BSP + jQuery Mobile, ou a aventura de SAP-webdev – Parte II: Frontend

26/05/2015 11:00

E aí seus consultor WD (Walking Dead), tudo bem? Aqui tá tudo certo também.

Nessa segunda parte da série eu vou falar de coisas que podem ser novidade pros desenvolvedores mais ABAPeiros: técnicas para desenvolver uma *view* com jQuery Mobile (que abreviarei como jQM neste *post*) para uma aplicação com *backend* BSP. Se você não está acostumado com desenvolvimento *web*, seria prudente checar os links que eu coloquei na primeira parte dessa série.

Ferramentas

Eu nunca subestimo o masoquismo das pessoas, mas eu acho que escrever e testar HTML, JavaScript e CSS na SE80 é no mínimo ruim. Felizmente, não é necessário fazer isso: você pode escrever e testar a lógica e definição das suas *views* em outros lugares / editores e depois importar para o SAP. Vou deixar aqui algumas dicas de ferramentas que eu uso:

Editor: uma indicação boa que peguei do próprio Maurício aqui do ABAP Zombie é o Sublime Text. É um editor bacana que tem várias opções de *plugins* de automação, *syntax highlighting* para várias linguagens, enfim, é muito bom.

Desenho de telas: às vezes é necessário montar alguns *mockups* de telas para entender a lógica, ou mesmo para mostrar como a aplicação vai se comportar. Eu gostei da versão gratuita do Moqups, porque os elementos já são bem parecidos com os do jQM.

Testes: em certas situações você acaba tendo que bolar algo ~~do capeta~~ mirabolante com CSS pra alguma tela, ou você precisa mesmo entender como e quando aquele evento ~~maldito~~ incomum do jQM é disparado, ou você apenas quer ver se a sua lógica ~~não deu merda~~ funciona. Existem vários *sites* bacanas para esse tipo de teste, e o que eu uso é o JSFiddle. Ele tem um controle de versões bem simples, e você pode compartilhar e criar *forks* de códigos de outras pessoas com facilidade. Eu tenho um *dashboard* com algumas coisinhas lá, e você pode criar a sua conta e criar um *dashboard* seu também.

Multi-HTML vs. *multipage*

Uma das primeiras decisões a se tomar é qual vai ser a arquitetura de páginas da aplicação. Seria melhor usar o padrão *multi*-HTML, onde cada página jQM fica num documento HTML separado, ou o padrão *multipage*, onde as páginas jQM estão todas no mesmo documento HTML? Existem argumentos a favor e contra o uso de cada um, como se vê nesse post aqui.

De maneira geral, se a sua aplicação tem uma pequena quantidade de telas, a escolha de arquitetura vai influir pouco na performance. No caso do projeto que eu estou usando como exemplo nessa série, porém, foram necessárias 10 telas com lógica (pesada) de negócio e outras 4 só com *links* para navegação. Não é prudente que uma aplicação para dispositivos móveis fique recarregando a cada mudança de página, dependendo assim bastante da rede, o que seria o caso se a arquitetura *multi*-HTML fosse a escolhida. Portanto, eu escolhi o padrão *multipage*, de modo que assim que o usuário fizer *login* corretamente, todas as páginas jQM da aplicação serão carregadas no DOM, e somente a primeira página, a página inicial, é exibida (esse comportamento é descrito na documentação do jQM).

Assim, a navegação é resolvida via Ajax e fica quase independente do *backend*, e a *view* precisará ser carregada apenas uma vez para cada execução da aplicação.

E por falar em Ajax...

A importância do Ajax

...praticamente TODA ação que resulta em navegação, leitura ou alteração dos dados na *view* pode ser (ou será) resolvida com Ajax. Isso se torna uma parte essencial da aplicação quando se escolhe a arquitetura *multipage*, porque nesse caso não temos *roundtrips* constantes ao servidor para esse fim. Como estamos usando jQuery, é portanto imperativo que se entenda e se utilize corretamente os métodos que expõem as funcionalidades Ajax, entre eles o método$.ajax(), o mais *low-level*, porém principalmente os métodos simplificados$.get() e$.post().

Um detalhe importante sobre as chamadas Ajax é que estas são **assíncronas**, o que quer dizer que embora saibamos exatamente o ponto de origem da chamada, não sabemos em que momento a resposta chegará do servidor.
Os métodos de Ajax do jQuery possuem vários parâmetros onde podem ser indicadas funções de *callback*, que serão chamadas dependendo do tipo da resposta e em pontos específicos antes ou depois do resultado.

Além disso, os próprios métodos Ajax retornam um objeto do tipo jqXHR, que é basicamente um encapsulamento jQuery do objeto XMLHttpRequest do *browser*, e de onde podem ser chamados outros métodos para cada tipo de resposta do servidor, em vez das – ou adicionalmente às – funções de *callback*. Então uma chamada Ajax típica ficaria, por exemplo, assim:

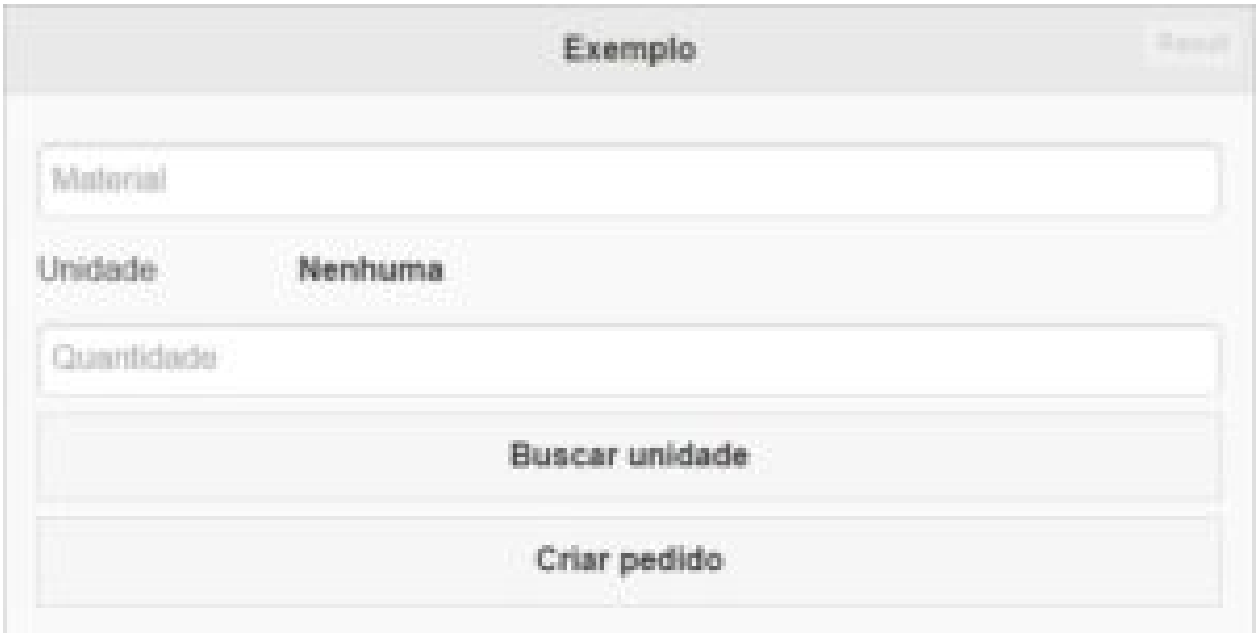
```
$.get('main.do?action=getServerParameters', function(data, textStatus, jqXHR) {
    /* Aqui tratamos a resposta caso a request tenha sido bem sucedida. O
    parâmetro "data" é o corpo da resposta */
    $('#server_parameters').text(data);

    /* Se a chamada retornar um objeto JSON em vez de HTML ou texto puro, é
    possível acessá-lo já interpretado através do parâmetro jqXHR */
    dateFormat = jqXHR.responseJSON.dateFormat;
})
.fail(function(jqXHR, errorStatus, errorThrown) {
    /* Este callback é parte da interface Promise do objeto jqXHR, que é
    chamado caso a request falhe, e que está sendo chamado aqui de forma
    encadeada a partir do retorno do $.get(). Se a resposta contiver dados,
    como uma mensagem de erro ou mesmo um popup, podemos acessá-los pelo
    parâmetro "jqXHR". Por exemplo: */
    $('#message_area').html(jqXHR.responseText).show('fast');
});
```

Outro detalhe importante é que os *forms* numa arquitetura *multipage* devem ser declarados **sem o atributoaction**, porque um *submit* em um *form* que contenha esse atributo resulta sempre em um POST no recurso indicado e num subsequente *refresh* no documento, que como eu mostrei lá no começo é um empecilho para uma arquitetura com apenas um documento HTML. Isso muda um pouco a maneira de submeter os dados de um *form*; na maioria das vezes será necessário passar diretamente os valores dos campos, ou usar algum método de jQuery que retorne uma *string* de parâmetros, como o.serialize() ou o$.param(), para então passar os valores como parâmetros de URL em um GET ou como corpo da *request* num POST. Um exemplo simples:

HTML

```
<div data-role="page">
    <div data-role="header">
        <h1>Exemplo</h1>
    </div>
```

```
    <div role="main" class="ui-content">
        <form id="tela_selecao">
            <input type="text" id="material" name="material" placeholder="Material" />
            <div class="ui-field-contain">
                <label style="margin-top:0">Unidade</label>
                <b><span id="unidade">Nenhuma</span></b>
            </div>
            <input type="text" id="quantidade" name="quantidade" placeholder="Quantidade" />
        </form>
        <a href="#" id="buscar" class="ui-btn">Buscar unidade</a>
        <a href="#" id="criar" class="ui-btn">Criar pedido</a>
    </div>
</div>
```



Exemplo renderizado no JSFiddle

JavaScript

```
/* Ao clicar no botão "Buscar unidade", queremos que o span "unidade" seja
preenchido com a unidade do material que está no campo "material", e não
pretendemos alterar nenhuma informação no servidor (portanto, usaremos o
método GET). O backend possui um método que interpretará o parâmetro "action"
como essa busca ("buscarUnidade"). Portanto: */
$('#buscar').on('click', function() {
    var url = 'main.do?action=buscarUnidade&material=' + $('#material').val();
    $.get(url, function(data) {
        $('#unidade').text(data);
    })
    .fail(function(jqXHR) {
        alert(jqXHR.responseText);
    });
});

/* Além da busca de unidades, temos outro botão que servirá para criar um
pedido com base no material e na quantidade, ou seja, usando todos os campos
do form. Nesse caso, a request alterará os dados no servidor, por isso
usaremos o método POST. O backend possui outro método justamente para criar o
pedido, que responde ao valor "criarPedido". Logo: */
$('#criar').on('click', function() {
    var url = 'main.do?action=criarPedido',
        postData = $('#tela_selecao').serialize();
    $.post(url, postData, function(data) {
        alert(data);
```

```
    })
    .fail(function(jqXHR) {
        alert(jqXHR.responseText);
    });
});
```

Nos exemplos acima eu acabei já descrevendo outra característica importante, embora já bem conhecida, para a definição dos métodos das *requests* de uma aplicação *web*: se a *request* vai apenas **buscar dados do servidor**, deve-se usar o método **GET**; se a *request* servirá para **alterar os dados no servidor**, mesmo que apenas dentro da aplicação, deve-se usar o método **POST**. Aqui tem alguns argumentos sobre o porquê disso.

Mais um detalhe: um *form* mesmo sem *action* **ainda pode sofrer um *submit* quando se pressiona ENTER**. Esse comportamento pode ser desabilitado com um pouco de JavaScript:

```
$(document).on('pagecontainercreate', function() {
// Fazendo o binding na criação do pagecontainer
    $(window).on('keydown', function(event) {
        if (event.keyCode === 13) {
            event.preventDefault();
            return false;
        }
    });
});
```

É interessante ter um elemento visual que indique quando a aplicação está parada esperando a resposta de uma *request* Ajax. Para isso, podemos usar o *loader* do jQM. A maneira mais simples é exibir o *loader* sempre que uma *request* for disparada, e escondê-lo quando a resposta chegar:

```
$(document).ajaxStart(function() {
    $.mobile.loading('show');
});
$(document).ajaxStop(function() {
    $.mobile.loading('hide');
});
```

Eventos

É importante lembrar que ao usar jQM a *view* está sujeita aos eventos do jQM (obrigado, Capitão Óbvio). Portanto, ao fazer *binding* dos eventos da tela – como cliques, abertura e fechamento de *collapsibles*, abertura e fechamento de *popups*, dentre outros – é melhor fazê-lo **dentro dos *callbacks* de outros *event bindings* de eventos do jQM**, como eu fiz no exemplo acima onde eu desabilitei a função do ENTER dentro do *callback* de criação dopagecontainer. **O mesmo vale para as chamadas Ajax.**

Isso parece ser um detalhe de menor importância, mas acredite quando eu digo que não é. O jQM tem a tendência de fazer algumas coisas malucas quando se tenta mesclar os seus *event bindings* com os do jQuery. Eu quase fiquei doido tentando descobrir porque os eventos dos botões que eu estava colocando na *view* estavam sendo disparados TRÊS VEZES para cada clique, até eu mudar a declaração do *binding* para dentro de um eventopagecreate. Então, em vez de fazer algo como...

```
$(document).ready(function() {
    $('#botao').click(function() {
        // Código do event handler
    });
});
```

...é melhor fazer:

```
$(document).on('pagecreate', '#pagina1', function() {
    $('#botao').click(function() {
        // Código do event handler
    });
});
```

Adaptando paradigmas e elementos de telas

Há certas coisas que o desenvolvedor ABAP está acostumado a fazer ao criar interfaces para SAP GUI que nem sempre são fáceis de se reproduzir fora desse meio. Em alguns casos, basta adaptar o elemento; em outros, é preciso revisar a funcionalidade da tela. Vejamos como fazer algumas adaptações de elementos e paradigmas de interface para jQM:

Passagem de parâmetros entre telas: isso é bem comum em telas ABAP, e resolvemos em JavaScript do mesmo jeito que em ABAP: usando variáveis globais. Declare as variáveis que servirão para passar parâmetros entre páginas diretamente na área de *scripts* do<head>, tomando o cuidado de não declarar variáveis com o mesmo nome dentro de funções. Se for necessário manipular as variáveis no meio do caminho (limpar, validar, etc.), é possível fazê-lo nos eventos dopagecontainer, como opagecontainerhide, que é disparado após a página origem da navegação ser totalmente escondida, ou opagecontainerbeforeshow, que é disparado antes da animação de exibição da página destino ser executada.

Matchcodes (ajudas de pesquisa, F4)**:** um *matchcode* simples com valores fixos sempre pode ser reproduzido fielmente com um<select>. Já um *matchcode* mais complexo, com parâmetros, abas ou ambos, vai exigir uma página ou um *popup* extra, e isso significa construir uma lógica de passagem de parâmetros entre telas e também pelo menos mais uma *request* Ajax para fazer a busca; portanto, só construa se for realmente necessário. Uma exceção neste caso é o *matchcode* para seleção de datas; ele pode ser perfeitamente refeito com<select>s para ano, mês e dia, mas neste caso eu prefiro usar um *plugin* chamado DateBox, que possui um ótimo seletor de datas, mais flexível do que o do SAP GUI inclusive.

Formatação de *input*:

- Na maioria dos casos, os *inputs* podem ser feitos comtype="text", e o atributomaxlength é a melhor maneira para limitar o comprimento do valor.
- Para campos baseados em variáveis ABAP do tipoc, é possível usar a propriedade CSStext-transform: uppercase para deixar o texto em maiúsculas, porém isso só vai modificar o texto que está **na tela** – portanto, é necessário transformar o texto do campo em maiúsculas **novamente** quando os valores estiverem sendo tratados pelo *backend*.
- Campos de data: uma boa maneira de formatá-los é utilizando o já mencionado DateBox – você pode estilizar os campos depois que a página foi criada, chamado o método.datebox() e passando as propriedades conforme a documentação do *plugin*. Uma boa prática é definir o formato da data a partir dos parâmetros do usuário: para isso,
    - chame aBAPI_USER_GET_DETAIL e escolha o formato correspondente ao valor do campoDEFAULTS-DATFM entre os formatos definidos no domínio desse campo;
    - traduza o formato SAP para o formato usado no DateBox: por exemplo, o tipo de formato de data mais comum, que é o 1 (DD.MM.AAAA), viraria%d.%m.%Y;
    - ao estilizar o campo, passe o formato definido para a opçãooverrideDateFormat.
- Formatação de quantidades / valores em moeda é outro problema para o qual precisamos de um *plugin* jQuery – nesse caso, eu uso o Number. A ideia é parecida com a que eu expliquei para o DateBox: pegue a definição a partir do campoDEFAULTS-DCPFM, compare com os valores fixos do domínio e passe os caracteres de

separador de milhares e de decimais para o método que estiliza os campos, conforme a documentação do *plugin*.

Mensagens: essa é uma área mais aberta. É possível adotar uma linha de *design* um pouco mais moderna usando elementos *toast* (que se chamam assim porque parecem com torradas pulando da torradeira) usando *plugins* como o jquery.mobile.toast, mas eu considero que os *popups* do jQM já servem bem pra esse propósito. Eu fiz esse *fiddle* aqui bem simples pra mostrar como exibir algumas mensagens em *popups* jQM.

Tabelas: não é muito prático ou funcional mostrar uma tabela com uma tonelada de informações numa tela de dispositivo móvel, como faríamos normalmente num ALV. As tabelas de jQM são pensadas para terem poucas colunas (de 5 a 7, no máximo, é um intervalo razoável) e serem responsivas, mudando de formato em resoluções menores – veja os exemplos de *reflow* nos *demos* de jQM. Uma solução para exibir itens com muitos dados é usar um *collapsibleset*, onde somente se exibe uma lista de descrições dos itens e os detalhes aparecem quando se clica em uma linha (escondendo detalhes de outro item que porventura já estejam sendo exibidos). Outra solução, sem usar elementos nativos de jQM, é implementar um **carrossel**, onde também se exibe um item por vez e a navegação entre itens é feita horizontalmente. Essa técnica precisa de um pouco de CSS e jQuery pra funcionar, porém é claro que já existem vários *plugins* que fazem isso, como o OWL Carousel ou o slick. Eu fiz, como exemplos, um *fiddle* com um carrossel bem rudimentar e um outro mais complexo, com mais alguns controles, ambos sem usar *plugins* e sem muita estilização, mas mostrando como o conceito funciona.

Tirando tudo isso, o uso de jQuery e jQM oferece possibilidades praticamente ilimitadas para desenvolver interfaces. Por exemplo, o projeto que eu estou usando de exemplo nessa série, que é uma aplicação para coleta de materiais, se beneficiaria muito com um leitor de código de barras. Como integrar isso na aplicação? Com um *plugin* de jQuery, claro. O WebCodeCam é uma implementação simples que usa a API getUserMedia para capturar a imagem da câmera e identificar códigos de barra de vários tipos. O único problema é que não funciona no Safari nem no IE.

Bom, é isso por enquanto. Esse *post* tem mais *links* do que artigo da Wikipedia, mas a maioria aponta pras documentações de jQuery e jQM, então basta ir lá pra ter uma referência mais completa e concisa. Espero que eu tenha conseguido ajudar quem está desenvolvendo ou precisando desenvolver *views* BSP com jQM a não ficar completamente sem rumo.

Até a próxima (e última) parte, onde eu vou falar do desenvolvimento de *backend* (e onde finalmente vai ter um pouco mais de ABAP).

Referências (também conhecido como “TL;DR” ou “tá de sacanagem que eu vou ter que ficar caçando *link*“)

- jQuery API / jQuery Mobile API e jQuery Mobile *Demos*
- Sublime Text
- Moqups
- JSFiddle / Meu *dashboard* lá
- *Multipage template* vs *Multi* HTML *template in* jQuery Mobile
- HTTP *Methods*: GET vs. POST
- *Carousel design pattern*
- Navigator.getUserMedia()
- Plugins:
  - DateBox
  - jQuery Number
  - jquery.mobile.toast
  - OWL Carousel
  - slick
  - WebCodeCam

Edit

Eu imaginei que fossem faltar ~~algumas~~ várias coisas mesmo nesse turbilhão de informações aí. Vamos lá:

- Não ficou claro no *post*, mas ao contrário do que se faz normalmente com BSP puro, a ideia aqui é construir a *view* **SEM NENHUM CÓDIGO ABAP**. O *markup* é praticamente só HTML5, totalmente independente do BSP, com exceção de *tags* de OTR para os textos (<%= otr(...) %>) e da *tag* de página (normalmente eu uso<%@page language="abap" otrTrim="true" %>). A ideia aqui é escrever a *view* com HTML, JavaScript e CSS de maneira que se ela fosse removida totalmente do BSP e implementada junto com outro *backend*, mantendo somente a estrutura dos recursos e substituindo as *tags* de BSP, a lógica da *view* funcionaria exatamente da mesma maneira.
- De maneira nenhuma, absolutamente, NUNCA, eu repito, **NUNCA**, coloque lógica de negócio no JavaScript. JavaScript é código que 1) roda no *browser*, 2) pode ser depurado e analisado em tempo de execução, e 3) fica em *cache*, portanto isso é um problema de **segurança**. O JavaScript da *view* deve somente fazer coisas muito triviais com relação aos dados, que também não tenham relação alguma com dados de negócio, como por exemplo montar um seletor de ano:

  ```
  $('#tela_de_selecao').find('select.ano').each(function() {
      var year = new Date().getFullYear();
      for(var i = year - 10; i < year + 11; i++)
          $(this).append('<option value="' + i + (i === year ? '" selected="selected">' : '">'
      $(this).selectmenu('refresh', true);
  });
  ```

  Lógica de negócio fica no código ABAP, e isso fica no BSP. Falarei disso no próximo *post*.

Comentários

> **Mauricio Miao** — *13/06/2019 17:19*
>
> Sera que consigo aprender JQuery kkkk

> **Mauro Laranjeira** — *27/05/2015 09:07*
>
> Grande post Leo... parabéns..
>
> Cara, estou com 1 milhão de duvidas, haha... Mas acredito que seja porque nunca trabalhei diretamente com o BSP puro, trabalhei bastante com o BSP do CRM, que é bem diferente.
>
> Você usa um editor externo e depois importa para o MINE do SAP? Como ficam as regras do negocio, tudo misturado no JS com ABAP? E a consulta no banco, ficam na pagina do BSP ou como montaria um MVC loko?
>
> Fiquei doido com esse post ae rs...
>
> Cara, curti muito esse post, mais uma vez parabéns... é nois 😉

> **Leo Schmidt** — *27/05/2015 09:39*

Valeu Mauro!

Eu infelizmente tive que deixar pra falar de BSP depois de falar de frontend, porque tem certas coisas que não dá pra explicar como fazer no BSP sem apresentar antes o que eu fiz com JS, como por exemplo as chamadas em Ajax. Mas, com o risco de adiantar aqui algumas coisas do próximo post, vamos lá:

– Sim, como eu disse ali eu escrevo tudo de HTML, JS e CSS no Sublime e depois eu importo pro BSP. Vou falar melhor sobre isso e sobre como isso é organizado na aplicação BSP no próximo post.
– Não, não tem e **não deve ter nenhuma lógica de negócio** dentro de JS. Código da view só pode servir pra manipular a view. Em alguns raros momentos você pode MONTAR JavaScript dentro do ABAP (que é o que o standard faz no caso da tela de login, como eu falei no outro post, por exemplo), e em vários casos eu monto HTML dentro do ABAP, mas lógica de negócio NUNCA (sério, NUNCA). O HTML que eu construo não tem NADA de ABAP. Nada mesmo.
– Eu pretendo explicar o MVC que eu estou usando aqui no próximo post também, mas sim, coisas que envolvam o banco de dados (portanto, lógica de negócio) acontecem no código do BSP e NUNCA (sério, NUNCA) aqui no JS.

Vou dar um edit aqui no post pra compartilhar essas coisas com a galera 🙂

É nóis!

> **Mauro Laranjeira** — *08/06/2015 21:27*
>
> Leo,
>
> Muito show.. estou com bilhões de perguntas, mas vou esperar os próximos posts, tomar vergonha na cara e tentar fazer alguma coisa usando o Ajax.
>
> vlw mesmo...

Mauricio Cruz — *27/05/2015 08:58*

Muito legal, principalmente os links.

JSFiddle é uma das melhores coisas que inventaram. Usei muito para conseguir “compartilhar” um erro em fóruns como o Stackoverflow e deixar que outras pessoas avaliem e possam me ajudar com um erro. Exemplo: eu perguntando sobre JQM.

Valeu, abs!