
      <input type = "checkbox" [(ngModel)] =
                "selectedItem.isComplete"/>
    </label>
  </div>
</div>

```

Когда `selectedItem` равен `undefined`, директива удаляет `div` из DOM. Когда пользователь выбирает элемент из списка, в `selectedItem` присваивается объект `ToDoItem` и директива `*ngIf` добавляет в DOM блок с детальной информацией.