

○ ○ ○ ○

PRESENTATION

# **SÉCURITÉ WEB**

○ ○ ○ ○

# ***TABLE DES MATIÈRES***

- Démontrer une injection SQL via un jeu de formulaires
- Démontrer une faille XSS
- Créer une système d'authentification forte via OTP email.





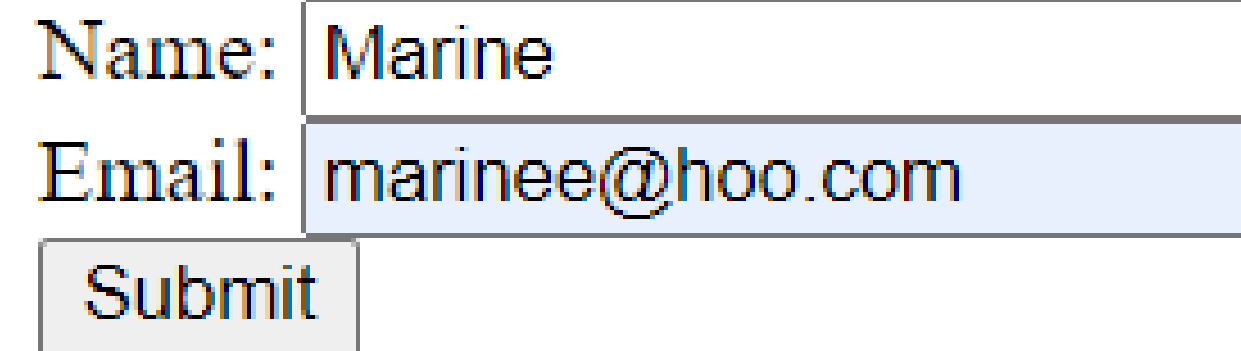
# DÉMONTRER UNE INJECTION SQL VIA UN JEU DE FORMULAIRES

## UTILISATION NORMALE

Notre formulaire est le suivant. Il comprend un nom et un email à rentrer.

Après avoir appuyé sur le bouton submit, notre code va rechercher si l'utilisateur, (ici Marine), est bien présent dans la base de données.

Si il l'est, il renvoi les informations.



Name:

Email:

```
[{"nom": "Marine", "email": "marinee@hoo.com"}]
```

# DÉMONTRER UNE INJECTION SQL VIA UN JEU DE FORMULAIRES

## UTILISATION DEVIANTE

Ici on injecte à la place de l'Email, la requête suivante :

```
%' UNION select * from bdd_mdp; --
```

En effet, la base de données bdd\_mdp contient déjà des mots de passes pour les utilisateurs.

En rentrant cette commande, on obtient le mot de passe de Marine.

Name:	<input type="text" value="Marine"/>
Email:	<input type="text" value="%' UNION select * from bdc"/>
<input type="button" value="Submit"/>	

```
[{"nom": "Marine", "email": "ceci_est_mon_mot_de_passe"}]
```

# DÉMONTRER UNE INJECTION SQL VIA UN JEU DE FORMULAIRES

## CONTRE CELA

Pour contrer cela, on programme notre appel à la base de données différemment.

## ANCIEN CODE

```
const query = `SELECT * FROM personne WHERE nom = '${name}' AND email = '${email}'`;
```

## NOUVEAU CODE



```
const query = {  
  text: 'SELECT * FROM personne WHERE nom = $1 AND email = $2',  
  values: [name, email]  
};
```

# DÉMONTRER UNE INJECTION SQL VIA UN JEU DE FORMULAIRES

## CONTRE CELA

Maintenant lorsque l'on appelle une requête dans le champ d'un formulaire, la requête ne s'exécute pas:

Name:	Marine
Email:	%' UNION select * from bdc
<input type="button" value="Submit"/>	



[]

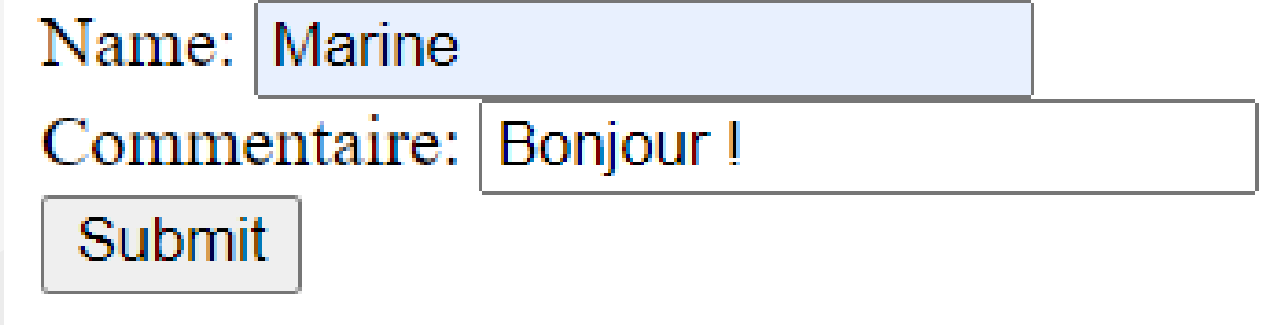
# ***DEMONTRER UNE FAILLE XSS***

# DÉMONTRER UNE FAILLE XSS

## UTILISATION NORMALE

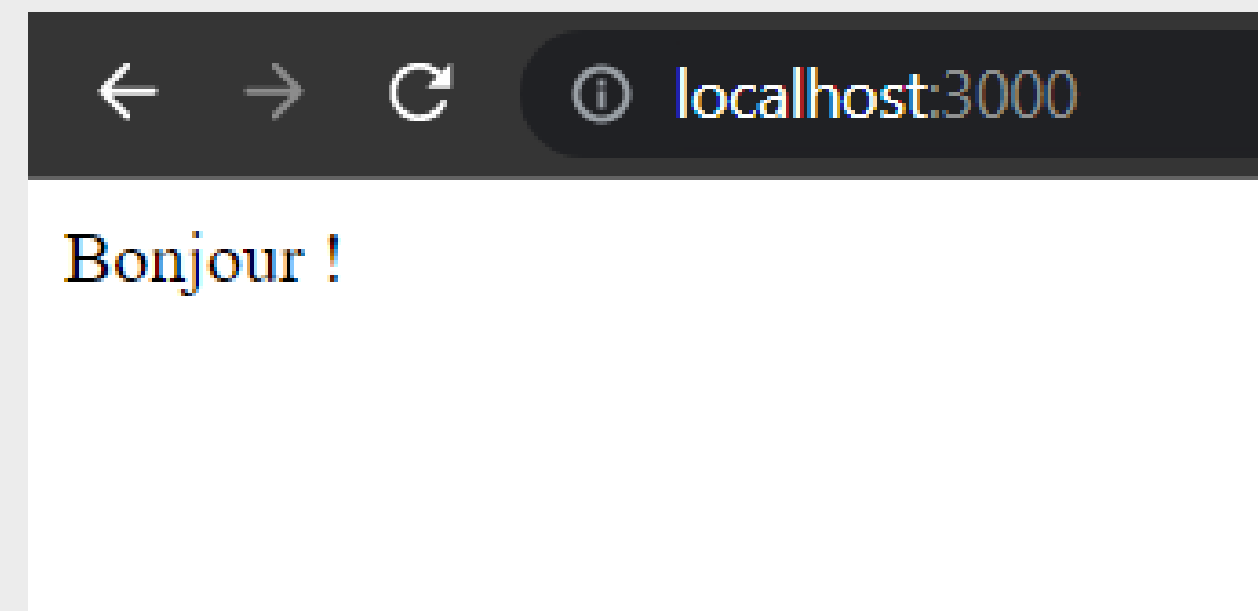
Notre formulaire est le suivant. Il comprend un nom et un commentaire à rentrer.

Après avoir appuyé sur le bouton submit, notre code va juste envoyer le commentaire sur la page.



Name:

Commentaire:





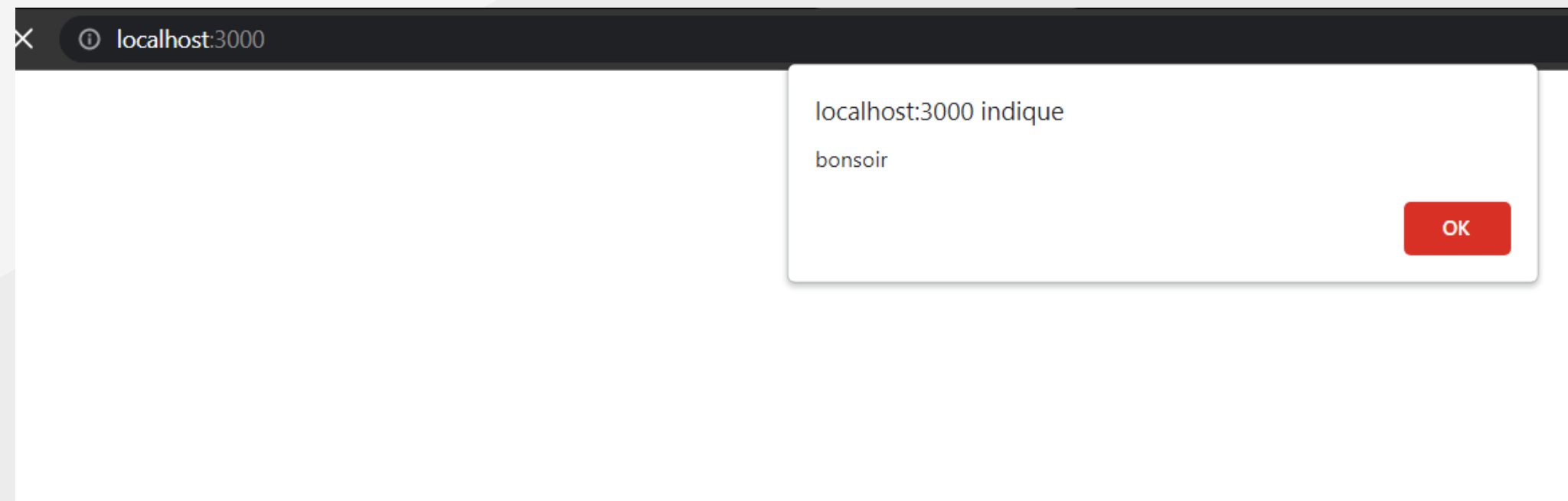
# DÉMONTRER UNE FAILLE XSS

## UTILISATION DEVIANTE

En faille XSS, on va mettre injecter du code JavaScript dans la zone de commentaire :

Bonjour, `<script>alert("bonsoirs")</script>` !

Cela nous affichera quelque chose que l'utilisateur n'a pas demandé, soit un pop up.



# DÉMONTRER UNE FAILLE XSS

## CONTREER CETTE ATTAQUE

Pour cela, en Node.js, nous allons utilisé validator.

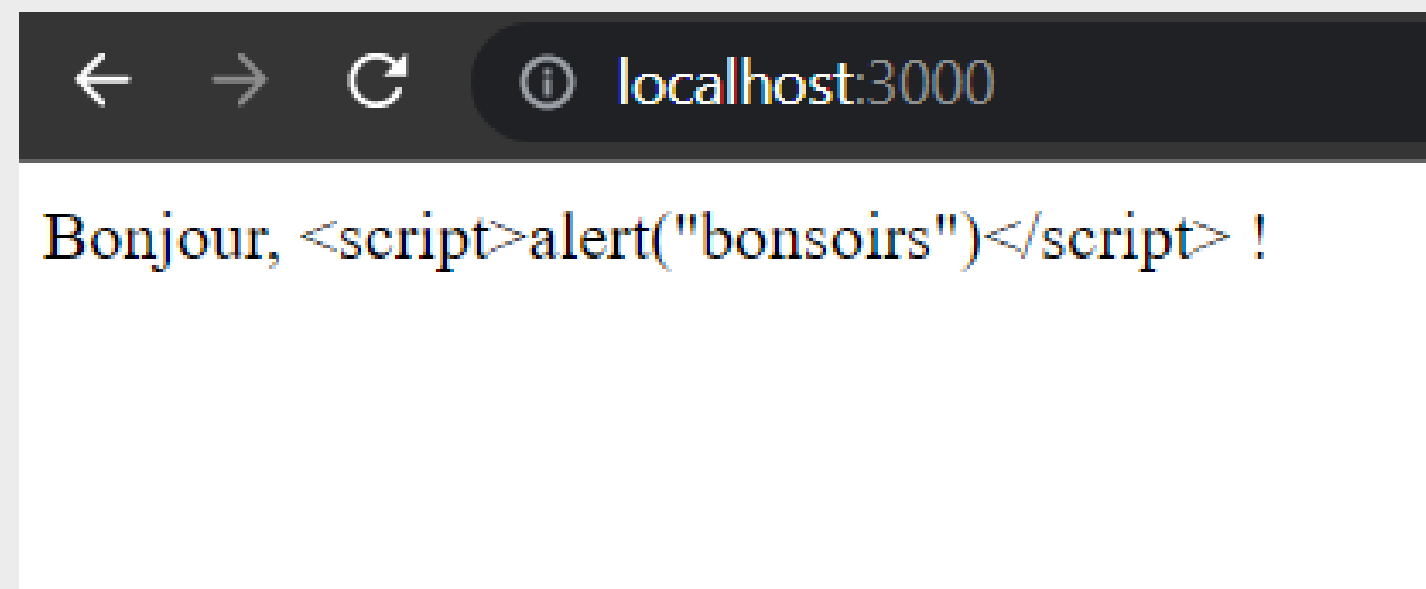
```
var commentaire_valide = validator.escape(commentaire);  
res.send(commentaire_valide);
```

Ce morceau de code va permettre de détecter si dans le champ "commentaire" de notre formulaire, est injecté autre chose que du string, ou un type accepté. En réalisant un `console.log(commentaire_valide)`, on remarque que notre injection de la page précédente (Bonjour, `<script>alert("bonsoirs")</script> !`), est sous la forme :

```
Bonjour, &lt;script&gt;alert(&quot;bonsoirs&quot;)&lt;&#x2F;script&gt; !
```

En effet, validator bloque les caractères non désirés comme `.`, `\`, `&quot;` etc...

Ainsi, dans notre page web, nous n'avons plus de pop up, mais un commentaire simple :



# ***CRÉER UNE SYSTÈME D'AUTHENTIFICATION FORTE VIA OTP EMAIL***

# AUTHENTIFICATION FORTE

Pour cette partie 3, nous créer un **nouveau** formulaire. L'utilisateur pourra désormais se connecter à son compte avec son email et son mot de passe.

## Formulaire de connexion

Mail :  Password:

Et **réarranger** notre base de données :

```
formulaire=# select * from identite_personne ;
      mail      | password | otp
-----+-----+-----
marinelangrez@outlook.fr | $2b$10$.udgYj9rG5joUYS3u7G5.uXLQWRJ0/Pzs.HPRG9YsS4HaVpTPTWx2 | 7CQN7H
(1 row)
```

Dans notre base de données, on a un mot de passe **déjà haché** (ce qui renforce la sécurité et nous aide pour la partie 4). Et également, une colonne OTP, qui sera **modifiée** à chaque nouveau test d'authentification de l'utilisateur.

# AUTHENTIFICATION FORTE

Quand l'utilisateur essaye de se connecter :

## Formulaire de connexion

Mail :  Password:

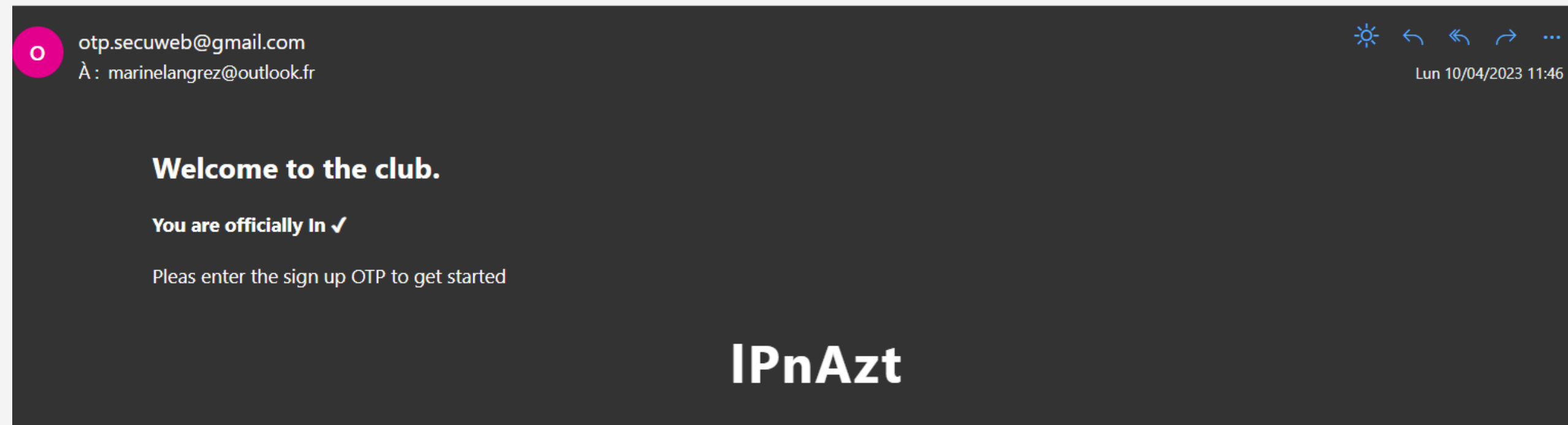
Le programme va **vérifier** que le **mot de passe est correct** avec celui de la base de données.  
=> Utilisation de la fonction `compare` (du package `bcrypt`) pour voir si le mot de passe rentré correspond avec celui haché de la bdd.

```
const validPassword = await bcrypt.compare(password_formulaire, motdepassebdd);
```

si le mot de passe est valide ; le code va **générer un OTP**, et **modifier** celui déjà existant dans la base de données par le nouveau.

# ***AUTHENTIFICATION FORTE***

Le nouveau OTP a été envoyé sur le **mail**, l'utilisateur recevra :



Il va par la suite être **rediriger sur un autre formulaire** qui va **vérifier** la validité de son OTP.



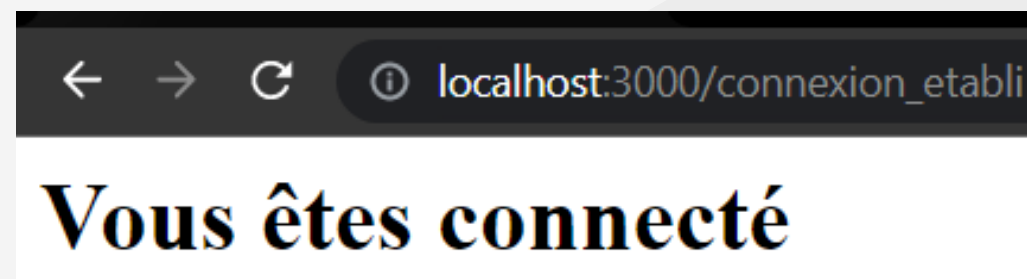
# ***AUTHENTIFICATION FORTE***

L'utilisateur devra rentrer le nouvel OTP reçu par mail sur cette page :

## OTP

Mail:  OTP :

Si l'OTP est validé, il sera sur la page :



***CONSTRUIRE UN  
FORMULAIRE AVEC  
CONTRÔLE FRONT ET  
CONTRÔLE MIDDLE DES  
DONNÉES***

# CONTRÔLE FRONT ET CONTRÔLE MIDDLE

Une fois que l'utilisateur est connecté, il a la possibilité de changer le mot de passe de son compte :

**Vous êtes connecté**

[Modifier le compte](#)

En cliquant sur Modifier le compte, l'utilisateur aura un formulaire de modification :

## Formulaire de modification

Mail :  Ancien Password:  New Password:

# CONTRÔLE FRONT ET CONTRÔLE MIDDLE

Nous avons décidé dans ce formulaire de rajouter un champ "ancien mot de passe" pour renforcer la sécurité et éviter qu'un autre utilisateur change le mot de passe d'un compte avec autant de facilité.

Le champ "ancien mot de passe" du formulaire est donc vérifié avec l'ancien mot de passe de la base de données (même principe que pour la connexion => fonction compare car les mots de passes sont hachés dans la bdd)

```
let ancien_pwrdd_bdd = result.rows[0].password
const validPassword = await bcrypt.compare(ancien_password, ancien_pwrdd_bdd);
```

# CONTRÔLE FRONT ET CONTRÔLE MIDDLE

Si cette vérification est effectuée, on peut modifier le mdp du compte avec le nouveau mot de passe choisie par l'utilisateur. On va donc utiliser un sel, et la fonction hash (pour haché la mot de passe et l'update dans la base de données)

```
if (validPassword) {
  const salt = await bcrypt.genSalt(10);
  let hashedpassword = await bcrypt.hash(newpassword, salt);

  const query_update = {
    text: 'UPDATE identite_personne SET password = $1 WHERE mail = $2;',
    values: [hashedpassword, email]
  };

  pool.query(query_update);
}
```

# ***CONTRÔLE FRONT ET CONTRÔLE MIDDLE***

Une fois que l'opération est effectuée, l'utilisateur retombe sur la même page :

**Vous êtes connecté**

[Modifier le compte](#)



**METTRE EN PLACE UN  
CHIFFREMENT  
APPLICATIF (LES  
DONNÉES STOCKÉE  
DOIVENT ÊTRE  
CHIFFRÉES DE MANIÈRE  
RÉVERSIBLE)**



# ***DONNÉES STOCKÉE DOIVENT ÊTRE CHIFFRÉES DE MANIÈRE RÉVERSIBLE***

Lors de nos dernières manipulations (aux dernières parties), nous avons déjà choisi de garder des données, des mots de passes cryptés dans notre base de données.

Ainsi, quand nous modifions notre mot de passe, celui-ci était chiffré avant d'être envoyé dans la base de données. Et déchiffré quand nous devions le vérifier (quand un utilisateur se connecte)

