AJAX et Symfony

AJAX, qu'est-ce que c'est? Formellement l'acronyme vient de **A**synchronous **Ja**vascript And **X**ML. La portion importante de cet acronyme est le **asynchronous**.

Que veut dire asynchrone dans le contexte client-serveur du Web?

Une requête synchrone : le client demande une page au serveur et **il attend** la réponse. Son interface utilisateur est **bloquée** tant qu'il ne l'a pas obtenue. Lorsque la réponse arrive la page est réaffichée au complet.

Une requête asynchrone : le client (le fureteur) émet une requête au serveur mais **ne bloque pas** son interface utilisateur. Plutôt, le client indique quoi faire lorsque la réponse arrivera, c'est-à-dire qu'il fournit une fonction à exécuter lorsque la réponse arrivera. Pendant ce temps l'utilisateur n'a pas ressenti la lourdeur d'envoyer une requête au serveur. La réponse arrivera lorsqu'elle arrivera. Le temps réponse d'une requête Web est très variable (d'une fraction de seconde à quelques minutes) tout dépend de ce que l'on demande au serveur et de la charge de celui-ci. Quand la réponse arrive, on exécute en arrière-plan la fonction que le client avait fourni lors de l'appel asynchrone. La page n'est pas réaffichée au complet, seuls les éléments concernés le sont, ce qui est beaucoup plus fluide et confortable pour l'utilisateur.

Une requête synchrone est plus simple mais moins ergonomique.

Une requête asynchrone est plus complexe mais plus ergonomique.

Asynchronisme et Babilon

Au moins trois zones de Babilon bénéficieraient de la touche magique de AJAX :

1- Mise en panier d'un produit
2- Vérification de l'unicité de l'utilisateur lors de l'inscription
3- Rafraîchissement du compte à rebours des annulations de commandes

Analysons-les à tour à tour :

Mise en panier d'un produit

Situation : Vous désirez deux produits qui se trouvent vers la fin du catalogue. Vous cliquez sur l'icône du premier pour le mettre dans le panier. Ceci provoque une requête au serveur.

Version synchrone : Le serveur reçoit la requête, met un produit dans le panier, et provoque le rafraichissement complet de la page du catalogue. Maintenant vous devez scroller encore une fois jusqu'à la fin du catalogue pour mettre le deuxième produit dans le panier.

Version asynchrone : Le fureteur envoi une requête AJAX au serveur, et indique quelle fonction exécuter lorsque la réponse à la requête arrivera. Le serveur reçoit la requête, met un produit dans le panier et envoi sa réponse. L'interface utilisateur du fureteur n'a pas bougé, la fonction à exécuter à la réception de la réponse du serveur est un rafraichissement d'une très petite portion de l'interface utilisateur : le compteur d'items du panier. Le confort de l'utilisateur est beaucoup plus grand car il se trouve au même endroit où il était et il peut ajouter immédiatement le deuxième produit au panier.

Technique de développement de la version asynchrone :

Dans le twig d'entête on avait déjà une zone pour afficher le compteur d'items au panier: voir le id en vert:

```
{#entete.html.twig#}
<div class="row" style="background: black;">
  <a href="{{ path('racine') }}">
    <img width="500" src="{{ asset('img/logoBabylon.jpg') }}" alt="logo”>
  </a>
</div>

<!-- row de la barre de navigation -->
. . .
  <ul class="navbar-nav">
    <li class="nav-item"><a class="nav-link" href="{{ path('contact')
                                                  }}">Contact</a>
    </li>
    . . .
    <li class="nav-item"><a class="nav-link" href="{{ path('panier_affiche')
                                                   }}">Panier</a></li>
  </ul>
  <!-- Navbar text-->
    <span class="navbar-text" id="compteur">{{ nbItems }} items</span>
.   . .
```

Dans le twig de produit on associe la classe CSS “ajouter_panier” au bouton-icone panier:

```
{# produit.html.twig#}
...
{% for produit in produits %}

   <div class="unProduit col-3">
 . . .
       <div class="col-2">
         <button id="{{produit.id}}" class="ajouter_panier">
            <img width="40" class="imgPanier" alt="ajouter au panier"
                  src="{{asset('img/panier.jpg')}}"/>
```

```
          </button>
        </div>
    </div>
{% endfor %}
```

Dans un script JavaScript on définit ce qu'on fait lorsqu'on clique sur le bouton-icône panier. Le code est écrit en JQuery. À chaque fois que la page est chargée, *$(document).ready,* on associe une fonction qui se déclenchera lors d'un clic sur un des bouton-icône. Cette fonction

1- Récupérera l'id du produit cliqué :
var id = this.getAttribute('id');
2- Construira une URL qui correspond à une URL routée de notre contrôleur :
var url = "ajouter_produit_panier/" + id;
3- Fera un appel AJAX à cette URL :
$.get(url,
4- Fournira la function à executer lorsque la réponse du serveur arrivera:
function(data, status){
$('#compteur').text(data);
C'est-à-dire que le texte de l'élément compteur sera mis à jour avec le contenu de la réponse du serveur

```
//biblio.js
$(document).ready(function(){
  $(".ajouter_panier").click(function(){
    var id = this.getAttribute('id');
    var url = "ajouter_produit_panier/" + id;
    $.get(url, function(data, status){
      $('#compteur').text(data);
    });
  });
});
```

Notre contrôleur doit offrir une route appelée par le clic sur le bouton-icône. Quand cette route est exécutée on vérifie si la requête provient bien d'une requête AJAX :
if ($request->isXmlHttpRequest()
Si c'est le cas on compte les items présents dans le panier et on retourne une *new Response* qui ne contient qu'un simple texte : le nombre d'items dans le panier!

```
//panierController.php
. . .
#[Route('/ ajouter_produit_panier/{id}', name:'ajouter_produit_panier')]
public function ajouterProduitPanierAction(Request $request)
{
  if ($request->isXmlHttpRequest())
  {
    // on récupère la session
    $panier = $request->getSession()->get('panier');

    // si le panier est initialisé et qu'il contient le produit
    if ($panier)
    {
. . .
      }
     }
```

```
    else
    {
      $panier = new Panier();
. . .
    }
  }
  $request->getSession()->set('panier', $panier);
  return new Response($panier->compteItems() . " items");
  }
}
. . .
```

Vérification de l'unicité d'un nom d'utilisateur

Situation : Quand on remplit le formulaire d'inscription on donne au début un nom d'utilisateur.

Version synchrone : Ensuite on remplit les autres informations et on clique sur « soumettre ». Ce n'est qu'à ce moment qu'on communique avec le serveur pour qu'il valide nos informations. Si on a fourni un nom d'utilisateur qui malencontreusement se trouvait déjà en BD ce n'est qu'à ce moment qu'on le saura. C'est frustrant pour l'utilisateur.

Version asynchrone : Dès que le champ du nom d'utilisateur est rempli on envoie une requête asynchrone au serveur qui la traite pendant qu'on remplit les autres champs et si le nom d'utilisateur est déjà pris on avise notre futur-client en temps réel. Il n'aurait pas besoin de se rendre jusqu'à la soumission du formulaire pour être mis au courant de l'erreur.

Technique de développement de la version asynchrone :

Dans le twig du formulaire d'inscription on ajoute un évènement **onBlur** au champ qui saisit le nom d'utilisateur proposé (OnBlur est un évènement HTML déclenché lorsqu'on quitte un champ de saisi) on associe à cet évènement la fonction JavaScript *validerUniciteUtilisateur()*. Il faut aussi ajouter notre fichier JavaScript contenant cette fonction

<script src="{{ asset('js/validerUniciteUtilisateur.js') }}"></script> :

```
{# client.html.twig
. . .
{{ form_start(form) }}
                    {% if contexte == 'CREATION' %}
                    <div class="row">
                       <section class='col-1'>
              {{ form_label(form.utilisateur, "Utilisateur:")}}
                       </section>

              {{ form_widget(form.utilisateur, {attr:{'class':'col-4 form-
control', 'onBlur':'validerUniciteUtilisateur()'}} ) }}

              <section class='rouge'>
                         {{ form_errors(form.utilisateur) }}
                       </section>
            </div>
            {% endif %}
. . .
{% block javascripts %}
    <script src="{{ asset('js/validerUniciteUtilisateur.js') }}"></script>
```

Voici le code JavaScript en question :

1- On récupère le nom proposé comme utilisateur
var NomUtilisateur = $("#client_utilisateur").val();
2- On appelle une requête AJAX codée en JQuery
$.get("validerUnicite?utilisateur=" + NomUtilisateur,
Le premier paramètre est l'URL appelée par la requête AJAX
3- Le deuxième paramètre est l'action à exécuter lorsque la réponse du serveur arrivera :
function(data, status){
if (data == "doublon")
alert("Attention! Ce nom d'utilisateur est déjà pris");
4- Finalement si la réponse du serveur est « doublon » on affiche un message avertissant l'utilisateur que son nom candidat est déjà utilisé

```
//validerUniciteUtilisateur.js
. . .
function validerUniciteUtilisateur()
{
  var NomUtilisateur = $("#client_utilisateur").val();
  $.get("validerUnicite?utilisateur=" + NomUtilisateur, function(data, status)
  {
    if (data == "doublon")
      alert("Attention! Ce nom d'utilisateur est déjà pris");
  });
}
```

Voici la route du contrôleur correspondant à la requête AJAX :

```
#[Route('/validerUnicite')]
public function validerUnicite(ManagerRegistry $doctrine, Request $request):
Response
{
   $reponse = "unique";
   if ($request->isXmlHttpRequest())
   {
     $nomUtilisateur = $request->query->get('utilisateur');
     $em = $doctrine->getManager();
     $client = $em->getRepository(Client::class)
                              ->findOneBy(['utilisateur'=>$nomUtilisateur]);
     if ($client)
        $reponse="doublon";
     return new Response($reponse);
    }
    die('Injection URL');
 }
```

Rafraichissement du compte à rebours

Situation : Dans l'historique des achats, le compte à rebours se met à jour périodiquement, disons à chaque 5 secondes. Imaginez qu'il y ait beaucoup de commandes et qu'on scrolle pour voir la plus ancienne. Juste comme j'arrive sur la commande que je voulais vérifier le timer de 5 secondes est déclenché.

Version synchrone : La page se rafraichit et je me retrouve en haut de la liste des commandes : frustration.

Version asynchrone : L'appel de rafraichissement se fait par Ajax, je demeure donc dans le bas de la page et je peux lire tranquillement les infos sur ma plus vieille commande sans être toujours déplacé au début de page à chaque 5 secondes.

Technique de développement de la version asynchrone :

Dans le contrôleur voici le code pour afficher l'historique de mes achats :

```
// CommandeController.php
. . .
#[Route('/commande_historique', name:'commande_historique')]
public function 'commande_historique (Request $req)
{
. . .
   return $this->render("commande/historique.html.twig",
       array('nbItems'         => $nbItems,
             'client_connecte' => $clientRepo,
             'tabCmdDelai'     => $tabCmdDelai,
             'nbItems'         => $nbItems,
             'messageEclair'   => $messageEclair));
}
```

Voici le twig rendu :

```
{# historique.html.twig #}
{% extends 'base.html.twig' %}
{% block principal %}
<div class="row">
  <div id="page" class="col-8">
   {% if tabCmdDelai|length > 0 %}
        <H2>Historique de vos commandes</h2>
. . .
{% endblock %}
{% block javascripts %}
   <script src="{{ asset('js/biblio1.js') }}"></script>
   <script src="{{ asset('js/TimerAnnulationAjax.js') }}"></script>
{% endblock %}
```

Remarquez la <*div id=*"*page*" (en vert), c'est elle qui sera rafraichie de façon asynchrone.

Voici le fichier JavaScript contenant le mécanisme du timer

```
//TimerAnnulationAjax.js
function startAutoRefresh(){
  setInterval(function(){DelaiMessageHandler()},5000);
}

function DelaiMessageHandler() {
    $.get("commande_historique_ajax", function(data, status){
            $('#page').html(data);
    });
}
```

La fonction **startAutoRefresh**() est associée à l'évènement **onLoad** de l'élément **<body>.** À tous les 5000 millisecondes la fonction DelaiMessageHandler() sera appelée. C'est cette dernière fonction qui fera l'appel AJAX ($.get). Le premier paramètre est l'URL appelé et le second la fonction à exécuter lorsque le serveur aura répondu. Cette fonction remplace le HTML de l'élément page par le HTML envoyé par le serveur.

Voici le contrôleur et la route invoquée par la requête AJAX : *commande_historique_ajax.*

Cette route est presque identique à *commande_historique* à l'exeption qu'elle render un autre twig : *historique_ajax.html.twig:*

```
// CommandeController.php
/**
 * @Route("/commande_historique_ajax",  name="commande_historique_ajax")
 */
 public function commandeHistoriqueAjaxAction(Request $req)
    {
       if ($req->isXmlHttpRequest())
. . .

        return $this->render("commande/historique_ajax.html.twig",
          array('nbItems'         => $nbItems,
             'client_connecte' => $clientRepo,
             'tabCmdDelai'     => $tabCmdDelai,
             'nbItems'         => $nbItems,
             'messageEclair'   => $messageEclair));
      }
      else
      {
        // injection URL on ne vient pas d'un appel AJAX
        return $this->RedirectToRoute('racine');
      }
    }
```

Voici le twig *historique_ajax.html.twig*. Il ressemble à *historique.html.twig* en plus simple: il n'y ni *extends* ni *block*. Il n'y a que le HTML présenté par l'élément dont le id est *page*

```
{# historique_ajax.html.twig #}
{% if tabCmdDelai|length > 0 %}
        <H2>Historique de vos commandes</h2>
. . .
{%endif%}
```