ورودی کاربر

اعمال کاربر مانند کلیک کردن بر روی یک لینک، فشار دادن دکمه و وارد کردن متن باعث پیش آمدن رویدادهای DOM می‌شوند. در این بخش به چگونگی مقید کردن این رویدادها به event handler ها کامپوننت توسط سینتکس مقیدسازی رویداد در Angular می‌پردازیم.

مقید شدن به رویدادهای ورودی کاربر

برای پاسخ دادن به هر رویداد DOM می‌توانید از مقیدسازی‌های رویداد Angular استفاده کنید. بسیاری از رویدادهای DOM توسط ورودی کاربر فعال می‌شوند. با مقید شدن به این رویدادها می‌توان ورودی‌های کاربر را دریافت کرد.

جهت مقید شدن به یک رویداد DOM اسم رویداد DOM را داخل پرانتز قرار دهید و یک دستور قالب نقل قول شده را به آن تخصیص دهید.

در مثال یک مقیدسازی رویداد نشان داده شده است که در آن از یک click handler استفاده شده است.

src/app/click-me.component.ts

```
<button (click)="onClickMe()">Click me!</button>
```

(click) واقع در سمت چپ علامت تساوی، رویداد کلیک را به عنوان هدف مقیدسازی شناسایی می‌کند. متن داخل نقل قول در سمت راست علامت تساوی همان دستور قالب است که با فراخوانی شدن متد onClickMe کامپوننت به رویداد کلیک پاسخ می‌دهد.

در زمان نوشتن یک مقیدسازی، حواستان به زمینه‌ی اجرای دستور قالب باشد. شناسه‌های موجود در یک دستور قالب به شیء زمینه‌ی خاصی تعلق دارند که معمولاً کامپوننت Angular قالب را کنترل می‌کند. در مثال بالا تنها یک خط از HTML نشان داده شده است؛ اما این خط به کامپوننت بزرگ‌تر زیر تعلق دارد:

src/app/click-me.component.ts

content_copy

```
@Component({
 selector: 'app-click-me',
```

```
template: `
<button (click)="onClickMe()">Click me!</button>
{{clickMessage}}`
})
export class ClickMeComponent {
clickMessage = '';

onClickMe() {
this.clickMessage = 'You are my hero!';
}
}
```

زمانی که کاربر بر روی دکمه کلیک می‌کند، Angular متد onClickMe را از ClickMeComponent فراخوانی می‌کند.

دریافت ورودی کاربر از شیء $event

رویدادهای DOM بار مفیدی از اطلاعات را با خود حمل می‌کنند که این اطلاعات می‌توانند برای کامپوننت مفید باشند. در این بخش به چگونگی مقید شدن به رویداد keyup مربوط به کادر ورودی می‌پردازیم تا بتوانیم بعد از هر بار فشرده شدن دکمه‌ها ورودی کاربر را دریافت کنیم.

کد زیر به رویداد keyup توجه می‌کند و کل بار مفید رویداد را ($event) به event handler کامپوننت می‌دهد.

src/app/keyup.components.ts (template v.1)

```
template: `
<input (keyup)="onKey($event)">
<p>{{values}}</p>
`
```

زمانی که کاربری کلیدی را فشار می‌دهد و رها می‌کند، رویداد keyup رخ می‌دهد و Angular در متغیر $event شیء رویداد DOM متناظری را فراهم می‌کند که این کد به عنوان پارامتری به متد onKey() کامپوننت می‌رود.

src/app/keyup.components.ts (class v.1)

```
export class KeyUpComponent_v1 {
 values = '';

 onKey(event: any) { // without type info
 this.values += event.target.value + ' | ';
 }
}
```

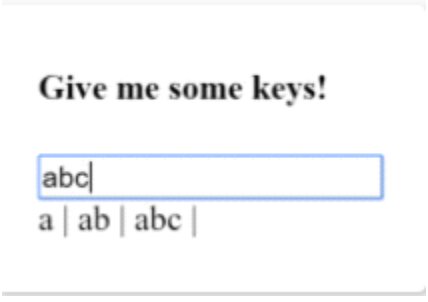
ویژگی‌های یک شیء $event براساس نوع رویداد DOM متفاوت است. برای مثال یک رویداد mouse نسبت به یک رویداد ویرایش کادر ورودی شامل اطلاعات متفاوتی است.

تمامی اشیاء رویدادهای استاندارد DOM دارای یک ویژگی target هستند که این ویژگی مرجعی برای عنصری است که باعث رخ دادن این رویداد شده است. در این صورت target به عنصر <input> اشاره می‌کند و event.target.value محتوای فعلی این عنصر را برگشت می‌دهد.

متد onKey() بعد از هر بار فراخوانی به محتوای مقدار کادر ورودی مربوط به لیست موجود در ویژگی values کامپوننت پیوست می‌شود که به دنبال این ویژگی از کاراکتر جدا کننده‌ی (|) استفاده می‌شود. میانگیری تغییرات روی هم رفته‌ی کادر ورودی حاصل از ویژگی values را نمایش می‌دهد.

فرض کنید کاربر حروف "abc" را وارد کند و سپس آن‌ها را یک به یک پاک کند. چیزی که رابط کاربری نمایش می‌دهد به صورت زیر است:

a | ab | abc | ab | a | |



به عنوان جایگزین می‌توانید برای کنار هم قرار دادن کلیدها به صورت مجزا جای event.key را با event.target.value عوض کنید. در این صورت ورودی تولید شده‌ی کاربر تفاوتی نخواهد کرد.

a | b | c | backspace | backspace | backspace |

نوع دادن به $event

در مثال بالا $event به عنوان نوع any قالب بندی (cast) می‌شود. این کار باعث ساده شدن کد می‌شود البته بهای آن را نیز باید پرداخت. هیچ نوعی از اطلاعات وجود ندارد که بتواند ویژگی‌های شیء رویداد را آشکار کند و از تصمیمات اشتباه جلوگیری کند.

در مثال زیر این متد همراه با نوع‌های آن بازنویسی شده است:

```
src/app/keyup.components.ts (class v.1 - typed)
export class KeyUpComponent_v1 {
 values = '';
 onKey(event: KeyboardEvent) { // with type info
 this.values += (<HTMLInputElement>event.target).value + ' | ';
 }
}
```

حالا دیگر $event یک KeyboardEvent مشخص است. همه‌ی عناصر ویژگی value را ندارند، به همین دلیل $event ، target را به صورت یک عنصر ورودی قالب بندی می‌کند. متد OnKey به صورت شفاف‌تر چیزی که از قالب نیاز دارد و چگونگی تفسیر رویداد را به صورت شفاف‌تر بیان می‌کند.

دادن $event به متغیر کار سؤال برانگیزی است

نوع دهی به شیء رویداد مشکوک بودن دادن کل رویداد DOM به متد را برملا می‌کند. کامپوننت نسبت به جزئیات قالب بیش از حد آگاه است به گونه‌ای که نمی‌تواند بدون کسب اطلاعات در رابطه با پیاده سازی HTML بیش از آن چیزی که باید بداند، اطلاعات را استخراج کند. این کار باعث می‌شود مجزا بودن رابطه‌ی بین قالب (چیزی که کاربر می‌بیند) و کامپوننت (آن طور که برنامه داده‌های کاربر را پردازش می‌کند) از بین برود.

در بخش بعد به چگونگی استفاده از متغیرهای مرجع قالب جهت رسیدگی به این مشکل پرداخته می‌شود.

دریافت ورودی کاربر از یک متغیر مرجع قالب

برای دریافت اطلاعات کاربر راه دیگری وجود دارد که این کار با استفاده از متغیرهای مرجع قالب در Angular انجام می‌شود. این متغیرها از درون قالب امکان دسترسی مستقیم به یک عنصر را فراهم می‌کنند. برای اعلان این متغیرها تنها کافی است جلوی آن‌ها از یک شناسه به همراه یک کاراکتر (#) (یا پوند) استفاده کنید.

در مثال زیر جهت اجرای یک حلقه‌ی برگشتی فشردن دکمه در یک قالب ساده از یک متغیر مرجعه قالب استفاده شده است.

src/app/loop-back.component.ts

```
@Component({
 selector: 'app-loop-back',
 template: `
<input #box (keyup)="0">
<p>{{box.value}}</p>
`
})
export class LoopbackComponent { }
```

اسم این متغیر box است، در عنصر <input> اعلان شده است و به خود این عنصر اشاره می‌کند. این کد برای دریافت value عنصر ورودی از متغیر box استفاده می‌کند و برای نمایش دادن آن، آن را میان تگ‌های <p> قرار می‌دهد.

این قالب کامل و مستقل است؛ به گونه‌ای که به کامپوننت مقید نمی‌شود و این کامپوننت هیچ کاری انجام نمی‌دهد.

در کادر ورودی چیزی را تایپ کنید و با هر بار فشردن دکمه به روز شدن صفحه نمایش را تماشا کنید.

این کار به هیچ وجه جواب نمی‌دهد مگر آن که شما به رویدادی مقید شوید.

Angular تنها در صورتی مقیدسازی‌ها (و به دنبال آن صفحه‌ی نمایش) را به روز می‌کند که برنامه در پاسخ به رویدادهای ناهمگام مانند فشردن دکمه، کاری انجام دهد. کد این مثال رویداد keyup را به عدد صفر که کوتاهترین دستور ممکن برای قالب است مقید می‌کند. با وجود این که این دستور هیچ کار مفیدی انجام نمی‌دهد، اما الزامات Angular را برآورده می‌کند تا Angular بتواند صفحه‌ی نمایش را به روز کند.

رسیدگی به کادر ورودی به کمک متغیرهای مرجع قالب نسبت به انجام این کار توسط شیء $event ساده‌تر است. در ادامه مثال keyup قبلی که در آن برای دریافت ورودی کاربر از متغیر مرجع قالب استفاده می‌شود، بازنویسی شده است.

src/app/keyup.components.ts (v2)

```
@Component({
 selector: 'app-key-up2',
 template: `
 <input #box (keyup)="onKey(box.value)">
 <p>{{values}}</p>
 `
})
export class KeyUpComponent_v2 {
 values = '';
 onKey(value: string) {
 this.values += value + ' | ';
 }
}
```

یکی از جنبه‌های زیبای این رویکرد این است که کامپوننت مقادیر داده‌ای رابدون دردسر از view دریافت می‌کند و دیگر نیازی به دانستن $event و ساختار آن ندارد.

فیلتر کردن رویداد key (به کمک key.enter)

ایونت هندلر keyup به تمامی کلیدهای فشرده شده توجه می‌کند. در برخی مواقع تنها دکمه‌ی Enter اهمیت دارد؛ زیرا این دکمه مشخص می‌کند که تایپ کردن کاربر تمام شده است. یکی از راه‌های کاهش نویز این است که تمامی $event.keyCode ها بررسی شوند و تنها زمانی اقدام به انجام کاری کرد که این کلید Enter باشد.

راه ساده‌تری نیز وجود دارد: مقید شدن به شبه رویداد Angular keyup.enter. با انجام این کار Angular تنها زمانی event handler را فراخوانی می‌کند که کاربر دکمه‌ی Enter را فشار دهد.

```
src/app/keyup.components.ts (v3)
@Component({
 selector: 'app-key-up3',
 template: `
 <input #box (keyup.enter)="onEnter(box.value)">
 <p>{{value}}</p>
 `
})
export class KeyUpComponent_v3 {
 value = '';
 onEnter(value: string) { this.value = value; }
}
```

نتیجه‌ی این کار به صورت زیر است.

رویداد blur

در مثال قبل اگر کاربر موس را حرکت دهد و بر روی چیز دیگری در صفحه کلیک کند بدون آن که قبل از آن دکمه‌ی *Enter* را فشار داده باشد، در این صورت حالت فعلی کادر ورودی از بین می‌رود. ویژگی value کامپوننت تنها زمانی به روز می‌شود که کاربر دکمه‌ی *Enter* را فشار دهد.

برای حل این مشکل باید هم به دکمه‌ی *Enter* و هم به رویداد blur توجه کرد.

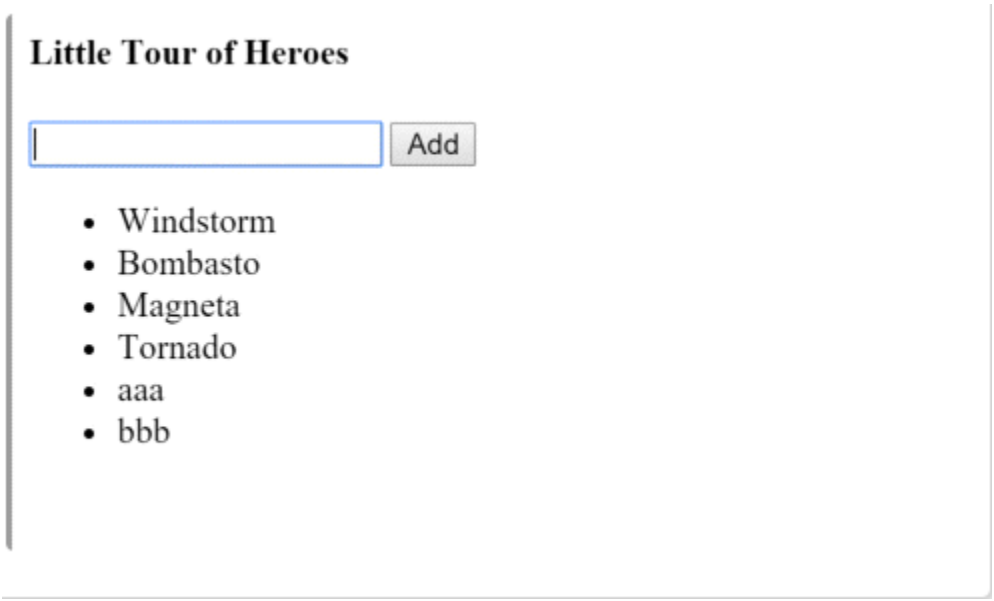
```
src/app/keyup.components.ts (v4)
@Component({
 selector: 'app-key-up4',
 template: `
 <input #box
 (keyup.enter)="update(box.value)"
 (blur)="update(box.value)">

 <p>{{value}}</p>
 `
})
export class KeyUpComponent_v4 {
 value = '';
 update(value: string) { this.value = value; }
}
```

ترکیب مطالب در کنار یکدیگر

در صفحه‌ی قبل چگونگی نمایش داده‌ها آموزش داده شد. در این صفحه به توضیح روش‌های مقیدسازی رویداد پرداخته شد.

حالا در یک برنامه‌ی کوچک این دو را در کنار یکدیگر قرار دهید به گونه‌ای که این برنامه فهرستی از هیروها را نمایش دهد و هیروهای جدیدی را به این فهرست اضافه کند. در این صورت کاربر می‌تواند با نوشتن اسم هیرو در کادر ورودی و کلیک کردن بر روی Add هیرویی را به این فهرست اضافه کند.



در ادامه می‌توانید کامپوننت "Little Tour of Heroes" را مشاهده کنید.

src/app/little-tour.component.ts

```
@Component({
 selector: 'app-little-tour',
 template: `
 <input #newHero
 (keyup.enter)="addHero(newHero.value)"
 (blur)="addHero(newHero.value); newHero.value='' ">

 <button (click)="addHero(newHero.value)">Add</button>

 <ul><li *ngFor="let hero of heroes">{{hero}}</li></ul>
 `
})
```

```
export class LittleTourComponent {

heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];

addHero(newHero: string) {

if (newHero) {

this.heroes.push(newHero);

}

}

}
```

مشاهدات

- برای اشاره به عناصر از متغیرهای قالب استفاده کنید. متغیر قالب newHero به عنصر <input> اشاره می‌کند. می‌توانید از تمامی فرزندان یا هم نیاهای عنصر <input> به newHero اشاره کنید.
- مقادیر را عبور دهید نه عناصر را. به جای عبور دادن newHero در متد addHero کامپوننت، مقدار کادر ورودی را دریافت کنید و این مقدار را به addHero بدهید.
- دستورات قالب را ساده نگه دارید. رویداد (blur) مقید به دو دستور جاوا اسکریپت است. اولین دستور addHero را فراخوانی می‌کند و دومین دستور (newHero.value='') بعد از اضافه شدن هیروی جدید به لیست کادر ورودی را پاک می‌کند.

سورس کد

در زیر می‌توانید تمامی کدهایی که در این صفحه به آن‌ها پرداخته شد را مشاهده کنید.

click-me.component.ts

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

@Component({
selector: 'app-click-me',
template: `
<button (click)="onClickMe()">Click me!</button>
{{clickMessage}}`
})
export class ClickMeComponent {
clickMessage = '';

onClickMe() {
this.clickMessage = 'You are my hero!';
}
```

```
}
```

keyup.components.ts

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

@Component({
selector: 'app-key-up1',
template: `
<input (keyup)="onKey($event)">
<p>{{values}}</p>
`
})
export class KeyUpComponent_v1 {
values = '';

/*
onKey(event: any) { // without type info
this.values += event.target.value + ' | ';
}
*/

onKey(event: KeyboardEvent) { // with type info
this.values += (<HTMLInputElement>event.target).value + ' | ';
}
}

////////////////////////////////////////

@Component({
selector: 'app-key-up2',
template: `
<input #box (keyup)="onKey(box.value)">
<p>{{values}}</p>
`
})
export class KeyUpComponent_v2 {
values = '';
onKey(value: string) {
this.values += value + ' | ';
}
}

////////////////////////////////////////

```

```
@Component({
selector: 'app-key-up3',
template: `
<input #box (keyup.enter)="onEnter(box.value)">
<p>{{value}}</p>
`
})
export class KeyUpComponent_v3 {
value = '';
onEnter(value: string) { this.value = value; }
}

////////////////////////////////////////

@Component({
selector: 'app-key-up4',
template: `
<input #box
(keyup.enter)="update(box.value)"
(blur)="update(box.value)">

<p>{{value}}</p>
`
})
export class KeyUpComponent_v4 {
value = '';
update(value: string) { this.value = value; }
}
```

loop-back.component.ts

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

@Component({

selector: 'app-loop-back',

template: `

<input #box (keyup)="0">

<p>{{box.value}}</p>

`

})

export class LoopbackComponent { }
```

little-tour.component.ts

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

@Component({
selector: 'app-little-tour',
template: `
<input #newHero
(keyup.enter)="addHero(newHero.value)"
(blur)="addHero(newHero.value); newHero.value='' ">

<button (click)="addHero(newHero.value)">Add</button>

<ul><li *ngFor="let hero of heroes">{{hero}}</li></ul>
`
})
export class LittleTourComponent {
heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
addHero(newHero: string) {
if (newHero) {
this.heroes.push(newHero);
}
}
}
```

خلاصه

تا به اینجای کار به اصول اولیه‌ی پاسخ به ورودی‌ها و اشارات کاربر مسلط شده‌اید.

این روش‌ها برای نمایش‌های در مقیاس کوچک مناسب هستند و در صورتی که بخواهید با مقادیر زیادی از ورودی‌های کاربر سروکار داشته باشید، به سرعت این کدها را بد ترکیب و پر از کدنویسی می‌یابید. برای جا به جایی مقادیر میان فیلدهای ورودی داده‌ها و ویژگی‌های مدل مقیدسازی دو طرفه راه جمع و جورتر و زیباتری است. در صفحه‌ی بعد در بخش فرم‌ها به چگونگی نوشتن مقیدسازی‌های دوطرفه به کمک NgModel می‌پردازیم.