Dynamiser la partie front

Table des matières

I. Contexte

Contexte

Dans ce cours, nous allons dynamiser notre application. En effet, nous avons jusque-là créé uniquement le visuel de notre application, mais il manque une certaine logique dynamique. Par exemple, nous avons des formulaires pour enregistrer des informations, mais nous ne faisons pas de contrôles sur ceux-ci. Nous ne gérons pas encore l'authentification des utilisateurs. Dynamiser un site est une partie fondamentale de la création d'un site web, car peu de sites sont uniquement statiques. Le JavaScript vous permettra d'avoir un site plus adapté aux besoins client, avec une interface plus lisible et plus agréable.

II. Le formulaire d'inscription

A. Introduction

Durée (en minutes) : 120

Environnement de travail :

- Ordinateur avec Windows, MacOS ou Linux
- Visual studio Code

Pré requis : connaître Git, et avoir des bases de Html, SASS, JS, et Bootstrap

Objectifs de la partie

- Savoir utiliser le JavaScript en web
- Savoir contrôler un formulaire

Contexte

Dans votre vie professionnelle, vous devrez sans doute dynamiser des applications web. Que ce soit pour une application de gestion, ou pour une application de présentation, le JavaScript vous aidera grandement. Vous pourrez faire des animations plus précises, vérifier la validité des champs, mais aussi et surtout gérer la logique de la partie front de votre application.

B. Vérifier les champs requis

Définition **Vérifier les champs en JS**

La vérification des champs d'un formulaire en JavaScript est une étape cruciale pour s'assurer que les données soumises par l'utilisateur sont valides et conformes aux attentes. Pour réaliser cette vérification, on utilise généralement des fonctions JavaScript qui s'exécutent à différentes étapes (modification de champs, clic, validation du formulaire, etc.). Ces fonctions peuvent vérifier différents aspects, tels que la présence d'informations obligatoires, le format correct des données (par exemple, une adresse e-mail valide), ou encore des valeurs dans des plages acceptables pour certains champs numériques. En cas de non-conformité, des messages d'erreur peuvent être affichés à l'utilisateur, l'incitant à corriger les champs concernés avant de réessayer la soumission. La vérification des champs en JavaScript contribue à améliorer l'expérience utilisateur en garantissant des données fiables et complètes pour le traitement ultérieur du formulaire.

Méthode **Capter l'évènement**

Lors de l'implémentation de la validité des champs d'un formulaire, nous devons définir à quel moment nous voulons effectuer cette vérification. Dans notre cas, nous allons effectuer la vérification lors de la saisie sur les champs de notre formulaire. Nous allons écouter l'évènement 'keyup'.

Définition **Événement keyup**

En JavaScript, l'événement `keyup` est un type d'évènement qui se déclenche lorsqu'une touche du clavier est relâchée après avoir été enfoncée. Il s'agit d'un évènement très couramment utilisé pour capturer les saisies de l'utilisateur dans un champ de saisie (input) ou une zone de texte (textarea).

Lorsque l'utilisateur appuie sur une touche du clavier, l'événement `keydown` se déclenche, puis lorsque la touche est relâchée, l'événement `keyup` est déclenché.

L'évènement `keyup` est souvent utilisé pour mettre à jour une interface utilisateur en temps réel en fonction de la saisie de l'utilisateur ou pour effectuer des actions spécifiques une fois qu'une certaine touche est relâchée.

Voici un exemple simple d'utilisation de l'événement `keyup` en JavaScript :

```
<input type="text" id="myInput">
```

JavaScript :

```
const inputElement = document.getElementById('myInput');

inputElement.addEventListener('keyup', (event) => {
  console.log('Touche relâchée : ', event.key);
});
```

Méthode **Vérifions les champs requis**

Nous voulons maintenant vérifier que les champs 'nom' et 'prénom' soient bien remplis. Nous voulons faire cette vérification chaque fois que l'utilisateur tapera sur le clavier dans un champ. On peut distinguer trois étapes à l'ajout d'un évènement en JS :

1. Définir la cible : le but est de sélectionner l'élément HTML sur lequel vous souhaitez écouter l'évènement. Cela peut être n'importe quel élément du DOM, comme un bouton, une balise div, un champ de saisie, etc. On peut sélectionner cet élément en utilisant différentes méthodes telles que document.getElementById, document.querySelector, ou en récupérant une liste d'éléments avec document.getElementsByClassName ou document.getElementsByTagName.
2. Définir l'évènement : on doit spécifier le type d'évènement que nous souhaitons écouter. Par exemple : click (clic de souris), keyup (relâchement d'une touche du clavier), submit (soumission d'un formulaire), etc.
3. Définir l'action à exécuter : c'est l'étape où on définit ce qui doit se passer lorsque l'évènement est déclenché. La méthode addEventListener permet d'attacher l'évènement à la cible et spécifier la fonction de rappel.

Nous obtenons ce résultat :

```
//Implémenter le JS de ma page

const inputNom = document.getElementById("NomInput");
const inputPreNom = document.getElementById("PrenomInput");
const inputMail = document.getElementById("EmailInput");
const inputPassword = document.getElementById("PasswordInput");
const inputValidationPassword = document.getElementById("ValidatePasswordInput");

inputNom.addEventListener("keyup", validateForm);
inputPreNom.addEventListener("keyup", validateForm);
inputMail.addEventListener("keyup", validateForm);
inputPassword.addEventListener("keyup", validateForm);
inputValidationPassword.addEventListener("keyup", validateForm);

//Function permettant de valider tout le formulaire
function validateForm(){
    validateRequired(inputNom);
    validateRequired(inputPreNom);
}
```

```
20
21 function validateRequired(input){
22     if(input.value != ''){
23         input.classList.add("is-valid");
24         input.classList.remove("is-invalid");
25     }
26     else{
27         input.classList.remove("is-valid");
28         input.classList.add("is-invalid");
29     }
30 }
```

C. Vérifier le mail

Définition **Expression régulière ou regex**

Une expression régulière, également appelée regex (de l'anglais regular expression), est une séquence de caractères qui définit un motif de recherche. Elle est utilisée pour rechercher et manipuler des chaînes de texte de manière puissante et flexible. Les expressions régulières sont largement utilisées dans la programmation, notamment pour valider des formats de texte, extraire des informations spécifiques ou effectuer des remplacements.

Exemple **Recherche des adresses e-mail dans un texte**

Expression régulière : `^[^\s@]+@[^\s@]+\.[^\s@]+$`

Dans cette expression régulière, nous cherchons à identifier une adresse e-mail. Voici ce que signifient les différents éléments de l'expression :

- ^ : début de la chaîne.
- [^\s@]+ : recherche un ou plusieurs caractères qui ne sont ni des espaces (\s) ni des arobases (@).
- @ : recherche le caractère « @ ».
- [^\s@]+ : recherche un ou plusieurs caractères qui ne sont ni des espaces (\s) ni des arobases (@).
- \. : recherche le caractère point (il doit être échappé car le point est un caractère spécial en expression régulière).
- [^\s@]+ : recherche un ou plusieurs caractères qui ne sont ni des espaces (\s) ni des arobases (@).
- $: fin de la chaîne.

Ce regex vérifie donc que l'adresse e-mail a au moins un caractère avant et après le « @ » et qu'il y a un point entre deux séquences de caractères.

Vous pouvez avoir plus d'informations sur ce regex sur ce lien (notamment l'onglet 'explanation' à droite : regular expressions 101[1]

1 https://regex101.com/r/ElLAYu/1

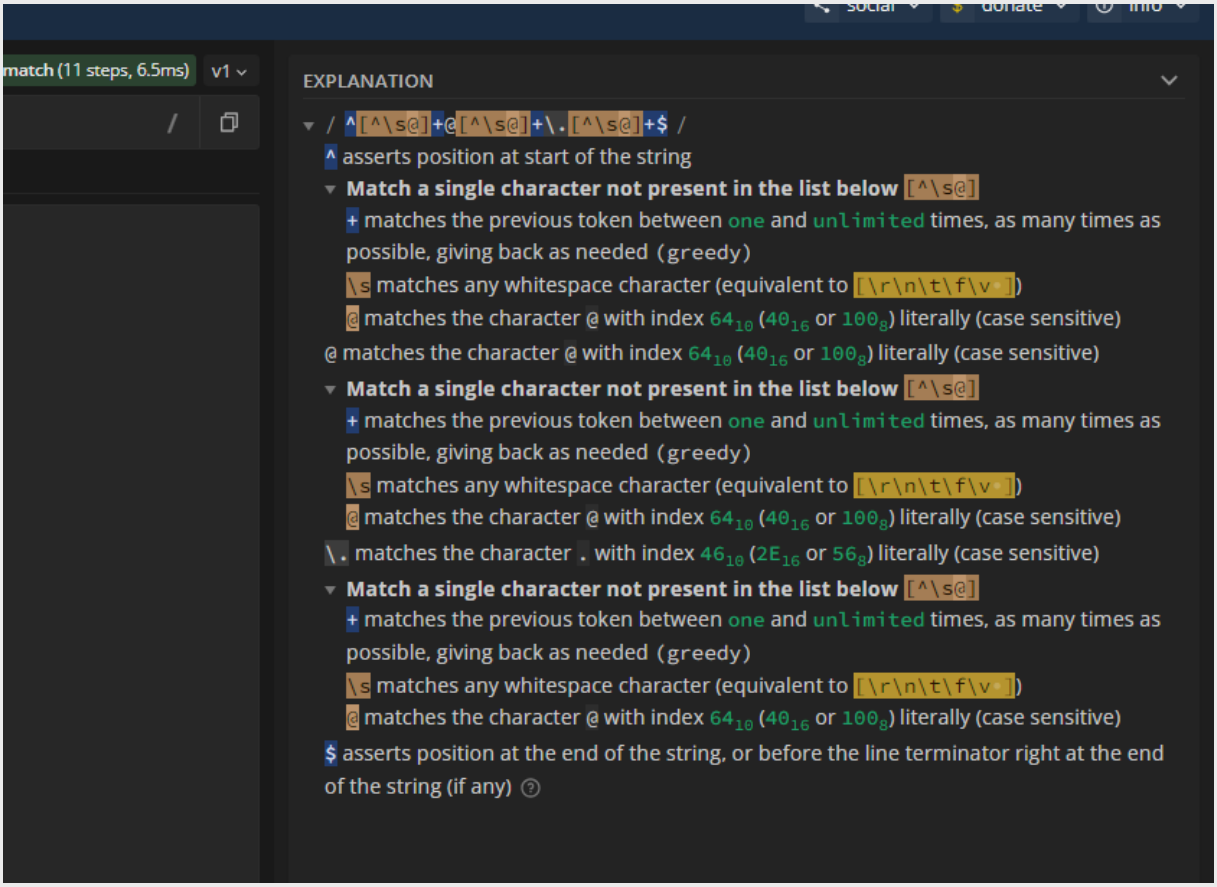

Image : ExplanationRegex.png

Outils utiles pour les regex

Des outils vous permettent de construire vos regex en comprenant ce que vous faites :

- Regular expressions 101[1]
- Untitled Pattern[2]
- ChatGPT

Méthode **Utiliser un regex en JS**

Vous pouvez utiliser votre regex en suivant la méthode suivante.

Tout d'abord, créez l'expression régulière en utilisant la notation littérale /.../ :

```
1 const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
```

Ensuite, utilisez l'expression régulière pour rechercher des adresses e-mail dans un texte en utilisant la méthode String.prototype.match() :

```
1 const text = "Bonjour, voici mon adresse e-mail : contact@example.com et une autre adresse : info@exemple.com";
2 const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
3 const matche = text.match(emailRegex);
```

1 https://regex101.com/
2 https://regexr.com/

La méthode match prend l'expression régulière emailRegex en argument et recherche toutes les occurrences correspondantes dans la chaîne de texte text. Elle renvoie un tableau contenant toutes les adresses e-mail trouvées dans le texte.

On peut donc faire différentes utilisations de la méthode match. On pourrait rechercher les différentes occurrences de notre regex dans un texte, ou nous pouvons aussi tester si une chaîne de caractère respecte un regex. Dans ce cas, le regex ne peut retourner qu'un seul élément. Donc on peut utiliser la variable matche comme un booléen.

```
const text = "Bonjour, voici mon adresse e-mail : contact@example.com et une autre adresse : info@exemple.com";
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
const matche = text.match(emailRegex);
if (matche) {
  console.log("Adresse e-mail valide !");
} else {
  console.log("Adresse e-mail invalide !");
}
```

Méthode **Vérifions les champs requis**

Nous devons donc vérifier le champ de mail, en utilisant un regex pour vérifier la validité du mail.

Nous obtenons donc ce résultat :

```
function validateMail(input){
    //Définir mon regex
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    const mailUser = input.value;
    if(mailUser.match(emailRegex)){
        input.classList.add("is-valid");
        input.classList.remove("is-invalid");
        return true;
    }
    else{
        input.classList.remove("is-valid");
        input.classList.add("is-invalid");
        return false;
    }
}
```

D. Vérifier le mot de passe

Vérifier un mot de passe

Plusieurs critères sont possibles pour vérifier un mot de passe, en fonction de vos préférences. Dans notre cas, le mot de passe est obligatoire, et doit avoir certaines normes de sécurité définies, pour éviter des mots de passe avec une sécurité trop faible.

Voici les règles que nous avons définies pour le mot de passe : au moins huit caractères, comprenant au moins une lettre majuscule, au moins une lettre minuscule, au moins un chiffre, au moins un caractère spécial.

Voici le regex permettant de vérifier ces conditions :

```
const passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[\W_])[A-Za-z\d\W_]{8,}$/;
```

Explication du regex :

- ^ : début de la chaîne.
- (?=.*[a-z]) : recherche d'au moins une lettre minuscule.

- (?=.*[A-Z]) : recherche d'au moins une lettre majuscule.
- (?=.*\d) : recherche d'au moins un chiffre.
- (?=.*[\W_]) : recherche d'au moins un caractère spécial (caractère non alphanumérique).
- [A-Za-z\d\W_]{8,} : recherche de huit caractères ou plus qui peuvent être soit des lettres (majuscules ou minuscules), soit des chiffres, soit des caractères spéciaux (caractères non alphanumériques).
- $: fin de la chaîne.

Exemple **Notre vérification de mot de passe**

Nous obtenons ce code :

```
function validatePassword(input){
    //Définir mon regex
    const passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[\W_])[A-Za-z\d\W_]{8,}$/;
    const passwordUser = input.value;
    if(passwordUser.match(passwordRegex)){
        input.classList.add("is-valid");
        input.classList.remove("is-invalid");
        return true;
    }
    else{
        input.classList.remove("is-valid");
        input.classList.add("is-invalid");
        return false;
    }
}
```

E. Vérifier la confirmation du mot de passe

Méthode **Confirmation du mot de passe**

Vérifier un champ de confirmation de mot de passe est assez simple. Il suffit, à chaque modification, de vérifier si le contenu du champ mot de passe est égal au contenu du champ de confirmation.

Voici le code nécessaire à cette étape :

```
function validateConfirmationPassword(inputPwd, inputConfirmPwd){
    if(inputPwd.value == inputConfirmPwd.value){
        inputConfirmPwd.classList.add("is-valid");
        inputConfirmPwd.classList.remove("is-invalid");
        return true;
    }
    else{
        inputConfirmPwd.classList.add("is-invalid");
        inputConfirmPwd.classList.remove("is-valid");
        return false;
    }
}
```

III. Gestion de la connexion et déconnexion

A. Connexion

Méthode

Nous allons simuler une gestion de connexion, qui sera amenée à évoluer dans le temps. Le but n'est pas d'avoir un module d'authentification abouti, mais de voir notre application réagir en mode connecté ou déconnecté. Nous allons pour cela faire ce qu'on appelle du bouchonnage.

Le bouchonnage sert à gérer de fausses données, au lieu de chercher des données réelles. Par exemple, pour la connexion, nous allons stocker en dur un mail et un mot de passe, et ce sera la seule manière que nous aurons pour l'instant de nous connecter.

Voici donc notre méthode de connexion :

```
const btnSingin = document.getElementById("btnSignin");

btnSingin.addEventListener("click", checkCredentials);

function checkCredentials(){
    //Ici, il faudra appeler l'API pour vérifier les credentials en BDD

    if(mailInput.value == "test@mail.com" && passwordInput.value == "123"){
        //Il faudra récupérer le vrai token
        const token = "lkjsdngfljsqdnglkjsdbglkjqskjgkfjgbqslkfdgbskldfgdfgsdgf";
        setToken(token);
        //placer ce token en cookie

        setCookie(RoleCookieName, "admin", 7);
        window.location.replace("/");
    }
    else{
        mailInput.classList.add("is-invalid");
        passwordInput.classList.add("is-invalid");
    }
}
```

Exemple **Méthodes pour les cookies**

Un cookie est un petit fichier de données qui est stocké dans le navigateur web de l'utilisateur lorsque ce dernier visite un site web. Il permet aux sites web de stocker des informations spécifiques liées à l'utilisateur et à sa navigation sur le site.

Il est stocké sous forme de chaîne de caractères, pour pouvoir récupérer ses données, il nous faut donc écrire des méthodes nous permettant de découper ces chaînes de caractères pour récupérer les bonnes données.

Nous pouvons utiliser ces méthodes :

```
function setCookie(name,value,days) {
    var expires = "";
    if (days) {
        var date = new Date();
        date.setTime(date.getTime() + (days*24*60*60*1000));
        expires = "; expires=" + date.toUTCString();
    }
    document.cookie = name + "=" + (value || "")  + expires + "; path=/";
}

function getCookie(name) {
    var nameEQ = name + "=";
```

```
    var ca = document.cookie.split(';');
    for(var i=0;i < ca.length;i++) {
        var c = ca[i];
        while (c.charAt(0)==' ') c = c.substring(1,c.length);
        if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length,c.length);
    }
    return null;
}

function eraseCookie(name) {
    document.cookie = name +'=; Path=/; Expires=Thu, 01 Jan 1970 00:00:01 GMT;';
}
```

Méthode **Gestion de la connexion**

Maintenant que nous pouvons gérer les cookies, nous voulons stocker en cookie notre token. C'est la présence de ce token qui définit si nous sommes connectés ou non. Nous pouvons donc tout simplement utiliser les méthodes créées au préalable, pour stocker et récupérer le cookie contenant le token.

```
const tokenCookieName = "accesstoken";

function setToken(token){
    setCookie(tokenCookieName, token, 7);
}

function getToken(){
    return getCookie(tokenCookieName);
}
```

Méthode **Savoir si on est connecté**

Nous pouvons désormais créer une méthode retournant un booléen, nous indiquant si nous sommes connectés ou non.

```
function isConnected(){
    if(getToken() == null || getToken == undefined){
        return false;
    }
    else{
        return true;
    }
}
```

B. Déconnexion

Gestion du rôle

Tout comme pour la connexion, nous allons bouchonner une gestion des rôles. C'est-à-dire que nous allons gérer un rôle en dur dans le code. Nous pouvons aussi ajouter des méthodes de gestion de rôle.

```
const RoleCookieName = "role";

function getRole(){
    return getCookie(RoleCookieName);
}
```

Dans le fichier signin.js, nous pouvons ajouter le rôle en cookie. Nous utilisons ce système temporairement, pour simuler la gestion du rôle de l'utilisateur.

```
        setCookie(RoleCookieName, "admin", 7);
```

Méthode **Gérer la déconnexion**

Pour déconnecter un utilisateur, il suffit de supprimer le cookie, et d'actualiser. Nous pouvons donc exécuter cette méthode au clic sur le bouton déconnexion dans le menu.

```
function signout(){
    eraseCookie(tokenCookieName);
    window.location.reload();
}
```

C. Afficher/masquer des éléments sur la page en fonction du rôle

Méthode **Afficher ou masquer des éléments en fonction du rôle**

Selon le rôle de l'utilisateur, certains éléments doivent être masqués, ou affichés. Par exemple, dans le menu, 'connexion' doit être affiché seulement si on est déconnecté, et 'déconnexion' uniquement si on est connecté.

Pour savoir pour quelles personnes ont accès à un noeud HTML, nous allons placer cette information sur notre noeud html, grâce à un data attribute.

Définition **data attribute**

Un « *data attribute* » (ou attribut de données) est un type d'attribut HTML personnalisé qui permet de stocker des données supplémentaires au sein d'un élément HTML. Ces attributs sont très utiles pour associer des données spécifiques à un élément HTML sans affecter le comportement standard de l'élément ni l'affichage à l'écran.

Un « *data attribute* » est défini en préfixant le nom de l'attribut avec le mot-clé `data-`. Le reste du nom peut être choisi librement, en suivant les règles habituelles pour les attributs HTML (par exemple, sans espaces ou caractères spéciaux). La valeur de l'attribut peut être définie pour stocker des données, qui peuvent être du texte, des nombres, des objets JSON, etc.

Voici notre data attribute, pour le lien 'connexion' :

```
<li class="nav-item" data-show="disconnected">
                        <a class="nav-link" href="/signin">Connexion</a>
                    </li>
```

Méthode **data attribute et JS**

Pour accéder aux « *data attributes* » en JavaScript, vous pouvez utiliser la propriété `dataset` de l'élément. La propriété `dataset` renvoie un objet contenant les « *data attributes* » sous forme de paires clé-valeur.

Voici comment utiliser les « *data attributes* » en JavaScript :

```
// Récupérer l'élément HTML
const myElement = document.getElementById('myElement');

// Accéder aux "data attributes" en utilisant la propriété dataset
const roleToShow = myElement.dataset.show; // "disconnected"
```

Méthode **Nos rôles**

Nous allons gérer quatre rôles, respectivement :

- Disconnected
- Connected (admin ou client)
- Admin
- Client

Il faut donc passer sur toute notre application, pour définir les éléments affichés en fonction du rôle.

Par exemple, nos boutons connexion et déconnexion :

```
<li class="nav-item" data-show="disconnected">
                    <a class="nav-link" href="/signin">Connexion</a>
                  </li>
                  <li class="nav-item" data-show="connected">
                    <button class="nav-link" id="signout-btn">Déconnexion</button>
                  </li>
```

Méthode

Maintenant que nos noeuds HTML contiennent le rôle pour lequel ils sont ciblés, nous devons les afficher ou les masquer après le chargement de la page.

Nous devons donc écrire une méthode, exécutée juste après avoir chargé le contenu de la page.

Juste après le changement du titre de la page, nous pouvons donc exécuter cette méthode :

```
// Changement du titre de la page
  document.title = actualRoute.title + " - " + websiteName;

  //Afficher et masquer les éléments en fonction du rôle
  showAndHideElementsForRoles();
```

Cette méthode devra récupérer tous les éléments ayant le data attribute ‘data-show’. Nous pouvons les récupérer via ce sélecteur :

```
let allElementsToEdit = document.querySelectorAll('[data-show]');
```

Une fois que nous avons récupéré tous ces éléments, nous devons en fonction de la valeur de ce data attribute, et de la valeur de notre rôle, masquer l’élément.

Voici le code de cette fonction, exécutée après le chargement de la page.

```
function showAndHideElementsForRoles(){
    const userConnected = isConnected();
    const role = getRole();

    let allElementsToEdit = document.querySelectorAll('[data-show]');

    allElementsToEdit.forEach(element =>{
        switch(element.dataset.show){
            case 'disconnected':
                if(userConnected){
                    element.classList.add("d-none");
                }
                break;
            case 'connected':
                if(!userConnected){
                    element.classList.add("d-none");
                }
                break;
```

```
            case 'admin':
                if(!userConnected || role != "admin"){
                    element.classList.add("d-none");
                }
                break;
            case 'client':
                if(!userConnected || role != "client"){
                    element.classList.add("d-none");
                }
                break;
        }
    })
}
```

D. Authentification dans le routage

Méthode

Certains éléments sont affichés ou masqués. Mais cela n'est pas suffisant, il nous faut restreindre l'accès à certaines pages en fonction de notre rôle. Pour ça, il faut modifier notre classe Route, pour spécifier pour chaque route quels sont les rôles autorisés.

```
export default class Route {
    constructor(url, title, pathHtml, authorize, pathJS = "") {
      this.url = url;
      this.title = title;
      this.pathHtml = pathHtml;
      this.pathJS = pathJS;
      this.authorize = authorize;
    }
}

/*
[] -> Tout le monde peut y accéder
["disconnected"] -> Réserver aux utilisateurs déconnecté
["client"] -> Réserver aux utilisateurs avec le rôle client
["admin"] -> Réserver aux utilisateurs avec le rôle admin
["admin", "client"] -> Réserver aux utilisateurs avec le rôle client OU admin
*/
```

Nous pouvons désormais modifier toutes nos routes, pour limiter leurs accès aux rôles définis.

```
//Définir ici vos routes
export const allRoutes = [
    new Route("/", "Accueil", "/pages/home.html", []),
    new Route("/galerie", "La galerie", "/pages/galerie.html", [], "/js/galerie.js"),
    new Route("/signin", "Connexion", "/pages/auth/signin.html", ["disconnected"],
"/js/auth/signin.js"),
    new Route("/signup", "Inscription", "/pages/auth/signup.html", ["disconnected"],
"/js/auth/signup.js"),
    new Route("/account", "Mon compte", "/pages/auth/account.html", ["client", "admin"]),
    new Route("/editPassword", "Changement de mot de passe", "/pages/auth/editPassword.html",
["client", "admin"]),
    new Route("/allResa", "Vos réservations", "/pages/reservations/allResa.html", ["client"]),
    new Route("/reserver", "Réserver", "/pages/reservations/reserver.html", ["client"]),
];
```

Maintenant, dans notre fichier router.js, nous devons, avant de charger le contenu d'une page en fonction de l'url, vérifier si l'utilisateur a bien le droit d'accéder à cette route. Il suffit pour cela de vérifier si le rôle de l'utilisateur est présent dans le tableau 'authorize' de notre route.

```
const path = window.location.pathname;
  // Récupération de l'URL actuelle
  const actualRoute = getRouteByUrl(path);

  //Vérifier les droits d'accès à la page
  const allRolesArray = actualRoute.authorize;

  if(allRolesArray.length > 0){
    if(allRolesArray.includes("disconnected")){
      if(isConnected()){
        window.location.replace("/");
      }
    }
    else{
      const roleUser = getRole();
      if(!allRolesArray.includes(roleUser)){
        window.location.replace("/");
      }
    }
  }
```

IV. Essentiel

Dans ce cours, nous avons dynamisé notre application à l'aide de JavaScript. Pour commencer, nous avons pu effectuer une validation en direct d'un formulaire. Grâce à ça, l'utilisateur peut voir ses erreurs en direct, et n'est pas frustré de voir que les données qu'il a voulu valider sont erronées. Nous avons ensuite pu intégrer l'authentification dans le module de routage de notre application, gérant ainsi l'accès aux pages et aux différents éléments en fonction du rôle de l'utilisateur.

Nous ne sommes pas sur la version finale de notre application, mais visuellement, nous commençons à avoir un rendu qui se rapproche du rendu final.

V. Pour aller plus loin

- **Ajouter un loader**. Le chargement de la page n'étant pas très fluide, nous pourrions décider d'ajouter un loader. Il se masquerait après avoir appelé la méthode 'showAndHideElementsForRoles()'.
- **Valider les autres formulaires** de notre application.