BSP + jQuery Mobile, ou a aventura de SAP-webdev – Parte III: Backend

24/06/2015 10:00

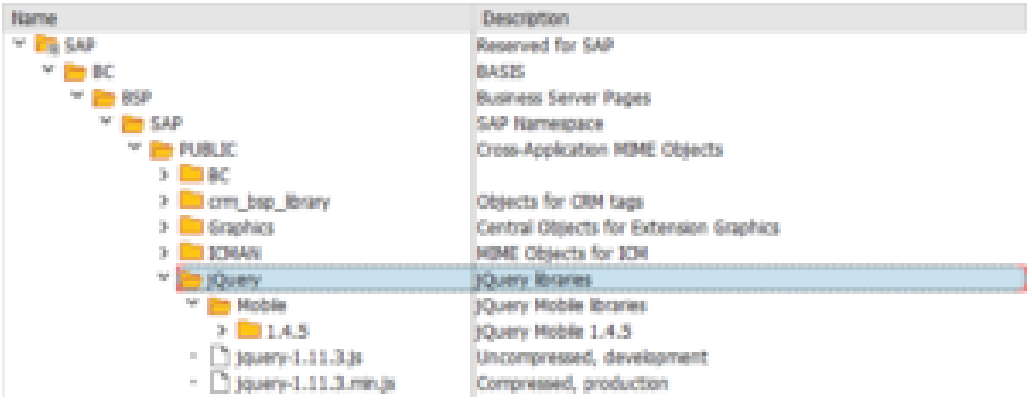
Acho que não demorou tanto quanto a saga do Freeza, mas sim caros zumbis, chegamos ao final desse guia! Vou falar agora da parte principal do desenvolvimento: a aplicação BSP. Aqui vou mostrar como estruturar os recursos no *backend* e também como ele vai responder a tudo o que acontece no *frontend*.

...mas antes, um pouco mais de *frontend*

Lembra, lá primeira parte, quando eu disse que é necessário subir as bibliotecas de jQuery e jQuery Mobile e os *plugins* no repositório MIME do SAP? Então, se você fez o *frontend* fora da SE80, é bem provável que o código que você fez ainda não esteja referenciando as bibliotecas que estão no repositório. Os seus *headers* podem estar mais ou menos assim:

```
<!doctype html>
<html>
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css" />
        <script src="http://code.jquery.com/jquery-1.11.3.min.js"></script>
        <script src="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js"></script>
        <!-- outros scripts, plugins e stylesheets vão aqui -->
    </head>
    <body>
        <!-- resto da página aqui -->
    </body>
</html>
```

O endereço da aplicação BSP, na maioria das vezes, vai ser algo comohttp://[servidor]:[porta]/sap/bc/bsp/[namespace]/[aplicação]. Aproveitando essa estrutura, podemos referenciar de maneira indireta as bibliotecas que estão disponíveis globalmente. Por exemplo, se você tiver organizado as bibliotecas no repositório MIME mais ou menos com a mesma estrutura do CDN do jQuery, desse jeito:



Você consegue mudar as declarações acima por algo assim (os endereços sempre são *case insensitive*):

```
<!doctype html>
<html>
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="../public/jquery/mobile/1.4.5/jquery.mobile-1.4.5.min.css" />
        <script src="../public/jquery/jquery-1.11.3.min.js"></script>
        <script src="../public/jquery/mobile/1.4.5/jquery.mobile-1.4.5.min.js"></script>
        <!-- outros scripts, plugins e stylesheets vão aqui -->
    </head>
    <body>
        <!-- resto da página aqui -->
    </body>
</html>
```

E agora eu posso finalmente explicar o porquê de usar o repositório MIME e fazer essas mudanças no *header*: é pra atender a política de segurança dos navegadores que se chama ***same-origin policy*** (política de mesma origem). Isso serve tanto para *views* que você vai usar no BSP como para as que você definiu nos métodos da classe de *login*.

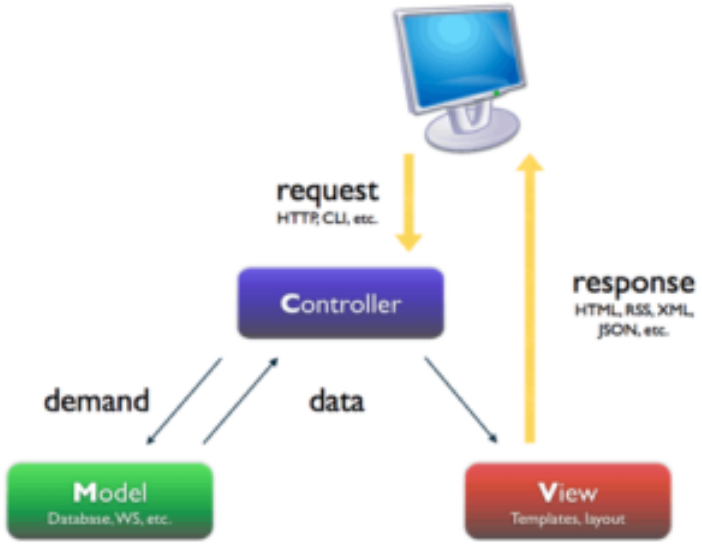
Outro detalhe que dá pra perceber pela imagem e pelo código: uma prática comum em desenvolvimento *web* é fazer versões ***minified*** (minimizadas ou comprimidas) de todos os arquivos que compõem as páginas, com o mesmo nome mais a extensão.min.[extensão original], de modo que a aplicação trafegue uma quantidade menor de dados entre servidor e navegador. Em tempo de desenvolvimento ou de *debug*, referencia-se o arquivo original, e depois que tudo estiver pronto, comprimimos e mudamos as referências para apontar para os arquivos comprimidos. O Sublime Text, que eu indiquei no *post* anterior, tem alguns *packages* que fazem isso, como por exemplo o Minify (para a glória da obviedade!), que comprime JavaScript e CSS. Eu não achei ainda nenhum *package* bacana pra minimizar HTML no Sublime, então eu uso ferramentas *on-line* mesmo como o HTML Minifier.

Estruturando a aplicação

Eu pensei em colar aqui uma imagem com o famoso diagrama do MVC, mas eu não sabia qual escolher:

Tem até um com o Homer ali

Vamos tentar o segundo ali mesmo:



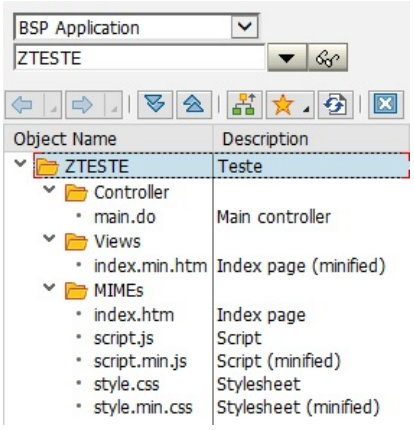
O fluxo representado por esse diagrama é basicamente o que a aplicação BSP deve fazer:

1. Usuário envia uma *request* HTTP ao *controller*;
2. *Controller* requisita dados do *model*;
3. A resposta depende do conteúdo: se a *request* foi comum, pedindo por uma *view*, a aplicação a instancia e esta é exibida ao usuário; se a *request* foi feita via Ajax, os dados interpretados são devolvidos diretamente do *controller* para a *view* pela *response*, que então os interpreta e exibe o resultado ao usuário.

Dessa maneira, uma aplicação BSP mais básica que use jQuery / jQuery Mobile será composta de uma classe para o *controller* e uma classe para o *model*. A *view* não precisa necessariamente de uma classe própria, porque a ideia é que a manipulação da *view* seja feita através dos *scripts* inseridos na página, sem intervenção ABAP – mas se você tiver que montar HTML muito complexo ou de maneira muito frequente dentro dos métodos do *controller* (vou falar sobre isso mais pra baixo), talvez seja melhor ter uma classe que faça somente isso. Essa estrutura pode ser repetida quantas vezes forem necessárias; se a lógica de tratamento das *requests*/*responses* ou da seleção dos dados é suficientemente complicada para que haja separação em mais *controllers* ou *models*, esse é o caminho.

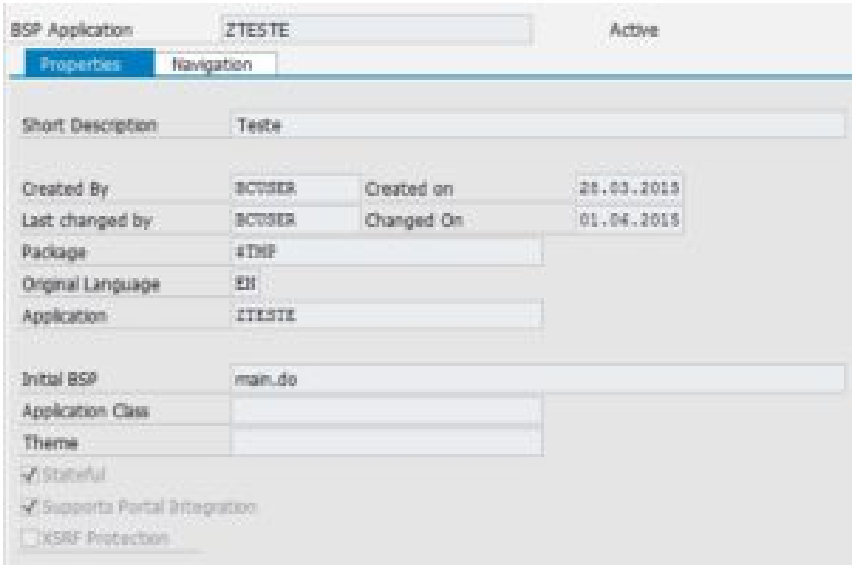
"Mas Leo, eu posso escrever tudo direto num *controller* só! Pra quê eu vou criar trocentas classes só pra UMA aplicação?" Porque eu espero que você vai ser uma pessoa boa e seguir a arquitetura MVC. Afinal, você não quer deixar um ninho de rato em forma de classe pra quem for dar manutenção na aplicação depois, certo? CERTO? Bom então.

A estrutura do BSP, usando a arquitetura *multipage* no *frontend*, ficaria mais ou menos assim:



Por partes, vamos.

Características da aplicação



Nada muito complexo por aqui:

- Em ***Initial* BSP** (“BSP página inicial”) se indica o **nome do *controller* principal** da aplicação;
- Normalmente não é necessário indicar uma ***Application Class*** (“Classe de aplicação”), mas se for necessário segregar a lógica da aplicação da aplicação em si, podemos indicar uma classe aqui, que pode ser acessada através do atributoAPPLICATION do *controller*;
- Não é necessário indicar nada em ***Theme*** (“Tema”), porque carrega-se os temas diretamente pelo *markup* com as *tags*<link>;
- A questão do estado da aplicação é aberta. É possível desenhar a aplicação de maneira que não se dependa de dados guardados no *model*, caso onde a opção ***Stateful*** (“Com status”) pode ficar desmarcada. Porém, se a aplicação trata uma quantidade muito grande de dados ou tem um fluxo lógico muito complexo, é melhor manter estados e essa opção precisaria ficar marcada;
- ***Supports Portal Integration*** (“Suporta integração Portal”) depende da sua aplicação. A integração com o Portal geralmente tem a ver com navegação e gerenciamento de sessões, parecido com o que acontece com aplicações Webdynpro ABAP. Ative conforme necessário;
- **XSRF *Protection*** (“Proteção XSRF”) é a proteção do *framework* BSP para prevenir *cross-site request forgery*. Esta SAP *note* explica o que é e como usar essa opção.

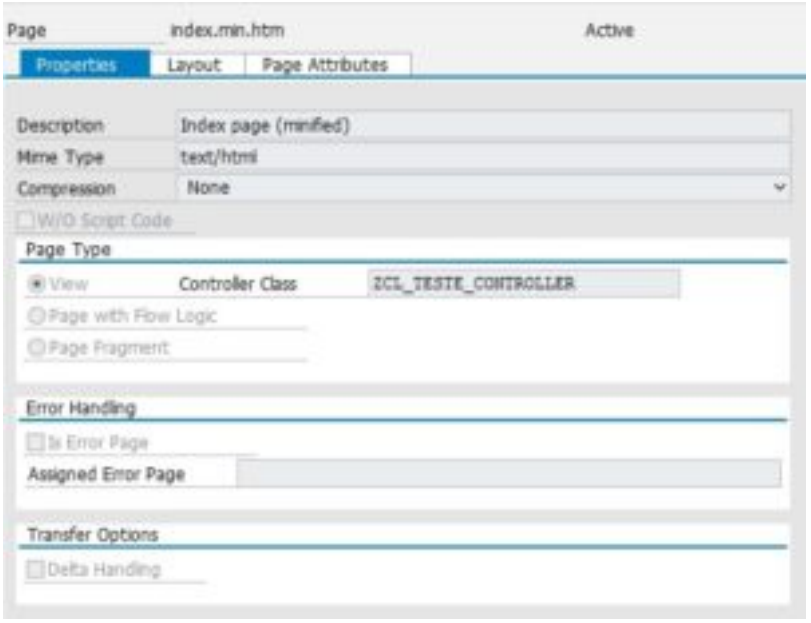
Objetos MIME

Aqui é onde se deve importar o código específico da sua *view*: o HTML descomprimido, os *scripts* comprimidos e descomprimidos, os *stylesheets* comprimidos e descomprimidos, e quaisquer outros objetos estáticos específicos para a sua aplicação: logotipos, vídeos, áudios, PDFs, etc. No HTML é possível referenciar diretamente os objetos que você importar usando somente o nome dos mesmos. O exemplo do cabeçalho que eu usei acima, então, ficaria:

```
<!doctype html>
<html>
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="../public/jquery/mobile/1.4.5/jquery.mobile-1.4.5.min.css" />
        <script src="../public/jquery/jquery-1.11.3.min.js"></script>
        <script src="../public/jquery/mobile/1.4.5/jquery.mobile-1.4.5.min.js"></script>
        <!-- outros scripts, plugins e stylesheets vão aqui -->
        <link rel="stylesheet" href="style.min.css" />
        <script src="script.min.js"></script>
```

```
    </head>
    <body>
        <!-- resto da página aqui -->
    </body>
</html>
```

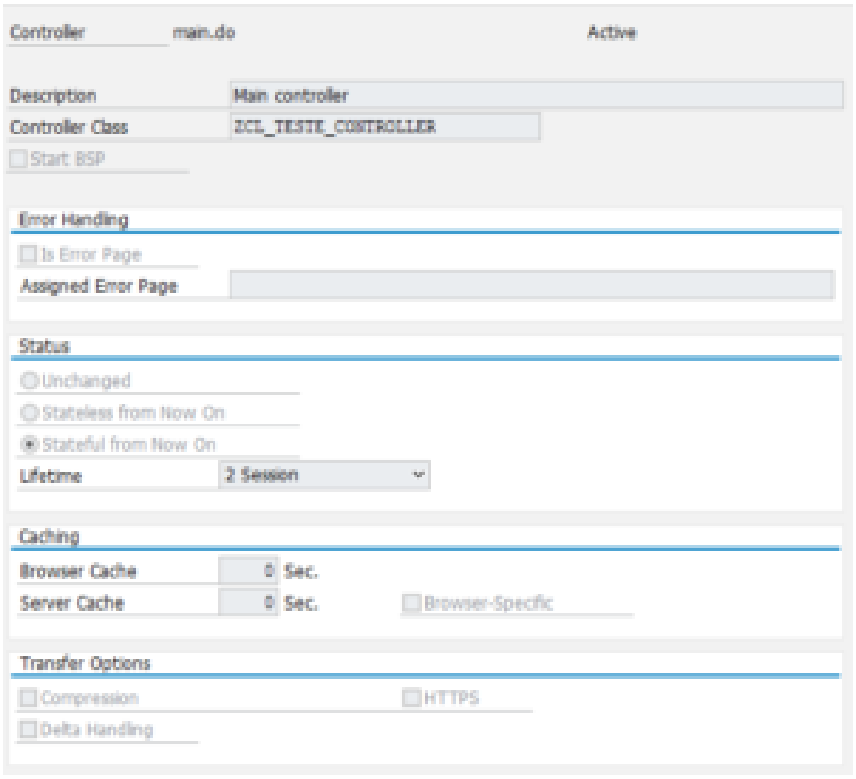
View



Aqui, também, nada extraordinário:

- Observe, em primeiro lugar, que a *view* contém o código HTML **comprimido**, que ficará na aba ***Layout*** – portanto o nomeindex.min.htm. A página com o código original descomprimido fica como objeto MIME do BSP, para futura referência/manutenção;
- É importante deixar o atributo ***Compression*** ("Comp.páginas") em branco (com o valor "*None*" ou "nenhuma"), porque os outros tipos de compressão podem afetar a sintaxe do HTML que já esteja minimizado;
- O atributo ***W/O Script Code*** ("sem cód.*script*") deve ficar **desmarcado**, pois a página contém *scripts*;
- O ***Page Type*** ("Tipo de página") é ***View*** ("Visão"), e você pode indicar qualquer *controller* ali (contanto que a classe seja descendente daCL_BSP_CONTROLLER – você pode inclusive indicar ela própria);
- Esta página não será uma página de erro (falei sobre elas na primeira parte da série), portanto o atributo ***Is Error Page*** fica **desmarcado**. Normalmente não se atribui aqui uma página de erro específica, pois as mensagens podem ser exibidas com outras técnicas (que eu expliquei na segunda parte da série), porém em ***Assigned Error Page*** ("Pág.erro atribuída") você pode indicar uma outra página dentro da aplicação que servirá como página de erro;
- Por fim, mantém-se **desmarcado** o atributo ***Delta Handling*** ("Tratmto.delta"), porque não queremos que o *framework* do BSP se intrometa na maneira como carregamos as páginas da aplicação.

Controller



Assim como nas outras páginas de configuração, nada muito estranho:

- O nome do *controller* pode ser como você quiser, e não precisa necessariamente terminar com.do. Para o *controller* principal é comum usarmain;
- Em ***Controller Class*** ("Classe controlador") indica-se a classe criada para servir de *controller* da aplicação. Esta classe deve ser descendente daCL_BSP_CONTROLLER;
- A opção ***Start* BSP** ("BSP início") funciona em conjunto com a opção XSRF *Protection* da aplicação, portanto veja a nota que eu mencionei mais acima para entender como usá-la;
- As opções de página de erro funcionam aqui da mesma maneira que funcionam na *view*;
- Com relação ao estado, a escolha aqui é a mesma que foi feita para a característica da aplicação em si, com a diferença que o estado do *controller* pode ser mantido (*stateful*) independentemente do estado da aplicação;
- O armazenamento em *cache* pode ter um tempo maior do que zero, mas isso funciona melhor em aplicações *stateless*. Em aplicações *stateful* as opções de ***Caching*** podem ficar em zero mesmo;
- ***Compression*** ("Compressão") e **HTTPS** são opcionais. A primeira comprime a página antes de enviá-la ao navegador, e a segunda trafega-a usando SSL;
- Assim como na *view*, deixe desmarcada a opção ***Delta Handling*** ("Tratmto.delta").

Classe do *controller*

Essa classe é responsável por receber, tratar e responder as *requests*. Como já falei algumas vezes aqui, essa classe precisa ser descendente daCL_BSP_CONTROLLER.

A ideia é aproveitar o mínimo dessa herança apenas para fazer com que o *controller* responda as *requests*, tanto as geradas pelo usuário quanto as feitas via Ajax. Para isso, é necessário:

1. **Redefinir e implementar o métodoDO_INIT.** A implementação desse método tem uma função muito simples, que é **inicializar o(s) *model(s)*** e guardar a referência da instância. Para isso, basta chamar o métodoCREATE_MODEL:

```
METHOD do_init.
  me->model ?= me->create_model(
      class_name = 'Z_MINHA_CLASSE_DO_MODEL'
      model_id   = 'MINHA_CLASSE_DO_MODEL'
  ).
ENDMETHOD.
```

   (FINALMENTE chegamos no ABAP, hein?) O parâmetromodel_id serve para encontrar a instância de *model* correta no atributoM_MODELS através do métodoGET_MODELS. Se você vai trabalhar com apenas um *model* (como é o caso desse exemplo), é mais prático guardar a instância do *model* diretamente num atributo novo da classe, como esseme->model que eu usei no exemplo acima, que no caso referencia a classeZ_MINHA_CLASSE_DO_MODEL.

2. **Redefinir e implementar o métodoDO_REQUEST.** Essa é a implementação principal. O fluxo é basicamente como nesse exemplo aqui:

```
METHOD do_request.

  DATA: cdata        TYPE string,
        code         TYPE i,
        content_type TYPE string,
        form_fields  TYPE tihttpnvp,
        message_type TYPE sy-msgty,
        reason       TYPE string.
```

```
* Assim como para o model, temos um atributo separado na classe para
* guardar a instância da view. Usamos aqui esse atributo para saber se a
* view já foi chamada:

  IF me->view IS INITIAL.

*   A instância da view ainda está vazia, portanto faremos só a
*   inicialização e chamada da view:

    me->view = me->create_view( view_name = 'index.min.htm' ).
    me->call_view( me->view ).

  ELSE.

*   A instância da view, aqui, já existe, portanto sabemos que a request
*   só pode ter sido feita via Ajax. Lembram dos exemplos de request
*   Ajax do post sobre frontend? Eles tinham endereços mais ou menos
*   assim:

*   main?action=algumaCoisa&...

*   Pois bem, esse atributo de URL chamado ACTION é a chave para
*   entender a request. Então vamos buscá-lo:

    CASE me->request->get_form_field_cs( 'action' ).

*     Uma request GET espera uma resposta com dados em algum formato
*     (HTML, JSON, XML, etc.) a partir de algum parâmetro. Então podemos
*     ter um método que retorne somente os dados, como por exemplo:

      WHEN 'getMaterialDescription'. "GET

        cdata = me->get_material_description( me->request->get_form_field_cs( 'material_number' ) ).

      WHEN 'getUnitDescription'. "GET

        cdata = me->get_unit_description( me->request->get_form_field_cs( 'material_unit' ) ).

*     Se tivermos vários parâmetros, a interface do método muda um pouco:

      WHEN 'getMaterials'. "GET

        me->request->get_form_fields_cs(
          CHANGING
            fields = form_fields
        ).

        me->get_materials(
          EXPORTING
            im_form_fields  = form_fields
          IMPORTING
            ex_cdata        = cdata
            ex_content_type = content_type
            ex_message_type = message_type
        ).

*     WHEN 'get...'. "GET

*       cdata = me->get_...

*     Já uma request POST pretende alterar dados a partir de parâmetros
*     informados pelo usuário. Os parâmetros de uma request POST
*     normalmente são enviados no corpo da request, codificados em
*     string da mesma maneira que seriam numa request GET - portanto
*     para lê-los precisamos da ajuda da classe CL_HTTP_UTILITY. Como
*     retorno, normalmente temos uma mensagem.

      WHEN 'createGoodsMovement'. "POST

        me->create_goods_movement(
          EXPORTING
            im_form_fields  = cl_http_utility=>string_to_fields( me->request->get_cdata( ) )
          IMPORTING
            ex_cdata        = cdata
            ex_content_type = content_type
            ex_message_type = message_type
        ).

      WHEN 'saveInventoryCount'. "POST

        me->save_inventory_count(
          EXPORTING
            im_form_fields  = cl_http_utility=>string_to_fields( me->request->get_cdata( ) )
          IMPORTING
            ex_cdata        = cdata
            ex_content_type = content_type
            ex_message_type = message_type
        ).

*     WHEN 'set...'. "POST

*       me->set_...(
```

```
      WHEN OTHERS.

*       Neste caso temos uma request Ajax sem o atributo action, ou seja,
*       não é uma request Ajax legítima para a aplicação. Isso pode
*       acontecer, por exemplo, quando o usuário executa um refresh (F5)
*       na página. Portanto, sabemos que a view já está criada, e basta
*       chamá-la novamente:

        me->call_view( me->view ).
        RETURN.

    ENDCASE.

*   Se o processamento da request Ajax resultou em erro, precisamos
*   informar um código de erro na resposta para que o frontend a
*   interprete corretamente:

    IF message_type = 'E'.
      code   = 422. "Unprocessable Entity - RFC 4918
      reason = 'Erro ao processar a request'(e01).
    ELSE.
      code   = 200. "OK
      reason = 'OK'.
    ENDIF.

    me->response->set_status(
        code   = code
        reason = reason
    ).

*   Os métodos de Ajax do jQuery são capazes de reconhecer
*   automaticamente o tipo do conteúdo da resposta, mas podemos tornar
*   as coisas mais claras indicando explicitamente valores como
*   'application/json' ou 'application/xml', conforme o retorno do
*   handler:

    IF NOT content_type IS INITIAL.
      me->response->set_content_type( content_type ).
    ENDIF.

*   Por fim, anexamos o conteúdo da resposta:

    IF NOT cdata IS INITIAL.
      me->response->set_cdata( cdata ).
    ENDIF.

  ENDIF.

ENDMETHOD.
```

"Ué Leo, mas de onde veio esse código 422 pra erro? Não é pra usar o 500?" Não, porque 500 é um código genérico pra dizer que aconteceu algum erro no servidor, e o *framework* do BSP já usa esse código pra informar *dumps*, o que é adequado. Os erros que acontecem já dentro dos *event handlers* na maioria das vezes são provocados por **entradas mal-formadas**, por culpa do usuário ou do processamento ABAP, e não necessariamente por causa do servidor. Por isso o erro devolvido aqui fica melhor na categoria 4xx (*client error*).

"Ah... mas então porque não usar logo o 400 – *Bad request* mesmo?" Porque o problema não é com a *request* em si, mas sim com os **dados contidos** (pra falar mais empolado, não é sintático, é semântico). Por exemplo: se o código do material não existe, isso não é necessariamente um problema com a *request*, mas sim com o próprio código que pode estar errado. Eis um *post* explicando um pouco sobre isso.

3. **Criar *event handlers* específicos para cada *action* Ajax.** Não precisam ser *event handlers* reais no contexto de ABAP *Objects*, mas sim apenas métodos que respondem a cada *action* Ajax definida no *frontend*, como eu mostrei nesse último exemplo. O fluxo de cada método vai ser simples: interpreta-se os parâmetros da *request*, transformando os dados conforme necessário, chama-se o método do *model* que tratará os dados e formata-se a resposta a ser devolvida aoDO_REQUEST. No exemplo abaixo, temos um *event handler* que vai ao *model* buscar uma lista de materiais a partir do centro e do depósito:

```
METHOD get_materials.

  DATA: material              TYPE string,
        material_descriptions TYPE makt_tab,
        message_text          TYPE string,
        number                TYPE string,
        plant                 TYPE werks_d,
        storage_location      TYPE lgort_d.

  FIELD-SYMBOLS: <field>                TYPE ihttpnvp,
                 <material_description> TYPE makt.

* Antes de mais nada é preciso traduzir a string de parâmetros que vem
* da request para variáveis que podemos usar em ABAP. O material
* continua em string aqui porque a ideia é que seja possível buscar
* tanto pelo MATNR como pelo MAKTX

  LOOP AT im_form_fields ASSIGNING <field>.
    CASE <field>-name.
      WHEN 'number'.
        material = <field>-value.
      WHEN 'plant'.
        plant = to_upper( val = <field>-value ).
      WHEN 'storageLocation'.
        storage_location = to_upper( val = <field>-value ).
```

```
      WHEN OTHERS.
    ENDCASE.
  ENDLOOP.

* Chamamos o model:

  me->model->get_materials(
    EXPORTING
      im_material              = material
      im_plant                 = plant
      im_storage_location      = storage_location
    IMPORTING
      ex_material_descriptions = material_descriptions
      ex_message_text          = message_text
      ex_message_type          = ex_message_type
  ).

  IF ex_message_type IS INITIAL.

*   Se o model não devolver nenhuma mensagem, montamos um JSON com os
*   dados:

    ex_cdata = |\{"materials":[|.

    LOOP AT material_descriptions ASSIGNING <material_description>.

      CALL FUNCTION 'CONVERSION_EXIT_MATN1_OUTPUT'
        EXPORTING
          input  = <material_description>-matnr
        IMPORTING
          output = number.

      ex_cdata = |{ ex_cdata }\{| &
                 |"number":"{ number }",| &
                 |"description":"{ <material_description>-maktx }"|.

      IF sy-tabix < lines( material_descriptions ).
        ex_cdata = |{ ex_cdata }\},|.
      ELSE.
        ex_cdata = |{ ex_cdata }\}|.
      ENDIF.

    ENDLOOP.

    ex_cdata = |{ ex_cdata }]\}|.

    ex_content_type = 'application/json'.

  ELSE.

*   Caso contrário, chamamos outro método para construir um HTML com um
*   popup para a mensagem:

    me->build_message_data(
      EXPORTING
        im_type         = ex_message_type
        im_text         = message_text
      importing
        ex_cdata        = ex_cdata
        ex_content_type = ex_content_type
    ).

  ENDIF.

ENDMETHOD.
```

O resultado desse método é um objeto JSON contendo um *array* de objetos representando os registros da MAKT que correspondem à busca feita pelo usuário, ou um outro objeto JSON com o tipo e texto da mensagem que será exibida em *popup*.

Existe um certo dilema sobre o tipo de conteúdo que deve ser usado na resposta de uma *request* Ajax. É melhor retornar JSON e montar a exibição pelo *script*, ou devolver HTML montado e somente integrá-lo na página, também pelo *script*? Essa decisão cabe ao desenvolvedor, e pode sem problemas resultar em implementações diferentes para cada método. No projeto que eu estou usando de exemplo, há um pouco de ambas as técnicas: alguns métodos devolvem HTML montado, outros devolvem JSON. A regra que eu uso é a seguinte: se for simples montar o HTML pelo *script*, o método retorna JSON; caso contrário, o método retorna o HTML já montado.

Classe do *model*

Aqui é onde acontece a "mágica", se por mágica entendermos o núcleo do trabalho ABAP – validação complexa das entradas, operações com o banco de dados, chamadas de funções, etc. Esta classe precisa ser descendente daCL_BSP_MODEL, porém, para este tipo de uso, não necessita da redefinição de nenhum dos métodos herdados.

Se estivermos trabalhando com uma aplicação *stateful*, os resultados das operações da aplicação ficarão armazenados em atributos dessa classe, e será necessário controlar em alguns pontos específicos os valores desses atributos. Por exemplo, uma tabela que represente o resultado da busca de um relatório pode precisar ser limpa toda vez que se chama a página do relatório, da mesma maneira que se faz com variáveis globais em um grupo de funções. Para isso é necessário criar uma chamada Ajax na entrada de cada página, e também métodos correspondentes no *controller* e no *model* para fazer a limpeza dos atributos.

É bom lembrar também que, numa aplicação *stateful*, as *requests* Ajax que resultam em um estado intermediário (ou seja, *actions* do tiposet*que apenas gravam dados nos atributos do *model*) **também estão alterando dados no servidor** e portanto devem ser iniciadas com o método **POST**.

Esta classe deve esperar parâmetros de entrada em seus métodos com dados já convertidos para os tipos ABAP, o que deve ser feito anteriormente pelo *controller*. Da mesma forma, os dados gerados pelo processamento do *model* são apenas retornados para o *controller*, que será responsável por formatá-los adequadamente e incluí-los na resposta, conforme o exemplo de *event handler* que está acima – portanto essa classe não deve ser responsável por montar HTML ou JSON. As mensagens, por exemplo, podem ser chamadas com o uso deMESSAGE ... INTOe retornadas em uma *string*, retornando também o valor deSY-MSGTY.

Assim chegamos ao fim desse guia maluco. Espero ter ajudado vocês a terem uma ideia de como usar tecnologias de *frontend*, e de como integrar isso com o BSP, e qual é o papel de cada um numa aplicação. Não é a coisa mais nova em folha em termos de desenvolvimento *web* pra SAP, mas pra quem gosta de interface isso dá um sabor diferente de WebDynpro ABAP e de SAP GUI. Espero que eu consiga mais pra frente fazer um guia espelho desse, usando Gateway e UI5, mas por enquanto é isso.

Dúvidas, reclamações ou sugestões, favor entrar em contato com o departamento de comentários abaixo ou via e-mail. Até mais!

Comentários

Max — *15/09/2015 15:55*

Opa...blz cara... ficou muito bom o post!! só tenho um problema na tela de login: ao trocar as referencias css e javascript para o endereço local, é requirida a autenticação para o css e javascript?! se eu ignorar esta autenticação, ele dá “401 Unauthorized” no cabeçalho de cada include.. se eu me autenticar, ele ignora a tela de login!
Você poderia me ajudar?! vlw!

> **Leo Schmidt** — *15/09/2015 16:35*
>
> Fala Max, valeu!
>
> Agora que vc falou, eu acabei testando aqui e tem esse problema mesmo. Realmente, não dá pra usar o mesmo caminho dos recursos na tela de login e na aplicação em si.
>
> A solução, que no caso é meio gambiarrenta, é subir as bibliotecas de jQuery e jQuery Mobile TAMBÉM em outro diretório no repositório. Um diretório público que a tela de login consegue enxergar é esse aqui:
>
> `[host:porta]/sap/public/bc/ur`
>
> Nesse diretório ficam as bibliotecas de Unified Rendering, que são responsáveis pelo layout da tela de login standard. Pode criar um diretório novo ali e subir tudo de jQuery que a tela de login vai conseguir exibir sem problema.
>
> Vou colocar essa informação nos posts. Valeu de novo!
>
> > **Max** — *17/09/2015 16:32*
> >
> > Vlw meu caro!
> > Fiz isso e funcionou perfeitamente! estou utilizando o bootstrap, mas se aplica da mesma forma!
> > Vlw a dica!

Mauricio Cruz — *06/07/2015 09:39*

Ficou animal! (só acabei de ler hoje haha).

Tendo trabalho com você com bsps tempos atrás, é muito legal ver quantas melhorias você fez da nossa singela arquitetura de primeira viagem, principalmente de performance (com a minificazizacação) e uso de JSONs 🙂

Parabéns e abs mano!

Flavio Furlan — *25/06/2015 17:29*

Muito interessante! Quando você fizer esse guia na SAPUI5 + Gateway estarei na primeira fila para pegar meu exemplar 🙂

Confesso que meu critério para escolha de qual diagrama MVC escolher foi o mesmo que o seu:
http://abap101.com/2012/04/01/falsa-programacao-orientada-objetos/

Abraços!