Usando XML HTTP Request para aplicações Web sem reload.

- Um pouco de história...

Um dos grandes problemas relativos à usabilidade e/ou acessibilidade das páginas e aplicações Web-Based antigas estavam ligados à necessidade de recarregar a página a cada interação com o servidor. Isso era imutável, e um problema com o qual todos tinham que conviver. Aí inventaram os frames.

Os frames foram uma revolução, e se disseminaram muito rapidamente. Poder fazer um site que trocasse o conteúdo mantendo o menu, o cabeçalho e o rodapé inalterados era a *hype* da internet, e os sites cheios deles (e absolutamente inacessíveis) se proliferaram pela web.

Num certo momento, alguém percebeu que, através de uma propriedade do objeto *window*, chamada *frames* (que retorna outro objeto *window*), do JavaScript, podia-se acessar propriedades e funções de outro frame. Assim, surgiram as primeiras aplicações "sem reload". Você executa uma ação no frame principal, e esse chama uma função num frame escondido, que dá o reload - apesar do usuário não perceber - e envia a resposta pro frame principal, que trata os dados e faz as modificações no conteúdo. Depois que os iFrames surgiram, a coisa se popularizou ainda mais, porque ganhou-se a possibilidade de ter um iFrame apenas em uma página do seu site, e executar as ações server-side que desejar, sem reload da página principal, e sem ter um frame escondido num canto do site durante toda a navegação, mas apenas quando necessário. Era uma boa solução em muitos casos, e tinha até certas vantagens, mas a inacessibilidade e a decisão da W3 de tirar os frames (inclusive os iFrames) das novas versões do XHTML, fizeram os desenvolvedores buscarem outras alternativas.

- A solução, enfim.

A Microsoft, ainda nos idos do Internet Explorer 5, implementou um controle ActiveX que provê um protocolo *client-side* que permite comunicação HTTP com o servidor, chamado "*Microsoft.XMLHTTP*". Posteriormente, com a criação da segunda versão do MSXML, esse controle foi atualizado, e ganhou o nome de "*Msxml2.XMLHTTP*". Ele permite requisições de conteúdo *on the fly*, sem reload e sem truques - mas usando tecnologia proprietária. Logo depois, a Mozilla Foundation implementou uma versão nativa e compatível com a criada pela Microsoft no Mozilla 1.0 (e no Netscape 7). A Apple logo fez o mesmo, a partir do Safari 1.2. O Konqueror, que usa o KHTML como *render engine*, assim como o Safari, também tem, atualmente, o XMLHTTPRequest implementado. A partir do Opera 7.6 beta,

esse objeto (nativo) também é suportado, mas nas primeiras versões ele ainda apresentava alguns bugs (como indicar duas vezes que a requisição foi concluída, fazendo as ações serem executadas em dobro) - talvez na versão 8 eles já tenham sido resolvidos.

O cenário de compatibilidade que temos atualmente é: Internet Explorer 5+, FireFox 0.9+(?), Netscape 7+, Safari 1.2+, Konqueror 3.3+ (apesar de ter um bug com o método POST, que é corrigido com um patch para o kdelibs-3.3), Opera 7.6+.

A propósito: A recomendação do DOM Level 3 Save and Load propõe uma funcionalidade similar à conseguida com o XMLHTTPRequest, mas dificilmente vamos poder usá-la nos próximos anos, posto que a maioria das recomendações da W3 leva uma quantidade de tempo no mínimo incômoda para ser implementada nos navegadores, e que impede o uso produtivo da tecnologia durante um longo período - o próprio XMLHTTPRequest já existe há tempos; a M$ o implementou desde o IE 5, que foi lançado em março de 1999, mas somente hoje podemos usá-lo de forma mais ampla, devido ao suporte mais difundido.

- ¿XMLHTTP pra que te quero?

Mais importante do que saber como usar, tecnicamente, o XMLHTTPRequest, é saber em que casos é conveniente aplicá-lo, e como aplicá-lo de forma a não prejudicar nenhuma outra estrutura do site, nem atrapalhar a usabilidade ou a acessibilidade - principalmente essa última. A dúvida mais comum é: como fazer um site cheio de "guéri-guéri" ser acessível a quem não dispõe dos recursos que você deseja usar em determinada página? A solução é simples: pense medíocre.

Relembrando o bom e velho Classicismo, e o famigerado preceito "aurea mediocritas", visualize a construção das aplicações Web *sem usar nenhum recurso adicional*. "Como fazer uma *selectbox* se popular segundo a seleção de uma outra, sem reload, e sem usar recursos adicionais?", você me pergunta. E a resposta é simples: "não dá" ;) O segredo está aí. Faça uma página com todos as funcionalidades que você quer, sem usar nada além de puros formulários e interação server-side: totalmente medíocre. Não "medíocre-pejorativo", como essa palavra é interpretada hoje, e sim no verdadeiro sentido: mediano, normal, sem nada a mais, nem a menos. Um site igual a dezenas de milhares de outros.

Depois que o site estiver funcionando perfeitamente sem nenhuma menção à JavaScript a mágica começa. Já ouviu falar em JavaScript não-obstrusivo? Pois pra XMLHTTPRequest é melhor que você entenda bastante (ao menos "o suficiente") do assunto, pra não prejudicar seu site e seus usuários. Infelizmente Unobtrusive JavaScript, por si só, já daria assunto pra outros 10 artigos, e ainda ficariam pontas soltas para terminarmos de amarrar, então vou ser bastante sucinto nesse caso, deixando a discussão sobre isso pra um momento mais oportuno.

Basicamente, JavaScript não-obstrusivo é o tipo de código JavaScript que *se aciona* - ou seja, se o navegador não dispõe de JavaScript, nada na página se perderá, apenas deverá ser acessado por meios comuns (e que, normalmente, gastam mais cliques e mais tempo), porque eles foram projetados pra isso, e o JavaScript entra como "camada de comportamento" (na divisão das três camadas que a página deve ter: "conteúdo", que é o XHTML; "apresentação", que é o CSS; e "comportamento", que é o JavaScript), como adicional, apenas. E, mesmo que o navegador suporte JavaScript, as boas maneiras mandam você, ainda, verificar se ele suporta as funções que você vai precisar, pra evitar que apareçam erros em navegadores mais antigos, que suportem algumas coisas, mas não tudo que você vai usar, e que normalmente frustram os usuários. *Browser sniffing*? Nada disso! A onda agora é verificar se ele suporta o que você quer, independente de qual seja ele, e qual versão dele. Devido à grande diferença na implementação de JavaScript nos diversos navegadores, e o suporte extremamente deficiente em alguns deles, é importante checar se ele suporta métodos e propriedades cruciais pras suas funções, pra garantir a compatibilidade com navegadores menos populares, mas que oferecem o suporte necessário. Por exemplo, por mais eficiente que pareça verificar se o navegador é "IE5+" ou "FF1+", este tipo de abordagem deixa, automaticamente, de fora, Opera, K-Meleon e vários outros.

Se, mesmo sabendo que a técnica de *browser sniffing* deve ser evitada, você queira usá-la, leia esse artigo da Mozilla.org que dá as dicas de como produzir um *browser sniffing* menos predisposto a erros.

Os programadores em geral, principalmente os que vieram de uma "cultura IE-Only" (conviveram/geraram código só pro IE a vida toda) têm uma tendência enorme a se empolgar com o advento de novas tecnologias e possibilidades, e esquecer que elas não existiam há 1 ano, e que uma parcela *muito* grande do público que vai acessar o site dele provavelmente usa navegadores de 1 ou 2 anos de idade, às vezes bem mais que isso (o IE 5.01, que é de 1999, ainda tem um público-usuário relativamente grande), e acaba por não permitir a essas pessoas o acesso a toda a funcionalidade, ou ao menos a todo o conteúdo, do site, o que pode fazer com que as pessoas se irritem e não voltem mais. Pois, sim, na internet, se seu site é muito complicado ou não tem acesso rápido à informação que os visitantes estão procurando, eles simplesmente fecham a sua aba e partem pro próximo resultado no Google.

Em suma: usem a tecnologia com responsabilidade (bah! isso parece comercial de camisinha :P)

- Aos códigos!

Num cenário ideal, criar um objeto XMLHTTPRequest deveria ser simples como fazer `tXHR=new XMLHttpRequest();` - leia bem: num cenário ideal, e a web e os navegadores para ela, hoje, passam muito longe de serem um cenário ideal. Como já foi citado, a implementação do XMLHTTPRequest (assim como, aliás, quase todas as novas tecnologias) tem disparidades nos diferentes navegadores (dividindo em duas partes simples, mas antagônicas: "navegadores que seguem o padrão" e "IE"), então, obviamente, precisamos de um pouquinho de trabalho pra fazer a coisa funcionar a contento.

Pra começar, verifica-se se o navegador que está acessando suporta nativamente o objeto.

```
if (window.XMLHttpRequest) { tXHR=new XMLHttpRequest(); }
```

Caso o navegador que esteja acessando suporte, acaba aí a função. Caso o navegador não suporte, tenta-se criar de acordo com o padrão do IE:

```
else { tXHR=new ActiveXObject('MSXML2.XMLHTTP'); }
```

Mas, como já foi citado, a M$ publicou duas versões do controle ActiveX, então, pra garantir a compatibilidade com versões mais antigas do navegador, tenta-se a outra alternativa, caso a primeira não funcione.

```
else {
 try { tXHR=new ActiveXObject('MSXML2.XMLHTTP'); }
 catch(e) { tXHR=new ActiveXObject('Microsoft.XMLHTTP'); }
}
```

Ainda há a possibilidade do navegador ser mais antigo e não ter nenhuma versão do XMLHTTP implementada, então, pra evitar erros, usa-se outra rotina "*try...catch*" encadeada.

```
else {
 try { tXHR=new ActiveXObject('MSXML2.XMLHTTP'); }
 catch(e) {
  try { tXHR=new ActiveXObject('Microsoft.XMLHTTP'); }
  catch(e) { /* O navegador não oferece suporte. */ }
 }
}
```

A função básica, completa, é essa:

```
function XMLHTTPRequest() { var tXHR=0;
 if (window.XMLHttpRequest) {
  tXHR=new XMLHttpRequest(); //objeto nativo (FF / Safari / Konqueror / Opera / etc)
 }
 else {
```

```
   try { tXHR=new ActiveXObject("Msxml2.XMLHTTP"); } //activeX (IE5.5+/MSXML2+)
   catch(e) {
    try { tXHR=new ActiveXObject("Microsoft.XMLHTTP"); } //activeX (IE5+/MSXML1)
    catch(e) { tXHR=false; } //O navegador não tem suporte
   }
  } return tXHR; //retornar resultado (objeto, ou "false", no caso de erro)
}
```

Com isso já pode-se criar um objeto XMLHTTP de forma cross-browser, e verificando no caso de erros. A função seria chamada da seguinte forma:

```
var tXHR=XMLHTTPRequest();
if (tXHR) {
 //ações
}
else {
 //exibir mensagem de erro
}
```

- Obaaa! Tenho um objeto! Mas, e agora?

Nem tudo foi tristeza quando criaram várias versões do XMLHTTPRequest, e ao menos algumas propriedades e métodos básicos (os essenciais, e provavelmente os únicos que você vai usar em aplicações simples) ficaram padronizados - isso significa que, depois de criado o objeto, você não precisará mais verificar diferentes versões de navegador e de forma de executar certa ação, o que facilita muito pro desenvolvedor.

Depois de criado o objeto, o processo é relativamente simples: prepara-se o objeto pra abrir uma requisição, dizendo os parâmetros (como servidor, método de requisição - que pode ser post, get, head...), e é só enviar. Com um evento, uma função e uma verificação de propriedade, bem simples, consegue-se facilmente o conteúdo de um arquivo.

Teoricamente, vamos precisar apenas do seguinte:

- ***open(método, url, async)*** - O principal método do objeto, e serve para prepará-lo para uma requisição. Recebe os parâmetros: "*método*", que indica o método da requisição (post, get, head etc); "*url*", que indica a URL para onde a requisição vai ser feita; e "*async*", que indica se a requisição é síncrona ou assíncrona (esse assunto será discutido mais abaixo).
- ***setRequestHeader(nome, valor)*** - Método que inclui um par nome/valor ao cabeçalho pra ser enviado junto com a requisição. Normalmente é usado para indicar que o conteúdo deve ser retornado como "*text/xml*", usando `objeto.setRequestHeader('content-type', 'text/xml');`
- ***send(conteúdo)*** - Método que, quando invocado, dispara a requisição. "*conteúdo*" é uma string que deve conter o conteúdo da "mensagem HTTP". Ex.: No caso de uma requisição via "post", esse parâmetro deve conter os dados que vão ser postados, no formato padrão de uma URL ("nome=valor&nome2=valor2&nome3=valor3"); caso você queira usar uma interface SOAP para requisitar dados, também é no "*conteúdo*" que vai o XML do "envelope"; e se você

quisesse fazer um upload sem recarregar a página, por exemplo, o arquivo, em si (dados binários, inclusive), iria nesse parâmetro.

- ***onreadystatechange*** - Um evento que é disparado quando o "estado" da requisição muda. Leia a propriedade "*readyState*" abaixo para entender melhor.
- ***readyState*** - A propriedade que indica o estado da requisição [cuja mudança dispara o evento "onreadystatechange"]. Ele pode conter os valores:
  - ***0***: não inicializado - O objeto foi criado, mas o método "*open*" não foi chamado ainda.
  - ***1***: carregando - O método "*open*" foi chamado, mas não o método "*send*".
  - ***2***: carregado - Não funciona no IE. Indica que a conexão foi estabelecida com sucesso com o servidor. Os cabeçalhos de resposta do servidor e o "*status code*" (404 = não encontrado, 200 = ok...) já podem ser acessados.
  - ***3***: interativo - Não funciona corretamente no IE. Nos outros navegadores, indica que o conteúdo começou a ser recebido, e pode ser acessado pela propriedade "*responseText*", mas ainda está incompleto (útil pra criar "*barras de progresso*"). É disparado a cada 4kb recebidos, exceto no IE, que o dispara apenas uma vez, ao começar a receber os dados.
  - ***4***: completo - O conteúdo foi totalmente recebido - a requisição foi finalizada. Não indica, entretanto, que o arquivo foi encontrado, ou que não houve erros no processamento do arquivo. Indica apenas que o processo de conectar ao servidor, fazer uma requisição HTTP simples e receber a resposta foi concluído, mas o arquivo pode não ter sido encontrado (nesse, caso a propriedade *status* teria o valor "*404*"), ou ter acontecido algum erro no processamento (*status* com valor "*500*").
- ***status*** - Código HTTP de retorno (HTTP Status Code) da requisição. Uma requisição bem sucessida retorna *status 200*, por exemplo.
- ***statusText*** - Texto que acompanha a propriedade *status*, complementando o valor de retorno dela. No caso de um erro 404 (arquivo não encontrado), por exemplo, a propriedade *statusText* teria valor "*Not Found*", e no caso de *status* "*200*", o valor seria "*Ok*".
- ***responseText*** - Essa propriedade armazena o retorno do servidor (conteúdo do arquivo requisitado, por exemplo), numa string normal.
- ***responseXML*** - A propriedade *responseXML* armazena a resposta do servidor, assim como a *responseText*, com a diferença de ser uma versão compatível com DOM, pra ser parseada como XML, e não tratada como texto comum.

- Fazendo sua primeira requisição

Depois de toda essa teoria, finalmente podemos fazer algo prático com nosso mais novo amigo, o XMLHTTPRequest.

Pra facilitar, eu criei uma suíte de funções básicas, que são sempre necessárias nos scripts não-obstrusivos, e vou usar algumas aqui. Pra identificarem quais funções não são nativas, vou destacá-las com outra cor. Quando encontrarem algo escrito assim, é só dar uma passada no basico.js que você encontra a função, buscando pelo nome dela - normalmente ela tem uma descrição comentada, caso o entendimento não seja óbvio. A propósito: nem todas as funções dessa suíte são usadas em todos os códigos, então você pode deixar no seu servidor apenas as que serão usadas, pra diminuir o tamanho do arquivo.

Primeiro, vamos ao HTML do exemplo (não espere XHTML válido - é só um exemplo básico):

```
<html>
<head>
<title>Exemplo XMLHTTPRequest - Requisição Básica</title>
<script type="text/javascript" src="basico.js"></script>
<script type="text/javascript" src="exemplo1.js"></script>
</head>
<body>

<a href="?exibirMensagem=1" id="linkCarregar">Carregar Texto.</a>

<div id="tContent"><!-- Aqui entrará o conteúdo retornado --></div>

</body>
</html>
```

Nota: Se você está lendo esse texto com o intuito de aprender apenas sobre XMLHTTPRequest, pule o próximo parágrafo..

Nesse ponto vem uma parte importante da construção de um código JavaScript, e que começa antes mesmo da primeira linha de JavaScript ser escrita: o planejamento da acessibilidade, através de código não-obstrusivo. Nesse caso, note que o link aponta para *"?exibirMensagem=1"*, ou seja, acessando o mesmo documento do link, mas passando o parâmetro "exibirMensagem" com o valor 1.

Com o HTML e o código *server-side* pronto, começamos a criar o JavaScript. A princípio, inserimos a função de criação do objeto de forma cross-browser (já demonstrada acima).

```
//Criação de objeto XMLHTTPRequest cross-browser - Parâmetros: N/A
function XMLHTTPRequest() { var tXHR=0;
 if (window.XMLHttpRequest) { tXHR=new XMLHttpRequest(); } //Objeto nativo (FF/Safari/Opera7.6+)
 else {
  try { tXHR=new ActiveXObject("Msxml2.XMLHTTP"); } //activeX (IE5.5+/MSXML2+)
  catch(e) {
   try { tXHR=new ActiveXObject("Microsoft.XMLHTTP"); } //activeX (IE5+/MSXML1)
   catch(e) { /* O navegador não tem suporte */ tXHR=false; }
  }
 } return tXHR;
}
```

Agora o que temos a fazer é tentar criar o objeto e verificar se o navegador suporta todas as funções que você vai usar (além de suportar o XMLHTTPRequest, óbvio):

```
var tXHR=XMLHTTPRequest();
```

```
if (tXHR && document.getElementById && document.createElement)
```

Depois, vem a parte na qual você adiciona os eventos aos elementos do seu documento. Lembram do Unobtrusive JavaScript? Tá na hora de começar a aplicá-lo:

```
 adEvento(window, 'load', iniciar);
```

Evento adicionado, browsers não-compatíveis devidamente ignorados, vamos às funções...

A primeira é a função básica, que serve apenas pra adição de eventos e definições de tudo na página, além da eventual alteração de algum elemento na mesma:

```
function iniciar() {
 adEvento(gE('linkCarregar'), 'click', carregaDados); //Note que eu não alterei o link, só atribuí uma função ao clique
}
```

Agora vamos criar a função pra receber o clique, e executar as devidas tarefas, além de anular a ação "default" do navegador (que seria seguir a URL indicada no link):

```
function carregaDados(e) { //O parâmetro dessa função serve pra cancelar o clique nos navegadores "Gecko".
 if (!tXHR) return false; //Segundo as boas maneiras, é bom verificar se o objeto ainda existe, pra evitar erros.

 //É interessante exibir algum tipo de mensagem enquanto o arquivo está sendo carregado,
 //pra indicar ao usuário que o processamento está em andamento. A linha abaixo faz isso.
 mudaConteudo('tContent', 'Carregando...'); //Essa função ainda não existe - vamos criá-la logo a seguir.
 tXHR.open('get', 'exemplo1.txt', true); //Prepara-se o objeto pra executar a requisição (ver método "open").
 tXHR.onreadystatechange=recebeInfo; //Define-se qual função será chamada (ver evento "onreadystatechange").
 //O parâmetro do método "send" é obrigatório
 //Caso não haja conteúdo pra enviar na requisição (no caso de uma requisição "get", por exemplo), use "null".
 tXHR.send(null); //Envia-se a requisição (ver método "send").

 //Anulando o click: (usa-se uma checagem simples, de "window.event", pra função ser compatível com o IE)
 if (window.event) { event.returnValue=false; /* Modo IE */ } else { e.preventDefault(); /* Modo Standard */ }
}
```

Na função acima, foi feita a referência à função "mudaConteudo", que ainda não existe. A idéia é que ela substitua o conteúdo de um elemento (cujo ID será indicado no primeiro parâmetro) com um texto que será passada pra ela (no segundo parâmetro).

```
function mudaConteudo(tID, tNC) {
 rEs(gE(tID).childNodes); //Limpar conteúdo atual do elemento.
 gE('tContent').appendChild(cTN(tNC)); //Inserir novo conteúdo.
}
```

Agora já estamos a um passo do término desse código. Só o que falta é criar a função "dadosCarregados", que será chamada no evento "readystatechange" (ver função acima).

```
function recebeInfo() { if (!tXHR) return false;
 if (!tXHR) return false; //Novamente, apenas pra evitar erros.

 if (tXHR.readyState == 4) { //Se a requisição estiver terminada (ver propriedade "readyState")
  if (tXHR.status == 200) { //Se a "status" retornado for "ok" (ver propriedade "status")
   mudaConteudo('tContent', tXHR.responseText); //inserir conteúdo (ver propriedade "responseText")
  }
  else { //Se o servidor retornou outro código que não "200", mostrar o erro.
   alert('Erro! "'+ tXHR.statusText +'" (erro '+ tXHR.status +')'); //(ver propriedade "statusText")
  }
 }
}
```

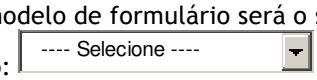
Tudo pronto. Veja os arquivos completos desse exemplo:

- ***exemplo1.html*** - Arquivo HTML. Exemplo funcional.
- ***basico.js*** - Suíte básica de funções comuns.
- ***exemplo1.js*** - Arquivo JS com as funções específicas do exemplo.
- ***exemplo1.txt*** - Arquivo TXT com o conteúdo que é carregado.

- Finalmente: Select's aninhados

Como exemplo ilustrativo final, e pra demonstrar um tratamento de erros mais completo, vou usar uma das demandas mais comuns quando o assunto é interação sem reload: caixas de seleção. Você tem duas caixas de seleção e, ao selecionar um valor na primeira, ele preenche a segunda com as opções. Normalmente você pode deixar todas elas no seu documento, escondidas, e exibir de acordo com a necessidade, mas há casos em que isso é inviável, pela quantidade de dados. Digamos que você tenha uma lista de países. Ao selecionar o país, ele lista os estados. Ao selecionar um estado, ele lista as cidades. Imagine isso pra todas as cidades do Brasil. E agora imaginei pra todas as cidades do mundo. Já pensou quanto HTML não usado teria que ficar no documento, aguardando sua hora de aparecer - se é que vai aparecer? Agora pense que você pode ter tudo isso num banco de dados qualquer (SQL Server, MySQL, Oracle, Postgree e até mesmo Access), e requisitar somente o que precisa, pra exibir no documento - sem precisar recarregar a página. Impossível? Não mesmo. Trivial; quiçá simplório.

O modelo de formulário será o seguinte:

Estado: 

Cidade: ---- Selecione um estado ----

Local: ---- Selecione uma cidade ----

A primeira coisa a se fazer é construir o HTML do formulário. Ele já está construído, aqui nesse documento, então não vou reproduzi-lo pra não ser redundante (e pra não deixar esse texto ainda maior) - quem quiser vê-lo, mande exibir o fonte. O ponto realmente importante é que cada *selectbox* tem um ID único (`id="slocal"`, por exemplo). Os IDs que eu usei, pras *selects* de estado, cidade e local, respectivamente, foram "*sestado*", "*scidade*" e "*slocal*". (criativo demais, eu, não? :P)

Depois precisamos de um modelo para o XML que será retornado, contendo as informações do banco de dados. Como, nesse exemplo, vou requisitar informações de estados e cidades, pra adicionar opções em "*select's*", vou definir o XML com o seguinte padrão:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dados SYSTEM "dados.dtd">

<dados>

 <opcao>   <capt> TEXTO </capt>   <valor> VALOR </valor>   </opcao>
 (...)
 <opcao>   <capt> TEXTO </capt>   <valor> VALOR </valor>   </opcao>

</dados>
```

O DTD dele seria algo próximo disso:

```
<!ELEMENT dados (option+)>
 <!ELEMENT option (capt, valor)>
   <!ELEMENT capt (#PCDATA)>
   <!ELEMENT valor (#PCDATA)>
```

Mais simples impossível. (Não sabe o que é DTD? Dê uma olhada no tutorial do W3Schools, então).

Como os locais vão conter informações diferenciadas, eles usarão um modelo de XML diferente, e um DTD diferente, também. Mas como esse XML pode variar muito de acordo com as necessidades de dados a serem exibidos, não vou gastar tempo explicando-o, e vou me concentrar no que é realmente importante.

Obviamente, esse XML deve ser gerado por uma aplicação server-side (em ASP, PHP, ColdFusion, Python, Miva, ou qualquer outra linguagem), buscando dados de um banco, porque com arquivos .xml estáticos boa parte disso simplesmente perderia o sentido,

devido à complicação que seria manter atualizada ou mesmo organizar dezenas de milhares de .xml's em pastas seguindo alguma estrutura inteligível e acessível ao script.

No caso de usar esse script produtivamente, o ideal é que acesse um arquivo como "arquivo.ext?estado=bla&cidade=bla&local=bla", para retornar os dados desejados - se for omitido o local, ele retorna a lista de locais na cidade especificada, e se for omitido o local e a cidade, ele retorna a lista de cidades do estado. Caso tudo seja omitido... Bom, aí não retorna nada, porque o script nunca vai fazê-lo, e é perda de tempo fazer tratamento de "exceções provocadas" (isso só aconteceria se alguém, deliberadamente, acessasse sem passar parâmetros - nesse caso simplesmente pare a execução sem mensagem nenhuma), a não ser que elas representem um risco de segurança, o que não é o nosso caso.

Como isso é só um exemplo, vou dar-me a liberdade de não fazer uma aplicação realmente dinâmica, e sim consultando arquivos estáticos. Os arquivos que vou usar chamam-se "rio.xml" e "sp.xml" pros estados, "mesquita.xml" e "ni.xml" pras cidades, e "diegoh.xml" e "crepe.xml" pros locais. Notavelmente, pelo número de arquivos, nem todas as opções vão funcionar, mas isso é, de certa forma, positivo, porque assim podemos estudar um pouco de tratamento de erros, e eu não preciso ficar criando um monte de arquivos e perdendo um tempo precioso só com uma exemplificação simples ;)

A primeira coisa a se fazer é tentar criar o objeto e verificar se o navegador suporta todas as funções que você vai usar:

```
var tXHR=XMLHTTPRequest();
if (tXHR && document.getElementById && document.createElement)
```

Depois, vem a parte na qual você adiciona os eventos aos elementos do seu documento. Lembram do Unobtrusive JavaScript? Tá na hora de começar a aplicá-lo:

```
adEvento(window, 'load', iniciar);
```

Evento adicionado, browsers não-compatíveis devidamente ignorados, vamos às funções...

A primeira é a função básica, que serve como base pra adição de eventos e definições de tudo na página:

```
function iniciar() {
 //Eventos: (Note que as funções que serão chamadas - atualizaSelects e carregaInfos - ainda não existem)
 adEvento(gE('sestado'), 'change', atualizaSelects);
 adEvento(gE('scidade'), 'change', atualizaSelects);
 adEvento(gE('slocal'), 'change', carregaInfos);

 //Definir subordinações:
 gE('sestado').dependente='scidade';
```

```
 gE('scidade').dependente='slocal';


 //Salvar textos iniciais
 gE('scidade').blankMsg=gE('scidade').options[0].innerHTML;
 gE('slocal').blankMsg=gE('slocal').options[0].innerHTML;
}
```

Nesse ponto eu já defini certas coisas específicas desse exemplo

http://members.lycos.co.uk/dnunes/artigos/xmlhttp/