

# Framework Angular



**Pr Adil ANWAR**

[anwar@emi.ac.ma](mailto:anwar@emi.ac.ma)

## Prérequis

- Connaître les fondamentaux du web (protocole http, Architecture client/server, etc.)
- Être à l'aise avec les technologies client side (HTML, CSS, JS, )

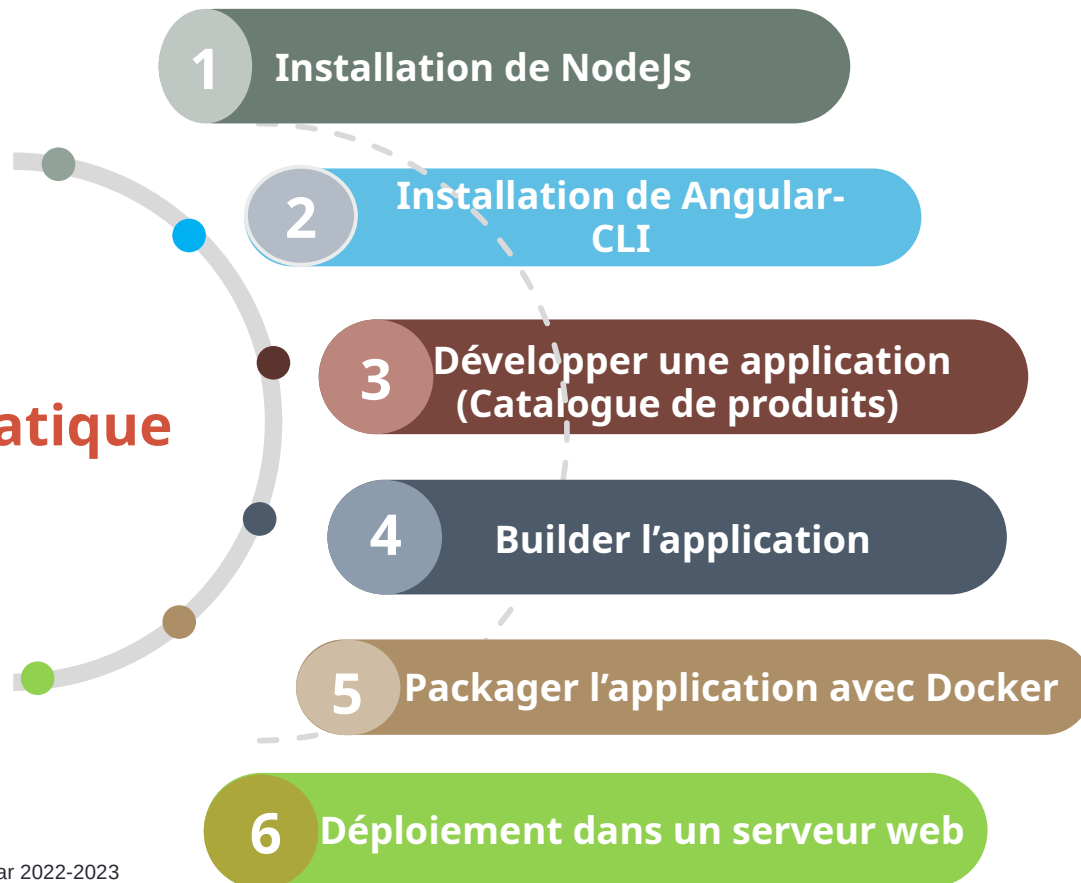
# Objectifs de formation

A la fin de ce cette formation, vous devez savoir :

- Fondamentaux de **Angular**
- Être capable de développer une application web avec Angular Etre
- capable de builder, déployer une application Angular



## Pratique



Adil Anwar 2022-2023

## Le routage



Adil Anwar 2022-2023

## Routage

- Afin de permettre aux utilisateurs de garder leurs habitudes de navigation en visitant une Single Page Application ou une Progressive Web Application, il est nécessaire d'utiliser un système de "Routing".
- Grâce au système de "Routing", les utilisateurs peuvent :
  - ♦ utiliser l'historique de leur navigateur (e.g. les boutons back et next),
  - ♦ partager des liens,
  - ♦ ajouter une vue à leurs favoris,
  - ♦ ouvrir une vue dans une nouvelle fenêtre via le menu contextuel,
  - ♦ ...

Angular fournit nativement un module de "Routing" pour répondre à ce besoin.

## Routage

- Angular utilise les composants pour gérer les fonctionnalités de routage. Les composants représentent les différentes «pages» auxquelles le routeur dirigera les utilisateurs.
- Pour rajouter la fonctionnalité de routage, vous devez suivre les étapes suivantes :
  1. Il faut importer le type **Routes** dans le fichier app.routes.ts:

```
import { Routes } from '@angular/router';
```
  2. déclarez le tableau des routes avec le type **Routes**

```
const appRoutes: Routes
```
- Ceci constitue un tableau (de type Routes) qui définit la configuration du routage de l'application Angular.
- Chaque élément de ce tableau est une configuration de route, qui associe une URL (via la propriété **path**) à un composant spécifique (via la propriété **component**) à afficher lorsque cette URL est visitée.

## Routage

3. initialiser la variable, qui est un tableau d'objets avec deux paramètres obligatoires: **path** et **composant**.

```
const appRoutes : Routes = [  
  { path : 'produits', component : ProductListComponent },  
  { path : 'auth', component : AuthComponent },  
  { path : 'produits/:id', component : EditProductComponent },  
  { path : 'panier', component : PanierComponent },  
  { path : '', component : ProductListComponent },  
  { path : 'admin', component : AdminComponent },  
  { path : 'not-found', component : FourOhFourComponent },  
  { path : '**', redirectTo : '/not-found' }  
]
```

- la route **\*\*** au bas de cette déclaration. Il s'agit de la route générique qui peut être utilisée pour afficher un **NotFoundComponent** - ou une page 404 - pour les routes qui ne correspondent pas
- Il est important de noter que l'ordre de déclaration des routes est très important.

Adil Anwar 2022-2023

## Routage

4. vous ajouterez la déclaration au tableau dans le tableau des fournisseurs de services dans le fichier de configuration `app.config.ts` pour que les routes que nous venons de configurer soient disponibles pour l'application.

```
providers: [  
  provideRouter(routes)  
],
```

5. La configuration du "Routing" permet de définir quel composant afficher en fonction de la route mais cela n'indique pas à Angular où injecter le composant dans la page.

Pour indiquer l'emplacement d'insertion du composant, il faut utiliser la directive `<router-outlet>` directement dans le "root component" `AppComponent`

Le tag `router-outlet` injectera le composant configuré pour la route présentée.

## Routage : Routes imbriquées

- Les routes imbriquées en Angular permettent d'organiser la navigation en regroupant plusieurs vues sous une même route parente.
- Par exemple, une route admin peut contenir des sous-routes comme users ou settings.
- Lorsqu'on souhaite rediriger automatiquement l'utilisateur vers une sous-route par défaut, Angular utilise la propriété **pathMatch** pour déterminer comment l'URL est comparée au chemin de la route.
- Dans le cas suivant, la route enfant avec path: " " redirige vers users uniquement lorsque l'URL correspond exactement à /admin, grâce à **pathMatch: 'full'** :
- Ainsi, l'URL **/admin** est redirigée vers **/admin/users**, tandis que **/admin/users** n'est pas affectée par la redirection. Sans **pathMatch: 'full'**, Angular utiliserait la correspondance par préfixe (**prefix**), ce qui ferait correspondre le chemin vide à toutes les URLs et entraînerait une redirection infinie.

```
export const routes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    children: [
      { path: 'users', component: UsersComponent },
      { path: 'settings', component: SettingsComponent },
      { path: ' ', redirectTo: 'users', pathMatch: 'full' }
    ]
  }
];
```

Adil Anwar 2022-2023

## Routage -- navigation

- Problème : Pour accéder à un composant, l'utilisateur doit connaître son chemin défini dans le tableau de routes (or ceci n'est pas vraiment très pratique)
- Solution : On peut plutôt définir un menu contenant des liens vers nos différents composants
- Ici, on utilise l'attribut **routerLink** à la place de *href* pour faire référence à un *path* défini dans notre tableau de routes.
- En utilisant des liens natifs <a href="/search">, le "browser" va produire une requête HTTP GET vers le serveur et recharger toute l'application.
- Pour éviter ce problème, le module de "Routing" Angular fournit la directive **routerLink** qui permet d'intercepter l'événement click sur les liens et de changer de "route" sans recharger toute l'application.
- Exemple :

<a **routerLink**="/search">Search</a>

Adil Anwar 2022-2023

## Routage – Création de routes avec paramètres

- La "route" /books/123 peut être construite avec des paramètres :

```
<a [routerLink]="['/books', book.id]">détails</a>
```

où book.id = '123'.

- Il est également possible de **passer des paramètres optionnels** par "query string" via l'Input **queryParams**.
- Exemple :**

```
<a [routerLink]="'/search'" [queryParams]="{key: 'eXtreme Programming'}">search</a>
```

## Routage – navigation dans le code

- Il est également possible de naviguer entre les différentes routes en utilisant la méthode **navigate()** du service Router. Pour cela le service Router doit être injecté dans la classe composant.
- L'exemple suivant montre le code de la classe du composant AppComponent : app.component.ts

```
<button class ="btn btn-success" *ngIf ="!authStatus"
(click)="onSignIn()">se connecter</button>
```

```
onSignIn(){
  this.authService.signIn().then(()=> {
    //console.log('connexion réussie');
    this.authStatus = this.authService.iAuth;
    this.router.navigate( commands: ['produits']);
  });
}
```

## Routage -- Paramètres de routes

- Pour ajouter un paramètre de route, il suffit de créer la route appropriée à l'aide de path: identifier, puis d'inclure le paramètre, tel que id, dans cette entrée, comme illustré dans l'exemple suivant:

```
const appRoutes : Routes = [  
  { path : 'produits', component : ProductListComponent},  
  { path : 'auth', component : AuthComponent},  
  { path : 'produits/:id', component : EditProductComponent},
```

- La dernière entrée de cette section Routes est produits /: id.
- La partie: id est le paramètre de route et sera remplacée par la valeur transmise à l'URL
- Exemple : http: //localhost:4200/produits/12545

Adil Anwar 2022-2023

## Routage -- Paramètres de routes

- Le service **ActivatedRoute** décrit l'état actuel du "router".
- Il permet au composant associé à la "route" de récupérer les paramètres via l'objet **snapshot** et le tableau **params**.
- Exemple : Pour récupérer les paramètres d'une route de la forme : products/123
  - aller dans le composant concerné (ici, *EditProductComponent*.)
  - faire une injection de dépendance du service Angular **ActivatedRoute**
  - utiliser ce service dans la méthode *ngOnInit()* pour récupérer l'objet **snapshot** puis récupérer les paramètres

```
constructor(private productService : ProductService,  
             private route : ActivatedRoute,  
             private panierService : PanierService) { }  
  
ngOnInit() {  
  const id : number = this.route.snapshot.params['id'];  
  this.product = this.productService.getProductById(+id);  
}
```

## Routage -- Guards

- Certaines routes de l'application ne devraient pas être accessibles à tous. L'utilisateur est-il authentifié ?
- A-t-il les permissions nécessaires ? Bien sûr, il faut désactiver ou masquer les liens pointant sur ces routes inaccessibles.
- Le backend doit aussi empêcher l'accès aux ressources interdites à l'utilisateur courant. Mais cela n'empêchera pas l'utilisateur d'accéder aux routes qui lui sont interdites, simplement en entrant leur URL dans la barre d'adresse du navigateur.

Adil Anwar 2022-2023

## Routage -- Guards

- C'est là que le guard **CanActivate** intervient, lorsqu'un tel guard est appliqué à une route, il peut empêcher l'activation de la route

```
@Injectable({providedIn: 'root'})

export class AuthGuard implements CanActivate {
  constructor(private authService : AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean
  | UrlTree | Promise<boolean | UrlTree> | Observable<boolean | UrlTree> {

    return this.authService.userSubject$.pipe(
      map(user=>{
        const isAuthenticated = user ? true : false;
        if(isAuthenticated) {
          return true;
        }
        return this.router.createUrlTree(['/auth'])
      })
    );
  }
}
```

## Routage -- Guards

- Le Guard doit être appliqué sur une route donnée. Ceci est indiqué en configuration de la route.

```
const appRoutes : Routes = [  
  { path : 'produits', component : ProductListComponent},  
  { path : 'auth', component : AuthComponent},  
  {path : 'produits/:id', component : EditProductComponent},  
  {path : 'panier', component : PanierComponent},  
  {path : '', component : ProductListComponent},  
  {path : 'admin', component : AdminComponent, canActivate :[AuthGuardService]},  
  {path : 'not-found', component : FourOhFourComponent},  
  {path : '**', redirectTo : '/not-found'}  
]
```

Adil Anwar 2022-2023

## Les formulaires



Adil Anwar 2022-2023

# Formulaires

- En Angular ou en développement web en général, les formulaires sont très utiles pour valider les saisies de l'utilisateur, afficher les erreurs correspondantes, ou aussi pour déclarer des champs obligatoires ou non, ou qui dépendent d'un autre champ,
- Elles permettent de réagir sur les changements de certains, etc. On a aussi besoin de tester ces formulaires,
- En Angular, il y a deux grandes méthodes pour créer des formulaires :
  - **La méthode « piloté par le template »**
    - vous créez votre formulaire dans le template, et Angular l'analyse pour comprendre les différents inputs et pour en mettre à disposition le contenu ,
    - utile pour des formulaires simples, sans trop de validation.
  - **la méthode réactive ou « piloté par le code »**
    - vous créez votre formulaire en TypeScript et dans le template, puis vous en faites la liaison manuellement à l'aide de directives
    - approche est plus complexe, mais elle permet beaucoup plus de contrôle et de validation
    - approche dynamique car ça permet de générer des formulaires dynamiquement.

Adil Anwar 2022-2023

## Formulaires -- template

- Dans cette méthode, on va mettre en oeuvre un ensemble de **directives** dans notre formulaire
- Toutes ces directives sont incluses dans le module **FormsModule**, nous devons donc l'importer dans notre module racine.

```
import {FormsModule} from '@angular/forms';

const appRoutes : Routes= [ ...
  @NgModule({
    declarations: [],
    imports: [
      FormsModule,
      CommonModule, RouterModule.forRoot(appRoutes)
    ],
  })
]
```

- **FormsModule** contient les directives pour la façon "pilotée par le template".
- il existe un autre module, **ReactiveFormsModule**, dans le même package **@angular/forms**, qui est nécessaire pour la façon "pilotée par le code".

Adil Anwar 2022-2023

## Formulaires -- template

- **ngForm**: cette directive transforme l'élément `<form>` en sa version Angular correspondante
- **ngModel**: La directive `ngModel` met à jour la valeur de l'input à chaque changement du modèle lié à l'input.
- elle génère aussi un événement à chaque fois que l'input est modifié par l'utilisateur
- cette directive est utilisée pour faire la liaison bidirectionnelle « two way binding »

```
<label>Username</label><input name="username" [(ngModel)]="user.username">
```

- **ngSubmit**: est émis par la directive `form` lors de la soumission du formulaire. Cela invoquera une méthode écrite dans le composant qui gère la template

```
<form (ngSubmit)="register()">
  <button type="submit">Register</button>
</form>
```

Adil Anwar 2022

## Formulaires -- template : ngForm

- Exemple : version 1

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm)" #userForm="ngForm">
  <div>
    <label>Username</label>
    <input name="username" required minlength="3"
      [(ngModel)]="user.username" >
  </div>
  <div>
    <label>Password</label>
    <input type="password" required name="password"
      [(ngModel)]="user.password">
  </div>
  <button type="submit">Register</button>
</form>
```

Adil Anwar 2022-2023

## Formulaires -- template

- Dans l'exemple précédent, on crée une variable locale `userForm` et lui assigner l'objet `NgForm` créé par Angular pour référencer le formulaire.
- C'est possible parce que la directive exporte l'instance de la directive `NgForm`,
- Notre méthode `register` est désormais appelée avec la valeur du formulaire en paramètre :

```
register(form : NgForm) {  
  const username : string = form.value['username'];  
  const password : string = form.value['password'];  
}
```

Adil Anwar 2022-2023

## Formulaires --template

- Angular nous offre une possibilité de lier un champ et une expression qui se mettrait à jour automatiquement dès que l'utilisateur entrait une valeur dans le champ. C'est la « liaison Bi-directionnelle » ou le two-way binding

```
<h2>Sign up</h2>  
<form (ngSubmit)="register(f)" #f="ngForm">  
  <div>  
    <label>Username</label>  
    <input name="username" minlength="3" required  
      [(ngModel)]="user.username" >  
  </div>  
  <div>  
    <label>Password</label>  
    <input type="password" name="password" required  
      [(ngModel)]="user.password">  
  </div>  
  <button type="submit">Register</button>  
</form>
```

Adil Anwar

## Formulaires – Validation

- La validation de données est traditionnellement une partie importante de la construction de formulaire. Certains champs sont obligatoires, certains dépendent d'autres, certains doivent respecter un format spécifique....
- Commençons par ajouter quelques règles basiques : **tous nos champs sont obligatoires**,

```
<h2>Sign up</h2>
<form (ngSubmit)="register(f)" #f="ngForm">
  <div>
    <label>Username</label>
    <input name="username"
      required minlength="3"
      [(ngModel)]="user.username">
    <small>{{ user.username }}</small>
  </div>
  <div>
    <label>Password</label>
    <input type="password" required
      name="password" [(ngModel)]="user.password">
  </div>
  <button type="submit">Register</button>
</form>
```

Adil Anwar 2022-2023

## Formulaires – Validation

- Un élément du formulaire a plusieurs attributs, dont :
  - **valid** : si le champ est valide
  - **invalid** : si le champ est invalide
  - **pristine** : l'opposé de dirty.
  - **dirty** : false jusqu'à ce que l'utilisateur modifie la valeur du champ.
  - **untouched** : l'opposé de touched.
  - **touched** : false jusqu'à ce que l'utilisateur soit entré dans le champ.
  - **value** : la valeur du champ.
  - **errors** : un objet contenant les erreurs du champ.

Adil Anwar 2022-2023

# Formulaires - Template - Erreurs de soumission

- Maintenant il nous faut afficher les erreurs sur chaque champ.
- on peut donc créer une variable locale pour accéder aux erreurs :

```
<h2>Sign up</h2>
<form (ngSubmit)="register(f)" #f="ngForm">
  <div>
    <label>Username</label>
    <input name="username"
      required minlength="3"
      [(ngModel)]="user.username" #userName="ngModel">
    <small>{{ user.username }}</small>
  </div>
  <div *ngIf="userName.dirty
    && userName.hasError( errorCode: 'required' )">
    User name is required
  </div>
```

Adil

# Formulaires - Template - Erreurs de soumission

- Evidemment, on veut que l'utilisateur ne puisse pas soumettre le formulaire tant qu'il reste des erreurs, et ces erreurs doivent être parfaitement affichées.

```
<h2>Sign up</h2>
<form (ngSubmit)="register(f)" #f="ngForm">
  <div>
    <label>Username</label>
    <input name="username"
      required minlength="3"
      [(ngModel)]="user.username">
    <small>{{ user.username }}</small>
  </div>
  <div>
    <label>Password</label>
    <input type="password" required
      name="password" [(ngModel)]="user.password">
  </div>
  <button type="submit" [disabled]="f.invalid">Register</button>
</form>
```

Adil Anwar 2022-2023

## Formulaires – Piloté par le code

- Angular introduit une déclaration impérative, qui permet de construire les formulaires par programmation.
- Désormais, on peut manipuler les formulaires directement depuis le code. C'est plus verbeux mais plus puissant.
- Pour construire un formulaire dans notre code, nous allons utiliser les classes: [FormControl](#) et [FormGroup](#)

## Formulaires – Piloté par le code : FormGroup

- Avec ces briques de bases on peut construire un formulaire dans notre composant. En utilisant les classes *FormControl* ou *new FormGroup*.
- [FormGroup](#) est une classe utilitaire qui représente un groupe de controls c'est-à-dire un ensemble de formControls

### Classe du composant

```
export class ReactAuth {  
  form = new FormGroup({  
    email : new FormControl(''),  
    password : new FormControl('')  
  })  
}
```

### Template du composant

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">  
  <h2 style="text-align:center"> s'authentifier</h2>  
  <div class="form-group">  
    <label for="email">E-mail</label>  
    <input type="email" [formControl]="form.controls.email"/>  
  </div>  
  <div class="form-group">  
    <label for="password">Password</label>  
    <input type="password" [formControlName]="password"/>  
  </div>  
  <div class="form-group m-2">  
    <button class="btn btn-primary" type="submit">Login</button>  
  </div>  
</form>
```

## Formulaires – Piloté par le code

- On utilise la notation avec crochets [`formGroup`]="form" pour relier notre objet form à formGroup.
- Chaque input reçoit la directive `formControlName` avec pour valeur le nom du contrôle auquel il est relié. Si l'on indique un nom qui n'existe pas, on aura une erreur. Comme on passe une valeur (et pas une expression), on n'utilise pas les `[]` autour de `formControlName`.

## Validation du formulaire

- Pour spécifier que chaque champ est requis, on va utiliser `Validator`.
- Un validateur retourne une map des erreurs, ou null si aucune n'a été détectée. Quelques validateurs sont fournis par le framework
  - `Validators.required` pour vérifier qu'un contrôle n'est pas vide
  - `Validators.minLength(n)` pour s'assurer que la valeur entrée a au moins n caractères
  - `Validators.maxLength(n)` pour s'assurer que la valeur entrée a au plus n caractères
  - `Validators.pattern(p)` pour s'assurer que la valeur entrée correspond à l'expression régulière p définie.

# Validation - configuration

- Les validateurs peuvent être multiples, en passant un tableau, et peuvent s'appliquer sur un FormControl ou un FormGroup.
- Comme on veut que tous les champs soient obligatoires, on peut ajouter le validator required sur chaque contrôle, et s'assurer que le nom de l'utilisateur fait 3 caractères au minimum.

```
export class RegisterFormComponent {  
  userForm: FormGroup;  
  constructor(fb: FormBuilder) {  
    this.userForm = fb.group({  
      username: fb.control('', [  
        Validators.required,  
        Validators.minLength(3)  
      ]),  
      password: fb.control('', Validators.required)  
    });  
  }  
}
```

Adil Anwar 2022-2023

## Validation - Contrôle de soumission

- Nous avons ajouté un champ userForm, du type FormGroup, à notre composant. Ce champ fournit une vision complète de l'état du formulaire et de ses champs, incluant ses erreurs de validation.

```
<h2>Sign up</h2>  
<form (ngSubmit)="register()" [formGroup]="userForm">  
  <div>  
    <label>Username</label>  
    <input formControlName="username">  
  </div>  
  <div>  
    <label>Password</label>  
    <input type="password" formControlName="password">  
  </div>  
  <button type="submit"  
    [disabled]="!userForm.valid">Register</button>  
</form>
```

Adil Anwar

# Validation - Affichage des erreurs

```
<form (ngSubmit)="register()" [formGroup]="userForm">
  <label>Username</label>
  <input formControlName="username">
  <div *ngIf="userForm.get('username').hasError('required')">
    Username is required
  </div>
  <label>Password</label>
  <input type="password" formControlName="password">
  <button [disabled]="!userForm.valid">Register </button>
</form>
```

Adil Anwar 2022-2023

## Validateurs spécifiques

- On souhaite valider le champ mot de passe que s'il contient des caractères spéciaux comme ! Ou ?. Comment fait-on cela ? En créant un validateur spécifique. Pour ce faire, il suffit de créer une méthode qui accepte un *AbstractControl*, teste sa valeur, et retourne un objet avec les erreurs, ou null si la valeur est valide

```
function customPasswordValidator(control : AbstractControl){
  if(control.value.includes('?') || control.value.includes('!')){
    return null;
  }
  return {
    passwordMustContainsSpecialCharacters : true
  }
}
```

Adil Anwar 2022-2023

## Validateurs spécifiques : utilisation

```
form = new FormGroup({
  email : new FormControl('', [
    Validators.required,
    Validators.email
  ]),
  password : new FormControl('',
    [Validators.required, Validators.minLength(6), customPasswordValidator]
  )
})
```

Adil Anwar 2022-2023

## Réagir aux modifications

- Avec un formulaire piloté par le code : on peut facilement réagir aux modifications, grâce à l'observable **valueChanges**.
- Par exemple, disons que notre dans le TypeScript, vous allez créer un Observable `filmPreview$` qui émettra des objets de type `filmValue`
- Branchez cet Observable aux changements de valeur du formulaire avec son attribut `valueChanges` , un Observable qui émet la valeur du formulaire à chaque modification :

```
filmForm!: FormGroup;
filmPreview$: Observable<Film>;
```

```
this.filmPreview$ = this.filmForm.valueChanges.pipe(
  map(formValue=>({
    ...formValue,
    createdAt: new Date(),
    score: 0,
    id: 0
  })))
);
```

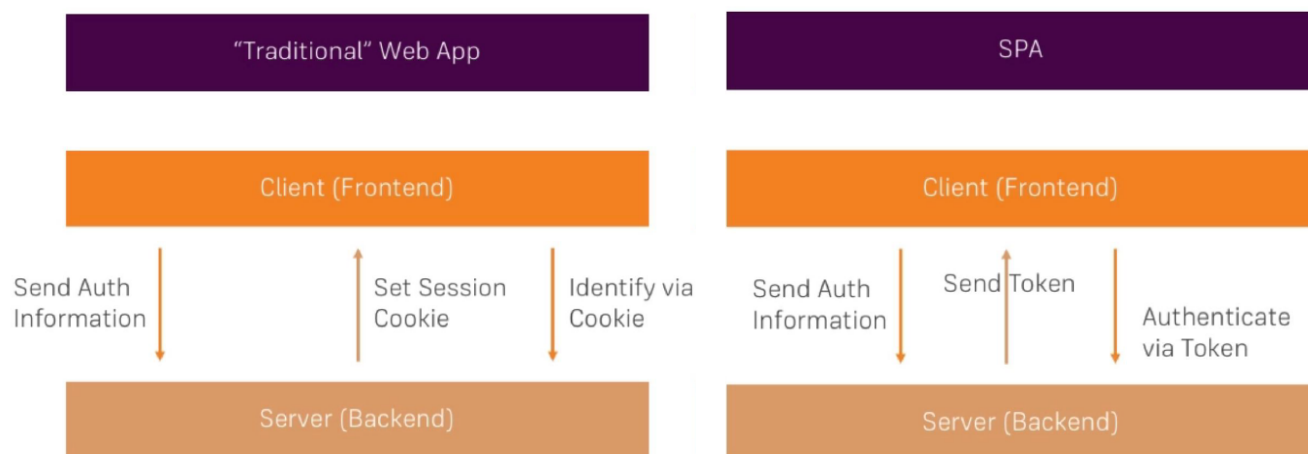
Adil Anwar 2022-

# Authentication



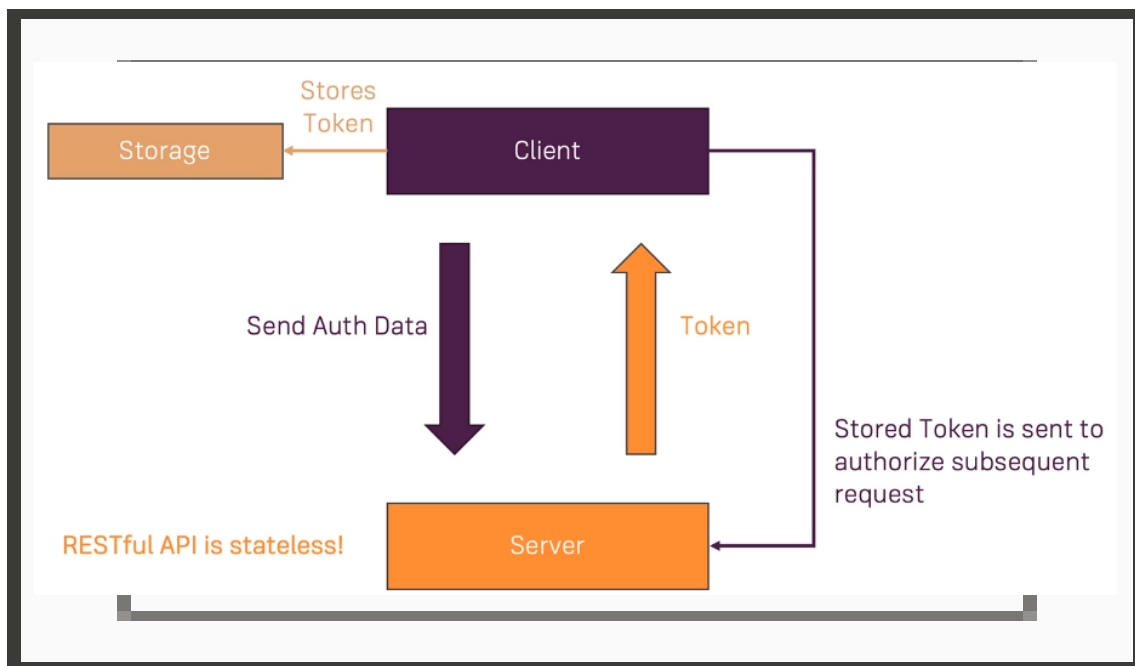
Adil Anwar 2022-2023

## Workflow d'authentification



Adil Anwar 2022-2023

## Workflow d'authentification



Adil Anwar 2022-2023

## Technique d'authentification

- Méthode d'utilisation de l'API Rest de Firebase
  1. Créer une application dans firebase
  2. Activation de l'authentification par mail/password dans la console de firebase
    - Pour utiliser l'authentification par e-mail et mot de passe proposée par Firebase, il faut d'abord l'activer dans la console Firebase :

The screenshot shows the 'Adresse e-mail/Mot de passe' (Email/Password) authentication method in the Firebase console. The 'Activer' (Activate) toggle switch is turned on. Below the toggle, there is a description: 'Permet aux utilisateurs de s'inscrire avec leur adresse e-mail et leur mot de passe. Nos SDK proposent également la validation de l'adresse e-mail, la récupération du mot de passe et les primitives de modification de l'adresse e-mail. [En savoir plus](#)'. At the bottom, there are two buttons: 'ANNULER' (Cancel) and 'ENREGISTRER' (Register).

Adil Anwar 2022-2023

# Technique d'authentification

1. Récupérer la clé de l'application firebase et créer une variable API\_KEY dans un service AuthService

```
const firebaseConfig = {  
  apiKey: "AIzaSyC_y17o5SnybalVaoH9-VQUIRR0p_-P3oE",  
  authDomain: "shopping-app-f7668.firebaseio.com",  
  databaseURL: "https://shopping-app-f7668-default-rtdb.firebaseio.com",  
  projectId: "shopping-app-f7668",  
  storageBucket: "shopping-app-f7668.appspot.com",  
  messagingSenderId: "901463732897",  
  appId: "1:901463732897:web:1a2711abe0b9a51d43f6e0"  
};
```

```
export class AuthService {  
  API_KEY : string = "AIzaSyCEfhjdjcQwlbCBXYHo_uXhZkkYXFFJ_APQ"
```

Adil Anwar 2022-2023

# Technique d'authentification

1. Créer la classe AuthService et injecter le service HttpClient pour communiquer avec l'api rest de firebase
2. Créer les méthodes login et signUp en utilisant les « endpoints » fournis par l'api rest de firebase

```
login(email : string,password : string){  
  return this.http.post<AuthResponseData>('https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key='  
    +this.API_KEY,{  
    email : email,  
    password: password,  
    returnSecureToken : true  
  })  
}
```

```
signup(email: string,password: string){  
  return this.http.post<AuthResponseData>('https://identitytoolkit.googleapis.com/v1/accounts:signUp?key='  
    +this.API_KEY,{  
    email : email,  
    password: password,  
    returnSecureToken : true  
  })  
}
```

3. Une fois l'authentification passée récupérer le token et enregistrer le dans localStorage

Adil Anwar 2022-2023

# Atelier Authentification

- Créer un service AuthService bouchonnant les appels serveurs d'authentification sur l'application. Le service expose les méthodes suivantes:
  - **login**(email, passwd): void
  - **Signup**(email, passwd): void
  - **isAuthenticated**(): boolean
  - **logout**(): void
  - **getToken**(): boolean
- Au chargement de l'application, par défaut nous arrivons sur la page d'authentification (Formulaire: login, mdp), une fois authentifié, l'utilisateur est redirigé vers la vue Admin

## Intercepteurs



## Client HTTP - Intercepteurs

- Le nouveau client HTTP introduit la notion d'intercepteurs de requête/réponse.
- A chaque envoi de requête, l'intercepteur intercepte l'appel.
- Cela permet par exemple de rajouter des informations au header, de formater certaines données avant envoi..

```
export class AuthInterceptor implements HttpInterceptor {  
  constructor(private router: Router, private authService: AuthService) {}  
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
    const isAuthenticated = this.authService.isAuthenticated();  
    if (isAuthenticated) {  
      const authReq = req.clone({  
        headers: req.headers.set('Authorization', 'Bearer ' + this.authService.getToken()) }  
      );  
      return next.handle(authReq);  
    }  
    return next.handle(req);  
  }  
}
```

## Client HTTP - Intercepteurs

- Pour utiliser l'intercepteur il faut l'enregistrer dans le tableau des providers dans le fichier app.config.ts

```
providers: [{  
  provide: HTTP_INTERCEPTORS,  
  useClass: AuthInterceptor,  
  multi: true  
}],
```

# Déployer votre application



Adil Anwar 2022-2023

## Stratégies de déploiement

- Lors de la mise en production d'une application Angular/NodeJS, deux scénarios ou options peuvent être choisis pour le déploiement. Chacun de ces scénario présente des avantages et des inconvénients.
  1. Scénario 1 : [Déploiement séparé](#)
    - deux applications séparées qui s'exécutent sur deux serveurs (machines) différents.
  2. Scénario 2 : [Déploiement combiné](#)
    - une seule application Node Rest API qui va juste rendre l'application Angular

Adil Anwar 2022-2023

## Déploiement séparé

- Deux applications s'exécutant sur deux [domaines/ports](#) différents
- L'application Angular en Front-end tourne sur un host statique qui n'exécute pas un code serveur, ce host doit juste être capable d'exécuter un code HTML, CSS et JS.
  - Exemple : Apache, AWS S3, Firebase hosting (google),
- L'application back-end nodeJS/Express tourne sur un host qui est capable d'exécuter un code nodeJs
  - Exemple : AWS EC2/ EB, Heroku
- Il est important d'ajouter les entêtes CORS dans la partie NodeJS/express (domaines différents)

## Déploiement combiné

- Une seule application qui s'exécute sur un host capable d'exécuter un code NodeJs et qui retournent les fichiers Angular.
- Pas besoin de régler les entêtes CORS vu que Angular s'exécute sur le même domaine que la partie NodeJs/express

# Étapes de déploiement dans Heroku

## 1. Builder l'application Angular

- La première étape consiste à builder le projet Angular en utilisant la commande `ng build --prod`
- Ceci génère un dossier nommé **dist** que vous pouvez optionnellement renommer (ex, Angular) et transporter tel quel dans un des dossiers gérés par votre serveur préféré (Apache, Node.js, http-server...).

## 2. Déplacer le dossier **dist** généré dans le dossier racine de l'application NodeJs/express.

## 3. Préparer le code dans votre contrôleur principal (server.js) pour rediriger les appels vers http vers l'index.html.

```
app.get('/', (req, res) => {  
  res.sendFile(path.join(__dirname, 'angular', 'index.html'));  
});
```