

# UNIT 3

## Ionic



### **Part 2** **Lifecycle. More components.** **Modals. Custom events.**

Client-side Web Development  
2nd course – DAW  
IES San Vicente 2020/2021  
Author: Arturo Bernal Mayordomo

# Index

Ionic Pages Lifecycle.....	3
More Components.....	4
Action Sheets.....	4
Fabs.....	4
Grid system.....	5
CSS Utilities.....	5
Loading.....	5
Popover.....	6
Range.....	7
Refresher.....	7
InfiniteScroll.....	8
Searchbar.....	9
Scroll.....	10
Reorder.....	10
Segment.....	11
Select.....	12
Slides.....	12
Toolbar.....	13
Virtual Scroll.....	14
Gestures.....	14
Skeleton.....	15
Modal windows.....	16
Custom events.....	17

# Ionic Pages Lifecycle

---

Ionic pages are Angular components with a thin layer on top of them added to them. This means we can use all Angular events in their lifecycle (`ngOnInit`, `ngOnDestroy`, etc.), but Ionic adds some more events.

For example, not every time we enter a page, `ngOnInit` will be called (or `ngOnDestroy` when we leave). This is because Ionic's router sometimes stores in cache the previous page (when we navigate forward or open a modal window). The `ngOnDestroy` method only gets called when a component is destroyed, and the `ngOnInit` when it's created, but when we go back to a page stored in cache it won't be called again.

Ionic introduces 4 new events to the pages lifecycle:

- **ionViewWillEnter** → Fires when the we are going to enter the page but the transition (animation) hasn't started yet. It will execute just after `ngOnInit` when the component is created the first time, and will be called every time we navigate to the page (back or forward).
- **ionViewDidEnter** → Fires after `ionViewWillEnter`, just after the transition animation has ended.
- **ionViewWillLeave** → Fires when we are about to leave the page (before the animation starts). Unlike `ngOnDestroy`, this event fires even if the page isn't destroyed.
- **ionViewDidLeave** → Fires after `ionViewWillLeave`. When the transition animation has ended.

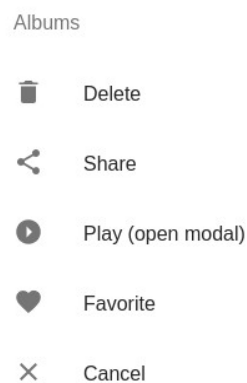
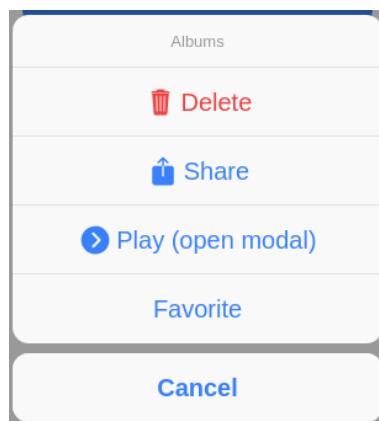
<https://ionicframework.com/docs/angular/lifecycle>

# More Components

## Action Sheets

An [Action Sheet](#) is a type of dialog that appears from the window's bottom and has some options to choose from. This component has a title (and optionally a subtitle) and an array of buttons. Each button has a text and a handler. After clicking a button, the default behavior is to close the Action Sheet, unless its handler returns false.

A button can also have an icon and a role (only visible on iOS). Role can be "destructive" (only the first button, appears in red) or "cancel" (only the last button, appears separated from the rest).



## Fabs

[Fabs](#) are floating buttons put inside a **ion-fab** container. Button containers can be aligned in an absolute position using **vertical** (top, bottom, center), **horizontal** (start, end, center) and **edge** attributes. They also can contain a [list of buttons](#) that appear when we click on the main (Fab) button.

```
<ion-fab vertical="bottom" horizontal="start" slot="fixed">
  <ion-fab-button>
    <ion-icon name="share"></ion-icon>
  </ion-fab-button>
  <ion-fab-list side="end">
    <ion-fab-button>
      <ion-icon name="logo-vimeo"></ion-icon>
    </ion-fab-button>
    <ion-fab-button>
      <ion-icon name="logo-facebook"></ion-icon>
    </ion-fab-button>
    <ion-fab-button>
      <ion-icon name="logo-instagram"></ion-icon>
    </ion-fab-button>
    <ion-fab-button>
      <ion-icon name="logo-twitter"></ion-icon>
    </ion-fab-button>
  </ion-fab-list>
</ion-fab>
```



## Grid system

Ionic has a simple [grid system](#) based on Flexbox. We can divide some content in columns if needed. In order to do this, we create row ([ion-row](#)) elements and columns ([ion-col](#)) inside of them. We can put rows inside an [ion-grid](#) element to have additional padding to rows.

The grid is divided in 12 columns (like Bootstrap). By default, all columns take the same space. We can use attributes like **size="auto"** (a column only takes the minimum space necessary), **size="1"** (1, 2, 3, 4, ..., 12 → Number of columns it occupies), **offset="1"** (Number of columns pushed to the right), etc. Almost all attributes have the equivalents only for small, medium, large, ... screens (size-sm, size-md, offset-lg, etc.).

```
<ion-grid>
  <ion-row>
    <ion-col>
      <p>This content takes 50% of space by default</p>
      <ion-button expand="block" color="primary">A button</ion-button>
    </ion-col>
    <ion-col>
      <p>This content also takes 50% of space</p>
      <ion-button expand="block" color="secondary">Other button</ion-button>
    </ion-col>
  </ion-row>
  <ion-row>
    <ion-col size="6" offset="3">
      This content has 25% margin from the left side and takes 50% of space.
    </ion-col>
  </ion-row>
</ion-grid>
```

This content takes 50%  
of space by default

This content also takes  
50% of space



This content has 25%  
margin from the left side  
and takes 50% of space.

## CSS Utilities

Ionic provides some attributes for generic CSS properties, like text alignment, padding, margin, or flex alignment. You can see examples of these properties in the official documentation for [CSS utilities](#).

## Loading

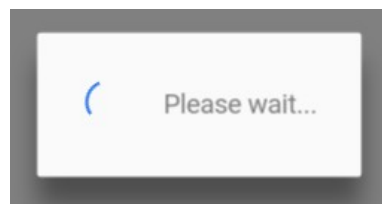
Usually, when doing a background task that can take more than a few milliseconds (connecting to a web service when the network is slow for example), we want to tell the user that he/she must wait for an important task to complete. Showing a [loading indicator](#) that blocks the screen is usually the best way to do this.

We create these loading indicators using Ionic's LoadingController. It's similar to showing an alert or toast message. After creating it, we must show it calling **present()**,

and it will disappear once we call **dismiss()** on it (after the task finishes), or after the maximum duration (if any established) ends.

```
async showLoading() {
  this.loading = await this.loadingCtrl.create({
    message: 'Please wait...',
    duration: 2000
  });

  return await this.loading.present();
}
```



## Popover

**Popovers** are like modal pages but don't use the entire screen. They're normally used when actions don't fit in the navigation bar, but they can be used anywhere.

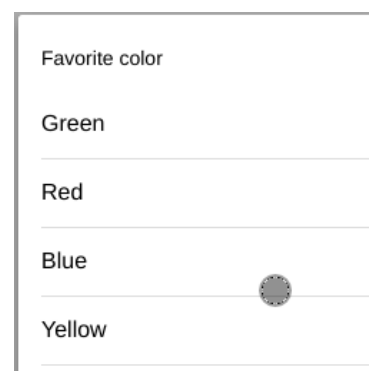
We create a Popover using the PopoverController. We need to have a component (with a template) class which controls the popover and its content. Parameters can be passed inside an object as the second argument of **create** (use NavParams to get the data passed inside the component).

```
@Component({
  selector: 'app-popovers',
  templateUrl: './popovers.page.html',
  styleUrls: ['./popovers.page.scss'],
})
export class PopoversPage implements OnInit {

  constructor(public popoverCtrl: PopoverController) {}

  ngOnInit() {}

  async showPopover() {
    const popover = await this.popoverCtrl.create({
      component: PopoverComponent,
      componentProps: {
        title: 'Favorite color' // Input parameters
      }
    });
    await popover.present();
    const result = (await popover.onDidDismiss()).data;
    console.log(result);
  }
}
```



```
@Component({
  selector: 'popover-component',
  template: `
    <ion-content>
      <ion-list>
        <ion-list-header>{{title}}</ion-list-header>
        <ion-item (click)="close('green')"><ion-label>Green</ion-label></ion-item>
        <ion-item (click)="close('red')"><ion-label>Red</ion-label></ion-item>
        <ion-item (click)="close('blue')"><ion-label>Blue</ion-label></ion-item>
        <ion-item (click)="close('yellow')"><ion-label>Yellow</ion-label></ion-item>
      </ion-list>
    </ion-content>
  `
})
export class PopoverComponent {
  @Input() title: string;

  constructor(public popoverCtrl: PopoverController) {}
}
```

```

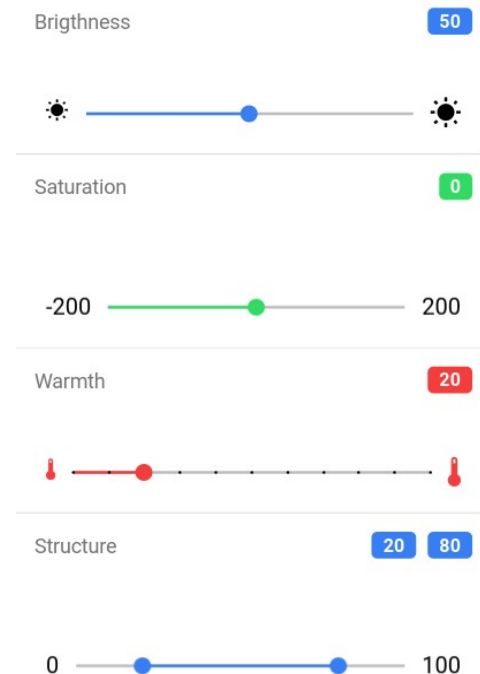
close(val: string) {
  this.popoverCtrl.dismiss(val);
}
}

```

## Range

The most efficient and comfortable way to make the user to select between a range of numbers is to use a [Range](#) component. This component behaves like a normal input (uses `[(ngModel)]`) and can be included inside a form), but we can be sure the user can only select a range of numbers.

Ranges are configurable using the attributes: **max** (maximum value), **min** (minimum value), **pin** (shows a pin with the current value when moving it), **step** (can only increment/decrement the value using this step), **snaps** (shows value points, useful with 'step'), **dualKnobs** (2 min/max values, **value.lower** and **value.upper**).



## Refresher

Ionic's [Refresher](#) let's you implement "pull to refresh" functionality. This means that when you pull down far enough the content with your finger, a loading icon and text will show until you hide it (when you get new data from the server). Default icons and text can be changed with parameter you'll find in the documentation.

This component is placed at the beginning of the ion-content element.

```

<ion-refresher slot="fixed" (ionRefresh)="refresh($event)">
  <ion-refresher-content
    pullingIcon="arrow-dropdown"
    pullingText="Pull to refresh"
    refreshingSpinner="circles"
    refreshingText="Refreshing..."
  </ion-refresher-content>
</ion-refresher>

```



```

export class SomePage {
  ...

  refresh(event) {
    console.log('Begin async operation');

    setTimeout(() => {
      console.log('Async operation has ended');
      event.target.complete();
    }, 2000);
  }
}

```

 Pull to refresh	 Refreshing...
Star wars	Star wars
Lord of the rings	Lord of the rings
Terminator	Terminator

## InfiniteScroll

[InfiniteScroll](#) is a component that allows to automatically call a method (usually to load more items) when you scroll to the bottom of the content. It's the opposite action as the Refresher component does. This component is placed at the end of the ion-content element.

We can customize this component by changing the spinner icon and the displayed text. We can also establish a **threshold** (distance to the bottom) when the event is fired (a string in px, % or other units. Default → 15%). Also, we can deactivate it with the **enabled** attribute (when no more items need to be loaded).

```
<ion-content padding>
  <ion-list>
    <ion-item *ngFor="let item of items">
      {{item}}
    </ion-item>
  </ion-list>
  <ion-infinite-scroll (ionInfinite)="loadMoreItems($event)"
[disabled]="finished">
    <ion-infinite-scroll-content
      loadingSpinner="bubbles"
      loadingText="Loading more data..."
    </ion-infinite-scroll-content>
  </ion-infinite-scroll>
</ion-content>
```

```
export class InfiniteScrollPage implements OnInit {
  items: String[] = [];
  num = 1;
  finished = false;

  constructor() {}

  ngOnInit() {
    this.loadMoreItems(null);
  }

  loadMoreItems(event) {
    // Simulating an external service call with a timeout
    setTimeout(() => {
      const max = this.num + 15;
      for (; this.num < max; this.num++) {
        this.items.push('Item ' + this.num);
      }

      if (this.num >= 60) { // We'll load a maximum of 60 items
        this.finished = true;
      }

      if (event != null) {
        event.target.complete(); // Hide the loader
      }
    });
  }
}
```



```
}, event === null ? 0 : 2000);
}
```

Item 43

Item 44

Item 45



Loading more data...

## Searchbar

Ionic's [Searchbar](#) component is like an input but with more options oriented to build a search bar (inside the content or inside a Toolbar for example). In the following example we use **debounce** to establish the time between the user stops writing and the event is fired (default: 250ms).

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-title>Searchbar example</ion-title>
  </ion-toolbar>
  <ion-toolbar>
    <ion-searchbar debounce="500" (ionChange)="filterItems($event)">
    </ion-searchbar>
  </ion-toolbar>
</ion-header>

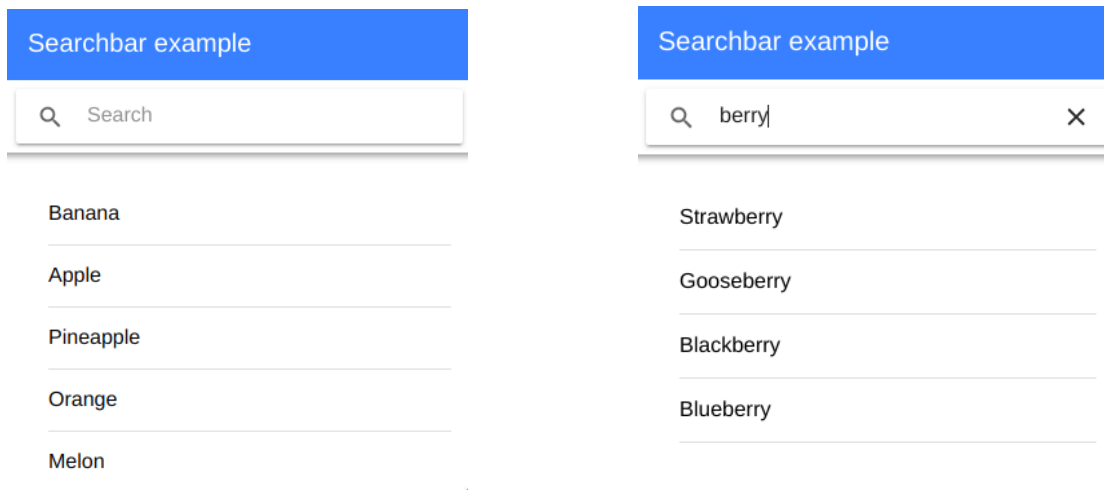
<ion-content padding>
  <ion-list>
    <ion-item *ngFor="let item of filteredItems">
      {{item}}
    </ion-item>
  </ion-list>
</ion-content>

export class SearchbarPage implements OnInit {
  items: String[] = [
    'Banana', 'Apple', 'Pineapple', 'Orange', 'Melon', 'Watermelon',
    'Peach', 'Strawberry', 'Gooseberry', 'Blackberry', 'Blueberry'
  ];
  filteredItems: String[];

  constructor() { }

  ngOnInit() {
    this.filteredItems = this.items;
  }

  filterItems(event) {
    let search: string = event.target.value;
    if (search && search.trim() !== '') {
      search = search.trim().toLowerCase();
      this.filteredItems = this.items
        .filter(i => i.toLowerCase().includes(search));
    } else {
      this.filteredItems = this.items;
    }
  }
}
```



## Scroll

In previous versions of Ionic (<4), there was an element called `ion-scroll` that we could use, for example, to create a horizontal scroll. Now, we must emulate that behavior with CSS.

```
<ion-content padding>
  <div class="scrollX">
    <ion-button *ngFor="let icon of icons" fill="clear">
      <ion-icon slot="icon-only" [name]="icon"></ion-icon>
    </ion-button>
  </div>
</ion-content>




.scrollX {
  overflow: auto;
  white-space: nowrap;
  &::-webkit-scrollbar {
    display: none;
  }
}

export class ScrollPage {
  icons: string[] = [
    "alert", "albums", "alarm", "analytics", "logo-angular", "logo-apple",
    "appstore", "archive", "at", "baseball", "basket", "battery-charging",
    "beer", "bicycle", "logo-bitcoin", "boat"
  ];
}
```

## Reorder

The [Reorder](#) element allows an `ion-item` to be dragged and dropped in a list. Items with this reordering element inside must be grouped inside a [Reorder Group](#). We can use the `disable` property to hide the reordering element. The event `ionItemReorder` is fired every time we change the position of any element, and we need to call the `complete()` method to modify the original array after this event.

```
<ion-list>
  <ion-reorder-group (ionItemReorder)="reorder($event)"
    [disabled]="disableOrdering">
```

<code>&lt;ion-item *ngFor="let food of foods"&gt;</code>	
<code>&lt;ion-label&gt;{{food}}&lt;/ion-label&gt;</code>	 Steak
<code>&lt;ion-reorder slot="start"&gt;&lt;/ion-reorder&gt;</code>	
<code>&lt;/ion-item&gt;</code>	
<code>&lt;/ion-reorder-group&gt;</code>	 Pizza
<code>&lt;/ion-list&gt;</code>	
 <code>export class ReorderPage implements OnInit {</code>	
<code>  foods: string[] = [</code>	
<code>    'Pizza', 'Banana', 'Hamburguer', 'Soup', 'Steak', 'Apple'</code>	 Hamburguer
<code>  ];</code>	
<code>  disableOrdering = false;</code>	
 <code>  reorder(event: CustomEvent) {</code>	
<code>    const elemFrom = this.foods.splice(event.detail.from, 1);</code>	 Apple
<code>    this.foods.splice(event.detail.to, 0, elemFrom[0]);</code>	
<code>    event.detail.complete();</code>	
<code>  }</code>	
<code>}</code>	

## Segment

In Ionic, [Segments](#) are like different sections in the same page. Internally they work as a **select** element (They are attached to a property by **[(ngModel)]** and each option has a value). They can be placed anywhere (in a toolbar, header or footer, or in the content). They affect the part of the content we want them to affect.

```
<ion-header>
...
<ion-toolbar color="light">
  <ion-segment [(ngModel)]="type">
    <ion-segment-button value="heroes">
      Heroes
    </ion-segment-button>
    <ion-segment-button value="villains">
      Villains
    </ion-segment-button>
    <ion-segment-button value="weapons">
      Weapons
    </ion-segment-button>
  </ion-segment>
</ion-toolbar>
</ion-header>

<ion-content padding>
  <div [ngSwitch]="type">
    <ion-list *ngSwitchCase="'heroes'">
      <ion-item *ngFor="let hero of heroes">
        {{hero}}
      </ion-item>
    </ion-list>
    <ion-list *ngSwitchCase="'villains'">
      <ion-item *ngFor="let villain of villains">
        {{villain}}
      </ion-item>
    </ion-list>
    <ion-list *ngSwitchCase="'weapons'">
      <ion-item *ngFor="let weapon of weapons">
        {{weapon}}
      </ion-item>
    </ion-list>
  </div>
</ion-content>

export class SegmentPage {
```

```

type: string = "heroes";
heroes: string[] = ["Batman", "Superman", "Spiderman", "Hulk", "Mazinger Z"];
villains: string[] = ["Dr Eggman", "The Joker", "Darth Vader", "Hannibal
Lecter"];
weapons: string[] = ["Missile", "Laser gun", "Tank", "X Rays"];
...
}

```

HEROES	VILLAINS	WEAPONS
Batman	Dr Eggman	
Superman	The Joker	
Spiderman	Darth Vader	

## Select

Ionic's [Select](#) element can be seen as a custom `<select>` element. The main difference is we have to use **ion-select** and **ion-option** elements. When you click on the element a dialog will appear where you select an option. We can choose which text the cancel and confirm button will have and if it should support multiple choices with **multiple="true"** (the property binded will be an array then).

```

<ion-list>
  <ion-item>
    <ion-label>Gaming</ion-label>
    <ion-select [(ngModel)]="selectedConsoles" multiple="true"
      cancelText="No!" okText="Okay!">
      <ion-select-option *ngFor="let console of consoles"
        [value]="console.val">{{console.title}}</ion-select-option>
    </ion-select>
  </ion-item>
</ion-list>
<p>
  Selected consoles: {{selectedConsoles}}
</p>

```

```

export class SelectPage {
  selectedConsoles: string[] = [];
  consoles: {val: string, title: string}[] = [
    {val: "nes", title: "NES"},
    {val: "n64", title: "Nintendo64"},
    {val: "ps", title: "Playstation"},
    {val: "md", title: "Mega Drive"},
    {val: "saturn", title: "Saturn"},
    {val: "snes", title: "SNES"},
  ];
  ...
}

```

Gaming

☐ NES
 ☐ Nintendo64
 ☒ Playstation
 ☒ Mega Drive
 ☒ Saturn

NO! OKAY!

## Slides

[Slides](#) by default take the full content's width and height. We can go from a slide to the next or previous by sliding on the the screen with our finger. There are also methods to automatically make transitions (also with a delay).

```

<ion-content>
  <ion-slides (ionSlideDidChange)="slideChanged()">
    <ion-slide>
      <h1>Slide 1 &gt;</h1>
    </ion-slide>
    <ion-slide>
      <h1>&lt;&lt; Slide 2 &gt;</h1>
    </ion-slide>
    <ion-slide>
      <h1>&lt;&lt; Slide 3</h1>
    </ion-slide>
  </ion-slides>
</ion-content>

ion-slides {
  height: 100%;
}

export class SlidesPage {
  @ViewChild(IonSlides) slides: IonSlides;

  ...

  slideChanged() {
    console.log("Current index: " + await this.slides.getActiveIndex());
  }
}

```

## Toolbar

[Toolbars](#) are elements that take all horizontal space, and are used to place titles, buttons, inputs, etc. There can be more than one in the same page and they can be placed inside an **ion-header**, **ion-footer** or **ion-content** (in this case scrolls with the content).

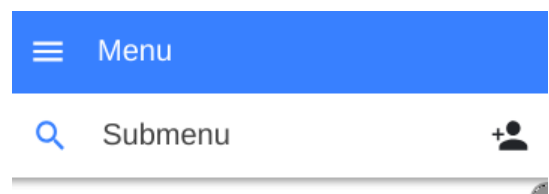
Buttons should be placed inside an **ion-buttons** element. Elements can be positioned with the attribute **slot** and these values:

- start** → Positioned at the far **left** of the element.
- end** → Positioned at the far **right** of the element.
- secondary** → Like start (left) but it's placed after.
- primary** → Like end (right) but it's placed before.

```

<ion-header>
  <ion-toolbar color="primary">
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title>Menu</ion-title>
  </ion-toolbar>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-button color="primary">
        <ion-icon slot="icon-only" name="search"></ion-icon>
      </ion-button>
    </ion-buttons>
    <ion-title>Submenu</ion-title>
    <ion-buttons slot="end">
      <ion-button color="dark">
        <ion-icon slot="icon-only" name="person-add"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

```



```
</ion-toolbar>
</ion-header>
```

## Virtual Scroll

[VirtualScroll](#) is recommended to be used instead of `*ngFor` with **very large** lists for performance reasons (but managing very complex list items may cause the performance to be actually worse). This kind of item repeater only inserts in the DOM the visible elements on the screen. When scrolling, VirtualScroll destroys (removes from the DOM) dynamically elements that are no longer visible and inserts the new ones.

Each item is represented with the directive `*virtualItem`. Optionally, we can use `*virtualHeader` to insert **list dividers** using a function.

```
<ion-virtual-scroll [items]="items" [headerFn]="headerFn">
  <ion-item-divider color="primary" *virtualHeader="let header">
    {{ header }}
  </ion-item-divider>

  <ion-item *virtualItem="let item">
    {{ item }}
  </ion-item>
</ion-virtual-scroll>
```

```
export class VirtualScrollPage implements OnInit {
  items: string[] = [];

  ngOnInit() {
    for (let i = 1; i <= 10000; i++) {
      this.items.push(`Item ${i}`);
    }
  }

  headerFn(record, recordIndex, records) {
    if (recordIndex % 20 === 0) {
      return `From ${recordIndex + 1} to ${recordIndex + 20}`;
    }
    return null;
  }
}
```

It's recommended for performance reasons to provide an approximate height of each item when it exceeds the default 45px. This is done passing a number to the `[approxItemHeight]` directive. If you can calculate or return the exact height of each element, is better to use the `[itemHeight]` directive instead, where you pass the name of a method that calculates the exact height. This will result in the best possible performance.

## Gestures

Ionic (and Angular) supports multiple touch gestures using the [HammerJS](#) library. All we need to do is install it (`npm i hammerjs`) and import it in the `polyfill.ts` file. After that (only in Angular 9+), import HammerModule inside the main AppModule:

```
import { BrowserModule, HammerModule } from '@angular/platform-browser';
...

@NgModule({
```

```

imports: [
  ...
  HammerModule
],
...
})
export class AppModule {}

```

You can detect new events like **tap** (finger touch), **press** (finger pressed for a few seconds), **pan** (finger moved while pressed), **swipe** (finger moved while pressed from side to side), **rotate** (rotation with two fingers), and **pinch** (pinch with two fingers). On every event, the special variable \$event represents the event object (with additional information such as finger position, etc.).

You can see the list of event names supported by Angular [here](#), and more info in [this page](#).

## Skeleton

This component allows to create placeholder content (grey bars for text, grey rectangles for images, with animations) before the real content is loaded and ready to show. This allows for a better user experience with the app.

```

// Default skeleton text (grey bar):
<ion-skeleton-text animated></ion-skeleton-text>
<ion-skeleton-text animated style="width: 70%"></ion-skeleton-text>

// Skeleton image
<ion-avatar slot="start">
  <ion-skeleton-text animated></ion-skeleton-text>
</ion-avatar>

<ion-thumbnail slot="start">
  <ion-skeleton-text animated></ion-skeleton-text>
</ion-thumbnail>

```

<https://ionicframework.com/docs/api/skeleton-text>

# Modal windows

---

[Modal windows](#) are presented outside the main (root) navigation stack. These windows are created the same way as **Popovers**. This means we need to have a component that controls the modal window and its template. We can also pass parameters from the parent window to the modal window (opening) and also pass data the opposite way (closing).

Modal windows are closed by themselves calling **dismiss()** on their injected modal controller. This method accepts (optional) an object which the parent window will receive (this reception is done in the **onDidDismiss()** method). Also, we can send parameters from the parent window to the modal while creating it with **create()**.

```
export class ProfilePage {
  ...
  constructor(public modalCtrl: ModalController,
               public userService: UsersService,
               public toast: ToastController) {}

  async openChangePass() {
    const modal = await this.modalCtrl.create({
      component: ChangePassComponent,
      componentProps: { name: this.user.name }
    });

    await modal.present();

    const result = await modal.onDidDismiss();
    if (result.data.changed) {
      this.showToast(3000, 'Password updated successfully!');
    }
  }
}

export class ChangePassComponent {
  @Input() name: string;
  password: string;
  password2: string;

  constructor(public modalCtrl: ModalController,
               public userServ: UsersService, public toast: ToastController) {}

  changePass() {
    if (this.password !== this.password2) {
      this.showToast(3000, 'Passwords do not match!');
      return;
    }

    this.userServ.changePassword(this.password)
      .subscribe(
        ok => this.modalCtrl.dismiss({changed: true}),
        error => this.showToast(3000, error)
      );
  }

  cancel() {
    this.modalCtrl.dismiss({changed: false});
  }
}
```



# Custom events

---

**Events** is an Ionic's service that let us pass data between components, pages or services that are not connected (like a global EventEmitter). We can use this also to load a page and pass data to it immediately after.

```
export class NewProductPage {
  ...
  constructor(public navCtrl: NavController, public events: Events,
    public toast: ToastController, public productService: ProductService,
  ) {}

  crearTarea() {
    this.productService.createProduct(this.product)
      .subscribe(
        async (newProd) => {
          await this.navCtrl.navigateBack(['/products']);
          this.events.publish('product:created', newProd);
        },
        (error) => this.showToast(3000, error)
      );
  }
  ...
}

export class ProductsPage implements OnInit {
  ...
  constructor(public navCtrl: NavController
    public productService: ProductService, public events: Events) { }

  ngOnInit() {
    ...
    this.events.subscribe('product:created', (prodData) => {
      let prod = <IProduct>prodData[0];
      this.products.unshift(prod);
    });
  }
  ...
}
```