Ajax (parte 2): encarando o mundo real

Introdução

Este artigo é uma continuação do primeiro: Ajax para quem só ouviu falar. Se você ainda não leu o primeiro, pare e faça isso agora, pois vou começar exatamente de onde parei lá.

Ao construir aplicações Ajax dificilmente você vai se deparar com alguma coisa como o primeiro artigo, em que precisa atualizar apenas uma única parte da tela. Geralmente o clique em uma região da tela vai atualizar duas ou mais regiões. É o caso daquela comuníssima aplicação onde a ação de selecionar em uma caixa de seleção (o onchange de um select) preenche outras.

Boa parte do que escrevi aqui é baseada em uma pesquisa e trabalho que estou fazendo para a EPTV. Naturalmente, você pode encontrar soluções melhores do que as que eu consegui desenvolver, ou encontrar falhas no meu código. Se isso acontecer, por favor, avise, vamos melhorar as coisas juntos.

Algumas pessoas me escreveram, depois de ler o primeiro artigo, com problemas com XMLHttpRequest ao tentar executar duas ou mais requisições ao mesmo tempo. Percebi então a necessidade de explicar melhor como funciona o objeto XMLHttpRequest. Não vou dar todos os detalhes sobre o objeto, vou apenas tentar explicar porque você pode estar tendo problemas. Se você quer saber mais sobre o que é o objeto XMLHttpRequest e o que pode fazer com ele, leia este artigo do Diego Nunes (o artigo não está completo, mas como ele não termina nunca você pode ir lendo e aprendendo enquanto isso...)

Para que possamos exercitar a atualização de mais de uma área da tela, fiz uma pequena mudança nesta aplicação. Coloquei o título da página atual ("Mudanças na Aplicação") separado do conteúdo, logo abaixo do título principal. Embora a mudança seja completamente inútil, vai ser um bom exercício.

No index.php, a única mudança foi a inclusão do título, assim:

```
<h2 id="titulo"><?=titulo($i)?></h2>
```

No funcoes.php alterei a função leconteudo, para não retornar o título, e criei a titulo, que retorna apenas o título:

```
/*
Lê o conteúdo de índice n. Aqui estou lendo de arquivos
html no disco, para não perdermos tempo com coisas que
fogem ao escopo do artigo. No mundo real, geralmente você
vai ler isso aqui do banco de dados, ou usar uma função
pronta disponibilizada por seu CMS.
*/
function leconteudo($n){
    $conteudo=split("\n",file_get_contents("$n.html"));
    $t="";
```

```
    for($i=1;$i<sizeof($conteudo);$i++){
        $t=$t.$conteudo[$i]."\n";
    }
    return $t;
}

/*
Lê o título de índice n.
*/
function titulo($n){
    $t=split("\n",file_get_contents("$n.html"));
    return $t[0];
}
```

Sei que essa não é a maneira mais elegante de se fazer isso em PHP, mas eu não queria perder muito tempo com os detalhes server side, uma vez que você pode estar usando outra linguagem para fazer isso.

Bom, apenas com isso e pequenos ajustes no CSS temos nossa aplicação rodando sem javascript, falta agora a parte Ajax da coisa.

Há duas maneiras de se fazer uma requisição com um objeto XMLHttpRequest, uma é síncrona, outra assíncrona. (Se você é um programador com experiência em threads, provavelmente já entendeu o que eu quis dizer e pode pular este e o próximo parágrafos.) No modo síncrono, quando você manda o objeto fazer uma requisição, o seu script é interrompido esperando pelo retorno. No modo assíncrono a requisição é feita em segundo plano (no jargão técnico, dizemos que ela é feita em outra thread) e seu script continua a ser executado.

Há um grave problema em trabalhar com o objeto XMLHttpRequest em modo síncrono, que é o fato de seu navegador permanecer congelado enquanto seu script é executado. O que acontece se um script demora muito para terminar? Primeiro, seu navegador congela, como você pode ver clicando aqui. Em meu computador o link anterior leva cerca de 3 segundos e meio para ser executado no Firefox, 2 segundos no IE e mais de 10 segundos no Konqueror, e durante esse período não posso sequer rolar a tela. O segundo efeito colateral é que, se um script realmente demora muito tempo, o navegador pergunta a você se quer interromper o script. É o que acontece comigo aqui no Konqueror, e deve estar acontecendo aí se você tem uma máquina bem mais lenta que a minha. Se sua máquina é rápida demais para que você possa ver este aviso, tente clicando nesse outro link. Isso é muito perigoso quando se trata de XMLHttpRequest, uma vez que o tempo para a requisição feita depende da sorte. Se o servidor estiver sobrecarregado, ou a internet congestionada, ou o modem do usuário com problemas, ou por qualquer outro motivo que depende dos humores do cobre e do silício, você trava o navegador do usuário. Isso é visto como algo muito antipático, e nós não queremos isso, não é mesmo.

Para escolher de que forma o objeto XMLHttpRequest vai trabalhar, usamos o terceiro parâmetro do método open. No nosso código do primeiro artigo:

```
//Abre a url
xmlhttp.open("GET", "funcoes.php?n="+n,true)
```

Esse true, no terceiro parâmetro, coloca o objeto em modo assíncrono. Ao fazer a requisição o objeto vai executar a função onreadystatechange, que criamos assim:

```
xmlhttp.onreadystatechange=function(){
    if(xmlhttp.readyState==4){
        /*Faz o que tem que fazer aqui*/
    }
}
```

Esse código vai ser executado várias vezes durante a requisição, por isso testamos readyState. Quando readyState é 4, isso significa que a requisição foi concluída e podemos ler o retorno e fazer o que precisarmos com ele. Problemas vão acontecer se tentarmos executar duas requisições, assim:

```
function atualizaDoisLugaresDiferentes(){

    xmlhttp.open("GET","url1.php",true)
    xmlhttp.onreadystatechange=function(){
        if(xmlhttp.readyState==4){
            /*Atualiza o lugar 1*/
        }
    }
    xmlhttp.send(null)

    xmlhttp.open("GET","url2.php",true)
    xmlhttp.onreadystatechange=function(){
        if(xmlhttp.readyState==4){
            /*Atualiza o lugar 2*/
        }
    }
    xmlhttp.send(null)

}
```

Ao executar a primeira vez o método send nosso script não vai ficar esperando pelo objeto XMLHttpRequest, ele continua sendo executado. A linha seguinte é uma chamada ao método open do mesmo objeto, que está ocupado com nossa primeira requisição. Os resultados são imprevisíveis, principalmente em navegadores diferentes, mas certamente isso não vai funcionar direito em nenhum deles.

As soluções são criar mais de um objeto XMLHttpRequest ou criar uma fila de requisições. A primeira solução pode parecer mais simples, mas ela terá que ser reescrita para cada aplicação. Além disso, ter vários objetos de requisição carregados no navegador pode ser uma fonte de problemas, principalmente se alguém resolver visitar seu site usando um velho e corajoso Pentium 233. Vamos então trabalhar numa fila de conexões, que possa ser reutilizada em vários projetos.

Além da fila de requisições, há algo mais que podemos fazer para otimizar nosso trabalho e que pode ser reaproveitado em vários projetos, que é escrever um método genérico para obter HTML do servidor (algo muito comum trabalhando com Ajax.)

Naturalmente você pode fazer mais do que isso com Ajax, então há espaço aberto para criar, por exemplo, uma função para executar javascript vindo do servidor e outra que preencha um select (e não consigo pensar em mais nada...)

Nossa idéia é ter uma função ajaxHTML, que recebe dois atributos: o id de um elemento HTML e assim:

```
function ajaxHTML(id,url){

    //Obtém o objeto HTML
    objetoHTML=document.getElementById(id)

    //Exibe "Carregando..."
    objetoHTML.innerHTML="<span class='carregando'>"+
                         "Carregando...</span>"

    //Abre a conexão
    xmlhttp.open("GET",url);

    //Função para tratamento do retorno
    xmlhttp.onreadystatechange=function() {
        if (xmlhttp.readyState==4){
```

```
            //Mostra o HTML recebido
            retorno=unescape(xmlhttp.responseText.replace(/\+/g," "))
            objetoHTML.innerHTML=retorno
        }
    }

    //Executa
    xmlhttp.send(null)
}
```

Assim você pode chamar `ajaxHTML("conteudo","funcoes.php?n=1")` e a função vai se encarregar de exibir "Carregando..." no div "conteudo", fazer a requisição e atualizar o conteúdo do div quando receber o código. Isto segue aquela premissa do primeiro artigo de que usando a mesma função no servidor para o Ajax e para a aplicação sem javascript é muito mais fácil fazer sua aplicação degradar bonito (ou seja, funcionar bem sem javascript.)

Você vai notar que no funcoes.php, além da nova função de título, escrevemos apenas o trecho:

```
if(isset($_GET["h"])){
    $t=titulo(intval($_GET["h"]));
    echo(urlencode($t));
}
```

Ou seja, se passarmos o parâmetro h vamos ter como retorno o título da página. Você vai se lembrar que já havíamos construído o código, no primeiro artigo, para que ao receber o parâmetro n obtenhamos o conteúdo da página. Agora só falta a fila de conexões.

Para começar criamos nossa fila de conexões, um Array, e uma variável com o índice do elemento da fila que vai ser executado:

```
//Fila de conexões
fila=[]
ifila=0
```

Agora nossa função ajaxHTML deve apenas exibir o "Carregando ...", inserir os parâmetros que ela recebe na fila e testar se a fila está em execução. Se não estiver, executa o elemento atual da fila. Fica assim:

```
//Carrega via XMLHTTP a url recebida e coloca seu valor
//no objeto com o id recebido
function ajaxHTML(id,url){
    //Carregando...
    document.getElementById(id).innerHTML="<span class='carregando'>"+
                                          "Carregando...</span>"
    //Adiciona à fila
    fila[fila.length]=[id,url]
    //Se não há conexões pendentes, executa
    if((ifila+1)==fila.length)ajaxRun()
}
```

Note a chamada à função ajaxRun. Ela vai ficar bem parecida com nossa ajaxHTML do passo anterior. Basicamente, vai criar a conexão e executá-la, com uma onreadystatechange que faz duas coisas: coloca o HTML recebido no objeto HTML e testa se há conexões na fila esperando execução, se houver, executa a próxima:

```
//Executa a próxima conexão da fila
function ajaxRun(){
    //Abre a conexão
    xmlhttp.open("GET",fila[ifila][1],true);
    //Função para tratamento do retorno
```

```
    xmlhttp.onreadystatechange=function() {
        if (xmlhttp.readyState==4){
            //Mostra o HTML recebido
            retorno=unescape(xmlhttp.responseText.replace(/\+/g," "))
            document.getElementById(fila[ifila][0]).innerHTML=retorno
            //Roda o próximo
            ifila++
            if(ifila<fila.length)setTimeout("ajaxRun()",20)
        }
    }
    //Executa
    xmlhttp.send(null)
}
```

Você pode ver o código completo no arquivo ajaxutil.js. Foi construído para que possa ser reaproveitado em qualquer projeto. Veja como o ajax.js ficou. Exatamento igual ao do primeiro artigo, exceto toda a chamada ao XMLHttpRequest, que foi substituída por:

```
//Carrega o HTML
ajaxHTML("titulo","funcoes.php?h="+n)
ajaxHTML("conteudo","funcoes.php?n="+n)
```

Se tivéssemos que carregar mais regiões, bastaria acrescentar linhas aqui.

Você pode querer incrementar sua aplicação, implementando chamadas diferentes ao servidor (como para executar javascript no client de acordo com o retorno, ou preencher um select, por exemplo,) facilitando a vida do seu usuário no client (com coisas como preenchimento automático de formulários de endereço de acordo com o CEP) e efeitos visuais (coisas como isso aqui para arrancar um "Uau!" de quem vê.)

Vai uma dica: tente manter as coisas separadas. Códigos server, chamadas a ele via Ajax, efeitos no navegador, não misture essas coisas. Tente trabalhar em "camadas" em seu javascript, como já faz com HTML e CSS. Uma sugestão de ordem de desenvolvimento:

1. HTML - cru, puro e simples;
2. Código server (PHP, ASP, JSP, Python, Ruby, XSLT, CFML...) - O objetivo aqui é ter sua aplicação completa, funcional, de modo que se puser apenas isso no ar, embora esteja feio, as pessoas vão conseguir usar. Não esqueça, você PRECISA validar os formulários e demais entradas do usuário no servidor. Validar no cliente é um adicional que vamos fazer mais tarde, mas não dispensa a validação do servidor;
3. CSS para a aplicação básica - Isso transforma seu projeto num site. Um site bastante convencional, sem nenhuma mágica javascript, é verdade, mas um site inteiro, completamente funcional, com layout acabado e tudo. Há quem divida esta etapa em duas: na primeira setam-se cores e fontes e tudo aquilo que vai funcionar até no velho Netscape 4 e na segunda se faz a montagem do layout para navegadores modernos (há mais sobre essa abordagem aqui, embora eu prefira ignorar o velho Netscape 4;)
4. Javascript básico - Validação de formulário, menus em cascata e tudo isso que a gente sempre tem feito (mais sobre isso aqui;)
5. CSS para o javascript básico - Você pode, por exemplo, deixar vermelhos os campos obrigatórios não preenchidos;
6. Javascript Ajax - XMLHttpRequest para evitar refresh e tornar a experiência mais confortável;
7. CSS Ajax - Pode, por exemplo, deixar bonito o "carregando..." e etc.;
8. Efeitos especiais - Javascript e CSS para encher os olhos, com fades, movimentos e etc.

Note que o site está pronto no passo 3, tudo o que vem depois é opcional e vem para enriquecer a experiência. A idéia é sobrepôr várias camadas simples, de modo que se as

últimas falharem seu site continuará perfeito e você mantém a simplicidade do conjunto inteiro. Simplicidade é garantia de desenvolvimento rápido e manutenção sem dores de cabeça.