# UNIT 2

## Angular

## Part 5
## 3rd Party modules

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

# Index

# SweetAlert2 with Angular

There's a library called ngx-sweetalert2 that provides integration for SweetAlert in Angular. This integration offers advantages like lazy loading the library (it's loaded the first time you open a dialog). You can use it as a directive (on a button, for example) or as a component (<swal>).

**npm install sweetalert2 @sweetalert2/ngx-sweetalert2**

You must import the SweetAlert2Module in your AppModule, calling the method **forRoot()** which registers the service that will lazy load the library.

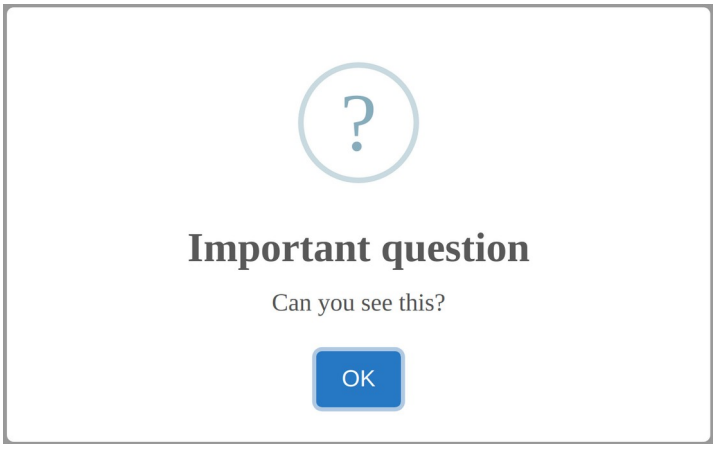
```
@NgModule({
  ...
  imports: [
    BrowserModule,
    SweetAlert2Module.forRoot() // Provides the service that loads the library
  ],
  ...
})
export class AppModule { }
```

In any other module you want to use Sweet alert, import the module without calling forRoot().

```
@NgModule({
  ...
  imports: [
    CommonModule,
    SweetAlert2Module
  ],
  ...
})
export class ProductsModule { }
```

Now, we can use the **swal** directive on any button (for example), so when the user clicks that button, the message will appear.

```
  <button [swal]="{ title: 'Important question', text: 'Can you see this?',
icon: 'question' }">
    Show SweetAlert question
  </button>
```



The options are more or less the same than when using the SweetAlert2 library in JavaScript. For example, we can show an input and get the value ($event) in the **confirm** ouput event

```
<button [swal]="{ title: 'Who are you?', input: 'text', icon: 'question' }"
  (confirm)="sayHello($event)" (cancel)="goBack()">
  Ask name
</button>
```


You can also use the <swal> component instead of a directive. It's more or less the same, but allows you to put custom content in the dialog and bind it to the component using the ***swalPortal** directive and the **SwalPortalTargets** service that allows to specify where in the dialog the custom content will show (closeButton, title, content, cancelButton, confirmButton, actions, and footer).

```
<swal title="Welcome" icon="success" #helloSwal>
  <ng-container *swalPortal="swalTargets.content">
    Hello {{name}}
  </ng-container>
</swal>

export class AppComponent {
  name = '';
  @ViewChild('helloSwal') private helloSwal!: SwalComponent;

  constructor(public readonly swalTargets: SwalPortalTargets) {}

  sayHello(name: string): void {
    this.name = name;
    this.helloSwal.fire();
  }
  ...
}
```

# Font-Awesome with Angular

Font Awesome also has Angular integration (you can always install the normal version and use CSS classes). This integration makes easier to use some advanced features like rotating, animations, layering, custom classes, etc.

You must specify which version to install. For example, version 0.8 is compatible with Angular 11. but not previous versions.

ng add @fortawesome/angular-fontawesome@"0.8.x"

After that, select which icons you want to use (if you don't pay, you can only use free icons).

```
? Choose Font Awesome icon packages you would like to use:
❯◉ Free Solid Icons
 ◉ Free Regular Icons
 ◉ Free Brands Icons
 ◯ Pro Solid Icons    [ Requires: https://fontawesome.com/pro ]
 ◯ Pro Regular Icons  [ Requires: https://fontawesome.com/pro ]
 ◯ Pro Light Icons    [ Requires: https://fontawesome.com/pro ]
 ◯ Pro Duotone Icons  [ Requires: https://fontawesome.com/pro ]
```

There are 2 ways of using the icons

- Explicit reference: importing icons inside the components where will be used, but also involves creating a variable/attribute for every icon.
- Icon library: Importing and registering the necessary icons in the AppComponent for the entire application.

Both options enable tree-shaking in Webpack (only the imported icons will be included in the final bundle). We'll choose the second option because it's usually cleaner (but also has its disadvantages).

First, we must import FontAwesomeModule in the AppModule and also in other modules which have components that display icons.

```
@NgModule({
  ...
  imports: [
    ...
    FontAwesomeModule
  ],
  ...
})
export class AppModule { }
```

Then, in the AppComponent, we'll import all the necessary icons and add it to the **FalconLibrary** service. It's important to notice that if we import the same icon name with solid and regular styles, for example, we must use an alias.

```
...
import { faUser as fasUser } from '@fortawesome/free-solid-svg-icons';
import { faUser as farUser } from '@fortawesome/free-regular-svg-icons';

...
export class AppComponent {
  ...
  constructor(
     ...
     library: FaIconLibrary
  ) {
    library.addIcons(fasUser);
    library.addIcons(farUser);
  }
  ...
}
```

We can now use this icon in any component.

```
User <fa-icon [icon]="['fas', 'user']"></fa-icon>
User2 <fa-icon [icon]="['far', 'user']"></fa-icon>
```

User User2

Example: Layering 2 icons (one inside the other):

```
<fa-layers [fixedWidth]="true">
  <fa-icon [icon]="['fas', 'square']" [styles]="{'color':'red'}"></fa-icon>
  <fa-icon [inverse]="true" [icon]="['fas', 'spinner']" transform="shrink-6"
[spin]="true"></fa-icon>
</fa-layers>
```

Example: Icon with a counter

```
<fa-layers [fixedWidth]="true">
  <fa-icon [icon]="['fas', 'envelope']"></fa-icon>
  <fa-layers-counter content="23"></fa-layers-counter>
</fa-layers>
```

# Bootstrap

Boostrap is a well known CSS/Javascript framework for creating responsive web pages. Boostrap's JavaScript code is written on top of JQuery, but there's no need to install JQuery as there exist Angular components and directives written for this framework (they also provide much better integration with your Angular code). This Angular module is called **ng-boostrap**: https://ng-bootstrap.github.io.

There are many services and directives that ng-bootstrap has. In this example, we are going to see how to show a **Modal window** asking the user is he/she wants to leave the current page. It will have 2 buttons (**yes,no**) and the response (true/false) is returned in a **promise**. We'll combine this with our **CanDeactivate guard** (remember that it accepts a boolean, a Promise<boolean> or an Observable<boolean>).

First of all, we'll install the plugin:

```
ng add @ng-bootstrap/ng-bootstrap
```

Then, we must include it in our App module, imports section. We also need to call the method forRoot() that will provide the necessary services:

```
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';
...

@NgModule({
  ...
  imports: [
    ...
    NgbModule,
  ],
  ...
})
export class AppModule { }
```

More information: https://ng-bootstrap.github.io/#/getting-started

Now, we are going to show a Modal Window in the **product/add** page for when the user tries to leave that page. Until now we've used a confirm dialog, but that's not very design friendly.

First, we need to create a component which will represent our modal content. In this case we're going to create a generic confirm dialog (**yes** and **no** buttons). As we want to be able to use it from everywhere, we'll add this component to a module called **modals**.

```
ng g module modals

ng g component modals/confirm-modal
```

This component will never be used from any HTML template (we will reference its class directly from a component), so we won't need to export it, and also won't have to import the ModalsModule anywhere.

In the component class, we'll just add 2 input values. The modal window **title** and the **body** text. We'll also inject the **NgbActiveModal** service that represents the current modal window object.

```
import { NgbActiveModal } from '@ng-bootstrap/ng-bootstrap';
...
@Component({
  selector: 'ap-confirm-modal',
  templateUrl: './confirm-modal.component.html',
  styleUrls: ['./confirm-modal.component.css']
})
export class ConfirmModalComponent implements OnInit {
  @Input() title: string;
  @Input() body: string;

  constructor(public activeModal: NgbActiveModal) {}

  ngOnInit() {}
}
```

And this is the Component's (modal) template. Notice that the **dismiss** method is like cancelling the modal window, and the **close** method is used for closing the window with a response (in this case true → yes, or false → no):

```
<div class="modal-header">
    <h4 class="modal-title">{{title}}</h4>
    <button type="button" class="close" aria-label="Close"
            (click)="activeModal.dismiss()">
        <span aria-hidden="true">&times;</span>
    </button>
</div>
<div class="modal-body">
    <p>{{body}}</p>
</div>
<div class="modal-footer">
    <button type="button" class="btn btn-success"
           (click)="activeModal.close(true)">Yes</button>
    <button type="button" class="btn btn-danger"
           (click)="activeModal.close(false)">No</button>
</div>
```

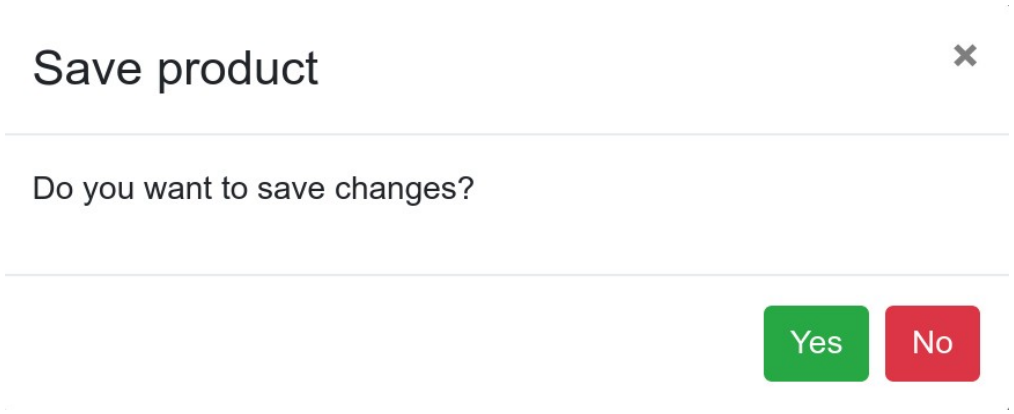
Now, we just need to import the shared module where we need to and use this modal component. In this case, from the **ProductComponent** (we must inject the **NgbModal** service), inside the **canDeactivate** method. Remember that this method is called from the **CanDeactivate guard**. In order to know which button has been pressed to close the modal, we need to watch the **result** property which is a Promise that resolves a value when **close** is called and rejects when **dismiss** is called.

If we want to work with observables, we can transform a Promise into an Observable using the **Observable.fromPromise** method.

```
export class ProductFormComponent implements OnInit, CanComponentDeactivate {
  @ViewChild('productForm') productForm: NgForm;
  ...

  constructor(
    ...
    private productsService: ProductsService,
    private modalService: NgbModal
  ) { }

  ...
```

```
  canDeactivate(): Observable<boolean> {
    console.log(this.productForm);
    if (this.productForm.dirty) { // Form changed and not saved
      const modalRef = this.modalService.open(ConfirmModalComponent);
      modalRef.componentInstance.title = 'Save product';
      modalRef.componentInstance.body = 'Do you want to save changes?';

      // Transform modalRef.result (Promise), into an Observable
      return from(modalRef.result).pipe(
        switchMap((yes: boolean) => { // returns another Observable
          if (yes) {
            return this.productsService.addProduct(this.newProduct).pipe(
              catchError(error => {
                // Show errors here
                return of(false);
              }),
              map(prod => true) // Changes saved successfully!
            );
          } else { // User said don't save changes!
            return of(true);
          }
        }),
        catchError(() => of(false))
      );
    }

    return of(true); // Form not modified
  }
}
```

**switchMap** is like **map**, but used to return an observable instead of a normal value. In this case we need it because we make an HTTP call if the user chooses to save changes.



# Angular material + flex layout

Another choice for a design framework is to use Angular Material. This is implementation of different components that follow the Google's Material Design specification, using Angular. You can visit its GitHub page here. Also, there's a sample application you can try here (github).

Also, keep in mind that Angular Material is a collection of components, directives and services. If you want to have a layout manager like Bootstrap has, you'll need to install Angular Flex Layout. Besides that, Material uses the Angular Animations module, so at the end, we'll need to install those 3 modules:

```
ng add @angular/material
ng add @angular/flex-layout
```

To use these installed modules, we'll need to import them in our own modules. Lets start by importing the layout and animations modules:

```
@NgModule({
  ...
  imports: [
    ...
    FlexLayoutModule,
    BrowserAnimationsModule
  ],
  ...
})
export class AppModule { }
```

Every component in Angular Material has its own module, so we'll need to import a module for every component we want to use. For example, if we want to use buttons, we'll import **MatButtonModule**, or if want to use checkboxes **MatCheckboxModule**. You can check all available components here.

A good idea could be to create a module (a shared module) to import and export all material modules you'll use in your application, and then, if you import that module somewhere else, you'll have access to all the components.

## Using icons

By default, Angular Material uses **Material icons** font to draw icons in your application. To show an icon, put the name of the icon between the **<mat-icon>** tag:

```
<mat-icon>home</mat-icon>
```

If you prefer to use another font like Font Awesome, install it (don't forget to include the "**../node_modules/font-awesome/css/font-awesome.css**" file in your **styles.css** or **angular-cli.json** file. Then, you can create an icon like this:

```
<mat-icon fontSet="fa" fontIcon="fa-home"></mat-icon>
```

Other options are specified here.

## Including a theme

Angular Material comes with some prebuilt themes. Creating a custom theme is beyond the scope of this course and unit, but you can do it. You just need to include one of those CSS files in your **angular.json** file. Those themes are:

```
"./node_modules/@angular/material/prebuilt-themes/deeppurple-amber.css"
"./node_modules/@angular/material/prebuilt-themes/indigo-pink.css"
"./node_modules/@angular/material/prebuilt-themes/pink-bluegrey.css"
"./node_modules/@angular/material/prebuilt-themes/purple-green.css"
```

## Creating a basic layout

A basic layout consists of a container (**fxLayout**) which defines the direction of its child elements (row, column) and how the will align with these directives:

- **fxLayout** → row | column | row-reverse | column-reverse
- **fxLayoutAlign** → <main-axis> <cross-axis>. Main axis: start |center | end | space-around | space-between. Cross axis: start | center | end | stretch.
- **FxLayoutGap** → % | px | vw | vh

And then, the child elements (**fxFlex**), which will take as much space as you let them (by default all of them take equal and proportional space.

- **fxFlex** → (empty) | px | % | vw | vh | <grow> <shrink> <basis>
- **fxFlexOrder** → number
- **fxFlexOffset** → % | px | vw | vh
- **fxFlexAlign** → start | baseline | center | end
- **fxFlexFill**

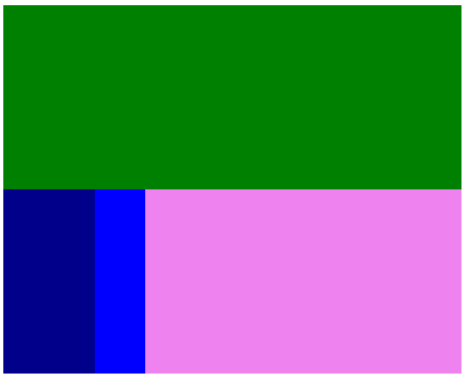
You can also create responsive layouts using aliases for media queries (fxFlex.alias) like: **xs** (extra small), **sm** (small), **md** (medium), **lg** (large), **xl** (extra large), gt-md (greater than medium), lt-lg (lower than large), etc.

```
<div fxLayout="row" fxLayoutWrap class="zero">
  <div fxFlex="33" fxFlex.lt-md="100" class="one"></div>
  <div fxFlex="33" fxFlex.xs="" fxLayout="column" fxLayout.xs="row" class="two">
    <div fxFlex="20" class="two_one"></div>
    <div fxFlex="40px" class="two_two"></div>
    <div fxFlex class="two_three"></div>
  </div>
  <div fxFlex fxHide.xs class="three"></div>
</div>
```

```
.zero {  height: 300px; }
.one { background-color: green; }
.two_one { background-color: darkblue; }
.two_two { background-color: blue; }
.two_three { background-color: violet; }
.three { background-color: red; }
```

md, lg and xl

sm



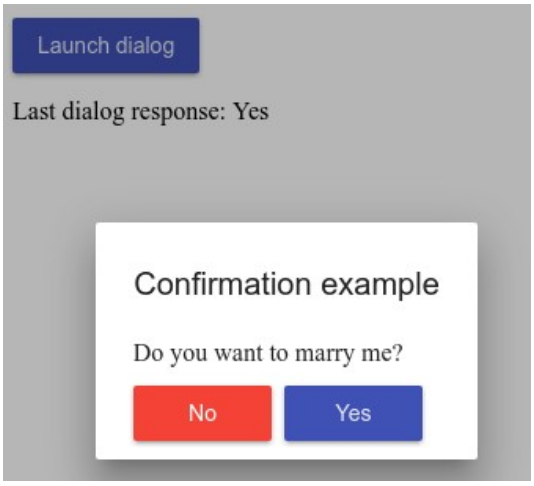
xs

## Displaying a modal dialog/window

Displaying a modal dialog is very similar to what we did with Bootstrap. You'll need to create an independent component with the content of the modal window you want to display and include it in your module. As this isn't a routed module or used in any HTML template, you need to include it in **entryComponents** to be compiled.

Also , we need to include at least 2 Material modules for our example (Button and Dialog modules):

```
@NgModule({
  declarations: [
    ...
    ModalConfirmComponent
  ],
  imports: [
    ...
    MatButtonModule,
    MatDialogModule
  ],
  entryComponents: [
    ModalConfirmComponent
  ],
  ...
})
export class AppModule { }
```




The service used to create and display modals is called **MatDialog**. It returns a reference to the dialog object (**MatDialogRef**), that allows, for example, to get the value returned using an observable:

```
export class AppComponent {
  response = '';

  constructor(private dialog: MatDialog) {}

  openDialog() {
    const dialogRef = this.dialog.open(ModalConfirmComponent, {
      data: {
        title: 'Confirmation example',
        body: 'Do you want to marry me?'
      }
    });

    dialogRef.afterClosed().subscribe(
      resp => this.response = resp ? 'Yes' : 'No'
    );
  }
}
```

And this is the template for the dialog's content:

```
<h2 mat-dialog-title>{{data.title}}</h2>
<mat-dialog-content>{{data.body}}</mat-dialog-content>
<mat-dialog-actions>
  <button mat-raised-button color="warn" [mat-dialog-close]="false">No</button>
  <button mat-raised-button color="primary" [mat-dialog-close]="true">Yes
  </button>
</mat-dialog-actions>
```

And the component:

```
export class ModalConfirmComponent implements OnInit {

  constructor(@Inject(MAT_DIALOG_DATA) public data: any,
              private dialogRef: MatDialogRef<ModalConfirmComponent>) { }

  ngOnInit() {}

}
```

Notice the way we are injecting the data received by the dialog in the constructor. This is the syntax used when the value injected is not an @Injectable class. Also, you can close the dialog from the **dialogRef** reference.

# Login / register with Google / Facebook

There are existing libraries that integrate your application with Login, Facebook or other providers authentication, like Angular Social Login. But we are going to create our own Angular code on top of the plain JavaScript libraries as an example.

First of all, we'll install the Typescript typings for the Google and Facebook API:

**npm i -D @types/gapi @types/gapi.auth2 @types/facebook-js-sdk**

In order for TypeScript to recognize these types, you'll need to add them in the types array (src/tsconfig.app.json):

```
"compilerOptions": {
  ...
  "types": ["gapi", "gapi.auth2", "facebook-js-sdk"],
},
```

## Login / Register with Google Plus

A typical way of login or register is using a 3rd party entity like Google or Facebook. They have an Oath2 authentication system that allows us to verify that the user data is authentic. We can use the data those platforms give us to create a new user in the server instead of using the normal register form.

Right now, there's no Angular plugin (at least not one that is well mantained and reliable) and we'll have to do some things manually. However, it shouldn't be too difficult.

First of all, we're going to create a **shared module** which will have a service to load the Google API and an attribute directive which will enable any element to become a Google login button. This module can be reused in any other application.

```
ng g module google-login
```

Now, lets go to the created folder and generate a service and a directive there:

```
ng g service google-login/services/load-google-api
```

```
ng g directive google-login/directives/google-login -m google-login
```

We want to get our Google **client_id** (needed for the authentication process) into the Angular injector (so it can be injected in our service for example). This **client_id** must be your Oath2 credentials created using the Google's developers console.

Angular allows to provide a value instead of a service (@Injectable), but we need an intermediary object (**InjectionToken**). We're going to create the injection token to inject the CLIENT_ID that Google requires. Just create it in a separate file like **google-login.config.ts**.

```
export const CLIENT_ID = new InjectionToken<string>('client_id');
```

When our module is imported in other modules, we need to have a method to get the Google client id value, and provide it as a global value using the InjectionToken from before. We can create a method called forRoot. This method will only be called from the AppModule and will inject the CLIENT_ID token in Angular dependency container using the providers array.

```
@NgModule({
  declarations: [GoogleLoginDirective],
  imports: [CommonModule],
  exports: [GoogleLoginDirective]
})
export class GoogleLoginModule {
  static forRoot(clientId: string): ModuleWithProviders<GoogleLoginModule> {
    return {
      ngModule: GoogleLoginModule,
      providers: [
        { provide: CLIENT_ID, useValue: clientId }
      ]
    };
  }
}
```

The service will be in charge of initializing the **Google API script**, but only when needed (lazy loading), so in pages where we don't place a login button it won't be loaded. Also, it has to make sure that it's only loaded once (it's loaded in the index.html document and we can't load it every time we go to the login page).

The script loading and the Google auth2 API initialization will be handled by observables, so we can get a response once everything is loaded.

```
import { Inject, Injectable, Optional } from '@angular/core';
import { fromEvent, Observable, ReplaySubject } from 'rxjs';
import { CLIENT_ID } from '../google-login.config';

@Injectable({
  providedIn: 'root'
})
export class LoadGoogleApiService {
  private initialized = false;
  // Similar to EventEmitter and replays last emitted values
  private loader$ = new ReplaySubject<gapi.auth2.GoogleAuth>(1);
  private clientId = '';

  constructor(@Optional() @Inject(CLIENT_ID) clientId: string) {
    if (!clientId) {
      throw new Error('GoogleLoginModule: You must call forRoot in your
AppModule to pass the CLIENT_ID');
    }

    this.clientId = clientId;
  }

  getAuthApi(): Observable<gapi.auth2.GoogleAuth> {
    if (!this.initialized) { // Initialization not started
      this.initialized = true;
      this.loadApi();
    }
    return this.loader$; // Return observable (emits auth API when loaded)
  }

  private loadApi(): void {
    const script = document.createElement('script');
```

```
    script.src = 'https://apis.google.com/js/api:client.js';
    script.setAttribute('async', '');
    script.setAttribute('defer', '');
    document.body.appendChild(script);

    fromEvent(script, 'load').subscribe(() => {
      gapi.load('auth2', () => {
        const gauth: gapi.auth2.GoogleAuth = gapi.auth2.init({
          client_id: this.clientId,
          cookie_policy: 'single_host_origin'
        });

        this.loader$.next(gauth);
        this.loader$.complete();
      });
    });
  }
}
```

Finally, this is the directive created for any button. It will get the auth2 object from the service and attach the click event to that button. It will emit 3 events:

```
@Directive({
  selector: '[appGoogleLogin]'
})
export class GoogleLoginDirective implements OnInit {
  @Output() loginOk = new EventEmitter<gapi.auth2.GoogleUser>();
  @Output() loginError = new EventEmitter<string>();
  @Output() loadingEnd = new EventEmitter<void>();

  constructor(
    private el: ElementRef,
    private loadService: LoadGoogleApiService
  ) { }

  ngOnInit(): void {
    this.loadService.getAuthApi().subscribe(
      auth2 => {
        auth2.attachClickHandler(
          this.el.nativeElement, {},
          (user: gapi.auth2.GoogleUser) => {
            this.loginOk.emit(user);
          },
          error => this.loginError.emit(error)
        );
        this.loadingEnd.emit();
      }
    );
  }
}
```

Now, in the AppModule, we import this module, calling forRoot, and passing the Google client id. In every other module we shouldn't call forRoot.

```
@NgModule({
  ...
  imports: [
    BrowserModule,
    GoogleLoginModule.forRoot('GOOGLE_ID.apps.googleusercontent.com')
    ...
  ],
  ...
})
export class AppModule { }

<button appGoogleLogin type="button" class="btn btn-danger"
```

```
(loginOk)="loggedGoogle($event)">
  <fa-icon [icon]="['fab', 'google']"></fa-icon>
  Login with Google
</button>


loggedGoogle(user: gapi.auth2.GoogleUser) {
    // Send this token to your server for register / login
    console.log(user.getAuthResponse().id_token);
    console.log(user.getBasicProfile().getName());
    console.log(user.getBasicProfile().getEmail());
    console.log(user.getBasicProfile().getImageUrl());
}
```

## Login / Register with Facebook

Like with Google, there's no good Angular plugin for Facebook login yet, so we are going to make our own login system. As we did with Google, we are going to load Facebook asynchronously. We'll create a module with a service and a directive just like we did before:

```
ng g module facebook-login
```

Now, lets go to the created folder and generate a service and a directive there:

```
ng g service facebook-login/services/load-fb-api
```

```
ng g directive facebook-login/directives/fb-login -m facebook-login
```

Now, lets create a file where we'll declare an InjectorToken for configuring the FaceBook API in the module folder, called **facebook-login.config.ts**.

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

export interface FBConfig {
  app_id: string;
  version: string;
}

export const FB_CONFIG = new InjectionToken<FBConfig>('fb_config');
```

In the module, we must create the **forRoot** method to provide the created service and the configuration data:

```
@NgModule({
  declarations: [FbLoginDirective],
  imports: [CommonModule],
  exports: [FbLoginDirective]
})
export class FacebookLoginModule {
  static forRoot(clientId: FBConfig): ModuleWithProviders<FacebookLoginModule> {
    return {
      ngModule: FacebookLoginModule,
      providers: [
        { provide: FB_CONFIG, useValue: clientId }
      ]
    };
  }
}
```

The service will have a method for loading the Fabebook API (aynchronous → Observable). Also, it will have methods to do login against Facebook and check if the user is already logged in.

```
import { Inject, Injectable, Optional } from '@angular/core';
import { Observable,  Observer, EMPTY, Subject, ReplaySubject } from 'rxjs';
import { catchError } from 'rxjs/operators';
import { FB_CONFIG, FBConfig } from '../facebook-login.config';

@Injectable({
  providedIn: 'root'
})
export class LoadFbApiService {
  private appId: string;
  private version: string;
  private initialized = false;
  private loader$ = new ReplaySubject<void>(1);

  constructor(@Optional() @Inject(FB_CONFIG) fbConfig: FBConfig) {
    if (!fbConfig) {
      throw new Error('FacebookLoginModule: You must call forRoot in your
AppModule to pass the APP_ID and Version');
    }

    this.appId = fbConfig.app_id;
    this.version = fbConfig.version;
  }

  loadApi(): Observable<void> {
    if (!this.initialized) {
      this.loadScript();
      this.initialized = true;
    }

    return this.loader$;
  }

  login(scopes: string): Observable<fb.StatusResponse> {
    return this.isLogged().pipe( // First, we'll see if it's already logged
      catchError(response => {
        return new Observable((observer: Observer<fb.StatusResponse>) => {
          FB.login((respLogin: fb.StatusResponse) => {
            if (respLogin.status === 'connected') {
              observer.next(respLogin);
            } else {
              observer.error(respLogin);
            }
            observer.complete();
          }, { scope : scopes });
        });
      })
    );
  }

  isLogged(): Observable<fb.StatusResponse> {
    return new Observable((observer: Observer<fb.StatusResponse>) => {
      FB.getLoginStatus(response => {
        if (response.status === 'connected') {
          observer.next(response);
        } else {
          observer.error(response);
        }
        observer.complete();
      });
    });
  }

  private loadScript(): void {
```

```
    if (this.initialized) { return; }

    (window as any).fbAsyncInit = () => {
      FB.init({
        appId: this.appId,
        xfbml: true,
        version: this.version
      });
      this.loader$.next();
      this.loader$.complete();
    };

    const script = document.createElement('script');
    script.id = 'facebook-jssdk';
    script.src = 'https://connect.facebook.net/es_ES/sdk.js';
    script.setAttribute('async', '');
    script.setAttribute('defer', '');
    document.body.appendChild(script);
  }
}
```

Finally, lets see the directive (very similar to the one we created for Google):

```
@Directive({
  selector: '[appFbLogin]'
})
export class FbLoginDirective {
  @Output() loginOk: EventEmitter<fb.StatusResponse> = new
EventEmitter<fb.StatusResponse>();
  @Output() loginError: EventEmitter<string> = new EventEmitter<string>();
  @Output() loadingEnd: EventEmitter<void> = new EventEmitter<void>();
  @Input() scopes!: string[];

  constructor(private el: ElementRef, private loadService: LoadFbApiService) {
    loadService.loadApi().subscribe(
      () => this.loadingEnd.emit()
    );
  }

  @HostListener ('click') onClick(): void {
    this.loadService.login(this.scopes.join(',')).subscribe(
      resp => this.loginOk.emit(resp),
      error => this.loginError.emit('Error with Facebook login!')
    );
  }
}
```

Now, we can import the new module in our AppModule. Don't forget to create the necessary credentials in https://developers.facebook.com/ (App id and API version).

```
@NgModule({
  imports: [
    ...
    FacebookLoginModule.forRoot({app_id: 'APP_ID', version: 'v9.0'})
  ],
   ...
})
export class AppModule { }

<button appFbLogin [scopes]="['email', 'public_profile']"
          class="btn btn-primary" (loginOk)="loggedFacebook($event)"
          (loginError)="showError($event)">
    <fa-icon [icon]="['fab', 'facebook']"></fa-icon>
    Facebook
</button>


export class AppComponent {
  ...

  loggedFacebook(resp: fb.StatusResponse) {
    // Send this to your server
    console.log(resp.authResponse.accessToken);
  }

  showError(error: any) {
    console.error(error);
  }
}
```

# Integration with Google Maps

The integration with Google Maps is really easy because there's a well mantained plugin you can find here: https://angular-maps.com/. To install it in you Angular project:

```
npm install @agm/core
```

Now (you must have a valid Google Maps API key first), we import this module in our App module:

```
import { AgmCoreModule } from '@agm/core';
...

@NgModule({
  ...
  imports: [
    ...
    AgmCoreModule.forRoot({
      apiKey: 'YOUR_KEY'
    })
  ],
  ...
})
export class AppModule { }
```

Now, in the component we want to add a map (with a marker), we just add in the HTML template:

```
<agm-map [latitude]="lat" [longitude]="lng" [zoom]="zoom">
  <agm-marker [latitude]="lat" [longitude]="lng"></agm-marker>
</agm-map>
```

We need to define values in our component for lat (latitude) and lng (longitude).

```
@Component({
  ...
})
export class AppComponent {
  ...
  lat = 38.4039418;
  lng = -0.5288701;
  zoom = 17;
}
```

Also in CSS we have to specify the map's dimensions:

```
agm-map {
  height: 300px;
}
```

# Google Places Autocomplete

The **angular-maps** module already loads the Google Maps library in the browser. We can include other libraries to be loaded easily by passing an array to the forRoot method. In this case, we'll need to also load the Google Places library:

```
@NgModule({
  ...
  imports: [
    ...
    AgmCoreModule.forRoot({
      apiKey: 'API_KEY',
      libraries: ['places']
    })
  ],
  ...
})
export class AppModule { }
```

First of all, import the Google Maps Typescript types definitions:

```
npm i -D @types/googlemaps
```

Then, include those types in the **src/tsconfig.app.json** file:

```
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    ...
    "types": ["googlemaps"]
  },
  ...
}
```

Now, for the Autocomplete functionality, we are going to create an attribute directive (we could create a component, but we don't want to manage the input HTML, other attributes and styles, etc.). This directive will load the Autocomplete functionality and generate events with search results.

```
ng g directive gmaps-autocomplete
```

This directive will wait for the Google Maps library to be loaded (using a service from Angular Maps) and then attach an Autocomplete to the element. When the Autocomplete emits the '**place_changed**' event, the directive will emit an event with the new latitude and longitude.

When we don't use observables, promises, timeouts, or Angular processed events. Other event types or asynchronous operations like the '**place_changed**' Google Maps event run outside of what is called **Angular Zone**. This means that Angular won't control what happens inside and won't make updates to the view. The solution is to wrap everything in an Observable like we did with Google and Facebook login, or in this case, is simpler to inject the NgZone service and emit the new LatLng object inside the Angular Zone with **NgZone.run** method. Now, Angular will notice when we call EventEmitter.emit and update the view if anything changes.

```
@Directive({
  selector: '[amGmapsAutocomplete]'
})
export class GmapsAutocompleteDirective {
  @Output() placeChanged: EventEmitter<google.maps.LatLng> =
                                          new EventEmitter<google.maps.LatLng>();

  constructor(private el: ElementRef, private mapsLoader: MapsAPILoader,
              private ngZone: NgZone) {
    this.mapsLoader.load().then(() => { // Wait until the GMaps is loaded
      const autocomplete =
                  new google.maps.places.Autocomplete(el.nativeElement);
      autocomplete.addListener('place_changed', () => {
        const place = autocomplete.getPlace();
        if (place.geometry && place.geometry.location) {
          this.ngZone.run(() => { // Run this inside Angular Zone
              this.placeChanged.emit(place.geometry.location)
          });
        }
      });
    });
  }
}
```

And that's all, we just need to use the directive and watch for changes.

```
<div>
  <input type="text" amGmapsAutocomplete placeChanged)="changePosition($event)">
</div>
<agm-map [latitude]="lat" [longitude]="lng" [zoom]="zoom">
  <agm-marker [latitude]="lat" [longitude]="lng"></agm-marker>
</agm-map>


...
export class AppComponent {
  lat = 38.4039418;
  lng = -0.5288701;
  zoom = 17;
  ...

  changePosition(pos: google.maps.LatLng) {
    console.log(pos);
    this.lat = pos.lat();
    this.lng = pos.lng();
  }
}
```

# Integration with Mapbox

If, instead of Google Maps, you want to integrate Mapbox with Angular, you can use the **ngx-mapbox-gl** module. To install this module, along with Typescript definitions for Mapbox, execute this:

```
npm i ngx-mapbox-gl mapbox-gl
npm i -D  @types/mapbox-gl
```

Include the necessary CSS in your **angular.json** file:

```
"styles": [
    "./node_modules/mapbox-gl/dist/mapbox-gl.css",
    ...
]
```

Import the module in your AppComponent, and configure the accessToken you have in your Mapbox account (https://www.mapbox.com/account/):

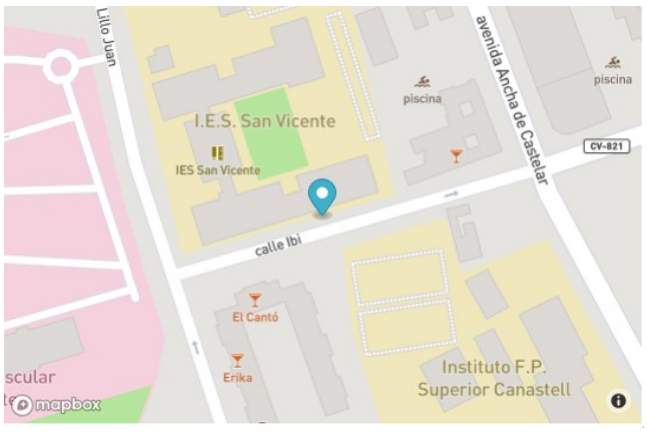
```
...
import { NgxMapboxGLModule } from 'ngx-mapbox-gl';

@NgModule({
  ...
  imports: [
    BrowserModule,
    NgxMapboxGLModule.withConfig({
      accessToken: 'TOKEN'
    })
  ],
  ...
})
export class AppModule { }
```

Import also in other modules if you need to use maps there, but the configuration is only necessary once, in the AppModule. Now, we'll create a map component with a marker inside.

```
<mgl-map [style]="'mapbox://styles/mapbox/streets-v11'"
  [zoom]="[9]" [center]="[-74.50, 40]"></mgl-map>
```

We need to define values in our component for lat (latitude) and lng (longitude).

```
export class AppComponent {
  ...
  lat = 38.4039418;
  lng = -0.5288701;
  zoom = 17;
}
```




Also in CSS we have to specify the map's dimensions:

```
mgl-map {
  height: 400px;
  width: 600px;
}
```

# Mapbox Gecoding (Places)

Integrating Mapbox geocoding is very easy, as the search control is already included in ngx-mapbox-gl. The first we'll do is include the geocoder CSS in our angular.json file:

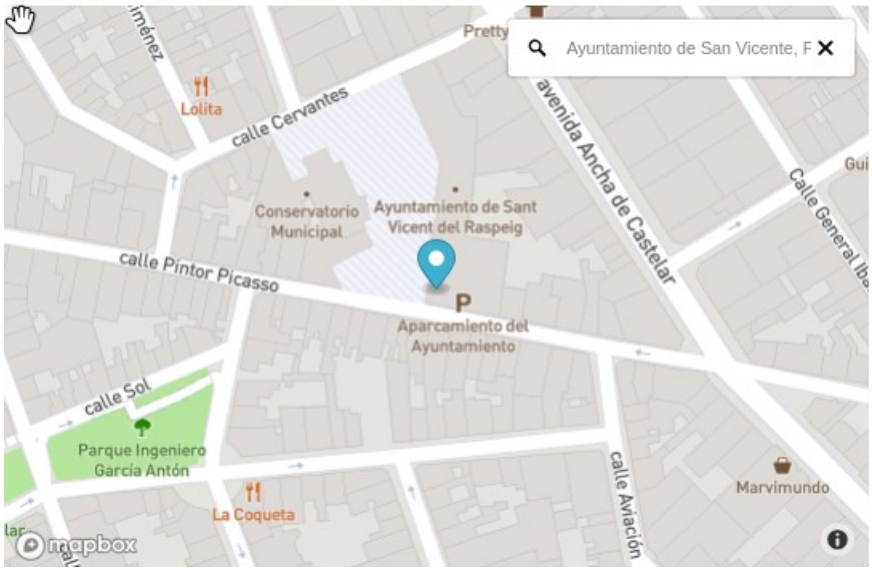
```
"styles": [
    "./node_modules/mapbox-gl/dist/mapbox-gl.css",
    "./node_modules/@mapbox/mapbox-gl-geocoder/lib/mapbox-gl-geocoder.css",
    ...
]
```

Then, just add a geocoder (**mglGeocoder**) control in any map you want, and listen for the **(result)** event which will have the information of the selected address:

```
<mgl-map [style]="'mapbox://styles/mapbox/streets-v11'" [zoom]="zoom"
[center]="[lng, lat]">
  <mgl-marker [lngLat]="[lng, lat]"></mgl-marker>
  <mgl-control mglGeocoder (result)="changePosition($event.result)">
  </mgl-control>
</mgl-map>
```

This is how you get and update the information in your component:

```
  changePosition(result: Result) {
    this.lat = result.geometry.coordinates[1];
    this.lng = result.geometry.coordinates[0];
    console.log('New address: ' + result.place_name);
  }
```



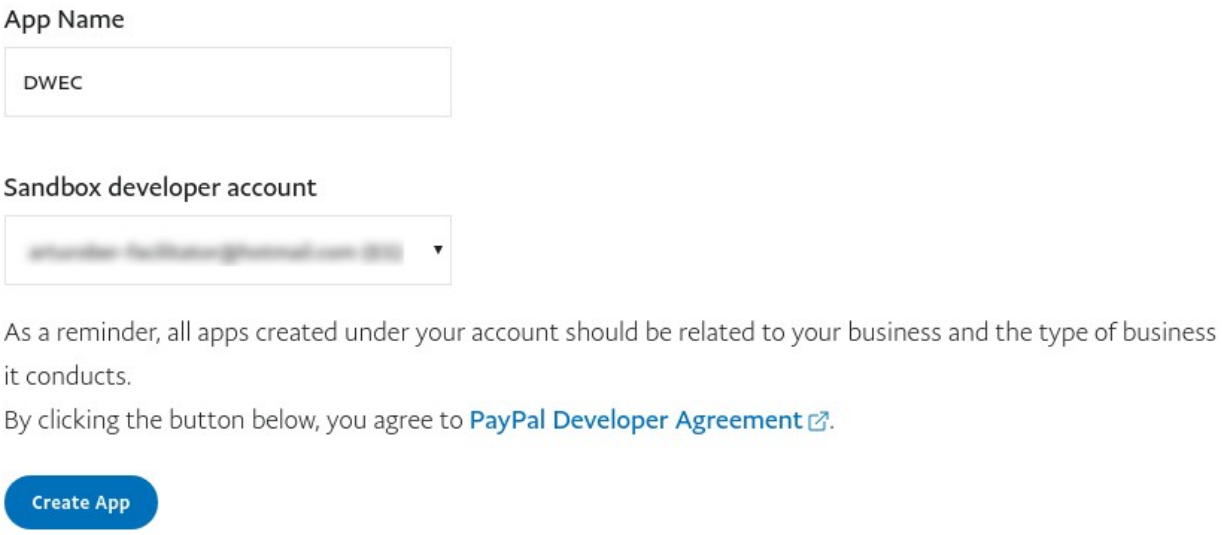
# Paypal

Integration with Paypal will be relatively similar to what we did with Google and Facebook login buttons. Instead of a directive we'll create a component (Paypal will render the button for us inside it), and a service to load the library only when it's required (when a Paypal button appears in the app) and not before.

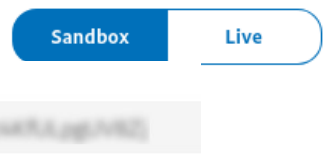
```
ng g module paypal-button

ng g component paypal-button/paypal-button

ng g service paypal-button/services/paypal-load
```

First of all, we need to create some access keys for Paypal. Generate a Sandbox access token (for testing purposes) from this page (you must be logged in).


Now, we must create the necessary keys from this page:




Copy the generated client id from the Sandbox tab. We'll need it for our Angular application:




Before implementing our code, we must set a password to the sandbox accounts used for testing. Do it from https://developer.paypal.com/developer/accounts/

Finally, we can implement our code.

**#paypal-button.config.ts**

```
export interface PaypalConfig {
  sandbox: string;
  production: string;
  environment: string;
}

export const PAYPAL_CONFIG = new InjectionToken<PaypalConfig>('paypal_config');
```

**#paypal-button.module.ts**

```
@NgModule({
  declarations: [PaypalButtonComponent],
  imports: [CommonModule],
  exports: [PaypalButtonComponent],
})
export class PaypalButtonModule {
  static forRoot(paypalConfig: PaypalConfig):
ModuleWithProviders<PaypalButtonModule> {
    return {
      ngModule: PaypalButtonModule,
      providers: [
        PaypalLoadService,
        { provide: PAYPAL_CONFIG, useValue: paypalConfig }
      ]
    };
  }
}
```

**#paypal-load.service.ts**

```
import { Injectable } from '@angular/core';
import { fromEvent, Observable, ReplaySubject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class PaypalLoadService {
  initialized = false;
  loading$ = new ReplaySubject<void>(1);

  constructor() { }

  loadPaypal(): Observable<void> {
    if (!this.initialized) {
      this.initialized = true;

      const script = document.createElement('script');
      script.src = 'https://www.paypalobjects.com/api/checkout.js';
      script.setAttribute('async', '');
      script.setAttribute('defer', '');
      script.setAttribute('data-version-4', '');
      document.body.appendChild(script);

      fromEvent(script, 'load').subscribe(
        () => this.loading$.next(),
        (error: any) => this.loading$.error(error),
        () => this.loading$.complete()
      );
    }

    return this.loading$;
  }
}
```

#paypal-button.component.ts

```
declare var paypal: any; // @type/paypal-checkout-components doesn't work well

@Component({
  selector: 'app-paypal-button',
  template: '',
  styleUrls: []
})
export class PaypalButtonComponent implements OnInit {
  @Input() amount!: number;
  // true (payment completed), false (payment cancelled)
  @Output() paymentCompleted = new EventEmitter<boolean>();

  constructor(
    @Inject(PAYPAL_CONFIG) private config: PaypalConfig,
    private paypalService: PaypalLoadService,
    private elementRef: ElementRef
  ) {}

  ngOnInit(): void {
    this.paypalService.loadPaypal().subscribe(
      () => this.render()
    );
  }

  private render(): void {
    paypal.Button.render(
      {
        env: this.config.environment,
        client: {
          sandbox: this.config.sandbox, // development (fake payments)
          production: this.config.production // In production (real payments)
        },
        payment: (data: any, actions: any) => {
          // Payment details
          return actions.payment.create({
            transactions: [
              {
                amount: {
                  total: this.amount,
                  currency: 'EUR'
                }
              }
            ]
          });
        },
        commit: true, // Display a "Pay Now" button rather than "Continue"
        onAuthorize: (data: any, actions: any) => {
          // Payment completed
          return actions.payment.execute()
          .then(() => {
            this.paymentCompleted.emit(true);
          })
          .catch(() => this.paymentCompleted.emit(false));
        },
        onCancel: (data: any) => {
          // Payment cancelled
          this.paymentCompleted.emit(false);
        }
      },
      '#' + this.elementRef.nativeElement.id
    );
  }
}
```

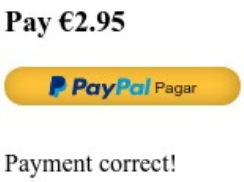
#app.module.ts

```
@NgModule({
  declarations: [AppComponent],
```

```
  imports: [
    BrowserModule,
    PaypalButtonModule.forRoot({
      sandbox: 'YOUR SANDBOX CLIENT_ID',
      production: '',
      environment: 'sandbox'
    }), ...
  ],
  providers: [],  bootstrap: [AppComponent]
})
export class AppModule {}
```

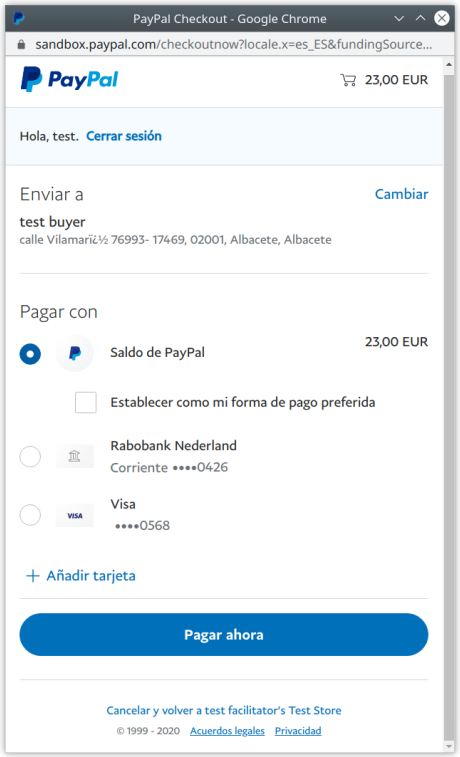
#app.component.html

```
<h3>Pay {{amount | currency:'EUR':'symbol'}}</h3>
<p>
  <app-paypal-button  id="payPalButton" [amount]="amount"
(paymentCompleted)="getPayment($event)"></app-paypal-button>
</p>
<p>
  {{payedMessage}}
</p>
```




#app.component.ts

```
export class AppComponent {
  title = 'ap';
  amount = 2.95;
  payedMessage = '';

  getPayment(ok: boolean) {
    this.payedMessage = ok ? 'Payment correct!' : 'Payment cancelled';
  }
}
```



# Socket.IO Angular integration

Using Socket.IO with Angular is very simple. We just have to create an Angular service that listens to Socket.IO events and emits them to other components/services using an EventEmitter (for example).

First of all, we'll create a simple Socket.IO server using NodeJS and Express that will just broadcast every message it receives. Don't forget to install dependencies:

**npm i express socket.io**

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server, { cors: { origin: '*' } });

server.listen(5000);

io.on('connection', socket => {
  socket.on('new-message', data => {
    socket.emit('message', data); // Boradcast messages
  });
});
```

Now, we are going to implement an Angular Service that will connect to the server and send / receive events. Install also the Socket.IO client.

```
npm i socket.io-client
```

```
# interfaces/message.ts
export interface Message {
  user: string;
  text: string;
  mine?: boolean;
}

# services/socket-io.service.ts
import { Injectable } from '@angular/core';
import { Observable, ReplaySubject } from 'rxjs';
import { io, Socket } from 'socket.io-client';
import { Message } from '../interfaces/message';

@Injectable({
  providedIn: 'root'
})
export class SocketIoService {
  private socket!: Socket;
  private connecting = false;
  public connection$ = new ReplaySubject<boolean>(1);
  public messages$ = new ReplaySubject<Message>(5);

  constructor() { }

  connect(port: number): Observable<boolean> {
    if (!this.connecting) {
      this.connecting = true;
      this.socket = io(`http://localhost:${port}`);

      this.socket.on('connect', () => {
        this.connection$.next(true);
```

```
      });

      this.socket.on('disconnect', () => {
        this.connection$.next(false);
      });

      this.socket.on('message', (msg: Message) => {
        this.messages$.next(msg);
      });
    }

    return this.connection$;
  }

  sendMessage(msg: Message): void {
    if (this.socket && this.socket.connected) {
      this.socket.emit('new-message', msg);
    }
  }

  disconnect(): void {
    if (this.socket && this.socket.connected) {
      this.socket.close();
    }
  }
}
```

**# chat/chat.component.html**

```
<div class="chat-container">
  <p *ngFor="let m of messages" [ngClass]="{'mine': m.mine}">
    <strong>{{m.mine ? 'me' : m.user}}:</strong> {{m.text}}
  </p>
</div>
<p>
  <input type="text" [(ngModel)]="newMsg">
  <button (click)="send()" [disabled]="!connected">Send</button>
  <button (click)="connect()" [disabled]="connected">Connect</button>
  <button (click)="disconnect()" [disabled]="!connected">Disconnect</button>
</p>
<p>Status: {{connected ? 'Connected as ' + user : 'Disconnected'}}</p>
```

**# chat/chat.component.css**

```
.chat-container {
  height: 400px;
  max-width: 300px;
  padding: 10px 20px;
}

p.mine {
  text-align: right;
  color: darkblue;
}
```

**# chat/chat.component.ts**

```
iimport { Component, Input, OnInit } from '@angular/core';
import { filter } from 'rxjs/operators';
import { Message } from '../interfaces/message';
import { SocketIoService } from '../services/socket-io.service';

@Component({
  selector: 'app-chat',
  templateUrl: './chat.component.html',
  styleUrls: ['./chat.component.css']
})
export class ChatComponent implements OnInit {
  @Input() user!: string;
  messages: Message[] = [];
```

```
  newMsg = '';
  connected = false;

  constructor(private ioService: SocketIoService) { }

  ngOnInit(): void {
    this.ioService.connection$.subscribe(
      connected => this.connected = connected
    );
    this.ioService.messages$.pipe(
      filter(m => m.user !== this.user)
    ).subscribe(m => {
        m.mine = false;
        this.messages.push(m);
    });
  }

  connect(): void {
    this.ioService.connect(5000);
  }

  disconnect(): void {
    this.ioService.disconnect();
  }

  send(): void {
    const msg: Message = {user: this.user, text: this.newMsg};
    this.ioService.sendMessage(msg);
    msg.mine = true;
    this.messages.push(msg); // Add the message directly to the chat
    this.newMsg = '';
  }

}
```

# app.component.html

```
<app-chat user="peter"></app-chat>
<app-chat user="john"></app-chat>
```

# Angular Charts (ngx-charts)

ngx-charts is an Angular library for displaying dynamic charts based on the popular d3js library. To install this dependency (along d3js which is required, and Angular Animations) run:

```
npm i @swimlane/ngx-charts

ng add @angular/cdk
```

You can see some chart examples here. Let's create a simple vertical bar chart which receives data dynamically and updates the chart.

First of all, we'll import the Animations and Charts modules in our module:

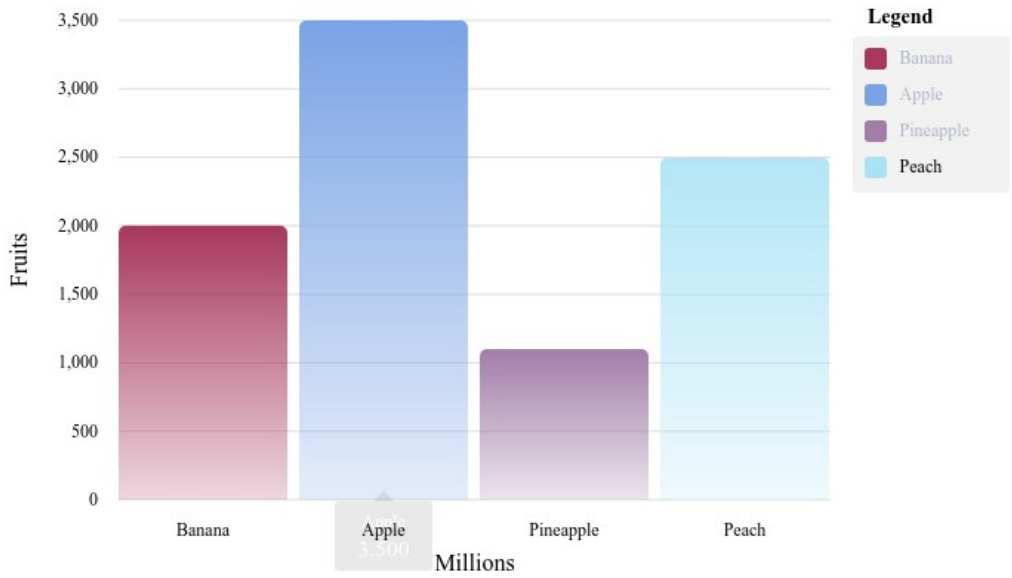
```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { NgxChartsModule } from '@swimlane/ngx-charts';
...

@NgModule({
  ...
  imports: [
    BrowserModule,
    FormsModule,
    BrowserAnimationsModule,
    NgxChartsModule
  ],
  ...
})
export class AppModule { }
```

We should place the corresponding component in our HTML (in this case for the vertical bar char, the component is **ngx-charts-bar-vertical**) with the corresponding options (some of them stored in variables). The parent element that contains the chart must be at least the same height and width as the chart.

```
<div style="height: 400px;">
  <ngx-charts-bar-vertical
    [view]="[600,400]" [results]="data" [gradient]="true"
    [xAxis]="true" [yAxis]="true" [legend]="true" [legendTitle]="title"
    [showXAxisLabel]="true" [showYAxisLabel]="true"
    xAxisLabel="Millions" yAxisLabel="Fruits" (select)="onSelect($event)">
  </ngx-charts-bar-vertical>
</div>

<h4>Add Fruit:</h4>
<p>Type: <input type="text" [(ngModel)]="newFruit.name"></p>
<p>Quantity: <input type="number" [(ngModel)]="newFruit.value"></p>
<p><button (click)="addFruit()">Add</button></p>
```

And this is the chart's parent component. Keep in mind that the chart doesn't update when we add or remove a fruit from the list, but when the entire list changes, so for every change we want to make, we must generate a new list and replace the old one. This is because the chart component is configured to use a different change detection strategy this way and it's an efficient way to update data for Angular.

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

@Component({
  selector: 'ac-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Fruits exports';

  data: {name: string, value: number}[] = [{
    name: 'Banana',
    value: 2000
  }, {
    name: 'Apple',
    value: 3500
  }];

  newFruit: {name: string, value: number} = {
    name: '',
    value: 0
  };

  onSelect(result) {
    console.log(result);
  }

  addFruit() {
    // We can't do a push, we need to replace the array
    this.data = [...this.data, this.newFruit];
    this.newFruit = { name: '', value: 0};
  }
}
```




**Add Fruit:**

Type:

Quantity: 0

Add