

**valtech\_**

# Labs Angular

# Sommaire

---

Nous verrons au cours des travaux pratiques les points suivants :

- [TP 1 - Installation](#)
- [TP 2 - Le composant Chronomètre](#)
- [TP 3 - Créer une liste d'utilisateurs en ligne](#)
- [TP 4 - Formulaires et routes](#)
- [TP 5 - Intégration des webservices](#)
- [TP 6 - Programmation reactive et websocket](#)

## Objectifs

---

Le labs a pour objectif de vous présenter les différentes fonctionnalités d'Angular. Au programme :

- Mise en place de l'environnement avec `@angular/cli`,
- Création d'un composant chronomètre et d'un Pipe,
- Utilisation des services Angular pour communiquer avec le backend,
- Création d'un formulaire de saisie,
- Mise en place du routage.

## TP 1

---

Installation de l'environnement

## Prérequis

---

Vérifier que vous avez les éléments suivants d'installé sur votre poste :

- Node v6 ou plus avec la commande `npm -v`,
- Git, nous l'utiliserons pour récupérer le projet initial,
- Webstorm (ou un autre IDE)

## Installation

---

### Initialisation du projet

Nous allons utiliser l'outil `@angular/cli` pour générer le projet. Placez-vous dans votre workspace et lancez les commandes suivantes :

```
npm install -g @angular/cli
ng new tp-angular
cd tp-angular
```

Une fois le projet créé vous pouvez l'ouvrir dans votre IDE.

Pour lancer le serveur de développement :

```
ng serve
```

## Installation d'Angular Material

Nous allons utiliser Angular material pour créer les interfaces de nos applications.

Lancer la commande suivante:

```
npm install --save @angular/material
npm install --save @angular/animations
```

Puis rajouter dans le fichier `app.module.ts` la dépendance du module comme suivant :

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { MaterialModule } from '@angular/material';
// other imports
@NgModule({
  ...
  imports: [
    BrowserAnimationsModule,
    MaterialModule
  ],
  ...
})
export class AppModule { }
```

En complément vous pouvez rajouter la feuille de style suivante dans la page index si vous voulez utiliser les icônes de la librairie Material :

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
```

Enfin dans le `styles.css`, ajouter le thème :

```
@import '~@angular/material/prebuilt-themes/indigo-pink.css';

body {
  padding: 0;
  margin:0;
  font-family: Roboto,"Helvetica Neue",sans-serif;
}
```

## Générer des Composants, Directives, Pipe, etc...

Vous pouvez utiliser la commande `ng generate` (ou `ng g`) pour générer des composants Angular :

```
ng generate component my-new-component
ng g component my-new-component # using the alias

# components support relative path generation
# if in the directory src/app/feature/ and you run
ng g component new-cmp
# your component will be generated in src/app/feature/new-cmp
# but if you were to run
ng g component ../newer-cmp
# your component will be generated in src/app/newer-cmp
```

Voici la liste des commandes possibles pour générer une fonctionnalité d'Angular:

Génération	Usage
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Guard	<code>ng g guard my-new-guard</code>
Interface	<code>ng g interface my-new-interface</code>
Enum	<code>ng g enum my-new-enum</code>
Module	<code>ng g module my-module</code>

# TP 2

## Le composant Chronomètre

Nous allons dans ce TP créer un composant Chronomètre. Nous verrons ainsi comment créer un composant ainsi que la création d'un Pipe pour formater l'affiche du composant.

## Notre premier composant

Dans le dossier `src/app`, créer un dossier `chrono`. Ce dossier contiendra les fichiers `html`, `css` et `ts`.

Voici la structure cible du projet :

```
src/app
├─ app.component.css
├─ app.component.html
├─ app.component.spec.ts
├─ app.component.ts
├─ app.module.ts
└─ chrono
   └─ chrono.component.css
      └─ chrono.component.html
         └─ chrono.component.ts
```

Vous devez donc créer les fichiers nécessaires au composant Chrono !

Ensuite éditez le fichier `chrono.component.ts` et collez le contenu suivant :

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

@Component({
  selector: 'chrono',
  templateUrl: './chrono.component.html',
  styleUrls: ['./chrono.component.css']
})
export class ChronoComponent implements OnInit {

  constructor() {

  }

  ngOnInit() {

  }

}
```

Puis ajoutez le composant dans la liste des composants de `AppModule` comme suivant :

```
@NgModule({
  declarations: [
    AppComponent,
    ChronoComponent // <- Ici
  ],
  ...
})
export class AppModule { }
```

Vous devez systématiquement déclarer votre composant en tant que dépendance du module pour l'utiliser !

## Exercice 1 - Template

Commencez par créer le template HTML nécessaire à l'affichage des données du chronomètre telles que:

- Les millisecondes,
- Les secondes,
- Les minutes.

Et n'oubliez pas le bouton start/stop du chrono !

Voici un exemple de ce qui est attendu :



Vous pouvez utiliser `@angular/material` pour créer un bouton et votre composant comme suivant :

```
<div>
  <span class="times">0</span>
  <button md-raised-button>Start</button>
</div>
```

Pour le css :

```
:host {
  text-align: center;
  display: block;
  width: 200px;
  height: 200px;
  margin: auto;
  background: #222;
  padding: 30px;
  border-radius: 100%;
  position: relative;
```

```

    z-index: 0;
  }
  :host>div {
    position: relative;
    z-index: 3;
  }
  :host:after {
    content: ' ';
    position: absolute;
    width: 206px;
    height: 206px;
    border: 5px solid #414141;
    border-radius: 100%;
    top: -8px;
    left: -8px;
    padding: 30px;
    z-index: 0;
  }
  :host .times {
    display: block;
    font-size: 50px;
    color: white;
    margin-bottom: 50px;
    padding-top: 30px;
  }

  :hover button {
  }

```

Note : le selecteur `:host` représente l'élément encapsulant votre composant Angular.

## Exercice 2 - Création de l'action click

À partir du cours, essayez de créer une action au click du bouton.

## Exercice 3

Maintenant que vous avez créé l'action au click du bouton, implémentez les méthodes `startTimer()` et `stopTimer()`.

## Notre premier Pipe

En l'état l'affichage des minutes, secondes et millisecondes n'est pas correct (cf. capture chrono).

Nous allons donc créer un Pipe, équivalent des filtres Angular 1, pour formater les données.

```

src/app
├─ app.component.css
├─ app.component.html
├─ app.component.spec.ts

```



```
├── app.component.ts
├── app.module.ts
├── chrono
│   ├── chrono.component.css
│   ├── chrono.component.html
│   ├── chrono.component.spec.ts
│   └── chrono.component.ts
└── dec-to-str.pipe.ts
```

Commencez par créer le fichier `dec-to-str.pipe.ts`. Créez la classe `DecToStrPipe` comme présenté dans le cours.

## Exercice

Le filtre doit transformer/formatter un entier vers un string sur deux chiffres !

Correction du TP : [tp2-solution](#)

## TP 3

---

Créer une liste d'utilisateurs en ligne

Dans ce TP, nous allons créer un tableau contenant la liste des utilisateurs avec leur statut en ligne.

Pour rappel, la structure initiale du projet est la suivante :

```
src/app
├── app.component.css
├── app.component.html
├── app.component.spec.ts
├── app.component.ts
└── app.module.ts
```

## Notre 1er Service

---

Dans le dossier `src/app`, créer votre 1er service sous le nom `UserService`.

Pour aller plus vite, vous pouvez utiliser l'outil `@angular/cli` pour créer votre service.

```
ng g service user
```

Genèrera les fichiers suivants :

---

```
src/app
├─ user.service.spec.ts
└─ user.service.ts
```

Ce service devra retourner la liste de tous les utilisateurs :

```
[
  {"id": 1, "email": "john.doe@gmail.com", "status": "online"},
  {"id": 2, "email": "jane.doe@gmail.com", "status": "online"},
  {"id": 3, "email": "jean.dupond@gmail.com", "status": "busy"},
  {"id": 4, "email": "jean.dupont@gmail.com", "status": "offline"},
  {"id": 5, "email": "jeanne.dupond@gmail.com", "status": "offline"},
  {"id": 6, "email": "joe.doe@gmail.com", "status": "online"}
]
```

## Le composant tableau des utilisateurs

Dans le dossier `src/app`, créer un dossier `user-table` qui contiendra tout le code source du composant `UserTableComponent`.

Ce composant `UserTableComponent` affiche la liste des utilisateurs avec leur statut dans tableau HTML.

Pour aller plus vite, vous pouvez utiliser l'outil `@angular/cli` pour créer votre composant.

```
ng g component user-table
```

Ci-dessous le code source ajouté au projet :

```
src/app
├─ user-table
│   ├── user-table.component.css
│   ├── user-table.component.html
│   ├── user-table.component.spec.ts
│   └─ user-table.component.ts
```

Ajouter les feuilles de style à votre composant pour que :

- Les lignes pairs soient en fonds gris clair,
- Le texte `online` soit en vert,
- Le texte `offline` soit en gris clair,
- Le texte `busy` soit en rouge,
- L'entête du tableau soit en fonds bleu clair, avec le texte en blanc.

Le tableau possède une checkbox (md-checkbox) permettant d'afficher soit les utilisateurs en ligne, soit tous les utilisateurs.

## Composant enfant

---

Créer un composant `UserTableRowComponent` représentant une ligne du tableau.

Pour aller plus vite, vous pouvez utiliser l'outil `@angular/cli` pour créer votre composant.

```
ng g component user-table-row
```

Modifier votre code pour intégrer ce composant dans le composant `UserTableComponent`.

Ci-dessous le code source ajouté au projet :

```
src/app
├── user-table-row
│   ├── user-table-row.component.css
│   ├── user-table-row.component.html
│   ├── user-table-row.component.spec.ts
│   └── user-table-row.component.ts
```

Le `UserTableRowComponent` devra gérer l'afficher des données de l'utilisateur ainsi que l'action de s'afficher ou se masquer en fonction du statut d'affichage demandé par le composant parent.







## Communication entres composants

---

Notre tableau affiche désormais les données. Nous allons maintenant utiliser l'annotation `@Output` pour émettre un événement lorsque l'utilisateur clique sur le bouton correspondant à la ligne de l'utilisateur. Cet événement sera remonté vers le composant `UserTableComponent` qui remontera cet événement vers le composant `AppComponent`.

Une fois l'information de l'utilisateur remonté dans `AppComponent`, vous devrez afficher les informations de l'utilisateur dans une boîte de dialogue (soit via une alert ou en utilisant la librairie [Material design](#)).

☐ Online only

ID	Email	Status	
1	john.doe@gmail.com	online	
2	jane.doe@gmail.com	online	
3	jean.dupond@gmail.com	busy	
4	jean.dupont@gmail.com	offline	
5	jeanne.dupond@gmail.com	offline	
6	joe.doe@gmail.com	online	

Dialog

john doe is online

Fermer

## Conclusion

Ce TP vous aura appris à :

- Utiliser les directives intégrées à Angular (*ngIf*, *ngFor*),
- Créer un service Angular et l'injecter dans un composant,
- Créer des composants avec une relation parent-enfant,
- Créer et utiliser un template,
- Communiquer entre les composants.

Correction du TP : [tp3-solution](#)

## TP 4

Formulaires et routes

Dans ce TP, nous allons créer les formulaires de connexion et d'authentification. Pour ce faire nous aurons besoin de mettre en place la module de routage d'Angular pour afficher plusieurs pages.

# Installation

Installez le module `@angular/router` avec la commande suivante :

```
npm install --save @angular/router
```

## Création des pages et des routes

Nous devons maintenant créer les composants Déclarez ensuite les routes dans un nouveau fichier `app.routes.ts` . Nous devons créer les routes suivantes :

Page	Route	Composant	Description
Accueil	/	HomeComponent	Affiche un message de bienvenue avec un bouton redirigeant l'utilisateur vers la page de connexion. Une fois authentifié, cette page affiche la liste des utilisateurs.
Login	/login	LoginComponent	Formulaire d'authentification
Signup	/signup	SignupComponent	Formulaire d'inscription

1. Commencez par créer les composants énoncés dans le tableau.
2. Ensuite, vous devrez écrire vos routes dans le fichier `app.routes.ts` en fonction de tableau ci-dessus.
3. Importez vos routes dans votre application `app.module.ts` comme suivant :

```
import { ROUTES } from "./app.routes";
import { RouterModule } from "@angular/router";

@NgModule({
  ...
  imports: [
    RouterModule.forRoot(ROUTES)
  ],
  ...
})
```

Enfin vous devez indiquer au routeur où seront affichées les vues dans votre application.

```
<router-outlet></router-outlet>
```

## Page d'accueil

La page d'accueil devra afficher la liste des utilisateurs une fois authentifié. Commencez par réintégrer la liste des utilisateurs dans HomeComponent. Nous nous occuperons des règles d'affichage plus tard.

## Page d'authentification

Nous allons maintenant utiliser les API Formulaires d'Angular. Nous utiliserons ici les formulaires pilotés par le template mais vous êtes libre d'utiliser la seconde approche.

Le formulaire de connexion présentera les éléments suivants :

- Un champ e-mail, qui est un champ obligatoire et doit respecter le pattern e-mail,
- Un champ mot de passe, qui est un champ obligatoire,
- Un bouton de connexion, actif que si toutes les données saisies sont valides,
- Un bouton ou m'enregistrer pour rediriger l'utilisateur vers la page d'inscription.

N'oubliez pas d'afficher les messages en fonction du type d'erreur du champs associé.

Voici un exemple du formulaire attendu en Material :

# Login

[ou m'enregister](#)

## Le service Authenticate

Une fois le formulaire réalisé, vous devrez créer le service `AuthenticateService` ( `ng g service authenticate` ) qui aura pour fonction d'authentifier l'utilisateur.

Ce service exposera les méthodes suivantes :

Method	Descripton
<code>authenticate(email: string, password: string): boolean</code>	Permet d'authentifier l'utilisateur. Stocke l'utilisateur dans le <code>localStorage</code> et émet un événement sur le flux <code>onSignin</code> .
<code>logout()</code>	Supprimer les infos stocker dans le <code>localStorage</code> et émet un événement sur le flux <code>onLogout</code> .
<code>getUser()</code>	Retourne l'utilisateur connecté et stocké dans le <code>localStorage</code> .
<code>get onSignin(): EventEmitter&lt;User&gt;</code>	Retourne l'objet (ou flux) <code>EventEmmitter</code> dédié à l'événement <code>onSignin</code> .

```
get onLogout():  
EventEmitter<boolean>
```

Retourne l'objet (ou flux) `EventEmitter` dédié à l'événement `onLogout`.

Nous utiliserons la class `EventEmitter` pour permettre à des composants de s'abonner à des événements.

## Utilisation du `localStorage`

```
// setter  
localStorage.setItem("currentUsers", JSON.stringify({firstName: "John"}));  
// getter  
const user = JSON.parse(localStorage.getItem("currentUsers"));
```

## La page d'accueil (fin)

Maintenant que vous avez un service `AuthenticateService` de prêt, vous pouvez gérer les règles d'affichage de cette page !

## Page d'inscription

Le formulaire d'inscription présentera les éléments suivants :

- Prénom, qui est un champ obligatoire,
- Nom, qui est un champ obligatoire,
- Un champ e-mail, qui est un champ obligatoire et doit respecter le pattern e-mail,
- Un champ mot de passe, qui est un champ obligatoire,
- Un bouton de enregistrer, actif que si toutes les données saisies sont valides,

Ensuite rajouter les fonctions permettant d'ajouter un utilisateur dans liste d'utilisateurs du service `UserService` .

## La navbar (bonus)

Une application est toujours plus belle lorsque cette dernière possède une `navbar` .

Cette navbar devra s'afficher sur l'ensemble des pages. Elle devra afficher les liens suivants :

- Home,
- Signin lorsque aucun utilisateur n'est authentifié,
- Signout lorsqu'un utilisateur est authentifié.

Pour vous aider, vous pouvez utiliser la librairie Material <http://material.angular.io>.



# TP 5

## Intégration des webservices

Nous avons maintenant besoin de faire fonctionner notre application avec des vrais webservice.

Nous allons donc utiliser le module Http d'Angular pour consommer des services Rest.

## Installation du serveur

Le serveur est déjà développé. Il expose un certain nombre de service que nous détaillerons par la suite.

Commençons par l'installer avec la commande suivante :

```
npm install --save labs-angular-backend
npm install --save-dev concurrently
```

Ensuite nous allons créer un fichier `server.js` à la racine du projet. Dans ce fichier copiez le code suivant :

```
const labsAngularBackend = require("labs-angular-backend");

new labsAngularBackend.Server().start().catch(er => console.error(er));
```

Pour nous simplifier la vie nous allons rajouter des commandes npm dans le `package.json` :

```
{
  "scripts": {
    "start": "npm install && concurrently \"npm run start:server\" \"npm run start:app\"",
    "start:app": "ng serve --proxy-config proxy.conf.json",
    "start:server": "node server.js"
  }
}
```

`npm run start` démarrera désormais l'application front et le serveur en même temps.

Il nous reste à configurer le proxy de `ng serve` pour que l'application web puisse consommer les webservices. Créez un nouveau fichier `proxy.conf.json` et copiez la configuration suivante :

```
{
```

```

"/api/*": {
  "target": "http://localhost:8080",
  "changeOrigin": true,
  "secure": false,
  "logLevel": "debug"
},
"/socket.io/*": {
  "target": "http://localhost:8080",
  "changeOrigin": true,
  "secure": false,
  "logLevel": "debug"
}
}

```

Enfin lancez la commande `npm run start` pour vérifier que l'ensemble fonctionne !

## Les webservice exposés

Voici la liste des webservice :

Method	Endpoint	Class method
ALL	/api/	RestCtrl.test()
GET	/api/html	RestCtrl.render()
POST	/api/users/	UserCtrl.create()
GET	/api/users/	UserCtrl.getList()
PATCH	/api/users/:email/:status	UserCtrl.updateStatus()
GET	/api/users/:idOrEmail	UserCtrl.get()
PUT	/api/users/:id	UserCtrl.update()
DELETE	/api/users/:id	UserCtrl.remove()
POST	/api/users/authenticate	UserCtrl.authenticate()

Avec les informations dont vous disposez concernant les webservice, vous pouvez intégrer ces services dans votre application.

Commencez par intégrer le service retournant la liste des utilisateurs ! Ensuite vous pouvez intégrer tous les autres services sauf la suppression (en bonus).

**Note :** Dans ce TP, vous n'avez pas pour obligation d'utiliser l'API reactive avec le module Http. Vous pouvez directement transformer l'objet Observable retourner par Http en Promise comme suivant :

```
import 'rxjs/add/operator/toPromise';

http.get(`/api/users`).toPromise(); // => Promise
```

Correction du TP : [tp5-solution](#)

## TP 6

---

Programmation réactive et websocket

Nous allons développer la gestion des changements de statut de connexion des utilisateurs en temps réel.

Autrement dit dès lorsqu'un utilisateur se connectera sur notre application, nous mettrons à jour la liste des utilisateurs avec les nouveaux statuts.

Pour ce faire, nous allons utiliser les WebSockets associés à la programmation réactive.

## Installation

---

Commençons par installer le WebSocket Client:

```
npm install --save socket.io-client
```

## Exemple d'utilisation

---

WebSocket nécessite très de configuration pour fonctionner. Il faut simplement lui donner l'url du serveur websocket. Voici un exemple :

```
import * as io from 'socket.io-client';

const socket = io("http://host:port");

socket.on('mycustomevent', (data) => console.log(data));
```

## Exercice 1

---

Notre objectif sera de mettre en place le socket client et de fournir un observable via votre nouveau service `UsersSocketService`.

Le socket client écoutera l'événement `users.update` et recevra la liste nouvelle liste d'utilisateurs à chaque changement de statuts.

Pour tester le changement de statuts, ouvrez un nouvelle onglet avec l'application est connectez-vous avec un autre compte utilisateur.

## Exercice 2

---

Une fois que vous avez réussi à récupérer la liste des utilisateurs à l'afficher dans la liste, essayez de rendre observable de bout en bout votre composant. Ainsi nous consommerons à la fois un webservice Ajax en mode Observable et du socket en Observable. Ces observables seront mergés pour ne donner qu'un seul Observable qui sera directement utilisé dans la vue.

Trop difficile ?

Je vous recommande donc de procéder par étape :

- Créer une méthode observable comme par exemple `UserService.getUsersObservable(): Observable<User[]>`,
- Merger les deux observables (UserService et UsersWebSocketService),
- Utiliser le pipe **async** dans le template pour connecter notre Observable avec le `*ngFor`.

À vous de jouer !

Correction du TP : [tp6-solution](#)