Manual do Usuário

ITX - Tecnologia da Informação Ltda. - Versão (IncognitaDigital)

Tabela de Conteúdos

1 Ajuda ao WebIntegrator

Versão 3.3

(Versão IncognitaDigital)

1.1 Introdução ao WebIntegrator

Introdução ao WebIntegrator (Versão IncognitaDigital)

A Tecnologia Web, denominação que se deu ao conjunto de protocolos e outros componentes de software sobre os quais se construiu a Internet, mostra a cada dia a sua superioridade, transformando-se na nova plataforma de suporte a sistemas de informação, capaz de integrar, sem fronteiras tecnológicas, as dimensões conhecidas como intranet, extranet e Internet. Em relação ao modelo cliente/servidor tradicional, esta tecnologia apresenta as seguintes principais vantagens:

1. Redução drástica do TCO. Diversos estudos comprovam que estações de trabalho baseadas em browsers (thin-client) apresentam redução do TCO de até quatro vezes em relação aos PC (fat-client).

2. Arquitetura aberta. A Tecnologia Web está fundada em protocolos e componentes de software totalmente abertos, muitos deles open-source. Isso significa mais flexibilidade para os desenvolvedores e independência de fornecedor, além de favorecer a interoperabilidade e a escalabilidade entre diferentes ambientes.

3. Uso eficaz de recursos. Operando sob demanda, as aplicações construídas com a Tecnologia Web utilizam com muito mais eficácia recursos normalmente caros, como processos

de usuário, base do licenciamento de sistemas operacionais e de bancos de dados. Um único processo de acesso a banco de dados, por exemplo, é capaz de atender a múltiplas requisições vindas de vários usuários. Ou seja, os recursos são alocados ao usuário apenas enquanto durar sua conexão, possibilitando o acesso de maior número de usuários concorrentes.

4. Onipresença. Enquanto no modelo tradicional é preciso instalar a camada cliente em cada uma das estações que irá executar determinada aplicação, no modelo Web o único requisito é a presença de um browser na estação de trabalho, que nem precisa ser um PC. Desse modo, resguardados os cuidados com a segurança, uma aplicação pode ser acessada de qualquer lugar, inclusive fora dos muros das organizações.

Nova interface. A interface gráfica da Tecnologia Web, baseada em linguagens de apresentação como HTML e JavaScript, representa um novo paradigma onde prevalece a simplicidade e facilidade de uso, aliados a mais produtividade na sua construção. Mesmo suportando conteúdo multimídia, a interface Web não requer super estações de trabalho para ser executada, ao contrário da interface GUI tradicional.

1.1.1 Visão Conceitual

Visão Conceitual

O WebIntegrator é um Servidor de Aplicações Web com ambiente de desenvolvimento integrado, capaz de agilizar o desenvolvimento de aplicações cliente/servidor que usam o cliente HTML (browser) acessando bancos de dados, com total independência entre os layouts das páginas e a codificação necessária para carregá-las.

Um projeto é composto por páginas que correspondem a todas as interfaces de interação com o usuário. As páginas devem ser criadas no ambiente de desenvolvimento do WebIntegrator (**WI Builder**) e editadas em qualquer editor HTML ou, até mesmo no próprio Builder. Tudo que existir na página e que não estiver entre | (pipes) será desconsiderado pelo sistema. As demais estruturas serão entendidas como variáveis (ou identificadores) e o sistema tentará substituí-las. Dessa forma, o desenvolvedor pode colocar Applets, Flash, links para páginas ASP, páginas PHP etc. sem que haja o menor tipo de problema de exclusão de código.

As variáveis podem ser do usuário, do sistema, grids, combos etc. e quando referenciadas, o sistema as processa em tempo de execução e substitue o seu conteúdo. Variáveis do usuário ou do sistema podem ser referenciadas em vários campos do **WI Builder**, como nos SQL, campos de definição etc.

Quando um formulário HTML é submetido todas as variáveis não começadas por "**`wi`**" são adicionadas ao contexto do usuário (o contexto é criado sempre que um novo browser é aberto) podendo, então, ser referenciadas a qualquer momento.

Sempre que um comando SQL é executado, o nome das suas colunas transformam-se em variáveis, passando a fazer parte do contexto e podendo ser referenciadas. Mas, diferentemente

das variáveis do usuário, após o término do processamento do SQL elas são removidas da memória, pois deixam de ter representatividade.

1.1.2 Tipos de Variáveis

Tipos de Variáveis

A princípio, quaisquer dados de campo de formulário, parâmetros passados por URL ou variáveis criadas a partir de componentes de pré ou pós-página, são armazendos pelo WebIntegrator como variáveis de sessão, mas a depender da utilização de alguns prefixos, o comportamente e o tempo de vida dessas variáveis, podem mudar.

Variáveis de sessão que comecem com o prefixo **`tmp.`** são conhecidas pelo WebIntegrator como variáveis temporárias. Esse tipo de variável, tem um tempo de vida pré-determinado onde as mesmas são removidas da sessão, automaticamente, pelo WebIntegrator, logo após a execução de um pré-página. Geralmente, a maioria das variáveis e identificadores de um projeto do WebIntegrator são formandos por variáveis temporárias.

Variáveis de sessão que comecem com o prefixo **`pvt.`** são conhecidas pelo WebIntegrator como variáveis privadas. Essas variáveis persistem durante todo o tempo de vida da sessão e apenas podem ser removidas, ou terem seus valores alterados, através do uso dos componentes de pré ou pós-páginas. Geralmente, variáveis privadas são utilizadas para armazenar informações que são acessadas com frequência pelo projeto.

Existe um grupo especial de variáveis **`pvt.debug`** e **`tmp.debug`** que serve para ativar diversos logs de debug do WI as quais podem não podem ser passadas por url e o desenvolvedor define se quer ou não que elas sejam da sessão.

Variáveis de sessão que comecem com o prefixo **`wi.`** são conhecidas pelo WebIntegrator como variáveis de sistema. As variáveis de sistema, normalmente são apenas de leitura, não sendo permitido ao desenvolvedor criar variáveis desse tipo. Ele apenas poderá acessar seus valores e, em alguns casos, atribuir valores. Para saber quais as variáveis de sistema disponíveis clique aqui.

O uso de qualquer outro prefixo nos identificadores de variáveis, fará com que o WebIntegrator sempre mantenha as variáveis armazenadas na sessão sendo que as mesmas poderão ser apagadas e terem seus valores alterados tanto por componentes de página ou através dos métodos GET (passagem de parâmetros via URL) ou POST.

1.1.2.1 Variáveis do Sistema

Variáveis do Sistema

O WebIntegrator oferece ao desenvolvedor algumas variáveis de sistema que sempre se encontram disponíveis independente da aplicação que esteja sendo executada ou dos componentes por ela usadas.

Variáveis com prefixo **`wi.`** são sempre reservadas para uso pelo WebIntegrator e apenas podem ter o seu conteúdo manipulado através dos componentes de pré-página ou pós-página. A grande maioria das variáveis reservadas do WebIntegrator são apenas de leitura, as únicas que não seguem essa regra são **`wi.proj.prev`** e **`wi.page.prev`** que são variáveis de leitura e escrita.

A variável **`wi.proj.prev`** serve para forçar um valor indicando qual o projeto que foi anteriormente executado, já a variável **`wi.page.prev`** serve para indicar qual pós-página o WebIntegrator irá executar.

Para ver todas as variáveis existentes numa requisição é só colocar **<wi:out/>** na página. É possível utilizar uma máscara colocando **<wi:out mask="tmp.*" />**.

Logo abaixo segue uma tabela com a relação das variáveis atualmente disponibilizadas pelo WebIntegrator, algumas dessas variáveis apenas estarão disponíveis se determinados tipos de componentes forem executados.

Variável	Descrição	Exemplo
wi.date.ansi	Retorna a atual data do sistema onde está instalado o WebIntegrator no formata ANSI aaaa-mm-dd.	2001-05-28
wi.date.day	Retorna o dia do mês atual de acordo com a data do sistema onde está instalado o WebIntegrator.	28
wi.date.dmy	Retorna a atual da data do sistema onde está instalado o WebIntegrator no formato dd/mm/aaaa.	28/05/2001
wi.date.hm	Retorna a hora atual do sistema onde está instalado o WebIntegrator no formato hh:mm.	09:25
wi.date.hms	Retorna a hora atual do sistema onde está instalado o WebIntegrator no formato hh:mm:ss.	09:25:36
wi.date.hour	Retorna as horas da atual hora do sistema onde está instalado o WebIntegrator.	09
wi.date.internal	Retorna o número de milisegundos desde de 1º de Janeiro de 1970, 00:00:00 GMT em relação à data-hora atual do sistema onde está instalado o WebIntegrator.	
wi.date.mdy	Retorna a data atual do sistema onde está instalado o WebIntegrator no formato mm/dd/aaaa.	05/28/2001
wi.date.min	Retorna os minutos da atual hora do sistema onde está instalado o WebIntegrator.	25
wi.date.month	Retorna o nome do mês da atual data do sistema onde está instalado o WebIntegrator.	Maio
wi.date.sec	Retorna os segundos da atual hora do sistema onde está instalado o WebIntegrator.	36
wi.date.wday	Retorna o nome do dia da semana da atual data do sistema onde está instalado o WebIntegrator.	Segunda-feira
wi.date.year	Retorna o número do ano com 4 dígitos da atual data do sistema onde está instalado o WebIntegrator.	2001

wi.date.ymonth	Retorna o número do mês da atual data do sistema onde está instalado o WebIntegrator.	05
wi.email	Retorna o endereço do e-mail para qual está sendo enviada a mensagem quando a opção "Processar cada cópia" estiver habilitada.	
wi.email.attid	Retorna o identificador global associado a um e-mail.	
wi.email.ok	Retorna o status final do envio de e-mails.	false
wi.error	Retorna um erro genérico do WebIntegrator.	
wi.headers	Retorna uma lista com os nomes dos HEADERS disponíveis para o browser em questão pois a depender do browser a lista varia.	accept-language, connection, accept, host, accept-encoding, user-agent, referer
wi.header.<nome-do-HEADER>	Retorna o valor de um HEADER específico.	
wi.mjava.ident	Retorna o identificador de qual rotina foi pedida para ser executada.	
wi.mjava.session	Retorna o valor da sessão do WebIntegrator.	
wi.mjava.nspace	Retorna o nome do *namespace* onde vai ser executada a rotina MUMPS.	
wi.mjava.user	Retorna o nome do usuário que foi usado para abrir conexão com o MJava.	
wi.mjava.pass	Retorna a senha do usuário que foi usada para abrir conexão com o MJava.	
wi.page.id	Retorna o identificador da página.	inserehtml
wi.page.name	Retorna o último token do identifcador da página. O símbolo de separação dos tokens é a barra (/) ou a barra invertida (\).	inserehtml
wi.proj.prev	Retorna o identificador do projeto que foi executado anteriormente.	
wi.page.prev	Retorna o identificador da página que foi exibida anteriormente.	inserehtml
wi.proj.title	Retorna o título do projeto.	
wi.page.title	Retorna o título da página.	
wi.session.ip	Endereço IP da sessão.	
wi.session.id	Id da sessão.	

wi.session.host	Nome da máquina cliente.	
wi.server.host	Nome do servidor.	
wi.server.port	Porta do servidor.	
wi.server.prot	Protocolo do servidor.	
wi.server.url	URL completa do servidor.	
wi.request.querystring	Querystring do metodo GET.	
wi.request.url	URL completa do que foi submetido.	
wi.request.parameters	Parâmetros recebidos na requisição.	

1.1.2.2 Variáveis de um Grid

Variáveis de um Grid

Logo abaixo seguem as listas de variáveis que se encontram disponíveis ao desenvolvedor quando se estiver utilizando um componente grid. O sinal de asterisco indica que as variáveis são de apenas leitura.

Nome da variável	Descrição
grid.<identificador>.from	Indica o número da linha do ResultSet usado pelo grid a partir do qual estão sendo exibidos os resultados.
grid.<identificador>.hasMore*	Retorna true se houver mais resultados a serem exibidos, caso contrário retorna false. Essa variável sempre retornará false quando o campo Quantidade não estiver definido.
grid.<identificador>.limit*	Retorna a quantidade de linhas que serão exibidas, no máximo, em cada página. Esse valor equivale ao que foi definido no campo Quantidade.
grid.<identificador>.link*	Retorna uma barra de navegação básica para o grid com os links para acessar os resultados anteriores e posteriores.
grid.<identificador>.linkFull*	Retorna a barra de navegação completa do grid, extendendo a funcionalidade da propriedade grid.<identificador>.link ao incluir os links para as páginas que exibem os primeiros e últimos registros.

grid.<identificador>.linkIndex*	Retorna uma barra de navegação que apenas contém os índices de navegação das páginas dos resultados do grid.
grid.<identificador>.linkIndexSize	Indica a quantidade máxima de links de navegação das páginas que podem ser exibidos juntamente com a barra de navegação (grid.<identificador>.link) formando um índice dos links para as páginas que contêm os resultados, semelhante aos que aparecem nas páginas de resultados de sites de pesquisas.
grid.<identificador>.linkBack*	Retorna o link que chama a página com os resultados anteriores do grid.
grid.<identificador>.linkGo*	Retorna um link que chama a página com os próximos resultados do grid.
grid.<identificador>.next	Indica o número da linha do ResultSet a partir da qual serão exibidos os próximos resultados.
grid.<identificador>.prev	Indica o número da linha do ResultSet a partir da qual linha serão exibidos os resultados anteriores.
grid.<identificador>.rowCount	Indica a quantidade de linhas que foram retornadas como resultado (ResultSet) para o grid. O funcionamento dessa variável depende de algumas funcionalidades que devem ser implementadas pelo driver JDBC. Caso o driver JDBC não ofereça tais funcionalidades, o desenvolvedor poderá atribuir um valor específico a essa propriedade.
rowID*	Retorna o identificador (número) específico de uma linha do resultado (ResultSet) exibido pelo grid. Essa variável apenas estará disponível no momento do processamento do grid, ou seja, ela apenas poderá ser referenciada no modelo do grid.
rowSeq*	Retorna o número de sequencia de uma linha da tabela do modelo do grid. Essa variável apenas estará disponível no momento do processamento do grid, ou seja, ela somente poderá ser referenciada no modelo do grid.
grid.<identificador>.size*	Retorna a quantidade de linhas que estão sendo exibidas por um grid. Caso o campo Quantidade esteja definido o valor máximo dessa variável será igual a grid.<identificador>.limit.
grid.<identificador>.to	Indica o valor até qual linha estão sendo exibidos os resultados.
grid.<identificador>.txtBack	Indica o conteúdo do link que chama a página que exibe os registros anteriores. O desenvolver poderá configurar essa propriedade para personalizar o conteúdo do link colocando textos, imagens, etc.
grid.<identificador>.txtBackOff	Indica o conteúdo do link que chama a página que exibe os registros anteriores mas quando não há mais registros a serem exibidos. O desenvolver poderá configurar essa propriedade para personalizar o conteúdo do link colocando textos, imagens, etc.

`grid.<identificador>.txtFirst`	Indica o conteúdo do link que chama a página que exibe os primeiros registros. O desenvolver poderá configurar essa propriedade para personalizar o conteúdo do link colocando textos, imagens, etc.
`grid.<identificador>.txtFirstOff`	Indica o conteúdo do link que chama a página que exibe os primeiros registros mas quando não há mais registros o serem exibidos. O desenvolver poderá configurar essa propriedade para personalizar o conteúdo do link colocando textos, imagens, etc.
`grid.<identificador>.txtGo`	Indica o conteúdo do link que chama a página que exibe os próximos registros. O desenvolver poderá configurar essa propriedade para personalizar o conteúdo do link colocando textos, imagens, etc.
`grid.<identificador>.txtGoOff`	Indica o conteúdo do link que chama a página que exibe os próximos registros mas quando não há mais registros a serem exibidos. O desenvolver poderá configurar essa propriedade para personalizar o conteúdo do link colocando textos, imagens, etc.
`grid.<identificador>.txtMid`	Indica o conteúdo que será exibido como separador entre `grid.<identificador>.txtLinkBack` e `grid.<identificador>.txtLinkGo`.

1.1.2.3 Variáveis de um XML

Variáveis de um processamento XML

Após o processamento de um conteúdo que tenha seguido o padrão de formatação XML e desde que o desenvolvedor tenha habilitado a decodificação o WebIntegrator gera automaticamente algumas variáveis de sessão. Todo elemento XML processado pelo WebIntegrator possui as seguintes propriedades:

Propriedade	Descrição
`all()`	Retorna todos os elementos e respectivos valores das propriedades associadas a cada elemento do conteúdo XML que foi processado.
`root()`	Retorna o nome do elemento raiz da estrutura XML.
`error()`	Retorna o erro que foi encontrado durante o processamento XML.
`size()`	Retorna o quantidade de elementos que há dentro deste nó.

Para acessar os elementos que fazem parte do nó raiz basta acessá-los atavrés de seus índices como se estivesse acessando valores armazenados em vetores. Para exemplificar suponha que o conteúdo do arquivo tenha sido importado para uma sessão do WebIntegrator e que a decodificação XML estivesse habilitada durante este processo.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ROOT [
  <!ELEMENT ROOT (#PCDATA)>
  <!ENTITY nbsp  >
  <!ENTITY cr &#13;>
  <!ENTITY lf &#10;>
]>
<root>
 <child>
  <rowid>1</rowid>
  <cod>3</cod>
  <nome>Fulano de Tal</nome>
   <root2>
    <rowid2>1</rowid2>
    <cod2>3</cod2>
   </root2>
 </child>
 <child>
  <rowid>2</rowid>
  <cod>4</cod>
  <nome>Sicrano</nome>
   <root2>
    <rowid2>2</rowid2>
    <cod2>4</cod2>
   </root2>
 </child>
</root>
```

Continuando com nosso exemplo digamos que o conteúdo desse arquivo foi gravado na variável **`xml`**. Para acessarmos os valores dos elementos que compõem esse arquivo usamos uma notação semelhante à de vetores:

```
|<identificador>|[<indice>].<nome-do-elemento>
```

sendo que o indice sempre começa de zero. Para demonstrar, se quiséssemos saber todo o conteúdo e os valores das propriedades dos elementos que se encontram em **`xml`** poderíamos fazer uso da propriedade **all()** que ela retornaria o seguinte:

```
xml.size()=2
xml[0].size()=4
xml[0].rowid.size()=0
xml[0].rowid=1
xml[0].cod.size()=0
xml[0].cod=3
xml[0].nome.size()=0
xml[0].nome=Fulano de Tal
xml[0].root2.size()=2
xml[0].root2.rowid2.size()=0
xml[0].root2.rowid2=1
xml[0].root2.cod2.size()=0
xml[0].root2.cod2=3
xml[1].size()=4
xml[1].rowid.size()=0
```

```
xml[1].rowid=2
xml[1].cod.size()=0
xml[1].cod=4
xml[1].nome.size()=0
xml[1].nome=Sicrano
xml[1].root2.size()=2
xml[1].root2.rowid2.size()=0
xml[1].root2.rowid2=2
xml[1].root2.cod2.size()=0
xml[1].root2.cod2=4
```

1.1.2.4 Todas as variáveis

Todas as variáveis do WI

Variáveis Internas da Sessao:

```
  bld-usr
  bld-ip
  bld-prj
  bld-page
  bld-report

  exp-sql0
    ...
  exp-sql9

  eng-context-wi
  eng-proj-default
  end-proj-admin
  eng-proj-teste
  eng-db-brs-connection (guarda a conexão)
  eng-db-brs-resultset  (guarda o ultimo resultset)
```

Variáveis do Contexto:

```
  wi.  - exclusivas do WI
  tmp.       - temporárias
  pvt.       - acesso privado
  grid.      - grid
  combo. - combo
  list.      - list (deprecated)
  file.  - file (deprecated)
```

Variáveis do update:

```
  result.status()  (valor retornado do BD na execução do update)
                   (Se for maio ou igual a zero com autocommit
desativado
                    indica a execução correta de todos os comandos)
  result[1]        (resposta em multiplos updates)
  result.size()    (quantidade de registros em multiplo)
```

Variáveis WI.

```
  wi.grid.form           //formulario do grid
  wi.prefix              //prefixo a ser adicionado aos campos sem "."
do formulario
  wi.jsp.filename        //arquivo jsp que esta sendo executado
  wi.token               //token gerado para a pagina
```

```
wi.token.ok        //validaçao do token recebido
  wi.token.received      //token recebido (uso interno)
  wi.token.proj          //projeto que enviou o token (uso interno
redirecionamento)
  wi.token.page          //pagina que enviou o token (uso interno
redirecionamento)
  wi.redirect            //indica o ID para onde esta sendo
redirecionado
  wi.mjava.ident         // acessiveis dentro do mumps
  wi.mjava.session              ...
  wi.mjava.nspace               ...
  wi.mjava.user                 ...
  wi.mjava.pass                 ...
  wi.mjava.proj                 ...
  wi.mjava.page                 ...
  wi.pwd.md5            (chave MD5 do WI)
  wi.email              (Email do usuario no eachone do Sendmail)
  wi.email.ok           (Resultado do envio de email SMTP)
  wi.email.ids          (Ids dos arquivos de email SMTP)
  wi.email.attid        (Identificador global do Attachment)
  wi.session.ip
  wi.session.id
  wi.session.host
  wi.server.host
  wi.server.port
  wi.server.prot         (protocolo: http ou https)
  wi.server.url
  wi.proj.title          (título do projeto)
  wi.page.title          (título da página)
  wi.headers             (lista de todos os headers enviados)
  wi.header.referer      (conteudo de um header especifico)
  wi.request.querystring (querystring do metodo GET)
  wi.request.url         (url completa do que foi submetido)
  wi.request.parameters  (parametros recebidos nessa requisição)
  wi.date.day
  wi.date.wday           (week day)
  wi.date.month
  wi.date.ymonth         (year month)
  wi.date.year
  wi.date.ansi           (2001-04-30)
  wi.date.dmy            (22/03/2001)
  wi.date.mdy            (03/22/2001)
  wi.date.hour
  wi.date.min
  wi.date.sec
  wi.date.hm             (hh:mm)
  wi.date.hms            (hh:mm:ss)
  wi.date.internal       (internal long date)
  wi.proj.path           (caminho do projeto)
  wi.upl.ok              (upload ok)
  wi.upl.ext             (upload extensao)
  wi.upl.path            (nome de diretorio produzido)
  wi.upl.filename        (nome do arquivo produzido)
  wi.xml.root            (gridxmlout)
  wi.xml.child           (gridxmlout)
--
  wi.sql.query (ultima query que deu erro)
  wi.sql.error (ultimo erro sql)
  wi.sql.msg   (mensagem do erro SQL - somente no update e login)
```

```
wi.error     (erro generico do WI)
--
 |$wi.brs(|tmp.docn|)$|  (recupera o documento)
  wi.brs.first
  wi.brs.last
  wi.brs.next
  wi.brs.prev

-- controles especiais --
  wi.proj.id
  wi.proj.parent
  wi.db.id              (apenas num componente que acesso o banco de
dados)
  wi.page.id
  wi.page.name          (nome da pagina)
  wi.proj.prev.bck      (prev do redirect)
  wi.page.prev.bck      (prev do redirect)
  wi.proj.prev
  wi.page.prev
  wi_proj_prev
  wi_page_prev
```

Variáveis Especificas do Upload:

```
  CAMPO.size = tamanho do arquivo enviado
```

Variáveis de Login:

```
  pvt.wilogin          (ativa o login do projeto usando |wi.session.ip|)
  tmp.msglogin         (mensagem do login)
```

Variaveis do Event

```
  wi.event.id           (id do select ou update que está sendo
executado)
```

Variáveis Grid.

```
  Parâmetros utilizáveis nos registros

  |rowid|
  |rowid0|
  |rowseq|

  Parâmetros utilizáveis nos conectores JavaGrid

  |grid.limit|               - limite em Grid Java
  |grid.id|                  - id em Grid Java

  Parâmetros internos dos Grids

  |grid.ALGO.recursivecontrol|  - controle de recursividade
  |grid.ALGO.reference|         - controle de referencia no XML

  Parâmetros públicos dos Grids

 *|grid.ALGO.next|
  |grid.ALGO.prev|
  |grid.ALGO.from|
```

```
|grid.ALGO.to|
  |grid.ALGO.limit|        - registros a serem mostrados (informado na
definição)
  |grid.ALGO.size|         - registros mostrados (to - from)
  |grid.ALGO.rowcount|     - máximo de registros encontrados (só no SQL
que suporte)
  |grid.ALGO.hasmore|      - true/false indica tem continuação
  |grid.ALGO.lastpage|     - indica o primeiro elemento da última página
  |grid.ALGO.contentlimit| - indica o limite no tamanho do conteudo em
mb (default 5)
  |grid.ALGO.linkfirst|
  |grid.ALGO.linkback|
  |grid.ALGO.link|
  |grid.ALGO.linkgo|
  |grid.ALGO.linklast|
  |grid.ALGO.linkfull|
  |grid.ALGO.linkindex|
 *|grid.ALGO.linkindexsize|
 *|grid.ALGO.txtfirst|
 *|grid.ALGO.txtfirstoff|
 *|grid.ALGO.txtback|
 *|grid.ALGO.txtbackoff|
 *|grid.ALGO.txtgo|
 *|grid.ALGO.txtgooff|
 *|grid.ALGO.txtlast|
 *|grid.ALGO.txtlastoff|
 *|grid.ALGO.txtmid|
```

Variáveis do sql:

```
  tmp.disableMaxRows - Desabilita a limitação de 50000 registros numa
requisição
  pvt.disableMaxRows - Desabilita a limitação de 50000 registros na
sessão
```

1.1.3 Tipos de Grid

Tipos de Grid

Os tipos de *grid* HTML que podem ser gerados encontram-se listados abaixo, clique em cada um deles para saber qual a função deles e quais as variáveis de contexto que se encontram disponíveis para cada um. Vale atentar para o fato de que essas variáveis são apenas de **leitura** (read-only).

- FTP

- LOCAL
- POP3
- ATTACH
- JAVA
- WIOBJECT

1.1.3.1 FTP

FTP

O tipo FTP serve para que uma lista de arquivos disponíveis em um certo diretório de um servidor FTP seja gerada. As variáveis disponíveis ao desenvolvedor para exibir as informações sobre os arquivos são as que seguem relacionadas logo abaixo:

Variável	Descrição
`rowid`	Retorna o identificador único de uma linha da tabela.
`rowseq`	Retorna o sequencial de uma linha da tabela.
`name`	Retorna o nome arquivo.
`type`	Retorna o tipo de arquivo, se for um diretório o valor retornado será **D** se for um arquivo, **F**.
`size`	Retorna o tamanho do arquivo em bytes.
`date`	Retorna a data da última modificação feita no arquivo.
`time`	Retorna a hora da última modificação feita no arquivo.
`permission`	Retorna quais as permissões do arquivo. O formato está no padrão UNIX.
`owner`	Retorna o nome do dono do arquivo.
`group`	Retorna o nome do grupo ao qual o arquivo pertence.

1.1.3.2 LOCAL

LOCAL

O tipo LOCAL serve para que uma lista de arquivos disponíveis em um certo diretório da máquina onde o WebIntegrator está instalado seja gerada. As variáveis disponíveis ao desenvolvedor para exibir as informações são exibidas quando o usuário gera o arquivo do grid pela primeira vez. A saber:

Variável	Descrição
`rowid`	Retorna o identificador único de uma linha da tabela.
`rowseq`	Retorna o sequencial de uma linha da tabela.
`name`	Retorna o nome arquivo.
`type`	Retorna o tipo de arquivo, se for um diretório o valor retornado será **D** se for um arquivo, **F**.
`size`	Retorna o tamanho do arquivo em bytes.
`date`	Retorna a data da última modificação feita no arquivo.
`time`	Retorna a hora da última modificação feita no arquivo.
`permission`	Retorna quais as permissões do arquivo. O formato está no padrão UNIX.

1.1.3.3 POP3

POP3

O tipo POP3 serve para que uma lista dos e-mails de um determinado usuário de uma conta num servidor POP3 seja gerada. As variáveis disponíveis ao desenvolvedor para exibir as informações são exibidas quando o usuário gera o arquivo do grid pela primeira vez. A saber:

Variável	Descrição
`rowid`	Retorna o identificador único de uma linha da tabela.
`rowseq`	Retorna o sequencial de uma linha da tabela.
`subject`	Retorna o valor do campo *Subject* (assunto) do e-mail.
`size`	Retorna o tamanho da mensagem em bytes.
`date`	Retorna a data que a mensagem foi recebida.
`from`	Retorna quem é(são) o(s) remetente(s) da mensagem.

1.1.3.4 ATTACH

ATTACH

O tipo ATTACH serve para que uma lista dos anexos de um e-mail seja gerada. As variáveis disponíveis ao desenvolvedor para exibir as informações são exibidas quando o usuário gera o arquivo do grid pela primeira vez.

Variável	Descrição
`rowid`	Retorna o identificador único de uma linha da tabela.
`rowseq`	Retorna o sequencial de uma linha da tabela.
`name`	Retorna o nome do anexo.
`binary`	Retorna **true** se o conteúdo do anexo estiver num formato binário caso contrário retorna **false**.
`mime`	Retorna o tipo MIME (MIME TYPE) do anexo.
`href`	Retorna a URL que realizará o download do anexo.
`link`	Retorna o link já montado de acordo com o valor de **href** e **name**, a estrutura retornada é sempre essa: `<a href="|href|">|name|</a>`

1.1.3.5 JAVA

JAVA

O tipo JAVA está sempre vinculado a um grid Java, para saber mais sobre o grid Java clique aqui.

1.1.3.6 WIOBJECT

WIOBJECT

O tipo WIOBJECT está sempre vinculado a um objeto do contexto do WebIntegrator que possua a propriedade `size()` e seus valores possam ser acessados semelhante à sintaxe de *`arrays`*, para saber mais sobre o grid WIOBJECT clique aqui.

1.1.4 Tipos MIME

Tipos MIME

# tipo MIME	Extensão
application/andrew-inset	ez
application/mac-binhex40	hqx
application/mac-compactpro	cpt
application/msword	doc
application/octet-stream	bin dms lha lzh exe class
application/oda	oda
application/pdf	pdf
application/postscript	ai eps ps
application/smil	smi smil
application/vnd.mif	mif
application/vnd.ms-excel	xls
application/vnd.ms-powerpoint	ppt
application/x-bcpio	bcpio
application/x-cdlink	vcd
application/x-chess-pgn	pgn
application/x-compress	Z
application/x-cpio	cpio
application/x-csh	csh
application/x-director	dcr dir dxr
application/x-dvi	dvi
application/x-futuresplash	spl
application/x-gtar	gtar
application/x-gzip	gz
application/x-hdf	hdf
application/x-koan	skp skd skt skm
application/x-latex	latex
application/x-netcdf	nc cdf
application/x-sh	sh
application/x-shar	shar
application/x-shockwave-flash	swf
application/x-stuffit	sit
application/x-sv4cpio	sv4cpio
application/x-sv4crc	sv4crc
application/x-tar	tar
application/x-tcl	tcl
application/x-tex	tex
application/x-texinfo	texinfo texi
application/x-troff	t tr roff
application/x-troff-man	man
application/x-troff-me	me
application/x-troff-ms	ms
application/x-ustar	ustar
application/x-wais-source	src
application/zip	zip

audio/basic	au snd
audio/midi	mid midi kar
audio/mpeg	mpga mp2 mp3
audio/x-aiff	aif aiff aifc
audio/x-pn-realaudio	ram rm
audio/x-pn-realaudio-plugin	rpm
audio/x-realaudio	ra
audio/x-wav	wav
chemical/x-pdb	pdb xyz
image/vnd.wap/wbmp	wbmp
image/bmp	bmp
image/gif	gif
image/ief	ief
image/jpeg	jpeg jpg jpe
image/png	png
image/tiff	tiff tif
image/x-cmu-raster	ras
image/x-portable-anymap	pnm
image/x-portable-bitmap	pbm
image/x-portable-graymap	pgm
image/x-portable-pixmap	ppm
image/x-rgb	rgb
image/x-xbitmap	xbm
image/x-xpixmap	xpm
image/x-xwindowdump	xwd
message/rfc822	eml
model/iges	igs iges
model/mesh	msh mesh silo
model/vrml	wrl vrml
text/css	css
text/html	html htm
text/plain	asc txt text
text/richtext	rtx
text/rtf	rtf
text/js	js
text/sgml	sgml sgm
text/tab-separated-values	tsv
text/x-setext	etx
text/xml	xml
text/vnd.wap.wml	wml
text/vnd.wap.wmlscript	wmls
video/mpeg	mpeg mpg mpe
video/quicktime	qt mov
video/x-msvideo	avi
video/x-sgi-movie	movie
x-conference/x-cooltalk	ice

1.1.5 Condição

Condição

No campo **Condição**(| | [1]) defini-se em que caso a operação relacionada deverá ser efetuada. Sempre que o resultado do processamento total da condição for verdadeiro a operação relacionada será realizada.

Os comparadores de condição são:

Comparador	Descrição
TRUE	é sempre verdadeiro
FALSE	é sempre falso
=	igual (compara a igualdade de 2 textos, não é "sensitive case")
!=	diferente (compara a desigualdade de 2 textos, não é "sensitive-case")
==	duplo igual (compara se são exatamente iguais, é "sensitive case")
>	maior do que (verifica se o identicador da esquerda é maior do que o da direita)
!>	não maior do que(verifica se o identicador da esquerda não é maior do que o da direita)
<	menor do que (verifica se o identicador da esquerda é menor do que o da direita)
!<	não menor do que (verifica se o identificador da esquerda não é menor do que o da direita)

Exemplos:

Condição	Resultado
nome = Nome	Retorna TRUE
nome == Nome	Retorna FALSE
|botao| = gravar	Retorna TRUE se o valor do identificador botao for gravar
texto > 5	Retorna FALSE pois em conversões numéricas feitas pelo WI os valores textos valem 0 (zero)

Os comparadores booleanos são:

Identificador	Descrição
&&	"E" booleano, apenas retorna verdadeiro se todos os segmentos forem verdadeiros.
||	"OU" booleano, apenas retorna falso se todos os segmentos forem falsos.

OSB: O **||** (or) tem precedencia sobre o **&&** (and)

Exemplos:

Condição	Resultado
`nome = webintegrator`	Retorna FALSE porque "nome" não é igual a "webintegrator"
`|nome| = webintegrator`	Pode retornar TRUE se o valor do identificador nome for igual a "webintegrator"
`nome = webintegrator || 3 < 5`	Retorna TRUE porque 3 é menor do que 5
`true || 3 < 5 && nome = webintegrator`	Retorna FALSE porque ambos os lados do "OU" são falsos.
`true || (3 < 5 && nome = webintegrator)`	Retorna TRUE pois com o uso dos parentesis o AND é executado primeiro.

Além dos exemplos citados acima, ainda é possível fazer uso de funções definidas pelo usuário onde essas deverão retornar "TRUE" ou "FALSE" para que a condição seja verdadeira ou falsa, respectivamente. Essas funções estão definidas em Funções.

A sintaxe para o uso de funções é simples, seguindo o modelo que é exibido abaixo:

```
|$<nome-da-funcao>([<parametro1>,...,<parametro#>])$|
```

1.1.6 Funções

Funções

Este é o local onde o administrador registra funções definidas pelo usuário para serem utilizadas em projetos do WebIntegrator. Funções definidas pelo usuário são classes Java que implementam a interface `br.com.itx.integration.InterfaceFunction`, para saber maiores detalhes sobre a implementação e o uso de funções clique aqui.

Uma função pode ser registrada globalmente, neste caso ela ficará acessível a todos os projetos, ou registrada especificamente em um projeto. Quando uma função for registrada globalmente o desenvolvedor deve se assegurar de que todos os projetos tenham acesso à função, para isso a classe Java ou o arquivo .jar que representa a função deverá ser instalada num local acessível a todos os projetos ou no repositório de classes interno de cada projeto, `<WEBAPP_PATH>/WEB-INF/lib` se for um arquivo **.jar** ou `<WEBAPP_PATH>/WEB-INF/classes` se for arquivo **.class**.

O campo `Classe` serve para que seja indicado o nome completo da classe Java que representa uma função do WebIntegrator.

O campo `Global` indica se a classe Java que representa uma função será registrada globalmente. Se esta opção estiver marcada no momento do registro da função, essa função ficará acessível a todos os projetos do WebIntegrator caso contrário ela apenas será acessada no projeto onde estiver sendo feito o registro.

Vale atentar que se a função for registrada globalmente a biblioteca ou classe Java que a representa deverá ser instalada em um local onde todos os projetos do WebIntegrator tenham acesso ou ser copiada para o repositório de classes interno do projeto, `<WEBAPP_PATH>/WEB-INF/lib` se for um arquivo **.jar** ou `<WEBAPP_PATH>/WEB-INF/classes` se for arquivo **.class**.

O campo `Nome` indica o nome pelo qual a classe Java que representa uma função será referenciada nos projetos do WebIntegrator.

1.1.7 Detalhes SQL

Detalhes SQL

O campo `SQL` é o local reservado para que o desenvolvedor possa digitar comandos a serem executados por um SGBD sendo que esse comando pode ser uma instrução SQL, uma chamada a uma rotina MUMPS, um método do CachéFactory® ou uma chamada a uma *`stored procedure`* que será submetida a um banco de dados específico. A instrução ou comando digitado poderá conter qualquer referência a variáveis que façam parte do contexto de um projeto do WebIntegrator.

Em virtude das diversas possibilidades de instrução clique naquela que você quiser saber maiores detalhes:

- SQL
- CachéFactory
- BR Search

1.1.7.1 SQL

SQL

SQL (**Structued Query Language**) é uma linguagem projetada para ser usada com banco de dados relacionais e seus comandos são divididos em duas categorias, os comandos DML (Data Manipulation Language) que lidam com dados, ou recuperando ou modificando-os para mantê-los atualizados; e os comandos DDL (Data Description Language) que lidam com criação ou alteração das tabelas, views (visões) e índices.

Dessas duas categorias a mais usada no WebIntegrator, certamente, será a DML. Logo abaixo segue uma lista dos comandos DML mais comuns.

- **SELECT:** usado para consultar e exibir dados de um banco de dados. O comando SELECT especifica quais colunas serão incluídas no resultado. A grande maioria das instruções SQL usadas nas aplicações são comandos SELECT;
- **INSERT:** adiciona novas linhas a uma tabela. INSERT é usado para popular uma tabela recém-criada ou para adicionar uma ou mais linhas a uma tabela já existente;
- **DELETE:** remove uma linha especificada ou um conjunto de linhas de uma tabela;
- **UPDATE:** altera um valor existente em uma coluna de uma tabela.

Logo abaixo temos alguns exemplos de como o campo SQL pode ser preenchido:

```
SELECT * FROM Funcionarios

INSERT INTO Funcionarios VALUES (|codigo|,"|nome|")

DELETE FROM Funcionarios WHERE cod = |codigo|

UPDATE Funcionarios SET
  nome = "|nome|", endereco = "|endereco|" WHERE cod = |cod|
```

Quando um comando sql é executado o valor das colunas é recuperado pelo getString mas se for um texto longo
o desenvolvedor pode especificar para recuperar usando o getAsciiStream através da função wi.longTextColumns. Ex:

```
|$wi.longTextColumns(3,5)$|
SELECT cod, nome, anamnese, nascimento, historico FROM Paciente
```

Por padrão o WI irá criar um Statement com a instrução substituída pelo resultado do processamento dos pipes (caso haja filtros de sql esses caracteres serão removidos das variáveis). Para fazer com que o WI processe a instrução usando um PreparedStatement é necessário que o pipe seja prececido, encostado, a um sinal "?". Ex:

```
INSERT INTO Pessoa (cod, nome) VALUES (?|tmp.cod|, ?|tmp.nome|)
```

Quando não é explicitamente informada o tipo de atribuição (cast) o WI irá utilizar o setString, mas é possivel fazer um
comando definindo qual comando set deverá ser utilizado:

```
select * from pessoa where nome like ?[str]|tmp.ini|
insert into pessoa (cod, nome) values (?[int]|tmp.cod|,
?[str]|tmp.nome|)
```

O tipos disponiveis para o colchete são:
asc (setAscii), bin (setBinary), dat (setDate), int (setInt), lon (setLong), flt (setFloat), dbl (setDouble), str (setString)

Quando é utilizado [bin] e o valor começa com "file:" será procurado o arquivo correspondente no disco.
Ex: O valor "file:/tmp/arquivo.jpg"

Quando o cast é especificado o WI faz o set correspondente e caso haja erro de cast ele define null
(resolve problemas de drivers que obrigam o set correspondente como no informix). Ex:

```
insert into teste (data) values (?[dat]|tmp.data|)
Se data for vazio fica null no bd.

insert into teste (data) values (?|tmp.data|)
Se data for vazio fica 0000-00-00 no bd.
```

Quando é especificado [str] é setString, mas se o texto for null (insensitivo) fica null para não violar uma chave
estrangeira, já o convencional, quando não especifica, ficaria o texto null.
Ex: Uma combo de estado onde a seleção não é obrigatorio (o item selecione teria value="null")

Existe uma função que retorna a lista de tabelas ou views de uma conexão:

```
|$wi.listmeta(table)$|
```

Existe uma função que retorna a meta-estrutura de uma tabela, e é ela:

```
|$wi.listmetastruct(tabela)$|
```

Para executar uma chamada a uma *`stored procedure`* utilize a sintaxe específica do sistema gerenciador de banco de dados (SGBD) que estiver sendo utilizado. No caso do SGBD Oracle a sintaxe a ser usada deve ser a seguinte:

```
{ call <nome-da-stored-procedure>([<param1>, ...]) }
{ call ? := <nome-da-stored-procedure>([<param1>, ...]) }
```

IMPORTANTE:

Ao chamar *`stored procedures`* Oracle assegure-se que:

1. se ela retornar um conjunto de resultados, os mesmos sejam referenciados através de um cursor. O código da stored procedure deve ser semelhante a:

```
CREATE OR REPLACE FUNCTION <nome-da-stored-procedure>([<param1>,
...])
RETURN types.ref_cursor
AS
   <nome-do-cursor> types.ref_cursor;
BEGIN
   OPEN <nome-do-cursor> FOR
   ...
   RETURN <nome-do-cursor>;
END;
```

2. se ela realizar atualizações e/ou inserções de dados que seja retornado um valor numérico. Valores negativos servem para indicar que houve algum erro durante a execução da stored-procedure ou valores positivos para indicar a quantidade de linhas que foram alteradas e/ou inseridas.

3. para passar parâmetros por referência é necessário colocar ".out" no nome da variável que usa Prepared Statement para indicar que o seu retorno dese ser capturado.
Ex (oracle): `{ call myprocedure(?|tmp.cod|, ?|tmp.nome.out|) }`

1.1.7.2 CachéFactory

CachéFactory

O tipo de conexão **CacheFactory**®, por se tratar de uma conexão de acesso a uma base de dados de objetos, possui três maneiras distintas e específicas de executar consultas ou alterações em um banco de dados uma seria usando a interface relacional, a outra usando as *Queries* relacionais definidas no Caché ou utilizando chamadas diretas aos métodos das classes.

Para utilizar a interface relacional basta colocar o prefixo **&sql** seguido da instrução SQL que se deseja executar. Lembre-se que durante a compilação o Caché informa o nome dado às tabelas. Logo abaixo segue uns exemplos:

```
&sql SELECT * FROM webintegrator_database.CacheTest

&sql INSERT INTO webintegrator_database.CacheTest VALUES
(|codigo|,|nome|,|telefone|)
```

Ao utilizar as *Queries* relacionais definidas no Caché é só colocar o prefixo **&sql query** seguido do nome da *query*. Podem ser usados **.** (ponto) ou _ (sublinhado) uma vez que um objeto

sempre é separado por `.` e o WebIntegrator converte os símbolos. Logo abaixo segue uns exemplos:

```
&sql query webintegrator_database.CacheTest_byName

&sql query webintegrator.database.CacheTest.byName(|name|)

&sql query webintegrator.database.CacheTest.byName(|name|)
```

Os métodos devem ser construídos segundo as regras abaixo:

- Método para efetuar um comando UPDATE deve receber os parâmetros necessários e retornar %Status. Exemplo:

  ```
  webintegrator.database.CacheTest.save(|codigo|,|nome|,|telefone|)
  ```

- Método para efetuar um comando SELECT deve ter o primeiro parâmetro como sendo do tipo webintegrator.database.CacheResultSet e retornar vazio (void). Pode ter outros parâmetros como valor usado na codificação. Na chamada dentro do WebIntegrator deve-se ignorar o parâmetro de referência (CacheResultSet). Durante a codificação utiliza-se métodos como: response.AddHeader, row.AddField, response.AddRow, etc. Exemplo:

  ```
  webintegrator.database.CacheTest.show()
  ```

1.1.7.3 BR Search

BR Search

O **BR/Search** é um produto voltado para o gerenciamento de informações não estruturadas. Trata-se de um banco de dados onde o tipo de dado "Texto Corrido" é tratado com o mesmo nível de importância que qualquer outro campo chave, tradicionalmente utilizado nos bancos de dados que suportam o modelo relacional, embora não siga tal modelo. Em virtude disso, usa-se uma sintaxe proprietária para recuperação e inserção de informações.

Para recuperar todos os documentos que constam na base de dados, a sintaxe é: `@docn`. Para listar apenas os documentos cujo `ID` seja maior do que `20`, a sintaxe é: `@docn > 20`. Em geral, a sintaxe usada para recuperar documentos baseados em `ID` é:

```
@docn<condição>
```

Para realizar a pesquisa por um determinado texto na base basta digitar o texto a ser pesquisado, caso queira que a pesquisa seja feita em apenas um determinado parágrafo especifique-o entre colchetes. A sintaxe é a seguinte:

```
<texto> [<parágrafo>]
```

Existe uma função que retorna a meta-estrutura de uma base, e é ela:

```
|$wi.listmetastruct$|
```

Você também poderá fazer buscas com operadores *booleanos* permitindo que termos sejam combinados por meio de operadores lógicos. Exemplo:

```
café [title] AND @docn > 20 AND NOT leite [title]
```

A expressão acima solicita que seja feita uma busca pela palavra **café** no parágrafo `title` cujos `IDs` dos documentos sejam maior do que `20` e que a palavra **leite** não apareça no parágrafo `title`.

Para personalizar a consulta existem alguns parâmetros que podem ser utilizados entre chaves no inicio da expressão e separados por "ponto e virgula" e são eles:

```
          hlon ou highlighton - define o inicio do highlight. Ex: <b>
        hloff ou highlightoff - define o final do highlight. Ex: </b>
 subparon - define o que deverá vir antes de cada subparagrafo (só funciona no
                                texto.subpars). Ex: <b>
   subparoff - define o que deverá vir após cada subparagrafo (só funciona no
                                texto.subpars). Ex: </b>
     sort - define a ordenação utilizada nos paragrafos. Ex: PAR1,PAR2,-PAR3
    plural - define o uso ou não de plural (default false). Ex: true ou false
         adjacentlevel - define o nível de adjacência da consulta. Ex: 3
      thesaurus - define o uso de um thesaurus. Ex: teste ou teste [BT,NT]
database - define um database a ser consultado (default o da conexão). Ex: SAMP
```

Para ativar o destaque das palavras pesquisadas defina as propriedades `hlon` e `hloff` com as tags que serão usadas para marcar o início e o fim do destaque, a sintaxe é a seguinte:

```
{ hlon=<tag-inicial> ; hloff=<tag-final> } <texto-de-pesquisa>
```

Por exemplo, para que a palavra **leite** venha destacada em negrito no resultado de uma pesquisa poderia-se usar o seguinte:

```
{ hlon=<b> ; hloff=</b> } leite
```

Um consulta ao BRSearch cria um vetor de subparágrafos para cada parágrado, baseando-se em seu nome e com suporte ao Grid ObjetoWI:

```
Parágrafo "Texto":
texto = valor do paragrafo
texto.subpars = valor de todos os subparagrafos (utiliza subpar on/off)
texto.subpars.size() = quantidade de subparagrafos
texto.subpars[1].value = valor de um determinado subparagrafo
```

É interessante para o desenvolvedor saber que a última consulta realizada ao BRSearch é armazenada no WI, sendo assim existe uma forma dinâmica de recuperar um determinado `docn` (detalhe) que é utilizando um gravar num objeto WI com a função `|$wi.brs(|tmp.docn|)$|` e também passarão a existir "`wi.brs.first, wi.brs.last, wi.brs.next, wi.brs.prev`"

Para realizar inserções na base de dados usa-se uma sintaxe semelhante àquela usada em

instruções SQL:

```
DELETE WHERE @docn=|tmp.id|

INSERT VALUES (|tmp.ti|,|tmp.tx|,|tmp.cr|)
INSERT (ti,tx,cr) VALUES (|tmp.ti|,|tmp.tx|,|tmp.cr|)
INSERT (ti longo,tx,cr longo) VALUES (|tmp.ti|,|tmp.tx|,|tmp.cr|)
INSERT (ti_longo,tx,cr_longo) VALUES (|tmp.ti|,|tmp.tx|,|tmp.cr|)

UPDATE VALUES (|tmp.ti|,|tmp.tx|,|tmp.cr|) WHERE @docn=|tmp.id|
UPDATE (ti,tx,cr) VALUES (|tmp.ti|,|tmp.tx|,|tmp.cr|) WHERE
@docn=|tmp.id|
UPDATE (ti longo,tx,cr longo) VALUES (|tmp.ti|,|tmp.tx|,|tmp.cr|) WHERE
@docn=|tmp.id|
UPDATE (ti_longo,tx,cr_longo) VALUES (|tmp.ti|,|tmp.tx|,|tmp.cr|) WHERE
@docn=|tmp.id|
```

onde `ti_longo` é o *`input name`* sem "`.`" e "`:`"

1.2 Definições

Definições

- Projeto
- Banco
- Servidores

1.2.1 Projeto

Definições de Projeto

Este é o local onde são feitas as definições gerais do projeto. Para fazer as definições de qual ou quais banco de dados e servidores (FTP, POP3, etc) serão usados no projeto, no canto esquerdo aparecem os links que levam às definições de Banco de dados e Servidores.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Pacote de classes java do projeto indica em que pacote serão colocadas as classes java do projeto que o WI necessitar gerar. Ex: br.com.empresa faz com que a geração de classe Teste fique em br.com.empresa.Teste.

O campo Modelo para as Páginas indica qual o arquivo que o WI Builder tomará como modelo na hora da criação das páginas. Esse arquivo deverá ficar no diretório `template` relativo ao repositório de definições projeto. Se esse campo for deixado em branco o WI Builder adotará o seu modelo de página padrão.

No conteúdo da página do modelo o desenvolvedor poderá fazer uso dos identificadores `|head.content|` ou `|body.content|` que servirão de referências para o WIzard saber em que parte do modelo deverá ser incluído os conteúdos do **HEAD** e do **BODY**, respectivamente, gerados por ele.

O campo Projeto Pai (herança) indica o projeto do qual serão herdados componentes, páginas, etc. (detalhes sobre herança de projeto)

O campo Variáveis Persistentes indica como será o gerenciamento do ciclo de vida de variáveis criadas por projetos do WebIntegrator. Com esta opção marcada o ciclo de vida de variáveis segue as regras abaixo:

- variáveis com prefixo `tmp.`, `combo.`, `grid.` (variáveis temporárias) são consideradas variáveis de requisição, ou seja, o tempo de vida delas é o tempo de vida de uma requisição;
- variáveis com prefixo `pvt.` (variáveis privativas) são consideradas variáveis de sessão da aplicação, ou seja, o tempo de vida delas é o mesmo tempo de vida da sessão. Variáveis privativas apenas podem ser criadas ou terem seus valores alterados através de componentes de pré ou pós-página;
- variáveis com prefixos diferentes dos citados acima ou sem algum prefixo também são consideradas variáveis de sessão. Ao contrário das variáveis privativas, esse tipo de variável também pode ser criado ou ter seu valor alterado diretamente através dos métodos **GET** e **POST**.

Com essa opção desmarcada o ciclo de vida de variáveis segue as regras abaixo:

- variáveis com prefixo `app.` são consideradas variáveis de contexto da aplicação, ou seja, o tempo de vida delas é o mesmo tempo de vida da aplicação. Esse tipo de variável apenas pode ser criada ou ter seu valor alterado através de componentes de pré ou pós-página;
- variáveis com prefixo `pvt.` são consideradas variáveis de sessão da aplicação, ou seja, o tempo de vida delas é o mesmo tempo de vida da sessão. Assim como as variáveis de contexto da aplicação, essas variáveis apenas podem ser criadas ou terem seus valores alterados através de componentes de pré ou pós-página;
- variáveis com prefixos diferentes dos citados acima ou sem algum prefixo são consideradas variáveis de requisição, ou seja, o tempo de vida delas é o tempo de vida da requisição que as gerou.

O campo Diretório de Logs indica aonde devem ser gerados os logs do projeto como webintegrator.log, dbtime.log, builder.log, etc. Se o campo estiver vazio será assumido o WEB-INF/logs do projeto. Ao alterar o valor desse campo será necessário efetuar o *restart* do projeto ou pelo builder ou pelo manager do tomcat.

O campo Página com lógica comum a todas serve para indicar uma página que terá o seu pré executado antes do pré de qualquer página assim como o pós. Exemplos de uso: autorização de usuários (colocado no pré), validação de WIToken (colocado no pós). Qualquer outra coisa que você quer que seja executada em todas as páginas você coloca nessa página e a indica no projeto. O desenvolvedor pode controlar a execução dos elementos através da condição usando a variável |wi.page.id| para identificar a página que está sendo executada.

O campo JSP com maior compatibilidade faz com que as páginas JSP geradas sejam mais compatíveis com outros ambientes (Ex: WebSphere). Quando essa opção está selecionada não serão incluídas as chamadas JSTL e os parâmetros passados pelos componentes do WI terão o "\r" removido e o "\n" transformado em espaço. Ao alterar essa opção no projeto será necessário regerar todos os arquivos JSP.

1.2.1.1 Login

Login

Este é o local onde o desenvolvedor define se o projeto será protegido pelo sistema de login do WI e quais são os parâmetros de configuração.

O campo Ativar Login serve para que o desenvolvedor indique se o projeto será protegido pelo sistema de login do WI.

O campo Usuário BD serve para que o desenvolvedor indique se o login será validado pela conexão do usuário ao banco de dados, ao invés de ser validado por um comando sql.

O campo Utiliza MD5 serve para que o desenvolvedor indique se o login deve utilizar criptografia MD5.

O campo Banco de Dados serve para que o desenvolvedor informe qual banco de dados será utilizado para o login.

O campo SQL serve para que o desenvolvedor indique o comando sql para validar o login. Quando não utiliza MD5 se algum registro for retornado o login será considerado válido e o valor das colunas será armazenado em pvt.login.[coluna]. Quando o login usa MD5 a primeira coluna deve ser o valor MD5 no banco de dados e as demais colunas serão armazenadas em pvt.login.[coluna]. A variável pvt.wilogin contendo o ip da sessão indica que a mesma está logada no projeto.

O campo Filtrar serve para que o desenvolvedor informe quais caracteres devem ser removidos das variáveis do comando sql que não estão precedidos por ? (PreparedStatement).

O campo Página Inicial ou de Login serve para que o desenvolvedor indique a página que deve ser apresentada na entrada do sistema e quando a sessão expirar.

O campo Mensagem de Login indica qual mensagem será dada quando o login não for válido.

1.2.1.2 Editor de arquivos texto

Editor de arquivos texto

Este é o local onde o desenvolvedor pode alterar o conteúdo de qualquer arquivo texto do projeto.

O campo Arquivo do projeto serve para que o desenvolvedor indique qual o caminho completo dentro do projeto para o arquivo a ser modificado. Lembrando que nos ambientes Unix faz difença letras maiúsculas e minúsculas.

Na lateral direita do campo Arquivo do projeto o usuário pode encontrar um ícone onde ao clicá-lo será mostrada um lista com todos os arquivos `"js,css,xml,txt,htm,html"` do projeto excluindo-se a pasta WEB-INF/definitions para facilitar sua escolha. Mesmo um arquivo do projeto que não apareça na lista pode ser editado bastando para isso escrever na caixa de texto o caminho completo para o arquivo e clicar em abrir.

Para renomear um arquivo basta antes de clicar no botão salvar alterar o nome do mesmo na caixa de texto e até mesmo o caminho dele se for da necessidade do usuário

OBS: Uma pasta vazia pode ser apaga usando esse utilitário, bastando para isso colocar na caixa de texto o caminho da pasta e clicar em remover.

1.2.1.3 Variáveis de Inicialização

Variáveis de Inicialização

Este é o local onde o desenvolvedor define variáveis de inicialização do projeto.

O campo Nome serve para que o desenvolvedor indique qual o nome da variável a ser inicializada.

O campo Valor serve para que o desenvolvedor indique qual o valor que deve ser inicializado.

1.2.1.4 Arquivo de Constantes

Arquivo de Constantes

No diretório WEB-INF do projeto o desenvolvedor pode criar um arquivo de constantes (constants.xml) que será carregado toda vez que uma página for solicitada e que pode definir variáveis com valor padrão a serem utilizadas pelo sistema e que podem ser configuradas sem necessitar o builder como segue exemplo abaixo.

```
<constants>
 <titulo_site>Projeto de Teste</titulo_site>
 <pvt>
   <nome>geraldo</nome>
 </pvt>
 <pvt_logsdir>/temp/mylogs</pvt_logsdir>
</constants>
```

1.2.1.5 Log de atualizações dos BDs

Log de atualizações dos banco de dados

Este é o local onde o desenvolvedor ativa o log automático do WI para atualizações feitas nos banco de dados.

O campo Banco de Dados serve para que o desenvolvedor indique qual será o banco de dados utilizado para armazenar o log.

O campo Tabela serve para que o desenvolvedor indique qual a tabela será utilizada para os logs.

A tabela deverá ter no mínimo 8 campos seguindo a ordem e nomeação abaixo:

- id = Id do registro (autoincrement)
- logdate = Data e hora
- databaseid = Banco de dados
- transactionid = Transacao
- sqlstatus = Status (T/F)
- sqlcommand = SQL
- sqlerror = Mensagem de Erro do BD (sqlerrormsg se for Caché)
- sqlparams = Parâmetros do SQL entre colchetes

Além das colunas obrigatórias o desenvolvedor pode definir outras colunas de texto e definir o valor a ser armazenado pela interface do builder. As colunas adicionais devem ser do tipo texto longo (Text ou CBLOB mas alguns BDs aceitam varchar) e não podem ser NOT NULL.

O campo Valor serve para que o desenvolvedor indique o valor a ser armazenado nas colunas específicas da aplicação.

Segue um exemplo do comando DDL para criar a tabela de logs no Mysql:

```
CREATE TABLE mylog (
   id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
   logdate DATETIME NOT NULL,
   databaseid VARCHAR(20) NOT NULL,
   transactionid VARCHAR(20) NOT NULL,
   sqlstatus CHAR(1) NOT NULL,
   sqlcommand TEXT NOT NULL,
   sqlerror TEXT NOT NULL,
   sqlparams TEXT NOT NULL,
   PRIMARY KEY (id)
)
```

Segue um exemplo do comando DDL para criar a tabela de logs no Postgres:

```
CREATE TABLE mylog (
   id serial NOT NULL,
   logdate timestamp NOT NULL,
   databaseid VARCHAR(20) NOT NULL,
   transactionid VARCHAR(20) NOT NULL,
   sqlstatus CHAR(1) NOT NULL,
   sqlcommand TEXT NOT NULL,
   sqlerror TEXT NOT NULL,
   sqlparams TEXT NOT NULL,
   PRIMARY KEY (id)
)
```

Segue um exemplo do comando DDL para criar a classe de logs no Caché:

```
Class User.MyLog Extends %Persistent [ ClassType = persistent, ProcedureBlock,
SqlRowIdName = id ]
{

Property logdate As %TimeStamp [ Required, SqlColumnNumber = 2 ];
```

```
Property databaseid As %String [ Required, SqlColumnNumber = 3 ];
Property transactionid As %String [ Required, SqlColumnNumber = 4 ];
Property sqlstatus As %String [ Required, SqlColumnNumber = 5 ];
Property sqlcommand [ Collection = characterstream, Required, SqlColumnNumber = 6 ];
Property sqlerrormsg [ Collection = characterstream, Required, SqlColumnNumber = 7 ];
Property sqlparams [ Collection = characterstream, Required, SqlColumnNumber = 8 ];

}
```

Segue um exemplo do comando DDL para criar a classe de logs no Oracle:

```
CREATE TABLE mylog (
   id INTEGER NOT NULL,
   logdate TIMESTAMP NOT NULL,
   databaseid VARCHAR2(20) NOT NULL,
   transactionid VARCHAR2(20) NOT NULL,
   sqlstatus CHAR(1) NOT NULL,
   sqlcommand CLOB NOT NULL,
   sqlerror CLOB NOT NULL,
   sqlparams CLOB NOT NULL,
   CONSTRAINT pk_mylog PRIMARY KEY (id)
);

CREATE SEQUENCE mylog_seq
   INCREMENT BY 1
   START WITH 1
   MAXVALUE 999999999
   MINVALUE 1
   nocycle
   nocache
   noorder;

CREATE TRIGGER mylog_insert BEFORE INSERT ON mylog
FOR EACH ROW BEGIN
   select mylog_seq.nextval into :new.id from dual;
END;
```

1.2.1.6 Backup

Backup

Este é o local onde o desenvolvedor executa o backup do projeto.

O campo Fazer Backup serve para que o desenvolvedor gere um arquivo zip que é o backup completo do projeto.

O campo Gerar War serve para que o desenvolvedor gere um arquivo war do projeto para ser utilizado em outro tomcat (produção). O diretório de definições (WEB-INF/definitions) não será levado, logo para o WI Builder o projeto não será visível. Para levar as definições altere o script ant build.xml do projeto.

O campo Gerar Ear serve para que o desenvolvedor gere um arquivo ear do projeto para ser utilizado em Jboss (produção). O diretório de definições (WEB-INF/definitions) não será levado pois o WI Builder não é suportado pelo Jboss.

O campo Novo Perfil permite que o desenvolvedor crie um novo perfil para as configurações de banco de dados, servidores e variáveis de inicialização.

O campo Perfil permite que o desenvolvedor selecione qual perfil deve ser usado na geração do War ou Ear.

O campo Fazer Deploy no Diretório permite que o arquivo War ou Ear gerado seja copiado para o diretório especificado ao invés de ser feito o download.

1.2.1.7 Como funciona a herança de projetos (Evite utilizar)

Como funciona a herança de projetos (Evite utilizar)

Suponhamos que temos 3 projetos chamados **A**,**B**,**C** e que no campo Projeto Pai de **A** está selecionado **B** e no campo Projeto Pai de **B** está selecionado **C**. Logo, **A** herda de **B** que por sua vez herda de **C**.

Quando uma página de **A** for solicitada, o WI irá verificar se existe realmente essa página no projeto **A**. Caso não exista, ele irá procurá-la em **B** e considerá-la como sendo também do projeto **A**. E, caso não exista, irá procurá-la no projeto **C**, e assim sucessivamente.

Se numa página do projeto **A** existir uma referência a um `Grid X`, o WI irá procurar esse grid no projeto **A**, não existindo procurará no projeto **B** e, se ainda não existir, procurará no projeto **C**. Mas ao encontrar, assumirá esse grid como sendo do projeto **A**.

Regra Geral: Quando o WI precisa de um elemento `X` (página, grid, etc) ele irá buscar no próprio projeto, mas caso não encontre irá buscar nos projetos pai até encontrar.

Os componentes do WI que são carregados por herança são as páginas, banco de dados, servidores, combos, grids, eventos, downloads, uploads e webservices.

A inicialização de variáveis também utiliza o mecanismo da herança.

O processamento de outros arquivos WSP utiliza a herança desde que no projeto pai este recurso esteja habilitado.

Para copiar um elemento do projeto pai pode-se utilizar no projeto filho o utilitário **Importar Elementos**.

O WI Builder só aceita a edição de elementos do próprio projeto, não permitindo a modificação de elementos herdados.

1.2.2 Banco de Dados

Banco de Dados

Este é o local onde o desenvolvedor define as configurações de um banco de dados que será usado durante o projeto tais como o tipo de conexão a ser usado, o limite máximo de conexões entre outras.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Identificador serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

O campo Alias define a string de conexão (JDBC ou nativa) de acordo com o driver de banco de dados que foi selecionado no campo Conexão. Para saber quais são as possíveis *strings* de conexão para os *drivers* de banco de dados já previamente cadastrados e homologados para funcionar com o WebIntegrator clique aqui.

O campo Conexão define qual o driver a ser utilizado para estabelecer conexão com um banco de dados. O WebIntegrator oferce suporte a conexões via drivers JDBC incluindo conexões ODBC, conexões com bases Mumps através da conexão MJava, conexão com o banco textual BR/Search e acesso a objetos do CachéFactory.

Além dos drivers de banco de dados que já se encontram previamente cadastrados e homologados, o desenvolvedor poderá cadastrar o driver JDBC de sua preferência para uso com o WebIntegrator configurando o arquivo <WI3_HOME>/builder/conf/drivers.def. Para saber maiores detalhes de como se configurar esse arquivo clique aqui.

O campo Usuário indica o nome do usuário que será usado para estabelecer a conexão com o servidor.

O campo Senha indica a senha correspondente ao nome do usuário para ser validado junto ao servidor.

O campo Time-Out da Conexão (pool) define o tempo máximo em minutos que uma conexão de banco de dados já estabelecida pelo WebIntegrator e que não esteja mais em uso por algum componente ficará ociosa até que seja encerrada. Configurando esse campo com um tempo

adequado às condições de execução de seu projeto fará com que o WebIntegrator aproveite ao máximo as conexões já abertas economizando a abertura de novas conexões com o banco de dados.

O campo Máximo de Conexões (pool) permite que o usuário limite o máximo de conexões que será permitida a abertura pelo WI.

O campo Time-Out de Login indica o tempo máximo em segundos que o WI esperará para que uma conexão com BD seja efetuada.

O campo Time-Out de Log de Consulta indica o tempo máximo em segundos que um consulta ao BD pode demora antes que um log seja gerado (default é 10 segundos).

O campo Time-Out de Consulta indica o tempo máximo em segundos que uma consulta ao BD pode demorar antes que ela seja abortada.

O campo Mensagem de Time-Out de Consulta indica a mensagem que o BD retorna quando o tempo de execução de uma consulta foi excedido. É usado pelo WI para validar se o erro foi o tempo excedido ou foi outro qualquer. Varia para cada BD de acordo com a versão.

Utilizando o algoritmo RoundRobin

É possível para o desenvolvedor configurar o Alias de modo que ele utilize o algoritmo RoundRobin para conectar alternadamente em vários banco de dados iguais, permitindo assim balanceamento de carga e o funcionamento sem paradas caso um banco venha a falhar.

Para utilizar o RoundRobin o desenvolvedor deve colocar as partes variáveis do alias entre colchetes e separadas por virgula, assim o WI irá utilizar a cada conexão um dos parametros e em caso de falha irá tentar os outros.

Exemplo de alias com Mysql:
//[server1,server2]:3366/dados
//[server1:3366,server1:3367]/dados

Exemplo de alias com Datasource:
java:/comp/env/jdbc/[default1,default2,default3]

1.2.2.1 Alias de Banco de Dados

Alias de Banco de Dados

O campo Alias deve ser preenchido com a string de conexão que corresponda ao tipo de conexão escolhido no campo Conexão. A seguir estão listados alguns dos tipos de conexões disponíveis, clique em uma delas para saber como deve ser configurado o campo Alias:

- ODBC - Genérico (não recomendado)
- Nativa - Datasource
- Nativa - CachéFactory
- Nativa - MJava
- Nativa - BRSearch
- JDBC - Cache
- JDBC - DB2
- JDBC - Informix
- JDBC - Interbase
- JDBC - MySQL
- JDBC - Oracle
- JDBC - PostgreSQL
- JDBC - Progress
- JDBC - SapDB
- JDBC - SQLServer
- JDBC - SQLServer 2000
- JDBC - Sybase
- JDBC - Sybase InetSoftware

1.2.2.1.1 Alias de Banco de Dados (Datasource)

Alias de Banco de Dados (Datasource)

Para utilizar a conexão Datasource o desenvolvedor precisa configurar no servidor de aplicações (Tomcat) as suas conexões com o BD e configurar o nome JNDI no alias do BD no WI.

1 - Configurando o Datasource para cada aplicação Web:

Ao utilizar esse tipo de configuração a aplicação wi3 (Builder) não terá acesso ao datasource então caso deseje utilizá-lo essas configurações deverão ser repetidas em seu context.xml e lembrar-se que quando o WI for atualizado o seu context.xml voltará ao padrão original.

- Criar dentro do projeto uma pasta META-INF com o arquivo context.xml conforme exemplo abaixo:

```
<Context>
   <Resource name="jdbc/exemploDB" auth="Container" type="javax.sql.DataSource"
             maxActive="100" maxIdle="10" maxWait="10000"
            username="root" password="root"
             driverClassName="com.mysql.jdbc.Driver"
             url="jdbc:mysql://localhost/teste"/>
  </Context>
```

- Criar uma conexão com o BD no WI e utilizar em conexão o valor Datasource e em alias o endereço JNDI.

Ex: java:/comp/env/jdbc/exemploDB

OBS: Lembrar que em certas configurações uma vez que o context.xml é criado o Tomcat o copia para a pasta "conf/Catalina/localhost/<app>.xml" e qualquer modificação feito nele não será refletida a menos que o arquivo <app>.xml seja removido.

2 - Configurando o Datasource compartilhado:

- Editar o arquivo server.xml na pasta conf do tomcat, localizar a tag <GlobalNamingResources> e colocar dentro dela o conteúdo abaixo:

```
<Resource name="jdbc/exemploDB" auth="Container" type="javax.sql.DataSource"
              maxActive="100" maxIdle="10" maxWait="10000"
             username="root" password="root"
              driverClassName="com.mysql.jdbc.Driver"
              url="jdbc:mysql://localhost/teste"/>
```

- Editar o arquivo context.xml na pasta conf do tomcat e adicionar dentro da tag <Context> o conteúdo abaixo:

```
<ResourceLink global="jdbc/exemploDB" name="jdbc/exemploDB"
type="javax.sql.DataSource"/>
```

- Criar uma conexão com o BD no WI e utilizar em conexão o valor Datasource e em alias o endereço JNDI.
Ex: java:/comp/env/jdbc/exemploDB

1.2.2.1.2 Alias de Banco de Dados (JDBC)

Alias de Banco de Dados (JDBC)

Nativa - CachéFactory
Nota:
Observe que o suporte ao CachéFactory apenas está disponível para as versões 4.0 ou superior do SGBD Caché.
Sintaxe:
`//<nome-da-máquina>:<porta>/<banco-de-dados>`
Exemplos:
`//localhost:1972/documento` `//192.0.0.20:1972/gestao`

JDBC - Caché
Nota:
Observe que o driver JDBC do Caché só está disponível para o SGBD Caché a partir da versão 3.2.
Sintaxe:
`//<nome-da-máquina>:<porta>/<banco-de-dados>`
Exemplos:
`//localhost:1972/documento` `//192.0.0.20:1972/gestao`

JDBC - DB2
Nota:
Para acesso a mainframe consulte a documentação da IBM.
Sintaxe:
`//<nome-da-máquina>:[<porta>]/<banco-de-dados>`
Exemplos:
`//localhost:3700/documentos` `//192.0.0.20/gestao`

JDBC - Informix
Nota:
Verifique o nome que foi dado ao serviço Informix durante a instalação.
Sintaxe:
`//<nome-da-máquina>:<porta>/<banco-de-dados>:informixserver=<nome-do-servico>`
Exemplos:
`//localhost:1533/documento:informixserver=myserver` `//192.0.0.20:1533/gestao:informixserver=gestao`

JDBC - Interbase
Nota:
Para utilizar o driver JDBC do Interbase será necessário instalar o InterServer que é um pacote fornecido pelo fabricante e que permite o acesso JDBC ao Interbase. Sem esse pacote instalado e o InterServer rodando, não será possível conectar-se via JDBC ao Interbase.
Sintaxe:
`//<nome-da-máquina>[:<porta>]/<banco-de-dados>`
Exemplos:
`//localhost/documento` `//192.0.0.20/gestao`

JDBC - MySQL
Nota:

Para permitir o acesso remoto ao MySQL será necessário executar esse dois comandos:
`GRANT ALL PRIVILEGES ON *.* TO user@localhost IDENTIFIED BY senha;`
`GRANT ALL PRIVILEGES ON *.* TO user@"%" IDENTIFIED BY senha;`
Onde `user` é o usuário e `senha` é a senha que serão preenchidos nos respectivos campos da definição de projeto no WI Builder.

Sintaxe:
`//<nome-da-máquina>[:<porta>]/<banco-de-dados>`
Exemplos:
`//localhost:3306/documento` `//192.0.0.20/gestao`

JDBC - Oracle
Nota:
O driver JDBC usado deverá ser da versão que corresponda à versão do SGBD Oracle, ou seja, se estiver sendo usado o Oracle 8i usar o driver JDBC para a versão 8i.
Sintaxe:
`//<nome-da-máquina>[:<porta>]/<banco-de-dados>`
Exemplos:

`//localhost:1521/documento`
`//192.0.0.20/gestao`
`//(description=(address=(host=myhost)(protocol=tcp)(port=1521))(connect_data=(service_name=orcl)(server=DEDICATED)))`

JDBC - PostgreSQL
Nota:
O serviço do SGBD PostgreSQL deve ser iniciado com o parâmetro "`postmaster -i`". Em alguns configurações é necessário configurar a variável **charSet** com o valor `LATIN1` para a correta visualização e inserção de caracteres especiais.
Sintaxe:
`//<nome-da-máquina>[:<porta>]/<banco-de-dados>[?charSet=LATIN1]`
Exemplos:
`//localhost/documento` `//192.0.0.20:2467/gestao`

JDBC - Progress
Nota:
Para solucionar qualquer dúvida vide documentação do fabricante.
Sintaxe:
`//<nome-da-máquina>[:<porta>]/<banco-de-dados>`
Exemplos:
`//localhost/documento` `//192.0.0.20/gestao`

JDBC - SapDB
Nota:
Executa comandos SQL nas estruturas.
Sintaxe:
`//<nome-da-máquina>[:<porta>]/<banco-de-dados>`
Exemplos:
`//localhost/documento` `//192.0.0.20/gestao`

JDBC - SQLServer
Nota:
Funciona com as versões 6.5, 7.0 e 2000 usando o driver Sprinta2000 (www.inetsoftware.de) com o máximo de 2 conexões simultâneas.
Sintaxe:
`//<nome-da-máquina>[:<porta>]?database=<banco-de-dados>[&sql7=true]`
Exemplos:
`//localhost?database=documento`

JDBC - SQLServer 2000
Nota:
Funciona apenas com o MS SQLServer 2000 usando o driver fornecido pela própria Miscrosoft (www.microsoft.com/sql/downloads/2000/jdbc.asp).
Sintaxe:
`//<nome-da-máquina>[:<porta>][;databasename=<banco-de-dados>][;selectmethod=cursor|direct]`
Exemplos:
`//localhost;databasename=documento`

JDBC - Sybase
Nota:
Funciona usando o JConnector 5.5.
Sintaxe:
`<nome-da-máquina>:<porta>/<banco-de-dados>`
Exemplos:
`localhost:1433/documentos`

JDBC - Sybase InetSoftware
Nota:
Funciona usando o driver SyTraks (www.inetsoftware.de) com o máximo de 2 conexões simultâneas.
Sintaxe:
`//<nome-da-máquina>[:<porta>]?database=<banco-de-dados>[&sql7=true]`
Exemplos:
`//localhost?database=documento`

1.2.2.1.3 Alias de Banco de Dados (MJava)

Alias de Banco de Dados (MJava)

Para utilizar a conexão nativa MJava será necessário instalar o módulo MJava em um SGBD compatível com a tecnologia M. Para isso basta instalar com o comando `%RIMF` o pacote `mjavaXXX.obj` que acompanha a distribuição do WebIntegrator e executar a rotina `^MJAVA`, para ativar o servidor MJava escolha a opção **1**.

O WebIntegrator consegue comunicar-se com vários sistemas gerenciadores de bancos de dados M através do servidor MJava. A porta padrão que o servidor MJava fica escutando é a 3500. Para definir o campo Alias de uma conexão MJava basta indicar o nome ou IP da máquina e opcionalmente o número da porta e o usuário, veja exemplos a seguir:

```
//servidor
//servidor/user
//192.0.0.30:3500
//192.0.0.30:3500/user
```

1.2.2.1.4 Alias de Banco de Dados (BR/Search)

Alias de Banco de Dados (BR/Search)

Para utilizar a conexão nativa BR/Search será necessário ter instalado a API Java para integração com tal banco de dados. Logo abaixo seguem alguns exemplos de configuração do campo Alias que podem ser usados para estabelecimento de conexão:

```
//<host>[:<porta>]/<paragrafo>
//localhost/SAMP
//127.0.0.1:6000/SAMP
```

1.2.3 Servidores

Servidores

Este é o local onde o desenvolvedor realiza as configurações dos servidores FTP, para realizar uploads e/ou downloads, dos servidores POP3, para recebimento de e-mails, e de socktes que serão utilizados no projeto.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Endereço indica qual o endereço (URL) do servidor. No caso de Web Services deve conter o caminho completo para o serviço inclusive com a descrição do protcolo (`http://...`).

O campo Identificador serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos grid. e combo. antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo Porta indica qual a porta que o servidor está escutando. Ao escolher os protocolos FTP, POP3 ou SMTP esse campo não poderá ser configurado pois o WebIntegrator assume as valores padrões para esses protocolos.

O campo Protocolo define qual o tipo de protocolo que o servidor trata.

O campo Senha indica a senha correspondente ao nome do usuário para ser validado junto ao servidor.

O campo Usuário indica o nome do usuário que será usado para estabelecer a conexão com o servidor.

1.3 Componentes de Projeto

Componentes de Projeto

Os componentes de projeto são definidos apenas uma vez e podem ser utilizados em vários pontos do projeto.

1.3.1 Página

Definição de Página

Nesta tela o desenvolvedor irá fazer as definições iniciais de uma nova página que será adicionada ao projeto. Essas definições iniciais incluem um título, um identificador único para essa página, o arquivo ou URL que representa essa página, entre outras coisas.

O campo Acessa sem Login permite seja feito o acesso à página sem validar se o cliente efetuou login no projeto. O WebIntegrator apenas fará essa validação automaticamente se o projeto estiver com a opção Ativar Login habilitada.

O campo Identificador serve para que o desenvolvedor entre com um identificador único para o componente de página.

A opção Permitir Cache do Browser indica se os cabeçalhos HTTP que instruem os browsers e servidores proxies a não fazerem cache de páginas serão enviados na resposta ao cliente.

O campo Página de Erro indica se a página é uma página de erro. Uma página de erro tem acesso ao objeto exception que pode ser utilizado para colher maiores detalhes sobre a exceção gerada. Normalmente as páginas de erro apenas deverão ser acessadas se ocorrer alguma exceção em outra página do projeto e essa página utilizar alguma página de erro para tratamento da exceção.

A opção Página de Sistema indicar se a página que está sendo definida apenas poderá ser acessada internamente pelo WI_Engine impedindo assim que os usuários do projeto possam acessá-la via URL. Recomenda-se habilitar esta opção quando a página que estiver sendo definida sirva como modelo de um componente grid.

A opção Segurança Ativa indica se o controle de segurança de acesso a páginas do projeto nativo do WebIntegrator estará ativado para essa página em particular. Com esse controle ativado uma página WSP do projeto apenas será acessada se a solicitação vier de uma outra página WSP. Quando se usam framesets poderá haver necessidade de desativar essa opção.

A opção Suportar Renderização Parcial ativa para a página a capacidade de acionar por javascript a função rerender conforme detalhes.

A opção Sincronizar Pré-Página ativa o sincronismo de execução para o pré da página selecionada. Isto é, existirá um tunel associado por um objeto que representa esse pré-página especifico e todas as requisições a esse pré passsaram por esse tunel, impedindo assim que (caso a lógica do desenvolvedor precise) duas requisições executem simultaneamente o mesmo trecho de código gerando inconsistências no BD.

A opção Sincronizar Pos-Página ativa o sincronismo de execução para o pos da página selecionada. Isto é, existirá um tunel associado por um objeto que representa esse pos-página especifico e todas as requisições a esse pos passsaram por esse tunel, impedindo assim que (caso a lógica do desenvolvedor precise) duas requisições executem simultaneamente o mesmo trecho de código gerando inconsistências no BD.

O campo Tipo define qual o tipo MIME que estará associado ao componente que estiver sendo definido. A especificação do tipo MIME é baseada numa extensão, para saber quais os tipos MIME suportados pelo WebIntegrator clique aqui.

O campo Título define qual será o título da página. O título da página estará acessível ao desenvolvedor através da variável wi.page.title.

O campo Usar Página de Erro deve conter o identificador de uma página indicando qual página de erro do projeto será usada caso ocorra algum tipo de erro ou exceção durante o processamento da página que está sendo definida. Na ocorrência de alguma exceção o processamento da página é interrompido e a página de erro correspondente é chamada, se nenhuma página de erro for utilizada e ocorrer algum tipo de erro durante o processamento de um componente da página o processamento da página como um todo não é interrompido.

1.3.1.1 Pré Página

Pré Página

Nesta tela o desenvolvedor poderá incluir ou remover elementos que serã processados antes que a página seja exibida. A ordem que os elementos aparecem na lista indica a ordem em que eles serão processados.

Os elementos download e desvio quando executados com sucesso interrompem a execução da lista, sendo assim, neste caso os elementos posteriores não serão processados.

1.3.1.2 Pré Página (referência)

Pré Página (referência)

Nesta tela o desenvolvedor poderá adicionar uma condição para a execução de um elemento de projeto (combo, grid, etc) que tenha sido incluído na página.

1.3.2 Combo

Combo

O componente Combo é um componente da camada de apresentação do projeto onde as opções de uma tag SELECT são montadas a partir do resultado de uma consulta feita a um banco de dados. Para acessar o conteúdo dessa variável basta que o desenvolvedor a referencie na página WSP que se queira exibir a combo-box usando a seguinte sintaxe:

```
|combo.<identificador-da-combo>|
```

Em casos onde o desenvolvedor tenha definido uma lista de seleção múltipla a informação sobre a quantidade de opções selecionadas será retornada pela propriedade `size` da variável que representa o elemento `SELECT` no formulário de uma página. Referenciando o nome da variável será retornado a lista dos valores selecionados separados por vírgula, para acessar cada um dos respectivos valores basta acrescentar `.<índice>` ao final da variável. Para exemplo, suponha que o seguinte formulário encontra-se definido numa página WSP:

```
...
<form action="/|wi.proj.id|/|wi.page.id|.wsp" method="post">
<select multiple name="tmp.select">|combo.minhaCombo|</select>
...
```

referenciando `|tmp.select|` serão retornados os valores selecionados separados por vírgula, `|tmp.select.size|` retornará a quantidade de elementos selecionados e para referenciar os valores selecionados individualmente usa-se `|tmp.select.<índice>|`, onde `<índice>` pode variar de `1` até `|tmp.select.size|`.

Uma outra característica suportada pelas combos é a passagem de parâmetros para a Combo a ser processada. Como exemplo, suponha que se queira passar para uma Combo se os elementos deverão vir ordenados em ordem crescente ou decrescente através de uma variável `tmp.orderBy` a sintaxe ficaria a seguinte:

```
|combo.<identificador-da-combo>?tmp.orderBy=desc|
```

Esse tipo de construção também aceita a passagem de múltiplos parâmetros, neste caso as atribuições das variáveis deverão estar separados por **&**. A sintaxe geral para um elemento Combo é a seguinte:

```
|combo.<identificador-da-
combo>[?<variável1>=<valor1>[&<variável2>=<valor2>...]]|
```

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo `Chave` indica o nome ou índice de uma das colunas retornada pelo resultado da pesquisa de onde serão pegos os valores para serem populados como valores-chaves das opções da combo.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executado pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo, suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e, o usuário da aplicação, ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome =` . Se o campo `Filtrar` não estiver definido com os caracteres `%`, então o WebIntegrator irá processar a seguinte instrução: `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND`

`publico = 1` retornando como resultado todos as linhas da tabela.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo `SQL` contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a `stored procedures`, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

O campo `Texto` indica o nome ou índice da coluna retornada pelo resultado donde serão pegos os dados para serem populados como os textos (valores) da combo.

Para gerar uma combo com as pastas IMAP de um servidor pode ser utilizada a opção Java de Banco de Dados e no comando sql coloca-se `br.com.itx.database.impl.ResultSetImapFolders:[id_servidor]` onde o id do servidor tanto pode estar fixo como ser uma variável entre pipes.

1.3.3 Grid

Grid

O componente Grid, oferece a possibilidade de processar as informações vindas de várias fontes de dados, podendo ser uma consulta a um banco de dados ou uma listagem de diretório.

1.3.3.1 SQL

Grid SQL

O componente Grid oferece ao desenvolvedor a possibilidade de se processar as informações vinda de uma fonte de dados podendo essa fonte de dados ser uma consulta a um banco de dados ou uma listagem de diretório e podendo exibí-las em uma forma tabular dispondo-as em linhas e colunas seguindo um modelo simples usando como base a estrutura de uma tabela HTML ou através de modelos múltiplos onde cada parte do grid (cabeçalho, rodapé, células vazias, células válidas, etc) está representada em modelos distintos.

Uma outra possibilidade que os grids possuem é a de que uma determinada página do projeto seja executada para cada linha válida retornada pela consulta à fonte de dados (banco de dados, servidor FTP, etc) oferecendo assim uma maneira de iteragir pelo resultado da consulta e para cada passo dessa iteração será a página correspondente será executada.

Ao utilizar o modelo simples a estrutura padrão a ser gerada será a seguinte tomando como exemplo um grid cuja consulta a um banco de dados retorne 4 colunas com os nomes `coluna1`, `coluna2`, `coluna3` e `coluna4` e o campo `Início na Linha` esteja definido com o valor 2:

```
<TABLE>
<!-- Cabeçalho (opcional) -->
<TR>
<TH>Coluna #1</TH>
<TH>Coluna #2</TH>
<TH>Coluna #3</TH>
<TH>Coluna #4</TH>
</TR>
<!-- Linha de detalhe -->
<TR>
<TD>|coluna1| </TD>
<TD>|coluna2| </TD>
<TD>|coluna3| </TD>
<TD>|coluna4| </TD>
</TR>
<!-- Mensagem a ser exibida quando não há dados a serem exibidos
(opcional) -->
<TR>
<TD ALIGN="center" COLSPAN="4">Nenhum registro selecionado</TD>
</TR>
<!-- Rodapé (opcional) -->
<TR>
<TD ALIGN="center" COLSPAN="4">Mensagem do rodapé</TD>
</TR>
</TABLE>
```

A principal funcionalidade do grid SQL é exibir informações vinda de uma consulta a um banco de dados aplicando sobre estas uma definição de um modelo simples ou múltiplo. Quando se usa

o modelo simples uma das definições a ser feita é qual linha da tabela HTML desse modelo irá ser populada com os dados do resultado da consulta ao banco de dados, essa definição é feita através do campo Início na Linha. Baseado na definição desse campo as linhas da tabela anteriores a essa, caso existam, formam o cabeçalho do grid; a linha posterior à linha de detalhe só irá aparecer caso o resultado da consulta não retorne nada e as linhas logo abaixo dessa formam o rodapé.

A linha de detalhe definida pelo campo Início na Linha além de referenciar os nomes das colunas também poderá referenciar outras variáveis que estejam na sessão inclusive referenciar outro grid para criar a visão de mestre-detalhe. Qualquer outra célula do grid também poderá fazer referência a variáveis de sessão do WebIntegrator.

O modelo múltiplo ao invés de trabalhar com um único modelo trabalha com a definição das várias partes que compõem o modelo do grid. Essas partes são o cabeçalho, início do registro, registro válido, registro vazio, fim do registro, sem registros e rodapé. Quando se usa o modelo múltiplo a sequência de evento de processamento das partes é a seguinte: primeiro o **WI Engine** processa o modelo relativo ao cabeçalho; em seguida se o resultado da consulta feita ao banco de dados retornou algum registro então os modelos relativos ao início do registro, registro válido e fim do registro são processados respectivamente e isso se repete até que todos os registros tenham sido processados ou que a contagem da iteração tenha alcançado o valor definido no campo Quantidade, senão se o resultado da consulta não retornar nenhum registro então o modelo "sem registros" é processado; finalmente o modelo relativo ao rodapé é processado.

O modelo relativo ao registro vazio apenas é processado quando o campo Horizontal estiver definido e indica como serão preenchidas as células restantes da tabela.

O campo Banco de Dados indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo Compartilhado indica qual o grid que terá seu modelo compartilhado, as alterações feitas no modelo do grid-pai irão se refletir nos grids-filhos. Nessa combo apenas serão exibidos os grids que possuem modelo simples ou múltiplo não sendo exibido os grids que possuem modelo compartilhado.

O campo Cor define o código hexadecimal de uma cor que será aplicada nas linhas pares das linhas de detalhe do grid. A cor padrão é aquela definida no arquivo que contém o esquema visual do grid e a cor das linhas pares é a definida neste campo. Lembrar que esse parâmetro altera a cor da linha, logo, se o desenvolvedor definir uma cor específica para uma célula, ela irá sobrepor esta definição.

Clicando em Cor aparecerá uma janela com algumas gamas de cores que ao se passar o mouse sobre elas, exibe-se o código hexadecimal correspodente à cor.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Executar a Cada Registro indica qual a página do projeto que será executada a cada iteração nos registros retornados como resultado da consulta feita ao banco de dados.

O campo Filtrar define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo SQL antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executada pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome = .` Se o campo `Filtrar` não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

A opção `Gerar` indica se o WebIntegrator irá gerar o modelo simples do grid baseado nas configurações atuais. O modelo só será criado após a gravação da definição. Esse campo apenas se refere aos grids que utilizem o modelo simples.

O campo `Colunado` define em quantas colunas serão exibidas as linhas de detalhe definidas no modelo do grid. Quando um grid é colunado as linhas que formam o cabeçalho e o rodapé não podem conter mais de uma célula.

O campo `Disposição` define a forma que os valores serão dispostos quando o grid é colunado.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo `Início na Linha` indica qual linha da tabela HTML que está definida como modelo do grid que será a linha de detalhe, ou seja, a linha que terá seu conteúdo processado e populado de acordo com o resultado da consulta. Esse campo apenas se refere aos grids que utilizem o modelo simples.

A opção `Múltiplo` marcada indica que o grid usará o modelo múltiplo para geração do conteúdo, para editar as várias partes do modelo múltiplo clique no ícone  que aparece na sessão **Modelo**. O modelo múltiplo trabalha com as várias partes que em conjunto irão criar o modelo do grid, as partes consistem na definição do conteúdo do cabeçalho, o conteúdo do início do registro, o conteúdo a ser iteragido quando forem encontrados registros válidos, o conteúdo a ser exibido em registros vazios (apenas utilizado quando o deslocamento horizontal está configurado), o conteúdo que indica o fim de um registro, o conteúdo a ser exibido quando nenhum registro for retornado pela consulta e o rodapé do grid.

O campo `Quantidade` define quantas linhas serão exibidas por página. Caso o número de linhas retornadas seja maior que a quantidade definida o usuário poderá fazer uso da propriedade `link` (`|grid.<identificador-do-grid>.link|`) que gerará uma barra de navegação com links para acessar as próximas linhas ou retornar às linhas anteriores.

O campo `SQL` contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a stored procedures, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas

chamadas clique aqui.

A opção Simples marcada indica que o grid usará o modelo simples para geração do conteúdo, para editar o modelo do grid clique no ícone ... que aparece na sessão **Modelo**. O modelo simples consiste na definição de uma tabela HTML onde cada linha tem um função dependendo do valor definido no campo Início na Linha. A linha indicada por esse campo será iteragida e populada com os valores vindos da consulta à fonte de dados, as linhas acima desta são consideradas como o cabeçalho do grid, as linhas que se encontrarem abaixo dela serão consideradas como as linhas de rodapé.

1.3.3.2 HTML

Grid HTML

O grid HTML assemelha-se em funcionalidade ao grid SQL, porém enquanto neste último a fonte dos dados é sempre o resultado de uma consulta feita a um banco de dados, no grid HTML a fonte dos dados é variável a depender do tipo escolhido, podendo ser uma listagem de um diretório de um servidor FTP ou de um diretório da máquina onde está instalado o WebIntegrator, ou a lista de mensagens disponíveis em um servidor POP3, etc. Para saber mais sobre os tipos de grid HTML clique aqui.

O funcionamento e a definição de modelos do grid HTML seguem o mesmo padrão de regras do grid SQL.

O campo Compartilhado indica qual o grid que terá seu modelo compartilhado, as alterações feitas no modelo do grid-pai irão se refletir nos grids-filhos. Nessa combo apenas serão exibidos os grids que possuem modelo simples ou múltiplo não sendo exibido os grids que possuem modelo compartilhado.

O campo Cor define o código hexadecimal de uma cor que será aplicada nas linhas pares das linhas de detalhe do grid. A cor padrão é aquela definida no arquivo que contém o esquema visual do grid e a cor das linhas pares é a definida neste campo. Lembrar que esse parâmetro altera a cor da linha, logo, se o desenvolvedor definir uma cor específica para uma célula, ela irá sobrepor esta definição.

Clicando em Cor aparecerá uma janela com algumas gamas de cores que ao se passar o mouse sobre elas, exibe-se o código hexadecimal correspodente à cor.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

A opção Gerar indica se o WebIntegrator irá gerar o modelo simples do grid baseado nas configurações atuais. O modelo só será criado após a gravação da definição. Esse campo apenas se refere aos grids que utilizem o modelo simples.

O campo `Colunado` define em quantas colunas serão exibidas as linhas de detalhe definidas no modelo do grid. Quando um grid é colunado as linhas que formam o cabeçalho e o rodapé não podem conter mais de uma célula.

O campo `Disposição` define a forma que os valores serão dispostos quando o grid é colunado.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo `Início na Linha` indica qual linha da tabela HTML que está definida como modelo do grid que será a linha de detalhe, ou seja, a linha que terá seu conteúdo processado e populado de acordo com o resultado da consulta. Esse campo apenas se refere aos grids que utilizem o modelo simples.

A opção Múltiplo marcada indica que o grid usará o modelo múltiplo para geração do conteúdo, para editar as várias partes do modelo múltiplo clique no ícone  que aparece na sessão **Modelo**. O modelo múltiplo trabalha com as várias partes que em conjunto irão criar o modelo do grid, as partes consistem na definição do conteúdo do cabeçalho, o conteúdo do início do registro, o conteúdo a ser iteragido quando forem encontrados registros válidos, o conteúdo a ser exibido em registros vazios (apenas utilizado quando o deslocamento horizontal está configurado), o conteúdo que indica o fim de um registro, o conteúdo a ser exibido quando nenhum registro for retornado pela consulta e o rodapé do grid.

O campo `Quantidade` define quantas linhas serão exibidas por página. Caso o número de linhas retornadas seja maior que a quantidade definida o usuário poderá fazer uso da propriedade `link` (`|grid.<identificador-do-grid>.link|`) que gerará uma barra de navegação com links para acessar as próximas linhas ou retornar às linhas anteriores.

O campo `Executar a Cada Registro` indica qual a página do projeto que será executada a cada iteração nos registros retornados como resultado da consulta feita ao banco de dados.

A opção `Simples` marcada indica que o grid usará o modelo simples para geração do conteúdo, para editar o modelo do grid clique no ícone que aparece na sessão **Modelo**. O modelo simples consiste na definição de uma tabela HTML onde cada linha tem um função dependendo do valor definido no campo `Início na Linha`. A linha indicada por esse campo será iteragida e populada com os valores vindos da consulta à fonte de dados, as linhas acima desta são consideradas como o cabeçalho do grid, as linhas que se encontrarem abaixo dela serão consideradas como as linhas de rodapé.

O campo `Tipo` indica o tipo de grid que será gerado, a depender do tipo escolhido determinadas variáveis de contexto estarão disponíveis para serem referenciadas no modelo. Para saber mais sobre os tipos clique aqui.

1.3.3.3 XMLOut

Grid XMLOut

O grid XMLOut é usado para a geração de conteúdo XML tendo como fonte de dados uma consulta feita a um banco de dados. A estrutura XML a ser gerada segue o seguinte esquema baseado nas definições dos campos:

```
[<valor-do-campo-HEADER>]
[<valor-do-campo-ROOT>]
  [<valor-do-campo-CHILD>
  |conteúdo-do-arquivo-do-campo-MODELO|
  </valor-do-campo-CHILD>]
</valor-do-campo-ROOT>
```

Nesta estrutura os elementos HEADER e ROOT apenas são processados uma vez enquanto que o elemento CHILD é processado tantas vezes quanto haja dados ainda a serem recuperados do banco de dados.

Para exemplificar caso quiséssemos gerar um conteúdo XML como o que segue logo abaixo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE NoRaiz [
  <!ELEMENT NoRaiz (#PCDATA)>
  <!ENTITY nbsp  >
  <!ENTITY cr &#13;>
  <!ENTITY lf &#10;>
]>

<NoRaiz>
  <NoFilho Id="0">
    <Elemento1>valor0.1</Elemento1>
    <Elemento2>valor0.2</Elemento2>
    <Elemento3>valor0.3</Elemento3>
  </NoFilho>
  <NoFilho Id="1">
    <Elemento1>valor1.1</Elemento1>
    <Elemento2>valor1.2</Elemento2>
    <Elemento3>valor1.3</Elemento3>
  </NoFilho>
  <NoFilho Id="2">
    <Elemento1>valor2.1</Elemento1>
    <Elemento2>valor2.2</Elemento2>
    <Elemento3>valor2.3</Elemento3>
  </NoFilho>
</NoRaiz>
```

nós usaríamos as seguintes definições:

Nome do Campo	Valor
Cabeçalho	<?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE |wi.xml.root| [<!ELEMENT NoRaiz (#PCDATA)> <!ENTITY nbsp > <!ENTITY cr
> <!ENTITY lf
>]>
Root	NoRaiz
Child	NoFilho Id="|rowid|"
Conteúdo	<Elemento1>|valor1|</Elemento1> <Elemento2>|valor2|</Elemento2> <Elemento3>|valor3|</Elemento3>

Logo abaixo segue a relação dos campos que são utilizados para se definir a estrutura do conteúdo que será montada pelo grid XMLOut.

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O rótulo `Cabeçalho` é um link para a página de edição do cabeçalho do conteúdo XML a ser gerado. O cabeçalho de um arquivo XML deverá possuir no mínimo uma entrada semelhante à:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Ainda no cabeçalho, opcionalmente, poderá ser definido qual o prólogo desse conteúdo XML.

O campo `Child` define como será a estrutura dos elementos filhos.

O rótulo `Conteúdo` é um link para a página de edição do conteúdo XML que será populado e gerado de acordo com o resultado vindo da consulta à fonte de dados.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executada pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável tmp.nome um valor como `% OR nome =` . Se o campo `Filtrar` não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

A opção `Gerar` indica se o WebIntegrator irá gerar o modelo simples do grid baseado nas configurações atuais. O modelo só será criado após a gravação da definição. Esse campo

apenas se refere aos grids que utilizem o modelo simples.

O campo `Identação` indica a quantidade de espaços em branco que será utilizada para realizar a identação do conteúdo XML gerado.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo `Root` define como será a estrutura do elemento raiz do conteúdo XML a ser gerado.

O campo `SQL` contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a stored procedures, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

1.3.3.4 Java

Grid Java

O grid Java é destinado àquele desenvolvedor que queira personalizar a construção do seu grid e que possua um conhecimento da linguagem de programação Java.

Para se construir um grid Java, primeiramente, o desenvolvedor deverá criar uma classe Java que implemente a interface `webintegrator.integration.InterfaceGrid`. Após compilada, colocar o arquivo .class juntamente com as classes do WI, atentando para o fato de que se foi criada dentro de um pacote (*package*) criar a estrutura de diretório correspondente.

O código-fonte da interface a ser implementada segue abaixo:

```
package webintegrator.integration;

import java.util.Hashtable;
```

```
import webintegrator.util.Hash;
  import webintegrator.integration.DatabaseAliases;

  public interface InterfaceGrid {
     public Hashtable[] execute(Hash context, DatabaseAliases
dbAliases);
  }
```

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Grid`, `Combo`, `Download` ou `Upload` lista quais componentes de projetos estão disponíveis para esse componente de acordo com o seu tipo, ou seja, na definição de um componente de listagem de e-mails serão listados os grids HTML do tipo **POP3**, num componente Grid SQL seriam listados os grids **SQL**, num componente Upload seriam exibidos os **uploads** definidos para o projeto, etc.

Se o campo possuir o link `Editar` o desenvolvedor poderá usá-lo para ser redirecionado para a tela de definição específica do componente que se encontra selecionado.

O campo `Nome da classe` deve conter o nome completamente qualificado da classe que representa um conector Java.

1.3.3.5 WI Object

Grid WI Object

O objetivo de um grid Objeto WI é recuperar os dados que estão armazendaos em um objeto de sessão que siga a padrão do WebIntegrator e exibí-los na forma de um grid usando um modelo definido pelo usuário, ou seja, quando o desenvolvedor quiser mostrar sob a forma de grid um objeto do ambiente WebIntegrator que contenha um array de registros.

Para um objeto de sessão estar no padrão do WebIntegrator os seguintes pontos devem ser atendidos:

1. o objeto deve possuir a propriedade **`size()`** que informará quantas instâncias desse objeto encontra-se na sessão.

2. o objeto deve oferecer suporte à sintaxe de referenciação de suas propriedades semelhante à notação de *`arrays`*. A sintaxe a ser suportada é:

```
<nome-do-objeto>[<indice>].<nome-da-propriedade>
```

3. a numeração do índice para referenciar uma determinada instância de um objeto deverá começar de `1` e ir até o valor da propriedade **`size()`** do objeto.

Os elementos Objeto e WebService são exemplos que seguem esse padrão de objetos do WebIntegrator. O desenvolvedor também poderá desenvolver seus próprios objetos que sigam esse padrão através de conectores Java.

Há também a possibilidade do desenvolvedor referenciar um grid Objeto WI diretamente na página WSP sem a necessidade de definí-lo no pré-página, para isso basta colocar na página WSP a referência ao grid e passar o parâmetro **id** com o nome do objeto que irá popular esse grid. A sintaxe é a seguinte:

```
|grid.<identificador-do-grid>?id=<identificador-do-objeto>|
```

O campo `Complemento do Identificador` indica qual o complemento a ser acrescentado ao identificador de um componente grid ou combo. A utilidade maior desse campo é percebida quando o desenvolvedor quer que o mesmo componente (combo ou grid) seja executado e exibido mais de uma vez na mesma página.

Como exemplo, suponha que numa página se queira exibir três vezes o mesmo compoenete combo, mas cada um deles vindo com valores selecionados diferentes. Para isso, basta que o desenvolvedor defina três combos e cada uma delas com um complemento do identificador e o valor a vir selecionado diferentes, e na página referenciar `|combo.<identificador-da-combo>.<complemento-do-identificador>|`.

Ao se utilizar um complemento de identificador, a atribuição de variáveis via `|?` não poderá ser feita através da própria página no momento em que o grid ou a combo estiver sendo referenciada. Ao invés disso, o desenvolvedor poderá usar a seguinte sintaxe no campo `Condição`:

```
<condicao> || |?<variavel>=<valor>|
```

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Grid`, `Combo`, `Download` ou `Upload` lista quais componentes de projetos estão disponíveis para esse componente de acordo com o seu tipo, ou seja, na definição de um componente de listagem de e-mails serão listados os grids HTML do tipo **POP3**, num componente Grid SQL seriam listados os grids **SQL**, num componente Upload seriam exibidos os **uploads** definidos para o projeto, etc.

Se o campo possuir o link `Editar` o desenvolvedor poderá usá-lo para ser redirecionado para a tela de definição específica do componente que se encontra selecionado.

O campo `Objeto WI` define o nome de um objeto que siga o padrão esperado pelo **WI Engine**.

1.3.4 Download

Download

O componente Download oferece ao desenvolvedor a possibilidade da realização de download do conteúdo de um arquivo localizado num diretório da máquina local (Download Local), em um servidor de FTP (Download FTP) ou em um banco de dados (Download de Dados). Veremos detalhes de cada um deles a seguir.

1.3.4.1 Local

Download (Local)

O componente Download Local oferece ao desenvolvedor a possibilidade da realização do download do conteúdo de um arquivo localizado em um diretório da mesma máquina onde esteja instalado o WebIntegrator. Caso o conteúdo do arquivo seja textual e nele sejam feitas referências a variáveis de sessão do WebIntegrator há a possibilidade de processá-las.

Algo a ser levado em consideração em relação aos compoenentes de downloads é que sempre que um download for executado com sucesso os restantes dos eventos que se encontram registrados num pré-página não serão processados.

O campo `Arquivo` indica o nome do arquivo que será enviado ao usuário.

O campo `Arquivo Substituto` indica o nome do arquivo que será enviado ao cliente caso o arquivo especificado no campo `Arquivo` não seja encontrado.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Diretório(||¹)` indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único

para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

A opção `Processa Variáveis` indica se será realizado o processamento de variáveis que possam existir no conteúdo do arquivo. Esta opção apenas será válida para arquivos com conteúdo textual, ou seja, que tipo MIME seja `text/*`.

O campo `Tipo` define qual o tipo MIME que estará associado ao componente que estiver sendo definido. A especificação do tipo MIME é baseada numa extensão, para saber quais os tipos MIME suportados pelo WebIntegrator clique aqui. O valor "attachment" força a janela de salvamento no browser.

1.3.4.2 FTP

Download (FTP)

O componente Download FTP oferece ao desenvolvedor a possibilidade da realização do download do conteúdo de um arquivo armazenado em um servidor FTP. Caso o conteúdo do arquivo seja textual e nele sejam feitas referências a variáveis de sessão do WebIntegrator há a possibilidade de processá-las.

Algo a ser levado em consideração em relação aos compoenentes de downloads é que sempre que um download for executado com sucesso os restantes dos eventos que se encontram registrados num pré-página não serão processados.

O campo `Arquivo` indica o nome do arquivo que será enviado ao usuário.

O campo `Arquivo Substituto` indica o nome do arquivo que será enviado ao cliente caso o arquivo especificado no campo `Arquivo` não seja encontrado.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Diretório(||¹)` indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse

identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

A opção `Processa Variáveis` indica se será realizado o processamento de variáveis que possam existir no conteúdo do arquivo. Esta opção apenas será válida para arquivos com conteúdo textual, ou seja, que tipo MIME seja `text/*`.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

O campo `Tipo` define qual o tipo MIME que estará associado ao componente que estiver sendo definido. A especificação do tipo MIME é baseada numa extensão, para saber quais os tipos MIME suportados pelo WebIntegrator clique aqui. O valor "attachment" força a janela de salvamento no browser.

1.3.4.3 Banco de Dados

Download (Banco de Dados)

O componente Download Banco de Dados oferece ao desenvolvedor a possibilidade da realização do download do conteúdo de um arquivo armazenado em banco de dados. Caso o conteúdo do arquivo seja textual e nele sejam feitas referências a variáveis de sessão do WebIntegrator há a possibilidade de processá-las.

Algo a ser levado em consideração em relação aos compoenentes de downloads é que sempre que um download for executado com sucesso os restantes dos eventos que se encontram registrados num pré-página não serão processados.

O campo `Arquivo` indica o nome do arquivo que será enviado ao usuário.

O campo `Arquivo Substituto` indica o nome do arquivo que será enviado ao cliente caso o arquivo especificado no campo `Arquivo` não seja encontrado.

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve

comentário sobre a ação a ser executada pelo componente em questão.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executada pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome =` . Se o campo `Filtrar` não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciados nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo `Nome ao Salvar` indica o nome que será enviado ao cliente relativo ao conteúdo que está sendo feito o download.

A opção `Processa Variáveis` indica se será realizado o processamento de variáveis que possam existir no conteúdo do arquivo. Esta opção apenas será válida para arquivos com conteúdo textual, ou seja, que tipo MIME seja `text/*`.

O campo `SQL` contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a `stored procedures`, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

O campo `Tipo` define qual o tipo MIME que estará associado ao componente que estiver sendo definido. A especificação do tipo MIME é baseada numa extensão, para saber quais os tipos MIME suportados pelo WebIntegrator clique aqui. O valor "attachment" força a janela de salvamento no browser.

1.3.5 Upload

Upload

O componente Upload oferece ao desenvolvedor a possibilidade da realização de upload do conteúdo de um arquivo localizado num diretório da máquina local (Upload Local), em um servidor de FTP (Upload FTP) ou em um banco de dados (Upload de Dados). Veremos detalhes de cada um deles a seguir.

1.3.5.1 Local

Upload (Local)

Este é o local onde o desenvolvedor define um componente que irá executar o upload de um arquivo escolhido pelo usuário em um campo de formulário HTML do tipo FILE para um diretório localizado na mesma máquina onde está instalado o WebIntegrator. O formulário HTML que estiver fazendo o upload deve ter como valor do atributo ENCTYPE o valor `multipart/form-data`.

O campo `Arquivo` indica o nome do arquivo que será dado ao conteúdo do upload.

A opção `Cria Diretório` indica se o diretório especificado na definição do componente deverá ser criado caso ele não exista.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Diretório`(| |[1]) indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo `Nome do Campo do Formulário` indica qual o nome do campo do tipo `FILE` do formulário HTML contém o nome do arquivo que estará sendo processado no upload. Isso permite que sejam executados vários uploads com um único formulário desde que se coloque um evento do tipo upload para cada campo do `FORM`.

1.3.5.2 FTP

Upload (FTP)

Este é o local onde o desenvolvedor define sob qual condição será realizado o upload de um arquivo escolhido pelo usuário em um campo de formulário HTML do tipo `FILE` em um diretório localizado em um servidor FTP. No formulário HTML que estiver fazendo o upload do arquivo deve constar o atributo `ENCTYPE` com o valor `multipart/form-data`.

O campo `Arquivo` indica o nome do arquivo que será dado ao conteúdo do upload.

A opção `Cria Diretório` indica se o diretório especificado na definição do componente deverá ser criado caso ele não exista.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Diretório`(| | [1]) indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo `Nome do Campo do Formulário` indica qual o nome do campo do tipo `FILE` do formulário HTML contém o nome do arquivo que estará sendo processado no upload. Isso permite que sejam executados vários uploads com um único formulário desde que se coloque um evento do tipo upload para cada campo do `FORM`.

O campo Servidor indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o endpoint do web service a ser invocado.

1.3.5.3 Banco de Dados

Upload (Banco de Dados)

Este é o local onde o desenvolvedor define sob qual condição será realizado o upload de um arquivo escolhido pelo usuário em um campo de formulário HTML do tipo FILE em um servidor de banco de dadps. No formulário HTML que estiver fazendo o upload do arquivo deve constar o atributo ENCTYPE com o valor multipart/form-data.

O campo Banco de Dados indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Filtrar define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo SQL antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executada pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1 e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável tmp.nome um valor como % OR nome = . Se o campo Filtrar não estiver definido com os caracteres % e o WebIntegrator irá processar a seguinte instrução SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1 retornando como resultado todos as linhas da tabela.

O campo Identificador serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos grid. e combo. antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo Nome do Campo do Formulário indica qual o nome do campo do tipo FILE do

formulário HTML contém o nome do arquivo que estará sendo processado no upload. Isso permite que sejam executados vários uploads com um único formulário desde que se coloque um evento do tipo upload para cada campo do FORM.

O campo SQL contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a stored procedures, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

1.3.6 WIEvent

WIEvent

O componente WIEvent oferece a funcionalidade às aplicações web desenvolvidas pelo WebIntegrator a troca (envio ou recebimento) de dados entre o cliente (browser) e o servidor sem a necessidade da submissão completa da página permitindo que o desenvolvedor, através da página, solicite ao servidor onde esteja hospedado uma aplicação WI a execução de comandos ao banco de dados e que as resultados dessas solicitações sejam retornados à página para que possam ser manipulados. Após os dados terem sido retornados ao cliente (browser) o WIEvent encerra a comunicação com o servidor e a partir daí começa a trabalhar com os dados localmente.

A manipulação dos dados no lado do cliente (browser) é feito através de chamadas a métodos JavaScript da biblioteca do WIEvent que se encontra no arquivo `wievent.js` que se encontra no diretório `js` dentro do próprio projeto. Esse arquivo será criado automaticamente no projeto quando for definido o primeiro componente WIEvent para esse mesmo projeto.

Para inserir a chamada à biblioteca do WIEvent numa página de um projeto usando o WIzard de páginas basta selecionar a opção <HEAD> e no campo `Link JS` adicionar o valor `/|wi.proj.id|/js/wievent.js`, o campo `Link JS` suporta que a definição de vários arquivos JS desde que cada chamada aos arquivos JS estejam separadas por vírgula.

Para inserir a chamada à biblioteca do WIEngine diretamente no código-fonte de uma página do projeto insira o trecho de código HTML que segue abaixo dentro da tag HEAD do documento HTML que representa a página.

```
<SCRIPT TYPE="text/javascript"
SRC="/|wi.proj.id|/js/wievent.js"></SCRIPT>
```

A configuração de um componente WIEvent é feito através do WIBuilder podendo ser do tipo SELECT ou UPDATE sendo que o primeiro apenas envia ao banco de dados comandos de pesquisa enquanto o último apenas envia comandos de atualização. Uma página solicita a execução de um componente WIEvent através do seu identificador sendo que para componentes WIEvent do tipo SELECT deve-se utilizar o método `selectdb()` enquanto para chamar componentes WIEvent do tipo UPDATE deve-se utilizar o método `updatedb()`.

Logo abaixo seguem listados os métodos disponíveis para interagir com componentes WIEvent, para saber como utilizar os métodos do WIEvent acesse a seção **Como...** da ajuda do WebIntegrator.

Método	Descrição
`WIEvent(timeout, prjId)`	Instancia um objeto WIEvent no lado cliente. O parâmetro `timeout` indica o tempo de espera em segundos para a resposta do WIEvent, o valor default é 30 segundos; o parâmetro `prjId` indica o nome do projeto onde está o compoente WIEvent a ser executado, o valor default é o nome do projeto chamado pela URL.
`execute(metodo)`	Solicita que o servidor execute o componente WIEvent correspondente e após o retorno da resposta do servidor executa a função JavaScript passada como parâmetro. Essa função JavaScript sempre receberá como primeiro parâmetro um objeto WIEvent que deverá ser usado para a chamada dos métodos necessários.
`preread(nome)`	Solicita a leitura da variável passada pelo parâmetro `nome`, a variável deve fazer parte da sessão do WebIntegrator. O conteúdo poderá ser lido posteriormente pelo método `readobj()` após a chamada do método `execute()`.
`readobj(nome)`	Retorna o conteúdo da variável passada pelo parâmetro `nome` que deve ter siso previamente solicitado através do método `preread()`.
`writeobj(nome, valor)`	Grava uma variável na sessão do WebIntegrator. A gravação somente irá se efetivar após a chamada ao método `execute()` ou `selectdb()`.
`selectdb(eventId)`	Solicita a execução de um componente WIEvent do tipo `SELECT` identificado pelo parâmetro `eventId`. O componente só será efetivamente executado ao se chamar o método `execute()`.
`updatedb(eventId)`	Solicita a execução de um componente WIEvent do tipo `UPDATE` identificado pelo parâmetro `eventId`. O componente só será efetivamnete executado ao se chamar o método `execute()`.
`next()`	Avança para o próximo registro ou linha de dados retornado pela chamada ao método `selectdb()`.
`column(nome)`	Retorna o valor da coluna indicada pelo parâmetro `nome`. O parâmetro `nome` pode ser tanto o nome da coluna quanto o seu índice no registro ou linha de dados, sendo que a contagem dos índices das colunas começa a partir de 1.
`go(linha)`	Move o cursor para o índice da linha indicada pelo parâmetro `linha` dentro dos registros ou linha de dados retornadas. A contagem dos índices das linhas começa a partir de 1.
`rowcount()`	Retorna o número de registros ou de linhas de resultado retornadas pela chamada ao método `selectdb()`.

`populateCombo(eventId, combo, código, valor)`	Método utilitário que popula a combo passada pelo parâmetro `combo` com os dados retornados pela execução de um componente WIEvent do tipo `SELECT` identificado pelo parâmetro `eventId`. O parâmetro `codigo` indica o nome ou índice da coluna que será usada para popular os valores da combo já o parâmetro `valor` indica o nome ou índice da coluna que será usada para popular os textos dos valores da combo, se esses dois parâmetros forem omitidos o método irá utilizar os dados da primeira coluna para popular os valores e os dados da segunda coluna para popular os textos dos valores da combo.
`setInputValue(eventId, input, coluna)`	Método utilitário que configura o atributo `VALUE` de um campo `INPUT` do tipo `TEXT` passado pelo parâmetro `input` com o valor da coluna indicada pelo parâmetro `coluna` retornado por um componente WIEvent do tipo `SELECT` identificado pelo parâmetro `eventId`, caso a execução do WIEvent retorne mais de uma linha de resultado será utilizado o valor da primeira linha.

1.3.6.1 SELECT

WIEvent SELECT

O componente WIEvent SELECT consiste na definição de um comando de consulta a ser executado pelo SGBD sendo que o resultado dessa consulta será retornado ao cliente (*browser*) para manipulação. Para solicitar a execução de um WIEvent e posteriormente manipular os resultados dessa execução o desenvolvedor deverá utilizar os métodos de um objeto WIEvent que deverá ser instanciado na página. Para saber maiores detalhes sobre o funcionamento do WIEvent como um todo clique aqui.

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da

execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executado pelo banco de dados evitando assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome = `. Se o campo `Filtrar` não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos respectivos identificadores para que o conteúdo correto seja exibido.

O campo `SQL` contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a `stored procedures`, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

1.3.6.2 UPDATE

WIEvent UPDATE

O componente WIEvent SELECT consiste na definição de um comando de atualização a ser executado pelo SGBD sendo que o resultado dessa consulta será retornado ao cliente (browser) para manipulação. Para solicitar a execução de um WIEvent e posteriormente manipular os resultados dessa execução o desenvolvedor deverá utilizar os métodos de um objeto WIEvent que deverá ser instanciado na página. Para saber maiores detalhes sobre o funcionamento do WIEvent como um todo clique aqui.

O `Auto-Commit` indica se um grupo de instruções SQL serão executadas individualmente ou

executadas agrupadamente em uma única transação.

Se este campo estiver marcado então todas as instruções serão executados individualmente, ou seja, a execução de uma instrução não influenciará na execução da outra.

Caso o campo não esteja marcado as instruções serão agrupados em uma única transação a ser enviada ao SGBD e ao término da transação um `commit` será executado para persistir todas as alterações, se ocorrer algum erro durante o processamento da transação um `rollback` será automaticamente executado revertendo as tabelas para o estado em que se encontravam antes do início da execução da transação.

Um detalhe a ser levado em consideração é que o suporte ao controle de transações depende exclusivamente do sistema gerenciador de banco de dados e/ou da versão do driver JDBC que estejam sendo usados portanto assegure-se disso antes de desabilitar o *check-box* `Auto-Commit`.

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Condição Para Cada Linha` define a condição a ser testada antes de que cada iteração do evento **Update Múltiplo** seja realizada. Caso o desenvolvedor tenha definido o campo `Quantidade` com `10` então durante as `10` iterações essa condição será testada.

Se este campo for deixado em branco e o desenvolvedor queira utilizar o recurso de update múltiplo a condição será avaliada como `FALSE` e o Update não será realizado em nenhuma das iterações. Então caso se queira que sempre todos os Updates sejam executados coloque `TRUE` como sendo a condição desse campo.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executado pelo banco de dados evitando assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome = .` Se o campo `Filtrar` não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

O campo `Identificador` serve para que o desenvolvedor entre com um identificador único para o componente de página ou de projeto que está sendo definido. Será através desse identificador que o componente e/ou seus valores serão referenciado nas páginas do projeto.

A depender do componente que se estiver definindo para que o mesmo seja referenciado numa página será necessário o acréscimo de um prefixo como é o caso dos componentes Grid e Combo, nesses dois casos é obrigatório o uso dos prefixos `grid.` e `combo.` antes dos

respectivos identificadores para que o conteúdo correto seja exibido.

O campo `Mensagem OK` define a mensagem que será gravada no objeto definido no campo `Resposta (objeto)` caso o evento tenha sido executado com sucesso. Quando se usa o recurso de update múltiplo é gerada uma mensagem para cada iteração do evento, para obter a mensagem específica de um dos passos da iteração use a seguinte sintaxe:

```
<objeto-resposta>.<passo-da-iteração>
```

Por exemplo, para acessar o segundo passo de uma iteração de um update múltiplo cujo objeto-resposta seja `tmp.respObj` bastaria refenciar `|tmp.respObj.2|`.

O campo `Prefixo` indica qual o prefixo comum dos nomes dos campos do formulário que serão usados durante a execução do componente Update. Esse prefixo irá auxiliar o recurso de update múltiplo a descobrir quais as variáveis que estão no contexto do WebIntegrator que serão usadas durante a iteração.

Para um correto funcionamento do recurso de update múltiplo os nomes dos campos do formulário referenciados pelo componente Update devem estar da seguinte forma:

```
<prefixo><1..quantidade>.<identificador-do-campo>
```

Para oferecer compatibilidade com a sintaxe de Objetos o desenvolvedor poderá definir o prefixo finalizando com `[]`. Com o prefixo terminando com `[]` se está informando ao WebIntegrator que as variáveis do contexto do WebIntegrator a serem utlizadas pelo update múltiplo estão na seguinte forma:

```
<prefixo>[<1..quantidade>].<identificador-do-campo>
```

Com esta última forma o desenvolvedor poderá fazer uso de resultados armazenados em um Objeto diretamente em update múltiplo.

O campo `Quantidade` define quantas linhas serão exibidas por página. Caso o número de linhas retornadas seja maior que a quantidade definida o usuário poderá fazer uso da propriedade `link (|grid.<identificador-do-grid>.link|)` que gerará uma barra de navegação com links para acessar as próximas linhas ou retornar às linhas anteriores.

O campo `Resposta (objeto)` indica o nome do objeto que receberá uma das mensagens definidas para o componente de acordo com o resultado final (sucesso/falha) da execução do comando enviado ao banco de dados.

O campo `SQL` contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a stored procedures, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

1.3.7 Servidor de WebServices

Definições de Servidor de WebServices

Este é o local onde o desenvolvedor pode configurar um método de web service para executar a lógica associada a uma página.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Método serve para informar o nome do método que se está definindo para esse Web Service.

O campo Objeto WI indica o nome do objeto que será inserido na sessão do usuário que servirá como o prefixo a ser inserido nos parâmetros de requisição de um cliente e em seguida esse objeto estará disponível para a página a ser processada representando a lógica desse método. Exemplo, se houver um objeto definido como tmp.ws e os parâmetros de requisição desse web service estiverem definidos como: <nome>|tmp.nome|</nome> Para recuperar o parâmetro nome que veio pela requisição do cliente o desenvolvedor deverá referenciar tmp.ws.nome.

O campo Página indica qual página do projeto contém a lógica de execução respectiva a esse método desse web service. Dessa página apenas o pré-página será executado e espera-se que ao final de seu processamento um grid XML seja gerado contendo a resposta a ser retornado ao solicitante da requisição.

O campo Requisição XML[1] indica como será o layout da mensagem que esse método espera receber. Pode ser definida a variável que vai receber o conteúdo como texto da tag. Ex: <Nome>|tmp.nome|</Nome>

O campo Resposta XML[1] indica como será o layout da mensagem a ser enviada como resposta ao solicitante. Tanto a requisição quanto a resposta podem ter um atributo type nas tags informando o tipo de dado que a tag contém. Exemplo: <codigo type="xsd:string"/>. Os possíveis valores para o atributo type são: string, int, array, float e boolean.

O campo Web Service serve para informar o nome do web service que se está definindo ou que se deseja invocar. É importante lembrar que ao fazer uma requisição a esse serviço é necessário informar pela URL o projeto do qual ele faz parte. Se você estiver definindo um web service pelo WebIntegrator esse serviço poderá ser invocado bastando acessar a URL no padrão http://<host>[:<porta>]/<projeto>/<nome-do-web-service>.ws. Para acessar o WSDL de um web service definido pelo WebIntegrator use uma URL no padrão http://<host>[:<porta>]/<projeto>/<nome-do-web-service>.wsdl.

1.3.7.1 Condição de Falhas

Condição de Falhas

Este é o local onde o desenvolvedor pode configurar condições para lançar falhas associadas a um webservice.

O campo Condição serve para que o desenvolvedor da aplicação informar qual condição deve ocorrer para que a falha seja lançada.

O campo Código indica o código da falha.

O campo Descrição indica uma descrição sobre a falha lançada.

O campo Lista de Falhas indica as falhas já cadastradas e sua ordem de validação.

1.4 Componentes de Página

Componentes de Página

Um dos recursos do WI que merece destaque, é a utilização de **Componentes**. Os componentes de interface e eventos pré-programados, dispensam conhecimentos específicos da linguagem Java para a sua utilização. Com eles o desenvolvedor da aplicação terá condições suficientes para a criação de intefaces dinâmicas JSP.

1.4.1 Ant Script

Ant Script

Este é o local onde o desenvolvedor executa um script Ant que deve estar dentro do WEB-INF/antscripts do projeto.

O campo `Condição` contém a expressão condicional a ser testada para saber se o script deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Arquivo` serve para que o desenvolvedor selecione o script a ser executado.

O campo `Alvo(target)` serve para que o desenvolvedor indique o target a ser executado. Se nenhum for informado será assumido o default target do script.

1.4.2 Apagar

Apagar

Este é o local onde o desenvolvedor define sob qual condição um ou mais objetos sejam apagados do contexto do WebIntegrator.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Objetos Separados por Vírgula` contém o(s) nome(s) do(s) objeto(s) que será(ão) apagado(s). Caso se queira apagar mais de um objeto eles terão de estar separados por vírgula. As seguintes máscaras também podem ser usadas para apagar um grupo de objetos:

wi.session.id Destrói a sessão do WebIntegrator (exceto se estiver no WI Builder).

***** Apaga apenas as variáveis que se encontram no nível raiz, ou seja, aqueles identificadores que não possuem ponto.

. Apaga todo o conteúdo de uma sessão de um projeto.

tmp.* Apaga todos os identificadores (variáveis ou objetos) que começem com `tmp.`.

pvt. Apaga todos os identificadores (variáveis ou objetos) que começem com `pvt.`, tem o mesmo efeito se usar `pvt.*` como máscara.

1.4.3 Arquivo

Arquivo

Componentes para manipulação de arquivos.

1.4.3.1 Importar

Importar arquivo

Este componente faz com que um arquivo sejá lido para posteriormente ser incluído em alguma página do projeto. O conteúdo do arquivo será armazenado no contexto da sessão de um projeto sendo que o mesmo poderá ser exibido fazendo uma referência ao nome do objeto que foi definido.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Condição Falsa` contém o caminho absoluto do arquivo que será processado caso a condição seja avaliada como falsa.

O campo `Condição Verdadeira` contém o caminho abosulto do arquivo que será processado caso a condição seja avaliada como verdadeira.

O campo `Decodificar XML` indica se um arquivo que siga uma formatação XML seja processado. Ativando essa opção o conteúdo do arquivo não será colocado no contexto, ao invés

disso serão armazenadas no contexto as variáveis resultantes do seu processamento. Para saber com maiores detalhes como acessar essas variáveis clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Objeto` define o nome do objeto que será inserido na sessão do usuário após a execução do componente, a depender do componente o objeto poderá possuir uma série de atributos e métodos. Por exemplo, o objeto definido em componentes do tipo Objeto possuem os métodos `size()` e `columnNames()` além de uma série de atributos representados pelos nomes das colunas vindas do resultado da consulta ao banco de dados.

No caso de um objeto declarado na definição de um web service esse objeto irá conter os parâmetros de resposta que serão retornados após a execução da chamada ao web service. Para saber quais os possíveis atributos e métodos de um objeto leia a documentação específica do componente.

A opção `Processa Variáveis` indica se será realizado o processamento de variáveis que possam existir no conteúdo do arquivo. Esta opção apenas será válida para arquivos com conteúdo textual, ou seja, que tipo MIME seja `text/*`.

1.4.3.2 Exportar

Exportar arquivo

Este é o local onde o desenvolvedor define sob qual condição o conteúdo de uma variável do contexto do WI será gravado em um arquivo.

O campo `Arquivo(||1)` indica o nome do arquivo a ser criado.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Conteúdo` indica o nome de uma variável contida na sessão do WebIntegrator que terá o seu conteúdo armazenado em arquivo.

A opção `Cria Diretório` indica se o diretório especificado na definição do componente deverá ser criado caso ele não exista.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Diretório(||1)` indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do

WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo `Incluir ao final` indica se o conteúdo será acrescido ao final do arquivo caso o arquivo já exista.

1.4.3.3 Apagar (Local)

Apagar arquivos locais

Este é o local onde o desenvolvedor define as condições sob as quais determinados arquivos de um diretório localizado na máquina onde está instalado o WI serão apagados.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Diretório(||1)` indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo `Máscara(||1)` contém a máscara que será aplicada pelo componente antes da execução de sua ação (listagem, remoção, etc). Caso queira que apenas os arquivos com extensão GIF, JPG e BMP sejam apagados a máscara seria "`*.gif, *.bmp, *.jpg`", sem as aspas.

O campo `Remover Diretório Vazio` indica se o diretório também deve ser apagado caso ele esteja vazio.

1.4.3.4 Apagar (FTP)

Apagar arquivos via FTP

Este é o local onde o desenvolvedor define as condições sob as quais determinados arquivos de um diretório localizado em um servidor FTP serão apagados.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Diretório`(| |[1]) indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo `Máscara`(| |[1]) contém a máscara que será aplicada pelo componente antes da execução de sua ação (listagem, remoção, etc). Caso queira que apenas os arquivos com extensão GIF, JPG e BMP sejam apagados a máscara seria "`*.gif, *.bmp, *.jpg`", sem as aspas.

O campo `Remover Diretório Vazio` indica se o diretório também deve ser apagado caso ele esteja vazio.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

1.4.3.5 Listar (Local)

Listar arquivos locais

Este é o local onde o desenvolvedor define sob qual condição será gerada uma lista com o conteúdo de um diretório que está localizado na mesma máquina onde está instalado o WI. A estrutura visual da lista está definida em um grid.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Diretório(||1) indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo Grid, Combo, Download ou Upload lista quais componentes de projetos estão disponíveis para esse componente de acordo com o seu tipo, ou seja, na definição de um componente de listagem de e-mails serão listados os grids HTML do tipo POP3, num componente Grid SQL seriam listados os grids SQL, num componente Upload seriam exibidos os uploads definidos para o projeto, etc.

Se o campo possuir o link Editar o desenvolvedor poderá usá-lo para ser redirecionado para a tela de definição específica do componente que se encontra selecionado.

O campo Máscara(||1) contém a máscara que será aplicada pelo componente antes da execução de sua ação (listagem, remoção, etc). Caso queira que apenas os arquivos com extensão GIF, JPG e BMP sejam apagados a máscara seria "*.gif, *.bmp, *.jpg", sem as aspas.

1.4.3.6 Listar (FTP)

Listar arquivos via FTP

Este é o local onde o desenvolvedor define sob qual condição será gerada uma lista com o conteúdo de um diretório que está localizado em um servidor FTP. A estrutura visual da lista está definida em um grid.

O campo Condição contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Diretório(||1) indica o caminho completo do diretório que será usado pelo componente. Verifique as permissões do diretório para saber se o usuário do processo do WebIntegrator tem as permissões necessárias para realizar as devidas ações.

O campo Grid, Combo, Download ou Upload lista quais componentes de projetos estão disponíveis para esse componente de acordo com o seu tipo, ou seja, na definição de um componente de listagem de e-mails serão listados os grids HTML do tipo POP3, num componente Grid SQL seriam listados os grids SQL, num componente Upload seriam exibidos os

uploads definidos para o projeto, etc.

Se o campo possuir o link `Editar` o desenvolvedor poderá usá-lo para ser redirecionado para a tela de definição específica do componente que se encontra selecionado.

O campo `Máscara( | |`[1]`)` contém a máscara que será aplicada pelo componente antes da execução de sua ação (listagem, remoção, etc). Caso queira que apenas os arquivos com extensão GIF, JPG e BMP sejam apagados a máscara seria "`*.gif, *.bmp, *.jpg`", sem as aspas.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

1.4.4 Combo, Grid, Download e Upload

Referência a Componente do Projeto

Este é o local onde o desenvolvedor faz referência a um componente do projeto (combo, grid, download, upload).

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

Os campos `Combo, Grid, Download e Upload` servem para que o desenvolvedor da aplicação selecione o componente do projeto em questão.

O campo `Complemento do Identificador` serve para que o desenvolvedor possa criar um sub-identificação para uma combo ou grid, permitindo que os mesmos sejam colocados mais de uma vez na página. No caso do grid se o complemento do identificador estiver preenchido não poderá ser usada a opção para gerar na montagem da página. Ex: |combo.estado.naturalidade| e |combo.estado.residencia|

1.4.5 Conector Java

Conector Java

Este é o componente responsável pela execução de uma classe Java que implemente a interface Java `br.com.itx.integration.InterfaceConnector` ou extenda a classe `br.com.itx.integration.AbstractConnector` da API do WebIntegrator. Classes que implementam essa interface são conhecidas pelo WebIntegrator como conectores Java, pois servem como uma camada de conexão entre o WebIntegrator e os recursos externos com os quais se desejam interagir.

Os conectores Java também podem implementar a interface `br.com.itx.integration.InterfaceParameters`, que tem como principal objetivo, definir quais são os parâmetros de configuração que devem ser passados para o conector, e fazer com que o **WI Builder** exíba-os como campos a serem preenchidos durante a configuração do conector. O conector que implementar essa interface não precisa se preocupar como os campos serão exibidos, apenas deverá informar qual será o rótulo e um possível valor padrão para o parâmetro.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Nome da classe` deve conter o nome completamente qualificado da classe que representa um conector Java.

O campo `Parâmetros` irá listar os parâmetros de configuração do conector Java desde que esse conector também tenha implementado a `interface br.com.itx.integration.InterfaceParameters`.

1.4.6 Cookie

Cookie

Componentes para leitura e gravação de Cookies.

1.4.6.1 Ler

Ler Cookie

Este componente faz leitura de *cookies* armazenando o valor lido em um objeto que depois poderá ser referenciado para ser usado pelas páginas do projeto.

O campo Condição contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Nome indica o nome do *cookie* a ser lido/gravado. No caso de leitura para acessar o valor do *cookie* basta referenciar o nome definido no campo Objeto.

O campo Objeto define o nome do objeto que será inserido na sessão do usuário após a execução do componente, a depender do componente o objeto poderá possuir uma série de atributos e métodos. Por exemplo, o objeto definido em componentes do tipo Objeto possuem os métodos `size()` e `columnNames()` além de uma série de atributos representados pelos nomes das colunas vindas do resultado da consulta ao banco de dados.

No caso de um objeto declarado na definição de um web service esse objeto irá conter os parâmetros de resposta que serão retornados após a execução da chamada ao web service. Para saber quais os possíveis atributos e métodos de um objeto leia a documentação específica do componente.

1.4.6.2 Gravar

Gravar Cookie

Este é o local onde o desenvolvedor cria ou atualiza o valor de algum *cookie* que será usado pelo projeto. Além do valor que será gravado, o desenvolvedor especifica a duração que este *cookie* terá validade, a condição na qual ele será atualizado/criado entre outras opções.

O campo Condição contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Domínio indica o domínio dentro do qual este *cookie* deveria estar presente. A forma do nome do domínio é especificado pela RFC 2109. Um nome de domínio começa com um ponto como `.foo.com` e significa que o *cookie* está visível aos servidores em uma zona de DNS específica por exemplo, `www.foo.com` mas não a `.b.foo.com`. Por *default* os *cookies* apenas são retornados ao servidor que os envia.

O campo Nome indica o nome do *cookie* a ser lido/gravado. No caso de leitura para acessar o valor do *cookie* basta referenciar o nome definido no campo Objeto.

O campo Path indica o caminho completo da página que está solicitando a gravação do *cookie*.

O *cookie* estará visível para todas as páginas no diretório que for especificado aqui e todas as páginas dos subdiretórios desse diretório.

O campo Validade indica o tempo máximo de duração do *cookie* em segundos.

Um valor positivo indica que o *cookie* expirará depois desse valor em segundos ter sido ultrapassado. Note que o valor é o tempo máximo quando o *cookie* irá expirar e não o tempo de duração do *cookie*.

Um valor negativo significa que o *cookie* não será armazenado persistentemente (gravado em disco) e será deletado qunado o browser for fechado. Um valor zero (0) apaga o *cookie*.

O campo Valor indica qual o valor a ser gravado no *cookie*.

1.4.7 Desvio

Desvio

Desvio Condicional (o desenvolvedor define sob qual condição ocorrerá um desvio para uma outra página) e **Desvio SQL** (o desenvolvedor define sob qual condição uma consulta SQL será executada e, a partir daí, será realizado um desvio para uma outra página do projeto).

1.4.7.1 Condicional

Desvio Condicional

Este é o local onde o desenvolvedor define sob qual condição ocorrerá um desvio para uma outra página do projeto.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Destino` indica a página ou URL para a qual será feito o redirecionamento caso a condição seja avaliada como veradadeira. O desvio sempre é feito tomando como caminho base o diretório de documentos do projeto do WebIntegrator. Caso se deseje fazer um redirecionamento para uma outra página que não faça parte do projeto inicie com `http://` e complete a URL que será chamada.

O campo `Retorna` indica se o **WI Engine** irá retornar à página que efetuou o desvio após o término do processamento da página para a qual foi desviada. O ponto de retorno à página será exatamente onde foi originado o desvio.

1.4.7.2 SQL

Desvio SQL

Este é o local onde o desenvolvedor define sob qual condição uma consulta SQL será executada e a depender do que ela retorne e do que esteja configurado será realizado um desvio para uma outra página do projeto.

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Destino` indica a página ou URL para a qual será feito o redirecionamento caso a condição seja avaliada como veradadeira. O desvio sempre é feito tomando como caminho base o diretório de documentos do projeto do WebIntegrator. Caso se deseje fazer um redirecionamento para uma outra página que não faça parte do projeto inicie com `http://` e complete a URL que será chamada.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executada pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome =` . Se o campo `Filtrar` não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

O campo `Objeto` define o nome do objeto que será inserido na sessão do usuário após a execução do componente, a depender do componente o objeto poderá possuir uma série de atributos e métodos. Por exemplo, o objeto definido em componentes do tipo **Objeto** possuem os métodos `size()` e `columnNames()` além de uma série de atributos representados pelos nomes das colunas vindas do resultado da consulta ao banco de dados.

No caso de um objeto declarado na definição de um web service esse objeto irá conter os parâmetros de resposta que serão retornados após a execução da chamada ao web service. Para

saber quais os possíveis atributos e métodos de um objeto leia a documentação específica do componente.

O campo SQL contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a `stored procedures`, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

O campo `Se não retornar registro` define se o desvio será executado quando o comando SQL não retornar nenhuma linha.

1.4.8 E-mail

E-mail

Os componentes de e-mail oferecem suporte a operações básicas mensagem eletrônica, isto é, temos a nossa disposição componentes para enviar, listar, receber e apagar mensagens.

1.4.8.1 Enviar

Enviar e-mail

O componente Enviar e-mail realiza o envio de uma mensagem eletrônica para um ou mais destinatários podendo srem usados os recursos de cópia (CC) e cópia oculta (CCO) incluisive o conteúdo da mensagem podendo ser em HTML e com a possibilidade de serem anexados arquivos.

O envio de anexos utiliza como repositório um diretório onde todo o seu conteúdo será anexado à mensagem e assim que a mensagem for enviada todo o seu conteúdo sera apagado, sendo assim recomendamos que ao definir o diretório de anexos de um e-mail seja utilizado algum identificador único, como a sessão do usuário (`wi.session.id`), no caminho desse diretório contendo apenas os anexos respectivos à mensagem.

O campo `Assunto(||1)` contém o assunto (`subject`) da mensagem a ser enviada.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Cópia(||1)` contém o e-mail a quem uma cópia da mensagem será enviada, isso equivale ao campo CC de uma mensagem. Caso o destino seja para vários endereços os mesmos terão de estar separados por vírgulas, por exemplo: `pessoa1@email.com.br, pessoa2@email.com.br, empresa1@email.com.br`

O campo `Cópia Oculta(||1)` contém o e-mail a quem uma cópia oculta da mensagem será enviada, isto equivale ao campo BCC de uma mensagem. Caso o destino seja para vários endereços os mesmos terão de estar separados por vírgulas, por exemplo: `pessoa1@email.com.br, pessoa2@email.com.br, empresa1@email.com.br`

O campo `De (e-mail)` contém o e-mail de quem está enviando a mensagem.

O campo `De (nome)` contém o nome de quem está enviando a mensagem.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Diretório dos anexos(||1)` indica em qual diretório o **WI Engine** irá buscar os anexos do email. Todos os arquivos existentes nesse diretório serão anexados ao email.

O campo `Executar a Cada E-mail` indica qual a página do projeto que será executada para cada e-mail que for enviado.

O campo `Formato HTML` indica se o tipo MIME da mensagem será "`text/html`". O tipo MIME padrão é "`text/plain`".

O ícone  do campo `Modelo` edita o arquivo de modelo que representa o modelo do corpo da mensagem. Nesse arquivo de modelo podem ser feitas referências às variáveis de sessão. Para que uma imagem anexa possa aparecer dentro do email (no corpo) o desenvolvedor pode usar .

O campo `Nome do servidor WI` serve para identificar o servidor onde está o WebIntegrator junto ao servidor de email. O padrão é deixar em branco.

O campo `Para(||1)` contém o e-mail a quem se destina a mensagem. Caso o destino seja para vários endereços os mesmos terão de estar separados por vírgulas por exemplo: `pessoa1@email.com.br, pessoa2@email.com.br, empresa1@email.com.br`

O campo `Processar cada cópia` será utilizado para enviar emails personalizados para uma lista de pessoas. Essa lista deve ser colocada no campo `Para`. Para cada elemento da lista o **WI Engine** irá processar o modelo passando o email que está sendo processado atualmente na variável `wi.email`.

O campo `Remover anexos` indica se o **WI Engine** deve apagar os arquivos do diretório de anexos do e-mail após o envio do mesmo.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

1.4.8.2 Listar

Listar e-mail

Este é o local onde o desenvolvedor entra com as definições para exibir a lista de mensagens disponíveis em um servidor POP3. Uma das opções de configuração é justamente o grid que será responsável pela exibição da lista.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Grid`, `Combo`, `Download` ou `Upload` lista quais componentes de projetos estão disponíveis para esse componente de acordo com o seu tipo, ou seja, na definição de um componente de listagem de e-mails serão listados os grids HTML do tipo POP3, num componente Grid SQL seriam listados os grids SQL, num componente Upload seriam exibidos os uploads definidos para o projeto, etc.

Se o campo possuir o link `Editar` o desenvolvedor poderá usá-lo para ser redirecionado para a tela de definição específica do componente que se encontra selecionado.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

1.4.8.3 Receber

Receber e-mail

Este é o local onde o desenvolvedor entra com as configurações do recebimento de mensagens através do WI. Caso uma mensagem seja composta por um ou vários anexos, o desenvolvedor também indicará qual o grid responsável pela visualização da lista de anexos.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Grid`, `Combo`, `Download` ou `Upload` lista quais componentes de projetos estão disponíveis para esse componente de acordo com o seu tipo, ou seja, na definição de um componente de listagem de e-mails serão listados os grids HTML do tipo POP3, num componente Grid SQL seriam listados os grids SQL, num componente Upload seriam exibidos os uploads definidos para o projeto, etc.

Se o campo possuir o link `Editar` o desenvolvedor poderá usá-lo para ser redirecionado para a tela de definição específica do componente que se encontra selecionado.

O campo `ID do e-mail(||¹)` serve para que o desenvolvedor entre com um identificador único para o e-mail a ser recebido.

O campo `Objeto` indica o nome do objeto a partir do qual o desenvolvedor poderá acessar as várias partes de um e-mail recebido. As propriedades desse objeto representam os dados do cabeçalho de um e-mail, como por exemplo: `id` que é o identificador do email, `body` que retorna o corpo do email, `from` que indica quem é o remetente do e-mail, `date` que informa a data da mensagem, `subject` que retorna o assunto do e-mail, etc.

Para listar todos os cabeçalhos e consequentemente as propriedades disponíveis de um e-mail recebido referencie em uma página apenas o nome do objeto, sem nenhuma propriedade.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

1.4.8.4 Apagar

Apagar e-mail

Este é o local onde o desenvolvedor configura quando e quais serão as mensagens a serem apagadas de um servidor. Note que para saber quais as mensagens que serão apagadas, o desenvolvedor fará uso de seus IDs que foram anteriormente definidos em `Listar e-mail`.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `IDs dos e-mails(||`[1]`)` contém a lista de IDs dos e-mails, separados por vírgula, que serão apagados do servidor.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

1.4.9 Gravar

Gravar

O componente Gravar permite que o desenvolvedor define sob qual condição um certo valor ou um conjunto de valores seja gravado no contexto do WebIntegrator.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Objetos` indica o nome de um ou mais objetos separados por vírgula que receberão os respectivos conteúdos que se encontram listados no campo `Se condição verdadeira` se a condição for avaliada como verdadeira. Se a condição for avaliada como falsa os valores do campo `Se condição falsa` apenas serão atribuídos se a opção `Processa condição falsa` estiver marcada.

Para gravar o conteúdo completo de um objeto em outro preencha esse campo com o nome do objeto que receberá o conteúdo seguido de ponto, assim: `meuNovoObjeto.`.

A opção `Processa Condição Falsa` indica se os valores definidos no campo `Se condição falsa` serão atribuídos aos objetos caso a condição seja avaliada como falsa.

O campo `Recursivo` indica se o processamento recursivo está ativado. Com esse recurso ativado se o valor de um objeto fizer referência a outro objeto, esse outro objeto será processado.

O campo `Salva Codificado` indica se o conteúdo da variável será armazenado criptografado. Ao se fazer referência à variável o seu valor é automaticamente descriptografado.

O campo `Se Condição Falsa` define o que será gravado nesse objeto caso a condição seja avaliada como falsa. Valem as mesmas regras do campo `Se condição verdadeira`.

O campo `Se Condição Verdadeira` define os valores que serão atribuídos aos objetos caso a condição seja avaliada como verdadeira. No caso de gravação de múltiplos objetos também deve-se separar os valores com vírgula (`,`). Caso um dos conteúdos a serem gravados contenha vírgula então todos eles devem estar delimitados entre chaves (`{}`) por exemplo:

```
{funcao1(arg1, arg2)}, {funcao2(arg3, arg4)}
```

No caso de gravação de objetos use o nome do objeto a ser copiado seguido de ponto e entre pipes, assim: `|meuAntigoObjeto.|`.

Existem duas variáveis funções especiais que podem ser utilizadas dentro do Gravar e são elas:

|$wi.context(...)$| - Grava numa variável os valores que estão no contexto usando a máscada passada entre parênteses.

|$wi.syncContext(...)$| - Sincroniza a sessão com as variáveis de sessão do contexto atual. Essa função é importante quando se usa includes (<jsp:include page="sub_page.wsp"/>)que precisam de variáveis pvt.

1.4.10 Lista

Lista

Este é o local onde o desenvolvedor define sob qual condição será gerada uma lista de valores baseada no resultado de uma consulta SQL. A lista gerada seguirá sempre o padrão descrito abaixo:

```
<antes-de-cada-elemento><valor1><depois-de-cada-elemento>[<separador-de-elementos><antes-de-cada-elemento><valor2><depois-de-cada-elemento>...]
```

Caso se queira gerar uma lista onde os valores venham um abaixo do outro, o desenvolvedor poderá fazer uso dos marcadores `\r` (retorno do carro) ou `\n` (nova linha). Se um elemento Lista for criado usando como separador de elementos a vírgula (`,`) e esta lista devendo retornar 5 linhas, alguns resultados possíveis são:

- Se todas as linhas forem diferentes de vazio, o WI retorna: `A,B,C,D,E,F`
- Se apenas as linhas 1 e 5 forem diferentes de vazio, o WI retorna: `A,,,,,F`
- Se apenas a linha 5 for diferente de vazio, o WI retorna: `,,,,,F`

O campo `Antes de Cada Elemento` define o conteúdo que será colocado no início de cada elemento da lista.

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Depois de cada Elemento` define o conteúdo que será colocado no final de cada elemento da lista.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executado pelo banco de dados, evitando assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome = `. Se o campo `Filtrar` não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

O campo `Objeto` indica o nome do objeto que servirá para fazer referência à lista a ser criada. O nome aqui digitado serve como um prefixo a ser colocado junto com o nome da coluna do banco de dados da qual queremos montar a lista.

Para exemplificar suponha que tenha sido criada a lista com o campo `Objeto` definido como `tmp.minhaLista` e que a consulta retorne um conjunto de valores nas colunas `colNome` e

colFone, então para criar uma lista com os valores da coluna colNome faria |tmp.minhaLista.colNome| e para uma lista com os valores da coluna colFone faria |tmp.minhaLista.colFone|.

O campo SQL contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?).

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a stored procedures, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

O campo Separador de Elementos define qual será o separador de elementos utilizado na contrução da lista.

1.4.11 Objeto

Objeto

Um componente do tipo Objeto representa um conjunto de atributos que ficam armazenados na sessãodo usuário possuindo como prefixo um identificador comum, sendo esse identificador comum o identificador definido para o Objeto. Os nomes e os valores dos atributos que fazem parte de um Objeto são o resultado da execução de uma consulta enviada a um banco de dados.

Os nomes das colunas retornados pela pesquisa indicarão quais serão os nomes dos atributos desse Objeto e para cada linha do resultado será definido um índice formando uma estrutura semelhante a uma matriz oferecendo assim uma maneira fácil e intuitiva para acessar um determinado valor.

A depender do banco de dados e do driver usado para acessá-lo o Objeto gerado também possuirá o método size() que informa a quantidade de linhas retornada pela pesquisa enviada ao banco de dados. Um outro método disponibilizado é o columnNames() que irá retornar os nomes das colunas retornados pela pesquisa separados por vírgula.

A sintaxe usada para acessar um determinado valor de um atributo armazenado num elemento Objeto do WebIntegrator é a seguinte:

```
<identificador-do-objeto>[<índice-da-linha>].<nome-ou-índice-do-atributo>
```

A ilustração a seguir exibe uma estrutura semelhante a uma matriz representando um Objeto

cujo identificador foi definido como `tmp.meuObj`. Para acessar o valor do atributo `atrib2` que está na segunda linha basta refenciar desta maneira `|tmp.meuObj[2].atrib2`.

	atrib1	atrib2	...	atribN
1	tmp.meuObj[1].atrib1	tmp.meuObj[1].atrib2	...	tmp.meuObj[1].atribN
2	tmp.meuObj[2].atrib1	tmp.meuObj[2].atrib2	...	tmp.meuObj[2].atribN
...	...	...	...	...
tmp.meuObj.size()	tmp.meuObj[size()].atrib1	tmp.meuObj[size()].atrib2	...	tmp.meuObj[size()].atribN

Para acessar os valores da primeira linha de um Objeto há uma sintaxe reduzida onde o valor do índice da linha pode ser omitido, então referenciando `tmp.meuObj.atrib1` seria retornado o valor do atributo `atrib1` que está armazenado na primeira linha do Objeto tmp.meuObj.

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executada pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome =` . Se o campo `Filtrar` não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

O campo `Objeto` define o nome do objeto que será inserido na sessão do usuário após a execução do componente, a depender do componente o objeto poderá possuir uma série de atributos e métodos. Por exemplo, o objeto definido em componentes do tipo Objeto possuem os métodos `size()` e `columnNames()` além de uma série de atributos representados pelos nomes das colunas vindas do resultado da consulta ao banco de dados.

No caso de um objeto declarado na definição de um web service esse objeto irá conter os parâmetros de resposta que serão retornados após a execução da chamada ao web service. Para saber quais os possíveis atributos e métodos de um objeto leia a documentação específica do componente.

O campo `Qtde. Colunas ID` indica quantas colunas serão usadas para ajudar a formar o identicador de uma linha caso o resultado retorne mais de uma linha. Se esse campo for deixado em branco o desenvolvedor ainda poderá referenciar uma determinada linha através de seu índice.

Geralmente quando um objeto armazena o resultado de uma consulta que contenha mais de uma linha os valores desses resultados são acessados através do índice da linha, configurando esse campo haverá uma nova possibilidade de acessar uma determinada linha através dos valores das colunas que foram especificadas. Como exemplo suponha que haja uma tabela com as seguintes informações:

mensagens	
codigo	**mensagem**
10	Mensagem #10
20	Mensagem #20
30	Mensagem #30
40	Mensagem #40

Se um objeto identificado por `tmp.obj` executar uma instrução que retorne como resultado todos os dados da tabela acima e o campo `Qtde. Colunas ID` estivesse definido com 1, ao referenciar `tmp.obj.mensagem.10` se recuperaria o valor da coluna **`mensagem`** cujo valor da coluna **`codigo`** seja `10`.

O campo `SQL` contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a `stored procedures`, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

1.4.12 Relatório

Referência a Relatório

Este é o local onde o desenvolvedor faz referência a um relatório.

O campo `Condição` contém a expressão condicional a ser testada para saber se o relatório deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Relatório` serve para que o desenvolvedor da aplicação selecione o relatório do

projeto em questão.

O campo `Diretório de Saída` serve para que o desenvolvedor indique o diretório onde o relatório deverá ser gerado ao invés de ser enviado para o browser.

O campo `Formato` serve para que o desenvolvedor indique qual será o formato de saída do relatório, podendo vir de uma variável.

OBS: É possível executar o relatório num outro servidor bastando para isso configurar a variável pvt.report.url (ou tmp.report.url) apontando para o servidor que deverá processar o relatório e configurar pvt.report.key tanto no projeto cliente como no projeto que vai executar o relatório com uma palavra chave que será usada para autenticar.

1.4.13 Socket

Socket

O componente Socket tem um funcionamento semelhante ao `Importar Arquivo` onde o conteúdo de algo que está vindo através da conexão estabelecida pelo socket pode ser armazenado em uma variável do contexto do WebIntegrator e caso esse conteúdo venha no formato XML há a possibilidade de processá-lo e convertê-lo em variáveis do contexto do WebIntegrator.

Além de receber,o Socket também pode enviar qualquer conteúdo de variáveis do contexto do WebIntegrator através dessa mesma conexão estabelecida, propiciando assim a implementação de troca de mensagens entre diferentes aplicações.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Decodificar XML` indica se um arquivo que siga uma formatação XML seja processado. Ativando essa opção o conteúdo do arquivo não será colocado no contexto, ao invés disso serão armazenadas no contexto as variáveis resultantes do seu processamento. Para saber com maiores detalhes como acessar essas variáveis clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Enviar Objeto` define qual o objeto que será enviado através dessa conexão.

O campo `Receber Objeto` define o nome do objeto que receberá o conteúdo que está chegando através da conexão.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em

casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

O campo `Timeout` define o tempo, em segundos, que será esperado para que o servidor retorne algum tipo de resposta. Caso o campo seja deixado em branco, o WebIntegrator assume 60 segundos.

1.4.14 Transação

Transação

O componente Transação permite que o desenvolvedor defina um conjunto de elementos no pré-página ou pós-página que devem ser executados dentro de um contexto transacional. Os elementos internos da transação podem usar diferentes Banco de Dados e caso todas as operações ocorram com sucesso será dado um *commit* em todas as conexões e caso alguma operação tenha falhado será dado um *rollback*..

Caso o elemento Finalizar Transação não tenha sido colocado será forçada uma finalização depois do pré-página e depois do pós-página.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a transação.

O campo `Resposta(objeto)` serve para que o desenvolvedor indique uma variável que irá conter a mensagem sobre como foi a execução da transação . O desenvolvedor pode usar objeto.status() para ter detalhes da execução, pois caso nada tenha sido feito estará vazio, estará TRUE se tudo deu certo e a mensagem de erro no BD caso tenha havido erro na execução. As variáveis individuais de cada Update dentro da transação continuam existindo normalmente.

O campo `Mensagem OK` serve para que o desenvolvedor informe a mensagem que deve ser colocada no objeto caso a transação tenha sido executada com sucesso.

O campo `Mensagem de erro padrão` serve para que o desenvolvedor informe a mensagem que deve ser colocada no objeto caso a transação tenha falhado.

O campo `Mensagem caso nenhum condição tenha sido atendida` serve para que o desenvolvedor informe a mensagem que deve ser colocada no objeto caso nenhum comando tenha sido executado nos BDs.

1.4.14.1 Finalizar

Transação

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a finalização da transação. Caso o elemento finalizar não tenha sido colocado será forçada uma finalização depois do pré-página e depois do pós-página.

1.4.15 TreeView

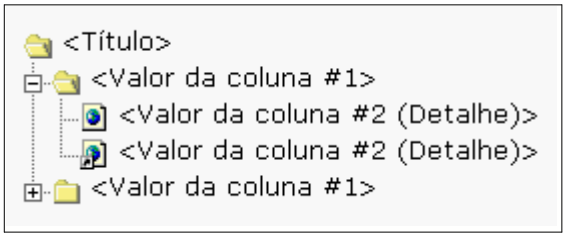
TreeView

O componente TreeView, oferece ao desenvolvedor a possibilidade de criar menus hierárquico utilizando estrutura de pastas semelhante ao Windows Explorer. Este recurso oferece uma aparência mais moderna e atual, além de serem mais fáceis de usar.

1.4.15.1 SQL

TreeView SQL

Com o componete TreeView o desenvolvedor poderá criar menus hierárquicos semelhantes à hierarquia de diretórios que é exibida pelo Windows Explorer. Eles são fáceis de usar e mais fáceis ainda de criar (isso se estiver usando o WebIntegrator). Visualmente, a saída do componente TreeView será algo semelhante com o que segue abaixo:



O exemplo de TreeView gerado acima possui apenas dois níveis hierárquicos sendo que o último nível é sempre considerado como nível de detalhe ou nó-folha pois a partir dele não há mais ramificações. Além das definições que são feitas diretamente pelo **WI Builder**, o componente TreeView oferece outras propriedades que podem ser alteradas pelo desenvolvedor. No momento em que as definições são gravadas, o WebIntegrator inseri no código-fonte da página WSP tags HTML que farão com que o TreeView seja montado e em uma dessas tags estão essas propriedades adicionais que o TreeView possui, a saber:

Propriedade	Descrição
`UniqueID`	Propriedade que contém o identificador único para esse TreeView. **Recomenda-se não alterar o valor dessa propriedade.**
`DocRoot`	Propriedade que informa o diretório-base a ser levado em consideração nos links do TreeView.
`ImgRoot`	Propriedade que informa o diretório onde está instalada a "pele" que está sendo usada pelo TreeView. O componente TreeView possui atualmente 4 tipos de "peles": •  •  •  • 
`FrameSet`	Propriedade que indica o nome da página que contém o FRAMESET, caso algum esteja sendo usado. Se o TreeView não fizer parte de um FRAMESET defina essa propriedade com uma *`string`* vazia ("").
`ImgWidth`	Propriedade que define a largura padrão das imagens exibidas pelo TreeView. **Recomenda-se não alterar o valor dessa propriedade.**
`ImgHeight`	Propriedade que define a altura padrão das imagens exibidas pelo TreeView. **Recomenda-se não alterar o valor dessa propriedade.**
`EntryHeight`	**Recomenda-se não alterar o valor dessa propriedade.**
`InitialKey`	Propriedade que define o valor inicial da chave usada internamente pelo TreeView. **Recomenda-se não alterar o valor dessa propriedade.**
`CurrPageBG`	Propriedade que define o valor da cor de *`background`* da página onde será exibido o TreeView.
`CurrPageFG`	Propriedade que define o valor da cor de *`foreground`* da página onde será exibido o TreeView.
`LinkCurrPage`	Propriedade que indica se os links serão para a própria página onde está sendo exibido o TreeView.
`TreeRootHint`	Propriedade que define o texto a ser apresentado como dica do nó raiz do TreeView.
`NormalPageHint`	Propriedade que define o texto a ser apresentado como dica de um link para uma página.
`LinkedPageHint`	Propriedade que define o texto a ser apresentado como dica de um link.
`OpenBookHint`	Propriedade que define o texto a ser apresentado como dica da figura que fecha um ramo da árvore do TreeView.
`ClosedBookHint`	Propriedade que define o texto a ser apresentado como dica da figura que abre um ramo da árvore do TreeView.
`OpenBookStatus`	Propriedade que define o texto a ser apresentado na barra de status do *`browser`* em relação à figura que fecha um ramo da árvore do TreeView.
`ClosedBookStatus`	Propriedade que define o texto a ser apresentado na barra de status do *`browser`* em relação à figura que abre um ramo da árvore do TreeView.
`window.defaultStatus`	Propriedade que define a mensagem que aparece na barra de status do *`browser`*.
`navExplain`	Propriedade que define a mensagem a ser exibida quando o TreeView não fizer parte de um FRAMESET e a propriedade checkFrames estiver setada para `true`.

FontFace	Propriedade que define a listagem das fontes a serem usadas pelo TreeView. Os nomes das fontes devem estar separados por vírgula.
compactTree	Propriedade que indica se a árvore que simboliza o TreeView será exibida numa forma compacta.
viewMatchCnt	
singleBranch	Propriedade que indica se apenas será visualizado um ramo extendido da árvore, ou seja, caso o usuário tente abrir um outro ramo da árvore e um já esteja aberto, esse último será fechado antes que o outro se ja expandido caso esta propriedade esteja definida como true.
checkFrames	Propriedade que indica se será feita a verificação se o TreeView faz parte de um FRAMESET.
baseHref	Propriedade que indica o nome do *target* dos links do TreeView, ou seja, o local onde será exibido o recurso para o qual o link aponta.

O campo Banco de Dados indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo Condição contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Filtrar define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo SQL antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executada pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome =` . Se o campo Filtrar não estiver definido com os caracteres `%` e o WebIntegrator irá processar a seguinte instrução `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1` retornando como resultado todos as linhas da tabela.

O campo Inserir Definição JS indica se o arquivo de definição JavaScript deverá ser colocado na página. Recomenda-se que na primeira vez que se esteja definindo um treeview esta opção esteja marcada.

O campo Link(| |[1]) define qual será a ação do link podendo ser uma chamada a uma função JavaScript, um endereço de uma página a ser chamada, etc. O link apenas será aplicado ao último nível da hierarquia.

O campo Níveis serve para indicar até qual coluna serão pegos os valores para a montagem do TreeView. Esse valor não pode passar da quantidade máxima de colunas que é retornada pela consulta definida no campo SQL. De acordo com o valor aqui digitado serão criados tantos níveis de hierarquia, sendo o último nível considerado como detalhe.

O campo Objeto define o nome do objeto que será inserido na sessão do usuário após a execução do componente, a depender do componente o objeto poderá possuir uma série de

atributos e métodos. Por exemplo, o objeto definido em componentes do tipo Objeto possuem os métodos `size()` e `columnNames()` além de uma série de atributos representados pelos nomes das colunas vindas do resultado da consulta ao banco de dados.

No caso de um objeto declarado na definição de um web service esse objeto irá conter os parâmetros de resposta que serão retornados após a execução da chamada ao web service. Para saber quais os possíveis atributos e métodos de um objeto leia a documentação específica do componente.

O campo `SQL` contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a `stored procedures`, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

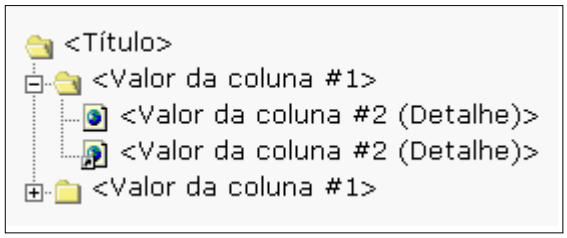
O campo `Texto(||1)` define o texto a ser exibido no nível da hierarquia que se encontra atualmente selecionado.

O campo `Título(||1)` serve para definir qual será o texto que será exibido como título do TreeView.

1.4.15.2 Manual

TreeView Manual

Com o componete TreeView o desenvolvedor poderá criar menus de índices que possuam uma funcionalidade semelhante à hierarquia de diretórios que é exibida pelo Windows Explorer. Eles são fáceis de usar e mais fáceis ainda de criar (isso se estiver usando o WebIntegrator). Visualmente, a saída do componente TreeView será algo semelhante com o que segue abaixo:



O exemplo de TreeView gerado acima possui apenas dois níveis hierárquicos sendo que o último nível é sempre considerado como nível de detalhe pois a partir dele não há mais ramificações. Além das definições que são feitas diretamente pelo **WI Builder**, o componente TreeView oferece outras propriedades que podem ser alteradas pelo desenvolvedor. No momento em que as definições são gravadas, o WebIntegrator inseri no código-fonte da página WSP tags HTML que farão com que o TreeView seja montado e em uma dessas tags estão essas propriedades adicionais que o TreeView possui, a saber:

Propriedade	Descrição
`UniqueID`	Propriedade que contém o identificador único para esse TreeView. **Recomenda-se não alterar o valor dessa propriedade.**
`DocRoot`	Propriedade que informa o diretório-base a ser levado em consideração nos links do TreeView.
`ImgRoot`	Propriedade que informa o diretório onde está instalada a "pele" que está sendo usada pelo TreeView. O componente TreeView possui atualmente 4 tipos de "peles": •  •  •  • 
`FrameSet`	Propriedade que indica o nome da página que contém o FRAMESET, caso algum esteja sendo usado. Se o TreeView não fizer parte de um FRAMESET defina essa propriedade com uma *`string`* vazia ("").
`ImgWidth`	Propriedade que define a largura padrão das imagens exibidas pelo TreeView. **Recomenda-se não alterar o valor dessa propriedade.**
`ImgHeight`	Propriedade que define a altura padrão das imagens exibidas pelo TreeView. **Recomenda-se não alterar o valor dessa propriedade.**
`EntryHeight`	**Recomenda-se não alterar o valor dessa propriedade.**
`InitialKey`	Propriedade que define o valor inicial da chave usada internamente pelo TreeView. **Recomenda-se não alterar o valor dessa propriedade.**
`CurrPageBG`	Propriedade que define o valor da cor de *`background`* da página onde será exibido o TreeView.
`CurrPageFG`	Propriedade que define o valor da cor de *`foreground`* da página onde será exibido o TreeView.
`LinkCurrPage`	Propriedade que indica se os links serão para a própria página onde está sendo exibido o TreeView.
`TreeRootHint`	Propriedade que define o texto a ser apresentado como dica do nó raiz do TreeView.
`NormalPageHint`	Propriedade que define o texto a ser apresentado como dica de um link para uma página.
`LinkedPageHint`	Propriedade que define o texto a ser apresentado como dica de um link.
`OpenBookHint`	Propriedade que define o texto a ser apresentado como dica da figura que fecha um ramo da árvore do TreeView.
`ClosedBookHint`	Propriedade que define o texto a ser apresentado como dica da figura que abre um ramo da árvore do TreeView.
`OpenBookStatus`	Propriedade que define o texto a ser apresentado na barra de status do *`browser`* em relação à figura que fecha um ramo da árvore do TreeView.

`ClosedBookStatus`	Propriedade que define o texto a ser apresentado na barra de status do *`browser`* em relação à figura que abre um ramo da árvore do TreeView.
`window.defaultStatus`	Propriedade que define a mensagem que aparece na barra de status do *`browser`*.
`navExplain`	Propriedade que define a mensagem a ser exibida quando o TreeView não fizer parte de um FRAMESET e a propriedade checkFrames estiver setada para `true`.
`FontFace`	Propriedade que define a listagem das fontes a serem usadas pelo TreeView. Os nomes das fontes devem estar separados por vírgula.
`compactTree`	Propriedade que indica se a árvore que simboliza o TreeView será exibida numa forma compacta.
`viewMatchCnt`	
`singleBranch`	Propriedade que indica se apenas será visualizado um ramo extendido da árvore, ou seja, caso o usuário tente abrir um outro ramo da árvore e um já esteja aberto, esse último será fechado antes que o outro se ja expandido caso esta propriedade esteja definida como true.
`checkFrames`	Propriedade que indica se será feita a verificação se o TreeView faz parte de um FRAMESET.
`baseHref`	Propriedade que indica o nome do *`target`* dos links do TreeView, ou seja, o local onde será exibido o recurso para o qual o link aponta.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Inserir Definição JS` indica se o arquivo de definição JavaScript deverá ser colocado na página. Recomenda-se que na primeira vez que se esteja definindo um treeview esta opção esteja marcada.

O campo `Link( | |`[1]`)` define qual será a ação do link podendo ser uma chamada a uma função JavaScript, um endereço de uma página a ser chamada, etc. O link apenas será aplicado ao último nível da hierarquia.

O campo `Nós` exibe os textos dos nós a serem visualizados pelo TreeView. Note que os textos estão dispostos de uma forma hierárquica semelhante àquela que será exibida quando em execução pelo TreeView.

O desenvolvedor pode a qualquer momento alterar a posição de um nó nesta hierarquia subindo ou descendo tanto no sentido horizontal quanto no vertical e também remover ou adicionar uma nova opção em qualquer posição.

Toda vez que se desloca um nó para esquerda ou direita esse nó estará subindo ou descendo, respectivamente, um nível na hierarquia.

O campo `Objeto` define o nome do objeto que será inserido na sessão do usuário após a execução do componente, a depender do componente o objeto poderá possuir uma série de atributos e métodos. Por exemplo, o objeto definido em componentes do tipo Objeto possuem os métodos `size()` e `columnNames()` além de uma série de atributos representados pelos nomes das colunas vindas do resultado da consulta ao banco de dados.

No caso de um objeto declarado na definição de um web service esse objeto irá conter os parâmetros de resposta que serão retornados após a execução da chamada ao web service. Para saber quais os possíveis atributos e métodos de um objeto leia a documentação específica do componente.

O campo `Texto( | | 1 )` define o texto a ser exibido no nível da hierarquia que se encontra atualmente selecionado.

O campo `Título( | | 1 )` serve para definir qual será o texto que será exibido como título do TreeView.

1.4.16 Update

Update

O componente **Update** possibilita que o desenvolvedor envie comandos de atualizações para banco de dados. Esse componente oferece um recurso chamado de update múltiplo que possibilita que o mesmo Update seja executado tantas vezes quanto um valor definido em uma única ação do usuário.

Para habilitar o recurso de update múltiplo o desenvolvedor precisará definir valores para os campos `Quantidade`, `Prefixo` e `Condição para cada linha`, o preenchimento do campo `Objeto` é opcional.

Para exemplificar o uso desse recurso suponha que se deseje criar um formulário de cadastro de novos itens de um estoque onde se possibilite o cadastro de vários itens simultaneamente. O código-fonte para esse formulário poderia ser algo parecido com o que segue abaixo:

```
...
<FORM ACTION="/|wi.proj.id|/|wi.page.id|.wsp" METHOD="post">
<TABLE>
<TR>
<TH>Código do Item</TH>
<TH>Nome do Item</TH>
<TH>Quantidade em Estoque</TH>
</TR>
<TR>
<TD><INPUT TYPE="text" NAME="tmp.update[1].codigo"></TD>
<TD><INPUT TYPE="text" NAME="tmp.update[1].nome"></TD>
<TD><INPUT TYPE="text" NAME="tmp.update[1].qtidade"></TD>
</TR>
<TR>
<TD><INPUT TYPE="text" NAME="tmp.update[2].codigo"></TD>
<TD><INPUT TYPE="text" NAME="tmp.update[2].nome"></TD>
<TD><INPUT TYPE="text" NAME="tmp.update[2].qtidade"></TD>
</TR>
```

```
<TR>
<TD><INPUT TYPE="text" NAME="tmp.update[3].codigo"></TD>
<TD><INPUT TYPE="text" NAME="tmp.update[3].nome"></TD>
<TD><INPUT TYPE="text" NAME="tmp.update[3].qtidade"></TD>
</TR>
<TR>
<TD COLSPAN="3" ALIGN="center">
<INPUT TYPE="submit" NAME="tmp.cadastrar" VALUE="Cadastrar">
<INPUT TYPE="reset"></TD>
</TR>
</TABLE>
</FORM>
...
```

Há alguns detalhes a serem percebidos no formulário acima. Primeiramente, todos os campos do formulário possuem o prefixo `tmp.update` em comum e o valor do passo da iteração encontra-se entre colchetes, então no momento da definição do componente **Update** o campo `Prefixo` deveria ser preenchido com `tmp.update[]`.

O outro detalhe a ser considerado é que para referenciar os campos do formulário no comando a ser enviado ao banco de dados o valor do passo da iteração deve ser omitido. Tomando como exemplo os campos do formulário acima bastaria referenciar `tmp.update.codigo`, `tmp.update.nome`, `tmp.update.qtidade` pois os valores do passo da iteração são substituídos internamente pelo componente **Update**.

O `Auto-Commit` indica se um grupo de instruções SQL serão executadas individualmente ou executadas agrupadamente em uma única transação.

Se este *check-box* estiver marcado então todas as instruções serão executados individualmente, ou seja, a execução de uma instrução não influenciará na execução da outra.

Caso o campo não esteja marcado as instruções serão agrupados em uma única transação a ser enviada ao SGBD e ao término da transação um commit será executado para persistir todas as alterações, se ocorrer algum erro durante o processamento da transação um *rollback* será automaticamente executado revertendo as tabelas para o estado em que se encontravam antes do início da execução da transação.

Um detalhe a ser levado em consideração é que o suporte ao controle de transações depende exclusivamente do sistema gerenciador de banco de dados e/ou da versão do driver JDBC que estejam sendo usados, portanto, assegure-se disso antes de desabilitar o campo `Auto-Commit`.

O campo `Banco de Dados` indica o identificador de um dos servidores de bancos de dados definidos para o projeto que será usado para executar o comando.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Condição Para Cada Linha` define a condição a ser testada antes de que cada iteração do evento **Update Múltiplo** seja realizada. Caso o desenvolvedor tenha definido o campo `Quantidade` com 10 então, durante as 10 iterações, essa condição será testada.

Se este campo for deixado em branco e o desenvolvedor queira utilizar o recurso de update múltiplo a condição será avaliada como FALSE e o Update não será realizado em nenhuma das iterações. Então caso se queira que sempre todos os Updates sejam executados coloque TRUE como sendo a condição desse campo.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Filtrar` define quais os caracteres que deverão ser filtrados do conteúdo das variáveis de sessão do WebIntegrator que estão sendo referenciadas no campo `SQL` antes da execução do comando. O objetivo é impedir que certos caracteres sejam passados para o comando a ser executada pelo banco de dados evitando que assim que o resultado do comando não seja intencionalmente alterado.

Como exemplo suponha que haja a instrução `SELECT * FROM tabCadastros WHERE nome LIKE |tmp.nome|% AND publico = 1` e o usuário da aplicação ao preencher o formulário coloque no campo correspondente à variável `tmp.nome` um valor como `% OR nome =` . Se o campo `Filtrar` não estiver definido com os caracteres `%,` o WebIntegrator irá processar a seguinte instrução: `SELECT * FROM cadastros WHERE nome LIKE % OR nome = % AND publico = 1,` retornando como resultado todos as linhas da tabela.

O campo `Mensagem OK` define a mensagem que será gravada no objeto definido no campo `Resposta (objeto)` caso o evento tenha sido executado com sucesso. Quando se usa o recurso de update múltiplo é gerada uma mensagem para cada iteração do evento, para obter a mensagem específica de um dos passos da iteração use a seguinte sintaxe:

`<objeto-resposta>.<passo-da-iteração>`

Por exemplo, para acessar o segundo passo de uma iteração de um update múltiplo cujo objeto-resposta seja `tmp.respObj` bastaria refenciar `|tmp.respObj.2|`.

O campo `Objeto` contém o identificador de um componente do tipo Objeto que tenha sido definido anteriormente no mesmo evento de página (pré-página ou pós-página) onde está sendo definido esse Update. A cada passo de iteração do evento Update Múltiplo esse componente Objeto será executado.

O campo `Prefixo` indica qual o prefixo comum dos nomes dos campos do formulário que serão usados durante a execução do componente Update. Esse prefixo irá auxiliar o recurso de update múltiplo a descobrir quais as variáveis que estão no contexto do WebIntegrator que serão usadas durante a iteração.

Para um correto funcionamento do recurso de update múltiplo os nomes dos campos do formulário referenciados pelo componente Update devem estar da seguinte forma:

`<prefixo><1..quantidade>.<identificador-do-campo>`

Para oferecer compatibilidade com a sintaxe de Objetos o desenvolvedor poderá definir o prefixo finalizando com `[]`. Com o prefixo terminando com `[]` se está informando ao WebIntegrator que as variáveis do contexto do WebIntegrator a serem utlizadas pelo update múltiplo estão na seguinte forma:

`<prefixo>[<1..quantidade>].<identificador-do-campo>`

Com esta última forma o desenvolvedor poderá fazer uso de resultados armazenados em um Objeto diretamente em update múltiplo.

O campo `Quantidade` indica quantas vezes esse evento será executado. Este campo apenas deve ser preenchido quando o recurso de update múltiplo for utilizado.

O campo `Resposta (objeto)` indica o nome do objeto que receberá uma das mensagens

definidas para o componente de acordo com o resultado final (sucesso/falha) da execução do comando enviado ao banco de dados.

O campo SQL contém o comando que será enviado para ser processado pelo banco de dados. Caso o comando seja composto por mais de uma instrução SQL separe-as por ponto-e-vírgula.

No componente Upload de Banco de Dados o valor da coluna relativo ao conteúdo deverá ser referenciado por um sinal de interrogação, exemplo: `INSERT INTO MinhaTabela (arquivo, conteudo) VALUES (|tmp.arquivo|, ?)`.

A estrutura do comando a ser executado é dependente do banco de dados, podendo ser feitas chamadas a stored procedures, rotinas M (através do MJava), métodos de objetos do CachéFactory, execução de instruções SQL, etc. Para saber maiores detalhes como fazer essas chamadas clique aqui.

1.4.16.1 Mensagens de Erro

Mensagens de Erro (Update)

Esta parte é usada quando se quer personalizar mensagens de erro de acordo com um código que é retornado pelo campo. Vale ressaltar que não é garantido que um código retornado por um SGBD, por exemplo o MySQL, seja igual ao de um outro SGBD, como por exemplo o PosrgreSQL, então as mensagens aqui definidas ficam dependentes do SGBD usado.

O campo Código indica qual o código de erro do banco de dados que terá sua mensagem personalizada.

O campo Mensagem indica qual a mensagem que estará associada ao código de erro digitado no campo Código.

O campo Mensagem de erro padrão(||[1]) define o texto da mensagem que será retornado caso o código de erro retornado pelo SGBD não coincida com aqueles que já tenham sido cadastrados. Para colocar a mensagem de erro completa vindo do BD utilize a variável |wi.sql.msg|.

1.4.17 WebService

WebService

Este é o local onde o desenvolvedor realiza as configurações para realizar uma chamada a um método de um web service através do WebIntegrator.

O campo `Condição` contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo `Código de Erro` indica qual o código de erro HTTP que deverá ser retornado pelo servidor caso o componente não consiga ser executado corretamente. Ao definir um código de erro para um componente, a ocorrência de um erro durante a execução do mesmo fará com que o processamento da página onde ele estiver sendo referenciado seja interrompido.

O campo `Descrição` serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo `Método` serve para informar o nome do método que se está definindo para esse Web Service.

O campo `Objeto` define o nome do objeto que será inserido na sessão do usuário após a execução do componente, a depender do componente o objeto poderá possuir uma série de atributos e métodos. Por exemplo, o objeto definido em componentes do tipo Objeto possuem os métodos `size()` e `columnNames()` além de uma série de atributos representados pelos nomes das colunas vindas do resultado da consulta ao banco de dados.

No caso de um objeto declarado na definição de um web service esse objeto irá conter os parâmetros de resposta que serão retornados após a execução da chamada ao web service. Para saber quais os possíveis atributos e métodos de um objeto leia a documentação específica do componente.

O campo `Parâmetros (XML)` contém a listagem de parâmetros num formato XML que deverá ser passada para o método selecionado.

O campo `Servidor` indica qual o servidor que será usado na execução do componente. Em casos de componentes de downloads ou uploads esse campo listará os servidores FTP cadastrados no projeto, já em componentes de manipulação de e-mail serão listados os servidores SMTP ou POP3. No caso da definição de um web service, o servidor escolhido deve ser o `endpoint` do web service a ser invocado.

O campo `URL para o WSDL` define a URL onde poderá ser encontrada a descrição WSDL para o serviço.

O campo `Web Service` serve para informar o nome do web service que se está definindo ou que se deseja invocar. É importante lembrar que ao fazer uma requisição a esse serviço é necessário informar pela URL o projeto do qual ele faz parte.

Se você estiver definindo um web service pelo WebIntegrator esse serviço poderá ser invocado bastando acessar a URL no padrão `http://<host>[:<porta>]/<projeto>/<nome-do-web-service>.ws`. Para acessar o WSDL de um web service definido pelo WebIntegrator use uma URL no padrão `http://<host>[:<porta>]/<projeto>/<nome-do-web-`

service>.wsdl.

1.4.18 JSP

JSP

O componente Código JSP permite ao desenvolvedor inserir trechos de código JSP como elemento de pré-página ou pós-página porém recomenda-se que não seja inserida nenhum tipo de lógica relacionada à interface nesse componenete, para esses casos o desenvolvedor poderá inserir trechos de código JSP no próprio código-fonte da página.

O campo Condição contém a expressão condicional a ser testada para saber se o componente deverá ser processado. Para saber maiores detalhes sobre as possíveis construções de condições clique aqui.

O campo Código JSP contém o código JSP que deverá ser inserido ao pré ou pós-página.

O campo Descrição serve para que o desenvolvedor da aplicação escreva um breve comentário sobre a ação a ser executada pelo componente em questão.

O campo Template indica qual o template de código JSP a ser inserido. As regras de montagem de um template JSP são as mesmas utilizadas no desenvolvimento de um componente para WIzard de Páginas.

1.5 WIzard de Páginas

WIzard

O WIzard é o módulo do WebIntegrator que oferece ao desenvolvedor um ambiente de desenvolvimento baseado nos conceitos RAD (Rapid Application Development) para a criação e/ou edição de páginas WSP. Neste tipo de ambiente a maior parte do tempo de desenvolvimento de uma página será destinado à definição dos valores das propriedades e dos eventos dos elementos que irão fazer parte da página sem ser preciso que o desenvolvedor se

importe com a codificação HTML necessária para gerá-la.

Um dos recursos que merece destaque no WIzard é a possibilidade de montagem automática de uma página baseada em informações coletadas de componentes do tipo **Objeto** definidos no Pré-Página. A primeira vez que o WIzard é executado para uma determinada página ou caso o desenvolvedor peça para recarregar a página o WIzard irá percorrer as definições de Pré-Página dessa página e assim que um componente Objeto é encontrado ele automaticamente insere na página WSP um formulário HTML com os campos correspondentes às colunas referenciadas pelo Objeto, caso a página venha a possuir mais de um Objeto serão criados tantos formulários quanto for a quantidade de Objetos existentes no Pré-Página.

NOTA

Como um sub-produto desse recurso há também a possibilidade da geração da lógica de Pós-Página para realizar inclusões ou alterações, essa lógica também será baseada nas definições do(s) Obejto(s) de Pré-Página. Para saber com maiores detalhes sobre o funcionamento da geração automática de páginas baseada nas definições de Objetos no Pré-Página clique aqui.

O ambiente do WIzard é composto de duas janelas: uma que exibe o layout da página como será visto pelos usuários e uma outra, a janela de Propriedades, onde são exibidas as propriedades do elemento selecionado. Na janela onde se exibe o layout da página, os elementos podem ser selecionados bastando clicar com o mouse sobre o ícone  e automaticamente a janela de Propriedades irá exibir os dados do elemento correspondente.

A janela de Propriedades contém em sua parte superior uma combo onde são listados todos os elementos que fazem parte da página e logo abaixo uma pequena barra de ferramentas que oferece as seguintes opções:

Botão	Descrição
	Insere mais um elemento na página. Esse novo elemento será inicialmente do tipo `Genérico` e será posicionado como o último elemento da página mas o desenvolvedor poderá alterar as suas propriedades a fim de reposicioná-lo e/ou trocar de tipo.
	Remove o elemento que se encontra atualmente selecionado.
	Facilitador para manutenção de página. Clicando nesse botão, o usuário obterá uma janela contendo todos os elementos que compõem sua página, podendo copiar, mover, e remover elementos.
	Desloca o elemento que se encontra selecionado uma posição acima ou à esquerda de onde ele se encontra atualmente. A propriedade `Sequência` será decrementada de 1.
	Desloca o elemento que se encontra selecionado uma posição abaixo ou à direita de onde ele se encontra atualmente. A propriedade `Sequência` será incrementada de 1.
	Exibe a página já processada pelo WIEngine.
	Gera a lógica de Pós-Página para a realização de inserções ou alterações baseada nas definições do(s) Objeto(s) de Pré-Página. Para cada Objeto de Pré-Página (formulário) serão criado 4 (quatro) elementos no Pós-Página: dois Updates, um Objeto e um Gravar.
	Recarrega as definições de Pré-Página a fim de inserir novos elementos na página. As alterações que foram feitas no layout da página não serão perdidas apenas serão inseridos os campos e/ou formulários (Objetos) novos que o WIzard encontre.
	Salva as posições e os dimensionamentos das janelas do WIzard.
	Abre a página que está selecionada na combo-box.
	Copia o elemento que está selecionado na janela de layout da página para a página que está selecionada na combo-box.
	Salva as propriedades do elementos de página.
	Este botão localiza-se no canto superior esquerdo da janela de definições da página. Ele abre a janela que contém os componentes padrão do WebIntegrator e os definidos pelo usuário.

Os elementos listados na *combo-box* localizada acima da barra de botões são aqueles que estão inseridos na página e podem representar um formulário HTML, um determinado tipo de campo de formulário HTML (TEXT, CHECKBOX, TEXTAREA, SELECT, etc.), a tag HEAD de um documento HTML ou um tipo genérico onde o desenvolvedor poderá criar links, inserir imagens ou colocar qualquer outra tag HTML. Para saber mais sobre o tipo `Genérico` clique aqui.

Como explicado anteriormente, para cada Objeto encontrado no Pré-Página será criado um formulário na página. Esse formulário aparecerá listado na *combo-box* identificado pelo mesmo identificador do Objeto a que ele se refere seguido de **FRM**. Para saber maiores detalhes sobre as propriedades de um formulário clique aqui.

O elemento que representa a tag HEAD está representado pela opção <**HEAD**> na *combo-box* servindo para que o desenvolvedor possa inserir informações sobre o documento. Os browsers não exibem as informações que se encontram dentro desta tag. Para saber maiores detalhes sobre as propriedades da tag HEAD clique aqui.

Já os elementos que representam tipos de campo de formulário HTML apresentam algumas características comuns sendo divididos em duas partes: o título e o campo. As propriedades relacionadas com o título são iguais para todos os componentes, já as propriedades relacionadas

ao campos há algumas que são específicas a depender do tipo escolhido.

Logo abaixo seguem relacionadas as propriedades comuns a todos os elementos que representam campos de formulário (**Campo**, **Grupo** e **Checkbox**).

O campo `Nome` indica o nome da variável do contexto do WebIntegrator representada por esse elemento. Em se tratando de um campo de formulário esse nome faz referência a uma coluna do Objeto representado pelo formulário.

O campo `Tipo` indica o tipo de elemento que está selecionado. Clicando no `link` se abrirá a janela de edição de `Elementos Definido pelo Usuário` para o elemento que estiver selecionado, caso o elemento não seja do tipo USR. Um elemento pode ser um dos 5 tipos listados a seguir:

- Genérico (GEN): indica um elemento que a depender das propriedades que estejam setadas poderá se comportar como um botão, um link, uma imagem, um texto com a possibilidade de conter tags HTML ou até mesmo uma combinação desses.
- Campo (TXT): indica um campo de entrada de dados de um formulário onde o usuário poderá digitar algum valor, exceção feita aos campos HIDDEN ou aqueles que se encontrem desabilitados e/ou com permissão de apenas leitura.
- Checkbox (CHK): indica um campo de formulário HTML do tipo CHECKBOX, este tipo de campo normalmente é usado para atributos *`booleanos`* simples onde um campo deve estar escolhido, ou não. Também pode ser utilizado para atributos que possam levar múltiplos valores ao mesmo tempo. O último é representado por um número de `CheckBoxes` com o mesmo nome.
- Grupo (GRP): indica um elemento que representa um grupo de valores donde apenas um poderá ser escolhido. Esse tipo de elemento se divide em mais dois tipos: **Combo** e **Radio** que representam respectivamente uma *`combo-box`* e um *`radio group`*.
- Form (FRM): indica um elemento que representa um formulário HTML. Normalmente será dentro desse tipo de elemento que elementos do tipo **Checkbox**, **Grupo** e **Campo** ficarão inseridos.
- Botão (BTN): indica um elemento que representa um botão.
- Elementos Definido pelo Usuário (USR): são simbolizados por # ou $ mais o nome do elemento.

O campo `Quebra Linha` indica se há uma quebra de linha logo após o elemento.

O campo `Sequência` indica a sequência que esse elemento possui na página. Ao se alterar o valor desse campo também se altera a posição (sequência) que o elemento irá aparecer na página. O desenvolvedor também poderá alterar a posição (sequência) do elemento na página usando os botões  e  que, respectivamente, deslocam-no uma posição acima ou abaixo.

O campo `Classe` indica o nome da classe CSS que será usada para aplicar um estilo visual a todo o elemento (título + campo). A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na *`combo-box`* o elemento **<HEAD>** e no campo `Link CSS` coloque o nome da arquivo desejado. Clicando em  serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo `Estilo CSS` indica o estilo visual que será aplicado a todo o elemento. Aqui o desenvolvedor especifica diretamente os códigos *`style-sheet`* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
background: blue;
```

O campo `Texto` indica o texto que será exibido como rótulo do elemento.

O campo `Classe` indica o nome da classe CSS que será usada para aplicar um estilo visual ao texto do elemento. A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na *combo-box* o elemento **<HEAD>** e na propriedade `Link CSS` indique o caminho relativo ao servidor web do arquivo CSS desejado. Clicando em ... serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo `Estilo CSS` indica o estilo visual que será aplicado ao campo do elemento. Aqui o desenvolvedor especifica diretamente os código *style-sheet* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
   background: blue;
```

O campo `Complemento` serve para que o desenvolvedor possa inserir atributos complementares aos elementos inseridos na página e que não esteja relacionado entre as propriedades disponíveis.

Já as propriedades relacionadas ao campo podem sofrer variações a depender do tipo escolhido. Para saber as propriedades relacionadas ao campo de cada um dos elemento que representam campos de formulário clique no links desejado que segue abaixo:

- Campo
- Checkbox
- Grupo
- Botão
- Check-Box
- Formulário
- Genérico
- HEAD
- Componente do Usuário

1.5.1 Funcionamento do WIzard

Detalhes do funcionamento do WIzard

O principal objetivo do WIzard é ajudar o desenvolvedor na criação de páginas WSP baseada

nas definições de elementos do tipo **Objetos** encontradas no Pré-Página. No momento da montagem da página o WIzard seguirá um modelo de página definido internamente por ele caso o projeto não ofereça nenhuma, caso o projeto use um modelo para a criação das páginas leia a nota que segue abaixo.

NOTA

A fim de obter um correto funcionamento do WIzard no caso onde um projeto esteja usando uma página que sirva como modelo para a criação das páginas WSP os seguintes requisitos tem que ser atendidos:

- Caso o modelo já possua uma `tag` **<HEAD>** faça referência à variável **|head.content|** dentro dessa tag. Querendo, o desenvolvedor também poderá deixar a cargo do WIzard a montagem dos elementos da tag **<HEAD>** desde que para isso ele faça referência à variável **|head|** logo após a tag **<HTML>**, neste caso o página-modelo não poderá possuir explicitamente a tag **<HEAD>**.

- O atributo **onLoad** da tag **<BODY>** deverá chamar a função **`init()`**. Esta função JavaScript encontra-se implicitamente definida no elemento **<HEAD>** da página quando gerada pelo WIzard.

- Dentro da tag **<BODY>** faça referência à variável **|body.content|**, será no local onde foi inserida essa variável que o WIzard irá inserir o(s) formulário(s) que corresponde(m) ao(s) Objeto(s) encontrado(s) no Pré-Página.

Logo abaixo segue um exemplo de código-fonte para uma página que servirá de modelo.

```
<HTML>
<HEAD>
|head.content|
</HEAD>
<BODY onLoad="init()">
<P ALIGN="center">MEU CABEÇALHO</P>
|body.content|
<P ALIGN="center">MEU RODAPÉ</P>
</BODY>
</HTML>
```

A primeira coisa a ser feita pelo WIzard quando o desenvolvedor solicita a sua ajuda será percorrer todos os elementos do Pré-Página procurando por **Objetos**, para cada elemento do tipo Objeto que for encontrado o WIzard irá inserir um formulário na página WSP e para cada coluna referenciada por um Objeto será criado um campo de entrada no formulário específico.

Como exemplo suponha que num Pré-Página haja um único Objeto identificado por **`tmp.obj`** com a seguinte instrução SQL: `SELECT col1, col2, col3 FROM tab`. Neste caso o WIzard iria criar apenas um formulário pois no Pré-Página só há um Objeto e nesse formulário seriam inseridos três campos de entrada que correspondem às três colunas (`col1, col2 e col3`) referenciadas pelo Objeto.

Ao montar a página o WIzard tentará descobrir se as colunas referenciadas pelo Objeto já possuem algum tamanho definido no banco de dados, assim quando os campos forem criados

algumas propriedades como limite e tamanho do campo serão automaticamanete preenchidas. Usando essas mesmas informações, o WIzard irá dispor os campos para que os mesmos ocupem o máximo espaço da página.

Quando na instrução SQL de um elemento do tipo Objeto existir a cláusula `WHERE`, o WIzard irá considerá-las colunas-chaves portanto não serão permitidas alterações em seus valores através do formulário gerado. Os valores com os quais as colunas citadas no `WHERE` são comparados irão aparecer no formulário como campos do tipo **`HIDDEN`** caso possuam o prefixo **`tmp`**. Já as colunas que foram citadas na cláusula `WHERE` e no `SELECT` irão aparecer na página como um *`label`*, ou seja, apenas o nome da coluna seguido de seu valor.

Como exemplo suponha que num Pré-Página haja um único Objeto identificado por **`tmp.obj`** com a seguinte instrução SQL: `SELECT col1, col2, col3 FROM tab WHERE col1 = |tmp.val1|`. Neste caso o WIzard iria criar apenas um formulário pois no Pré-Página só há um Objeto e nesse formulário seriam inseridos dois campos de entrada que correspondem às colunas `col2` e `col3`, um campo do tipo `HIDDEN` correspondente ao valor com o qual está sendo comparada a coluna `col1` e um *`label`* para apenas visualizar o valor da coluna `col1`.

Esta "restrição" imposta pelo WIzard é para garantir o correto funcionamento no momento da geração da lógica de Pós-Página.

IMPORTANTE:

A lógica dos elementos Update gerada para o Pós-Página pelo WIzard considera as colunas declaradas na cláusula WHERE do Objeto correspondente como sendo colunas que tem seus valores gerados automaticamente pelo banco de dados (valor auto-incremento, campo calculado, etc.) mas isso não impede que o desenvolvedor depois faça alterações nos elementos gerados com o intuito de alterar a sua lógica.

A lógica de Pós-Página só é gerada sob pedido do desenvolvedor quando se clica no ícone da janela de Propriedades porém para a lógica ser gerada o Objeto, a partir do qual está se construindo o Pós-Página, deverá possuir em sua instrução `SQL` as cláusulas `WHERE` e `FROM` e não pode usar mais de uma tabela.

Para cada Objeto do Pré-Página são criado 5 (cinco) elementos de Pós-Página: três Updates, um Objeto e um Gravar. Um dos Updates é responsável pela lógica de inserir os dados digitados no formulário no banco de dados através de um comando `INSERT`, outro Update serve para remover quando selecionada a opção excluir e o outro Update fica com a lógica para atualizar os dados através de um comando `UPDATE`. Os elementos Objeto e Gravar são utilizados quando no Objeto do Pré-Página a cláusula `WHERE` faz referência a alguma variável com prefixo **`tmp`**, o Objeto fica encarregado de retornar a(s) coluna(s) que é(são) considerada(s) chave(s) e o Gravar irá atribuir esses valores retornados pelo Objeto às respectivas variáveis temporárias.

Apagando o formulário correspondente a um Objeto os elementos de Pós-Página associadas a ele também serão removidos desde que o desenvolvedor não os tenha alterado.

1.5.2 Botão (BTN)

Botão (BTN)

Um elemento do tipo `Botão` define um *push button* onde o desenvolvedor poderá colocar conteúdos como textos ou imagens e ainda indicar qual a ação a ser executada. Esse elemento possui suas propriedades dividas em três sessões: `Texto`, `Imagem` e `Botão`. Para saber sobre as propriedades relacionada à sessão `Texto` clique aqui, são essas propriedades que irão definir a aparência de um texto do botão. Já as propriedades da **sessão `Botão`** lista aquelas relacionadas com a ação a ser executada e o estilo da visualização do botão, logo abaixo seguem as propriedades:

O campo `Valor` indica o valor que estará associado a esse botão.

O campo `Tipo` indica o tipo de botão que será criado. Há três tipos disponíveis:

- **button**: define um botão cuja ação a ser executada estará definida em um dos eventos disponíveis. Para uma lista dos possíveis eventos a serem configurados acesse a sessão Eventos da lista de propriedades de um botão.
- **submit**: define um botão cuja ação a ser executada será o envio dos dados do formulário ao qual o botão está associado.
- **reset**: define um botão cuja ação a ser executada será o retorno dos valores iniciais dos campos do formulário ao qual o botão está associado.

O campo `Classe` indica o nome da classe CSS que será usada para aplicar um estilo visual ao link. A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na combo-box o elemento **<HEAD>** e na propriedade `Link CSS` indique o caminho relativo ao servidor web do arquivo CSS desejado. Clicando em ... serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo `Estilo CSS` indica o estilo visual que será aplicado ao campo do elemento. Aqui o desenvolvedor especifica diretamente os códigos *style-sheet* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
background: blue;
```

O campo `Complemento` serve para que o desenvolvedor possa inserir atributos complementares ao botão que está sendo definido e que não estejam relacionados entre as propriedades disponíveis.

A **sessão `Imagem`** define as propriedades caso se queira inserir uma imagem como conteúdo do botão. As propriedades são:

O campo `URL` indica a URL onde se encontra a imagem a ser exibida.

O campo `Legenda` indica qual será a legenda que estará associada à imagem. O texto aqui definido irá aparecer quando o ponteiro do mouse ficar sobre a imagem.

O campo `À direita` indica se a imagem será colocada à direita do texto. Vale salientar que o texto no qual aqui se fala corresponde ao que foi digitado na sessão `Texto` e não ao que foi especificado na propriedade `Legenda`.

O campo `Complemento` serve para que o desenvolvedor possa inserir atributos complementares à imagem que está sendo definida e que não estejam relacionados entre as propriedades disponíveis.

A **sessão `Eventos`** serve para que o desenvolvedor insira os eventos que serão tratados por esse elemento. Para incluir basta selecionar um dos eventos que aparece listado na *`combo-box`* e em seguida definir a função ou o comando JavaScript a ser executado para o evento selecionado. Clicando em ... serão listadas as funções JavaScript que estão definidas nos arquivos JS que foram definidos para essa página.

Para saber maiores detalhes como definir os arquivos JS a serem utilizados por uma página acesse a ajuda sobre o elemento **<HEAD>** cliando aqui.

1.5.3 Campo (TXT)

Campo (TXT)

Um elemento do tipo `Campo`, assim como todos elementos que representam campos de formulário, é representado por um texto e o campo propriamente dito. Cada elemento `Campo` possui um sub-tipo que pode ser:

- Texto
- Senha
- Número
- Data
- Oculto
- Arquivo

Vale lembrar que além das propriedades listadas abaixo há as propriedades relacionadas ao texto do elemento. Para saber sobre elas clique aqui.

O campo `Tipo` indica o sub-tipo que um elemento do tipo `Campo` representa. Há as seguintes possibilidades:

- **Texto**: indica que nesse campo pode ser digitado qualquer valor alfa-numérico.
- **Senha**: indica que nesse campo pode ser digitado qualquer valor alfa-numérico mas o caracter de escape sempre será o asterisco (*). Campos desse tipo não têm os seus valores exibidos. Número: indica que nesse campo pode ser digitado apenas valores númericos. Pode-se aplicar uma máscara para formatação.
- **Data**: indica que nesse campo pode ser digitado apenas valores correspondentes a data/hora. Pode-se aplicar uma máscara para formatação.
- **Oculto**: indica campo cujo nome e valor não ficam visíveis ao usuário mas que ainda assim são enviados pelo formulário. Esse tipo de campo apenas possui as propriedades **Tipo** e **Valor**.
- **Arquivo**: indica campo que aceita a inclusão de arquivos juntamente com outras informações do formulário. Quando um formulário contém um campo desse tipo a propriedade Enctype do elemento que representa o formulário (FRM) deve estar configurada com multipart/form-data.

O campo `Valor` indica o valor que estará setado para a variável representada por esse elemento.

NOTA

Os campos do **`tipo Senha`** desconsideram o conteúdo que é digitado nessa propriedade. Já o **`tipo Arquivo`** não possui essa propriedade.

O campo `Só leitura` indica se esse campo de formulário será apenas de leitura.

O campo `Desabilitado` indica se esse campo de formulário ficará desabilitado.

O campo `Obrigatório` indica se esse campo de formulário é de preenchimento obrigatório.

O campo `Limite` indica a quantidade máxima de caracteres que esse campo de formulário aceita. Caso a quantidade de caracteres supere o limite especificado, os caracteres excedentes serão ignorados.

O campo `Tamanho` indica o tamanho em quantidade de caracteres que esse campo de formulário ocupará no formulário.

O campo `Linhas` (apenas disponível para campos **`tipo Texto`**) indica a quantidade de linhas que esse elemento terá. Ao especificar uma quantidade maior do que 1 será criada uma área-de-texto (TEXTAREA).

O campo `Máscara` (apenas disponível para campos **`tipo Data`** ou **`Número`**) indica a máscara de entrada que será aplicada ao valor atribuído a esse campo. O padrão da máscara para campos `tipo Número` é o mesmo adotado pela função NumberFormat e para campos `tipo Data` o mesmo padrão adotado pela DateFormat. Para saber mais consulte as respectivas documentações das funções.

O campo `Mínimo` (apenas disponível para campos **`tipo Número`**) indica o valor mínimo a ser digitado nesse campo. Propriedade apenas disponível para o sub-tipo Número.

O campo Máximo (apenas disponível para campos **tipo Número**) indica o valor máximo a ser digitado nesse campo. Propriedade apenas disponível para o sub-tipo Número.

O campo Classe indica o nome da classe CSS que será usada para aplicar um estilo visual ao link. A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na combo-box o elemento <HEAD> e na propriedade Link CSS indique o caminho relativo ao servidor web do arquivo CSS desejado. Clicando em ... serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo Estilo CSS indica o estilo visual que será aplicado ao campo do elemento. Aqui o desenvolvedor especifica diretamente os código *style-sheet* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
background: blue;
```

1.5.4 Grupo (GRP)

Grupo (GRP)

Um elemento do tipo Checkbox, assim como todos elementos que representam campos de formulário, é representado por um texto e o campo propriamente dito. O uso de elementos desse tipo é recomendado quando se quer oferecer ao usuário apenas duas opções onde geralmente uma é a negação ou o inverso da outra, exemplo: verdadeiro ou falso, habilitado ou desabilitado, sim ou não, etc.

Vale lembrar que além das propriedades listadas abaixo há as propriedades relacionadas ao texto do elemento. Para saber sobre elas clique aqui.

O campo Valor indica o valor representado por esse elemento. Se o valor aqui especificado corresponder ao valor definido em Valor Verd. o *check-box* virá marcado.

O campo Só leitura indica se esse campo de formulário será apenas de leitura.

O campo Desabilitado indica se esse campo de formulário ficará desabilitado.

O campo Obrigatório indica se esse campo de formulário é de preenchimento obrigatório.

O campo Valor Verd. indica qual o valor a ser considerado como verdadeiro, ou seja, caso o valor especificado na propriedade Valor seja igual ao valor definido aqui o *check-box* virá marcado.

O campo Valor Falso indica qual o valor a ser considerado como falso, ou seja, caso o valor especificado na propriedade Valor seja igual ao valor definido aqui o *check-box* virá desmarcado.

O campo Classe indica o nome da classe CSS que será usada para aplicar um estilo visual ao link. A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na *combo-box* o elemento <HEAD> e na propriedade Link CSS indique o caminho relativo ao servidor web do arquivo CSS desejado. Clicando em ... serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo Estilo CSS indica o estilo visual que será aplicado ao campo do elemento. Aqui o desenvolvedor especifica diretamente os código *style-sheet* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
background: blue;
```

1.5.5 Check-Box (CHK)

Check-Box (CHK)

Um elemento do tipo Checkbox, assim como todos elementos que representam campos de formulário, é representado por um texto e o campo propriamente dito. O uso de elementos desse tipo é recomendado quando se quer oferecer ao usuário apenas duas opções onde geralmente uma é a negação ou o inverso da outra, exemplo: verdadeiro ou falso, habilitado ou desabilitado, sim ou não, etc.

Vale lembrar que além das propriedades listadas abaixo há as propriedades relacionadas ao texto do elemento. Para saber sobre elas clique aqui.

O campo Valor indica o valor representado por esse elemento. Se o valor aqui especificado corresponder ao valor definido em Valor Verd. o *check-box* virá marcado.

O campo Só leitura indica se esse campo de formulário será apenas de leitura.

O campo Desabilitado indica se esse campo de formulário ficará desabilitado.

O campo Obrigatório indica se esse campo de formulário é de preenchimento obrigatório.

O campo Valor Verd. indica qual o valor a ser considerado como verdadeiro, ou seja, caso o valor especificado na propriedade Valor seja igual ao valor definido aqui o *check-box* virá

marcado.

O campo `Valor Falso` indica qual o valor a ser considerado como falso, ou seja, caso o valor especificado na propriedade `Valor` seja igual ao valor definido aqui o *check-box* virá desmarcado.

O campo `Classe` indica o nome da classe CSS que será usada para aplicar um estilo visual ao link. A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na *combo-box* o elemento <HEAD> e na propriedade `Link CSS` indique o caminho relativo ao servidor web do arquivo CSS desejado. Clicando em ... serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo `Estilo CSS` indica o estilo visual que será aplicado ao campo do elemento. Aqui o desenvolvedor especifica diretamente os código *style-sheet* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
background: blue;
```

1.5.6 Formulário (FRM)

Formulário (FRM)

Elemento do tipo `FRM` representa um formulário HTML que está inserido na página. Um formulário serve como contêiner de elementos, ou seja, é uma área que pode conter elementos de formulário. Elementos de formulário são elementos que permitem ao usuário entrar informações em um formulário.

Um formulário inserido numa página pelo WIzard aparece delimitado por uma caixa a qual pode ter associada um título. Visualmente o resultado é algo semelhante com o que se vê abaixo:

<Titulo>
Os campos do formulário serão inseridos aqui!

Lembre-se que um formulário gerado pelo WIzard representa um Objeto que foi definido no Pré-Página.

O campo `Nome` define o nome pelo qual esse formulário será identificado. O valor dessa propriedade já se encontra preenchido com o prefixo do Objeto a que este formulário se refere.

IMPORTANTE:
Não é recomendado que o desenvolvedor altere o valor dessa propriedade pois ele será usado internamente pelo WIzard.

O campo `Sequência` indica a sequência (posição) que esse elemento possui na página. Ao se alterar o valor desse campo também se altera a posição (sequência) que o elemento irá aparecer na página. O desenvolvedor também poderá alterar a posição (sequência) do elemento na página usando os botões  e  que, respectivamente, deslocam-no uma posição acima ou abaixo.

O campo `Ação` define qual a página que será acessada após os dados do formulário terem sido enviados ao servidor.

O campo `Método` indica qual será o método HTTP utilizado para a transmissão dos dados do formulário. Com o método GET os dados do formulário são enviadas como uma requisição com *?<dados-do-formulario>* adicionados ao fim da URL. Caso você não especifique um método este é o default.

Se algum dos dados do formulário contém caracteres que não sejam ASCII ou que possuam mais de 100 caracteres você deve usar o método POST, assim os dados são enviados no corpo da requisição. Esse método é o mais recomendado ser usado.

O campo `Enctype` especifica o tipo MIME usado para codificar os dados do formulário. Caso seja deixado em branco assume como valor *`default`* *`application/x-www-form-urlencoded`*. Se existir um elemento `Campo` do **`tipo Arquivo`** (`INPUT TYPE="file"`) essa propriedade deve estar definida com a codificação *`multipart/form-data`*.

O campo `Target` indica onde (em que janela do browser) deverá ser aberto o documento especificado no campo `Ação` (`atributo ACTION` do FORM). Usando **`_blank`** o documento será aberto em uma nova janela. Usando **`_self`** se abrirá o documento no mesmo *`frame`* onde foi clicado. Usando **`_parent`** fará com que o documento seja aberto no *`frameset`* pai. Usando **`_top`** o documento será aberto em toda a janela do *`browser`* disconsiderando a existência de algum *`frameset`*.

O campo `OnSubmit` serve para indicar qual a função JavaScript que será executada quando for solicitado o envio dos dados do formulário.

O campo `OnReset` serve para indicar qual a função JavaScript que será executada quando for solicitado que seja efetuado um reset nos campos do formulário.

O campo `Classe` indica o nome da classe CSS que será usada para aplicar um estilo visual ao formulário. A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na *`combo-box`* o elemento <HEAD> e na propriedade `Link CSS` indique o caminho relativo ao servidor web do arquivo CSS desejado. Clicando em  serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo `Estilo CSS` indica o estilo visual que será aplicado ao campo do elemento. Aqui o desenvolvedor especifica diretamente os código *`style-sheet`* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
background: blue;
```

O campo `Complemento` serve para que o desenvolvedor possa inserir atributos complementares ao formulário que está sendo definido e que não estejam relacionados entre as propriedades disponíveis.

A **`sessão Moldura`** relaciona as propriedades da moldura que delimita o formulário. Seus campos são:

O campo `Ativar` indica se a moldura será visualizada.

O campo `Título` define o título do formulário representado por esse elemento que será exibido na página.

O campo `Classe` indica o nome da classe CSS que será usada para aplicar um estilo visual à moldura do formulário. A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na *`combo-box`* o elemento <HEAD> e na propriedade `Link CSS` indique o caminho relativo ao servidor web do arquivo CSS desejado. Clicando em ... serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo `Estilo CSS` indica o estilo visual que será aplicado ao campo do elemento. Aqui o desenvolvedor especifica diretamente os código *`style-sheet`* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
background: blue;
```

1.5.7 Genérico (GEN)

Genérico (GEN)

O elemento do tipo `Genérico` é utilizado quando o desenvolvedor quiser inserir um link, uma imagem, um conjunto de tags HTML ou até mesmo uma combinação desses elementos. Vale lembrar que além das propriedades listadas abaixo há as propriedades relacionadas ao elemento. Para saber sobre elas clique aqui.

As propriedades abaixo referem-se à sessão Texto, é nesta sessão onde o desenvolvedor poderá definir *`tags`* HTML que serão inseridas na página ou definir o texto do link.

O campo `Texto` serve para o desenvolvedor inserir outras *`tags`* HTML que não estejam disponíveis diretamente pelo WIzard. O valor aqui digitado em conjunto com as definições de

propriedades do Link criam um link na página.

O campo `Incluir <BR>` indica se os símbolos de CARRIAGE RETURN (`\r`) e LINE FEED (`\n`) serão substituídos por
.

A sessão Link serve para que o desenvolvedor juntamente com o que foi definido na sessão Texto crie links, vejamos as propriedades:

O campo `URL` indica para qual recurso *web* apontará esse link. Por recurso *web* entenda que possa ser uma página HTML, uma imagem, um arquivo de som, etc.

O campo `Target` indica onde (em que janela do browser) deverá ser aberto o documento especificado no campo URL (atributo HREF da tag A). Usando **_blank** o documento será aberto em uma nova janela. Usando **_self** se abrirá o documento no mesmo frame onde foi clicado. Usando **_parent** fará com que o documento seja aberto no frameset pai. Usando **_top** o documento será aberto em toda a janela do *browser* disconsiderando a existência de algum *frameset*.

O campo `Classe` indica o nome da classe CSS que será usada para aplicar um estilo visual ao link. A classe CSS deve estar especificada no arquivo que serviu como modelo para a página ou deve estar em um arquivo CSS a ser incluído na página. Para incluir um arquivo CSS numa página gerada pelo WIzard selecione na *combo-box* o elemento <HEAD> e na propriedade Link CSS indique o caminho relativo ao servidor web do arquivo CSS desejado. Clicando em ... serão listadas as classes CSS que estão definidas nos arquivos CSS que foram definidos para essa página.

O campo `Estilo CSS` indica o estilo visual que será aplicado ao link. Aqui o desenvolvedor especifica diretamente os código *style-sheet* a serem aplicados ao elemento, exemplo:

```
font: italic 12px Verdana, Geneva, Arial, Helvetica, sans-serif;
background: blue;
```

A sessão Imagem serve para que um arquivo de imagem seja inserido na página. As propriedades dessa sessão combinada com as definições da sessão Link produzem um link simbolizado por uma imagem. Vejamos as propriedades:

O campo `URL` indica a URL onde se encontra a imagem a ser exibida.

O campo `Legenda` indica qual será a legenda que estará associada à imagem. O texto aqui definido irá aparecer quando o ponteiro do mouse ficar sobre a imagem.

O campo `À direita` indica se a imagem será colocada à direita do texto. Vale salientar que o texto no qual aqui se fala corresponde ao que foi digitado na sessão Texto e não ao que foi especificado na propriedade `Legenda`.

O campo `Complemento` serve para que o desenvolvedor possa inserir atributos complementares à imagem que está sendo definida e que não estejam relacionados entre as propriedades disponíveis.

1.5.8 HEAD

HEAD

A opção <HEAD> representa o conteúdo da tag HEAD da linguagem HTML a qual pode conter informações sobre o documento. Para isso outras *tags* são inseridas tais como <link>, <script>, <title>, etc. Com este elemento o desenvolvedor bastará definir o valor para as *tags* mais usadas mas com a liberdade de inserir outras.

O campo Título define qual será o título da página. Esta propriedade corresponde à *tag* <TITLE> de um documento HTML.

O campo Link CSS define quais são os arquivos de estilo CSS (*Cascade Style Sheet*) que são utilizados pela página. Ao definir o arquivo a ser usado coloque-o com o seu caminho relativo ao servidor web e caso queira usar mais de um arquivo separe-os por vírgula. Por *default*, esse campo já vem preenchido com **/|wi.proj.id|/page.css** que é o arquivo de estilos CSS que o WIzard usa no momento da criação das páginas. Esta propriedade corresponde à *tag* <LINK> com o atributo REL="stylesheet".

As classes que se encontram definidas nos arquivos CSS definidos aqui poderão ser listadas clicando no botão ... que aparece ao lado da propriedade Classe que alguns elementos possuem.

O campo Link JS define quais são os arquivos de *script* JS (*JavaScript*) que são utilizados pela página. Ao definir o arquivo a ser usado coloque-o com o seu caminho relativo ao servidor web e caso queira usar mais de um arquivo separe-os por vírgula. Por default, esse campo já vem preenchido com **/wi2/page.js** que é o arquivo de *scripts* JS do WebIntegrator que contém as funções de validação e formatação dos campos do formulário. Esta propriedade corresponde à *tag* <SCRIPT>.

As funções JavaScript que se encontram definidas nos arquivos JS definidos aqui irão ser listadas nos eventos dos campos de formulário bastando clicar no ícone ... que aparece ao lado do nome do evento. Para a função para ser listada, antes de sua declaração deve estar o identificador **//@list** o qual pode ser atribuído qual a assinatura padrão que a função terá. Como exemplo, suponha que o código abaixo seja relativo a um arquivo JavaScript:

```
  /*
   * com a declaração abaixo está sendo informado que a função
myFunction
   * será listada que sua assinatura padrão é myFunction(Alô Web!!).
   */
  //@list=myFunction(Alô Web!!)
  function myFunction(msg) {
    alert(msg);
  }
```

```
/*
   * esta função não será listada porque o identificador //@list não
   * aparece antes da declaração da função.
   */
  function myFunction2() {
    alert("Não sou listada!");
  }

  //@list=myFunction3(this)
  function myFunction3(obj) {
    alert(obj.value);
  }
```

O campo Código é uma área de texto livre onde o desenvolvedor poderá acrescentar outras *tags*. Vale lembrar que as *tags* que forem inseridas nesse campo serão colocadas dentro da *tag* <HEAD>.

1.5.9 Componente do Usuário (USR)

Componente do Usuário (USR)

Além dos elementos de páginas disponíveis pelo WIzard o desenvolvedor poderá criar seus próprios elementos personalizados a fim de que possa atender questões específicas. Um elemento definido pelo usuário (USR) consiste de um arquivo-modelo contendo o trecho de código HTML específico que representa o elemento. Esse código-fonte (HTML) poderá fazer referência a variáveis de sessão do WebIntegrator delimitadas por pipes (|)

Além dessas variáveis o desenvolvedor poderá criar variáveis do ambiente do WIzard. Variáveis do ambiente do WIzard são aquelas que começam com o prefixo **wiz.** e que cujos nomes servirão para a montagem da tela de propriedades do elemento onde poderão ter seus valores definidos através do ambiente de desenvolvimento do WIzard.

Para criar um elemento basta o desenvolvedor clicar no link da propriedade Tipo que se terá acesso à tela de edição. A tela é composta dos seguintes campos:

O campo Código é o local onde o desenvolvedor irá entrar com o respectivo código-fonte do elemento ou alterar o de algum já existente.

O campo Campo serve para que o desenvolvedor escolha um dos elementos já atualmente criados ou se escolha a opção [Novo...] para que seja criado um novo elemento.

O campo Nome indica o nome do elemento que está sendo editado/criado.

O campo Global indica se o elemento que está sendo criado será visível a todos os projetos do WebIntegrator ou apenas a esse projeto onde ele está sendo criado. Os elementos visíveis

globalmente possuem o prefixo **$**, já aqueles visíveis unicamente a um projeto possuem o prefixo #.

Para ver um exemplo detalhado sobre a construção de um elemento personalizado para o WIzard clique aqui.

1.6 Ferramentas

1.6.1 Importar Elementos

Importar Elementos

Esta parte destina-se ao desenvolvedor que queira importar elementos (páginas, grids, etc.) de um outro projeto para o projeto que esteja o uso. Durante a importação desses elementos o desenvolvedor poderá escolher se as referências e definições de servidores e/ou banco de dados serão sobrescitas.

O campo Projeto Origem serve para que o desenvolvedor selecione de qual projeto se quer importar os elementos.

O campo Páginas serve para que o desenvolvedor especifique os identificadores das páginas que devem ser importadas para o projeto em uso. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de páginas, exemplo, caso se queira importar todas as páginas que estão no diretório teste bastaria digitar teste/*.

O campo Combos indica quais componentes do tipo Combo devem ser importados para o projeto em uso. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de combos por exemplo, caso se queira importar todas as combos do projeto bastaria digitar *.

O campo Downloads serve para que o desenvolvedor especifique os identificadores dos downloads que devem ser importados para o projeto em uso. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de downloads, exemplo, caso se queira importar todos os downloads que começem com down bastaria digitar down*.

O campo Uploads serve para que o desenvolvedor especifique os identificadores dos uploads que devem ser importados para o projeto em uso. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de uploads, exemplo, caso se queira importar todos os uploads que começem com upl bastaria digitar upl*.

O campo Eventos serve para que o desenvolvedor especifique os identificadores dos eventos que devem ser importados para o projeto em uso. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de eventos, por exemplo caso se queira importar todos os eventos que começem com evt bastaria digitar evt*.

O campo Grids serve para que o desenvolvedor especifique os identificadores dos grids que devem ser importados para o projeto em uso. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de grids, exemplo, caso se queira importar todos os grids do projeto bastaria digitar *.

O campo Outros Arquivos indica quais os arquivos que estão no diretório do projeto que serão importados. Use caracteres coringas para montar a máscara.

O campo Relatórios serve para que o desenvolvedor especifique os identificadores dos relatórios que devem ser importados para o projeto em uso. O uso de máscara é permitido

oferecendo assim uma maneira mais fácil de importar um conjunto de relatórios, exemplo, caso se queira importar todos os relatórios que começem com rel bastaria digitar rel*.

O campo Sobrescrever BDs indica se as definições de banco de dados que venham a coincidir entre o projeto de origem e o projeto em uso devem ser sobrescritas.

O campo Sobrescrever Referências indica se as referências a elementos que venham a coincidir entre uma página importada e uma já existente devem ser sobrescritas.

O campo Sobrescrever Servidores indica se as definições de servidores que venham a coincidir entre o projeto de origem e o projeto em uso devem ser sobrescritas.

1.6.2 Revisão

Revisão

Esta parte destina-se ao desenvolvedor que queira gerar ou aplicar revisões (patch) de um outro projeto para o projeto que esteja o uso. Durante a geração da revisão o desenvolvedor poderá escolher se as referências e definições de servidores e/ou banco de dados serão sobrescitas.

O campo Páginas serve para que o desenvolvedor especifique os identificadores das páginas que devem ser incluídos na revisão. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de incluir um conjunto de páginas, exemplo, caso se queira importar todas as páginas que estão no diretório teste bastaria digitar teste/*.

O campo Combos indica quais componentes do tipo Combo devem ser incluídos na revisão. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de combos por exemplo, caso se queira incluir todas as combos do projeto bastaria digitar *.

O campo Downloads serve para que o desenvolvedor especifique os identificadores dos downloads que devem ser incluídos na revisão. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de downloads, exemplo, caso se queira incluir todos os downloads que começem com down bastaria digitar down*.

O campo Uploads serve para que o desenvolvedor especifique os identificadores dos uploads que devem ser importados para o projeto em uso. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de uploads, exemplo, caso se queira importar todos os uploads que começem com upl bastaria digitar upl*.

O campo Eventos serve para que o desenvolvedor especifique os identificadores dos eventos que devem ser incluídos na revisão. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de importar um conjunto de eventos, por exemplo caso se queira incluir todos os eventos que começem com evt bastaria digitar evt*.

O campo Grids serve para que o desenvolvedor especifique os identificadores dos grids que devem ser incluídos na revisão. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de incluir um conjunto de grids, exemplo, caso se queira importar todos os grids do projeto bastaria digitar *.

O campo Outros Arquivos indica quais os arquivos que estão no diretório do projeto que devem ser incluídos na revisão. Use caracteres coringas para montar a máscara.

O campo Relatórios serve para que o desenvolvedor especifique os identificadores dos relatórios que devem ser incluídos na revisão. O uso de máscara é permitido oferecendo assim uma maneira mais fácil de incluir um conjunto de relatórios, exemplo, caso se queira importar todos os relatórios que começem com rel bastaria digitar rel*.

O campo Incluir BDs indica se as definições de banco de dados devem ser incluídas.

O campo Incluir Referências indica se as referências a elementos devem ser incluídas.

O campo Incluir Servidores indica se as definições de servidores devem ser incluídas.

O campo Arquivo da Revisão serve para que o desenvolvedor possa selecionar o arquivo de revisão que deverá ser aplicado ao projeto.

1.6.3 DB Explorer

DB Explorer

Este é o local onde o desenvolvedor poderá digitar instruções SQL, inclusive chamadas a stored procedures, e ver o resultado imediato de sua execução. Aqui é um bom local para poder testar as instruções SQL antes delas serem colocadas no projeto para ver se elas retornam os valores esperados ou atualizam os dados corretamente.
A caixa-de-texto localizada no lado esquerdo é o local onde devem ser digitadas as instruções SQL, o campo do lado direito é o local onde ficam armazenadas as últimas instruções SQL digitadas.
Para ajudar ainda mais nas tarefas de debug das instruções SQL, as mesmas ainda podem conter referências a variáveis delimitadas por pipes que terão seus valores substituídos de acordo com os valores passados pelo desenvolvedor no momento da execução da instrução. Ao perceber que a instrução SQL faz referência a essas variávies é oferecido ao desenvolvedor um formulário com os nomes das variáveis a fim de que ele preencha os campos com os valores a serem testados.

O campo Banco de Dados define qual o banco de dados do projeto que será usado na execução da instrução SQL.

O campo Tipo define o tipo de instrução SQL que será executada. Caso se deseje executar comandos cujos resultados sejam conjuntos de linhas, como comandos SELECT, escolha o tipo

Select ou caso se deseje executar instruções que irão realizar atualizações em tabelas do banco de dados como INSERT, DELETE, etc. escolha o tipo Update.

O campo Opções Especiais exibe algumas opções especiais sobre os diversos tipos de tabelas que compõe o banco de dados escolhido. Para obter a relação das tabelas do banco de dados que são visíveis ao projeto escolha a opção TABLE.

O campo Formato define se o tipo de saída deve ser tabela ou CSV.

1.6.3.1 Relacionamentos

Relacionamentos

Este é o local onde o desenvolvedor informa o relacionamento entre as tabelas do banco de dados. O botão atualizar permite que o WI tente descobrir automaticamente os relacionamentos através das chaves estrangeiras.

O campo Chave Primária define qual tabela e coluna formam a chave primária do relacionamento.

O campo Chave Estrangeira define qual tabela e coluna formam a chave estrangeira do relacionamento.

O campo Tabelas com Relacionamentos informa quais tabelas tem relacionamentos definidos.

O campo Relacionamentos informa quais são os relacionamentos de uma dada tabela.

1.6.3.2 Filtros

Filtros

Este é o local onde o desenvolvedor informa quais filtros serão aplicados na montagem da instrução Sql.

O campo Título define qual será o título da coluna.

O campo Salvar Título define se para a coluna da tabela dada o título definido no campo título deverá ser sempre utilizado.

O campo Ordem define se deve ser aplicado um fator de ordenação na coluna e se ascendente ou descendente.

O campo Ocultar define se a coluna deve ser ocultada na projeção do Sql.

O campo Condição define se deve ser aplicada alguma condição à coluna.

O campo Negar define se a condição deve ser aplicada na forma negativa.

O campo Agregação define se a coluna deve ser agregada e qual operador será usado em sua agregação.

1.7 Administração

Administração

Este é o local onde o administrador faz as definições gerais do ambiente de desenvolvimento do **WI Builder**, como as definições de quais projetos terão os logs ativados, quais os tipos de eventos a serem logados, o idioma a ser usado nas telas do **WI Builder** entre outras opções. Obtenha detalhes das ferramentas de administração clicando nos links abaixo:

- Configuração
- Funções
- Plug-ins
- Usuários
- Permissões
- Mudar senha

1.7.1 Configuração

Definições do WI Builder

Este é o local onde o administrador faz as definições gerais do ambiente de desenvolvimento do **WI Builder** como as definições de quais projetos terão os logs ativados, quais os tipos de eventos a serem logados, o idioma a ser usado nas telas do **WI Builder** entre outras opções.

O campo Builder exibe uma lista dos projetos que estão fazendo logs das ações realizadas pelos usuários no **WI Builder**.

O campo Engine exibe uma lista dos projetos que estão fazendo logs do **WI Engine**.

O campo Comparar ao escrever arquivos jsp serve para informar ao builder que antes de escrever os arquivos jsps ele deve comparar seus conteúdos evitando que um arquivo tenha sua data modificada sem que seu conteúdo tenha sido modificado.

O campo Idioma define qual será o idioma padrão usado nas telas do **WI Builder** caso o idioma preferido pelo usuário não esteja disponível no ambiente do WebIntegrator. O **WI Builder** assume que o idioma preferido do usuário é o idioma padrão que está configurado no browser.

O campo Modo indica qual o modo de log a ser criado. Há o modo **`Completo`** onde qualquer ação que for executada é registrada nos arquivos de logs e o modo **`Simples`** onde apenas as exceções geradas são logadas.

O campo Permissão de rede define quais as redes que poderão ter acesso ao **WI Builder**. Há a possibilidade de uso de máscaras para a definição de uma faixa de redes, para saber mais clique aqui.

O campo Projeto serve para que o desenvolvedor escolha um projeto que terá registro de logs.

O campo SQL exibe uma lista dos projetos que estão fazendo logs da execução de comandos SQL.

1.7.2 Funções

Funções

Este é o local onde o administrador registra funções definidas pelo usuário para serem utilizadas em projetos do WebIntegrator. Funções definidas pelo usuário são classes Java que implementam a interface `br.com.itx.integration.InterfaceFunction`, para saber maiores detalhes sobre a implementação e o uso de funções clique aqui.

Uma função pode ser registrada globalmente, neste caso ela ficará acessível a todos os projetos, ou registrada especificamente em um projeto. Quando uma função for registrada globalmente o desenvolvedor deve se assegurar de que todos os projetos tenham acesso à função, para isso a classe Java ou o arquivo **`.jar`** que representa a função deverá ser instalada num local acessível a todos os projetos ou no repositório de classes interno de cada projeto, `<WEBAPP_PATH>/WEB-INF/lib` se for um arquivo **.jar** ou `<WEBAPP_PATH>/WEB-INF/classes` se for arquivo **`.class`**.

O campo Classe serve para que seja indicado o nome completo da classe Java que representa uma função do WebIntegrator.

O campo Global indica se a classe Java que representa uma função será registrada globalmente. Se esta opção estiver marcada no momento do registro da função, essa função ficará acessível a todos os projetos do WebIntegrator caso contrário ela apenas será acessada no projeto onde estiver sendo feito o registro.

Vale atentar que se a função for registrada globalmente a biblioteca ou classe Java que a representa deverá ser instalada em um local onde todos os projetos do WebIntegrator tenham acesso ou ser copiada para o repositório de classes interno do projeto, `<WEBAPP_PATH>/WEB-INF/lib` se for um arquivo **`.jar`** ou `<WEBAPP_PATH>/WEB-INF/classes` se for arquivo **`.class`**.

O campo Nome indica o nome pelo qual a classe Java que representa uma função será referenciada nos projetos do WebIntegrator.

1.7.3 Plug-ins

Plug-ins

Esta é o local onde o desenvolvedor poderá publicar conectores Java, grids Java ou funções como plug-ins do WebIntegrator. A idéia dos plug-ins é fazer com que as funções já sejam cadastradas automaticamente e que os conectores e grids Java sejam exibidos como componentes de página pelo WI Builder.

Para publicar um conjunto de conectores Java, grids Java ou funções como sendo plug-ins basta empacotá-los em um arquivo com extensão **`.jar`** juntamente com um arquivo **`plugins.xml`** que contém as definições de cada uma das classes Java e publicá-los. Para saber maiores

detalhes sobre a configuração e publicação de plug-ins acesse a seção **Como...** da ajuda do WebIntegrator ou clique aqui.

O processo de publicação de um plug-in faz com que o mesmo seja instalado no projeto atual em que está sendo a publicação ou, se a opção Global estiver marcada, em todos os projetos já criados. Caso queira instalar um plug-in já publicado em um projeto que foi criado posteriormente à publicação de um plug-in, basta que o administrador acesse esse projeto e selecione os plug-ins publicados que devem ser instalados nesse projeto e clicar no botão "**publicar**" para que os mesmos sejam efetivamente instalados no projeto.

O campo Arquivo JAR indica o arquivo JAR contendo os plug-ins a ser instalado.

O campo Global indica se o arquivo de plug-ins a ser instalado ficará disponível para todos os projetos ou apenas ao projeto onde está sendo feita a configuração. Se essa opção for marcada todos os projetos terão acesso aos plug-ins que serão registrados pela arquivo JAR.

1.7.4 Usuários

Usuários

Este é o local onde o administrador poderá controlar as permissões. A princípio, esse usuário não terá permissão a fazer nada pois o administrador ainda terá que definir a qual ou quais projetos ele terá acesso e que tipo de acesso será permitido a ele. Para realizar as configurações de permissões o administrador escolherá a opção Permissões.

O campo Confirmar Senha serve para que se entre com a senha digitada no campo anterior a fim de que a mesma seja confirmada.

O campo Descrição serve para que se entre com uma descrição do novo usuário.

O campo Master indica se o usuário terá poderes de super-usuário. Esse tipo de usuário pode acessar qualquer projeto, definir novos usuários e atribuir permissões a usuários já cadastrados.

O campo Nome para login indica o login do usuário a ser criado.

O campo Senha indica a senha do novo usuário que está sendo criado.

O campo Usuários exibe a lista de usuários já cadastrados no WI Builder. O usuário **admin** não irá aparecer nesta listagem por ser um usuário nativo do sistema do WI Builder.

1.7.5 Configuração do Servidor de Aplicações

Configuração do Servidor de Aplicações

Este é o local onde o administrador do projeto poderá configurar o servidor de aplicações.

O campo Tipo de Servidor indica o tipo de servidor que está sendo utilizado.

O campo Porta indica a porta que está sendo utilizada. Se for deixada vazio o WI tentará descobrir a porta atual em uso.

O campo Usuário indica o usuário de administração do servidor de aplicações (manager do tomcat).

O campo Senha indica a senha do usuário de administração do servidor de aplicações.

1.7.6 Permissões

Permissões

Este é o local onde o administrador do projeto poderá definir as permissões dos usuários cadastrados no WI Builder e os diretórios que poderão ser acessados pelos componentes do WebIntegrator.

O campo Grupo indica os grupos de usuários existentes no WI Builder.

O campo Usuários cadastrados contém uma lista com todos os usuários cadastrados no WI Builder.

O campo Usuários desse projeto contém a lista de usuários com permissão para acessar o projeto e a qual grupo eles pertencem.

Vale notar que um usuário cadastrado pode participar de um projeto fazendo parte de um grupo e em um outro projeto fazer parte de um outro grupo diferente pois as permissões são dadas por projeto e não por usuário.

1.7.7 Mudar Senha

Alterar dados do usuário

Este é o local onde o usuário poderá alterar a sua senha e a sua descrição.

O campo Confirmar senha serve para que se entre com a mesma senha que foi digitada no campo Nova senha a fim de que a mesma seja confirmada.

O campo Descrição indica qual a descrição deste usuário.

O campo Nova senha indica a nova senha que será criada.

O campo Senha atual serve para que o usuário entre com a senha atual que será trocada

1.8 Tutoriais

Tutoriais

Os tutoriais descrevem processos para o desenvolvimento de aplicações web usando: o servidor de aplicações WebIntegrator; gerar índices de busca através do módulo adicional do WebIntegrator, o WISearch, usando uma API baseada na Lucene; usar funções definidas pelo usuário no WebIntegrator; e, gerar imagens de gráficos baseadas em informações que estão armazenadas no WebIntergrator, através do WIChart.

Ao final dos tutoriais o usuário deverá possuir os conhecimentos básicos e iniciais para que ele possa aprimorar e desenvolver seus conhecimentos a medida que usa o WebIntegrator e todos os demais pacotes de recursos oferecidos, separadamente, para auxiliar no desenvolvimento de aplicações profissionais corporativas.

1.8.1 Tutorial do WebIntegrator

Tutorial do WebIntegrator

Clique aqui para acessar o tutorial do WebIntegrator.

1.8.2 Tutorial do WISearch

Tutorial do WISearch (introdução)

O WISearch é um módulo adicional do servidor de aplicações web WebIntegrator que oferece a capacidade de gerar índices a partir de conteúdos textuais onde posteriormente possam ser feitas buscas *full-text*. Esse módulo, composto por um conjunto de classes Java, gerencia um mecanismo de busca e indexação baseado na API Lucene oferecendo ainda a possibilidade de serem gerados índices hierárquicos baseados nas informações dos documentos indexados.

Todo documento, a ser publicado em um índice gerenciado pelo WISearch, além do seu conteúdo que deve ser representado por um arquivo também poderá ter associado propriedades adicionais definidas pelo próprio desenvolvedor como uma maneira de estender as informações disponíveis sobre um documento podendo essas propriedades adicionais também serem pesquisadas.
Basicamente o módulo WISearch é composto de 6 classes Java responsáveis pela geração e manutenção da estrutura dos índices, são elas:

- **br.com.itx.modules.search.WIIndexer**: conector Java responsável pela geração da estrutura inicial do índice.
- **br.com.itx.modules.search.WIPublisher**: conector Java responsável pela publicação de um documento no índice. Por publicação entenda como sendo o ato de indexar o conteúdo de um arquivo e as propriedades adicionais que irão compor um documento.
- **br.com.itx.modules.search.SearchGrid**: *grid* Java responsável pela montagem de um *grid* que servirá para exibir os resultados de uma pesquisa feita num índice.
- **br.com.itx.modules.search.SearchObject**: conector Java que retorna os resultados de uma pesquisa feita num índice numa estrutura de dados semelhante ao elemento Objeto do WebIntegrator.
- **br.com.itx.modules.search.WIHighlighter**: conector Java responsável por efetuar o destaque dos termos no conteúdo de um documento baseado na *query* que foi utilizada

na pesquisa.

- **br.com.itx.modules.search.WIRemover**: conector Java responsável pela remoção de um ou mais documentos da estrutura do índice. Vale notar que essa classe não fará a remoção física do documento no diretório apenas apagará as informações dele constantes no índice.

Para a construção e manutenção de um índice usando o WISearch basta-nos usar essas classes Java, através de definições de conectores e/ou *grids* Java e de algumas variáveis que são usadas internamente por elas. A princípio todos os conectores precisam saber com qual índice ele deverá trabalhar, essa informação deverá ser passada para os conectores através da variável `tmp.indexName`.

1.8.2.1 Criando um Índice

Criando um Índice

Para iniciar a criação de um índice defina alguns pontos anteriormente. Primeiro, todo documento antes de ser indexado ele deverá ter o seu conteúdo, que sempre deve ser representado por um arquivo, publicado em um diretório que chamamos de diretório de publicação. Sugere-se que o diretório de publicação de arquivos fique localizado em uma das pastas internas ao projeto. Segundo, defina quais serão as propriedades adicionais que um documento irá possuir.

Para a criação de um índice o desenvolvedor precisará usar o conector Java `br.com.itx.modules.search.WIIndexer` juntamente com a definição de algumas variáveis que servirão como parâmetros para o conector. Logo abaixo segue a lista de variáveis de ambiente que deverão estar configuradas para que o índice possa ser criado corretamente.

Variável	Descrição
`tmp.indexName`	variável que deverá conter o nome do índice que será criado ou que terá suas configurações lida.
`tmp.properties`	variável que deverá conter a listagem separada por vírgula dos nomes das propriedades adicionais que um documento poderá ter.
`tmp.dirName`	variável que deverá conter o nome do diretório de publicação dos arquivos.
`tmp.mask`	variável que deverá conter as extensões de arquivos que serão processados pelo índice. Se o conteúdo dessa variável for vazio o **WI Search** tentará indexar qualquer arquivo que for publicado.
`tmp.parseHTML`	variável que deverá conter o valor true para habilitar o processamento de conteúdo HTML dos documentos quando publicados.

Para exemplificar a criação de um índice inicie um projeto no WebIntegrator e crie uma página chamada **`configindice.wsp`** e coloque a listagem abaixo como sendo código-fonte da página.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Indexador</TITLE>
<SCRIPT>
function reindexar() {
  with (document.forms[0]) {
    elements["tmp.acao"].value="update";
    submit();
  }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM ACTION="|wi.page.id|.wsp" METHOD="post">
<INPUT TYPE="hidden" NAME="tmp.acao" VALUE="create">
<INPUT TYPE="hidden" NAME="tmp.storeContent" VALUE="">
<INPUT TYPE="hidden" NAME="wi.page.prev" VALUE="|wi.page.id|">
<TABLE BORDER="1" CELLSPACING="0" CELLPADDING="2">
<TR>
<TD>Nome do índice: <INPUT TYPE="text" NAME="tmp.indexName"
VALUE="|tmp.indexName|" SIZE="50"></TD>
</TR>
<TR>
<TD>Propriedades adicionais: <INPUT TYPE="text" NAME="tmp.properties"
VALUE="|tmp.properties|" SIZE="50"></TD>
</TR>
<TD><INPUT TYPE="checkbox" NAME="tmp.parseHTML"> Processar conteúdo
HTML</TD>
</TR>
<TR>
<TH BGCOLOR="#C0C0C0">Fonte de Dados</TH>
</TR>
<TR>
<TD>
<TABLE CELLSPACING="0" CELLPADDING="2">
<TR>
<TD ALIGN="left" NOWRAP><INPUT TYPE="radio" NAME="tmp.dsType"
VALUE="dir" CHECKED> Diretório de Arquivos</TD>
</TR>
<TR>
<TD NOWRAP>Diretório de pesquisa: <INPUT TYPE="text" NAME="tmp.dirName"
VALUE="|tmp.dirName|" SIZE="70"></TD>
</TR>
<TR>
<TD>Extensões de arquivos a serem processados: <INPUT TYPE="text"
NAME="tmp.mask" VALUE="|tmp.mask|"></TD>
</TR>
<TR>
<TD NOWRAP><INPUT TYPE="checkbox" NAME="tmp.recursive"> Pesquisar em
sub-diretórios</TD>
</TR>
</TABLE>
</TD>
</TR>
<TR>
<TD ALIGN="center"><INPUT TYPE="submit" VALUE="Indexar">
<INPUT TYPE="button" VALUE="Reindexar" onClick="reindexar();"</TD>
</TR>
```

```
</TABLE>
</FORM>
</BODY>
</HTML>
```

Para completar a construção da página adicione ao pós-página dela um evento Conector Java com as seguintes configurações:

Campo	Valor
Condição	`true`
Nome da classe	`br.com.itx.modules.search.WIIndexer`

Note que o código-fonte faz referência àquelas variáveis necessárias ao conector `br.com.itx.modules.search.WIIndexer`. O importante é que todo projeto do WebIntegrator que precise de uma esquema de indexação e pesquisa que utilize o WI Search precisará de uma página semelhante a essa para a criação da estrutura inicial do índice.

Para executar essa página clique no ícone Visualizar do WI Builder identificado por uma página com uma lupa (). Como estudo de caso crie um índice onde os documentos publicados terão como propriedades adicionais uma data e um título, para criar a estrutura inicial de um índice com essas características siga as informações da tabela abaixo.

Campo	Descrição
Nome do índice	Informe aqui o nome do índice a ser criado, o restante do tutorial assume que tenha sido definido o nome do índice como **modelo**. Todo índice a ser gerado possui um nome que será usado para referenciar esse índice nas ações de indexação, pesquisa e/ou remoção através da variável `tmp.indexName`. Baseado nesse nome será criado um diretório em **`<projeto>/WEB-INF/index`** onde serão armazenados os arquivos proprietários do índice.
Propriedades adicionais	No WI Search um documento é composto de um conteúdo e de 0 (zero) ou mais propriedades adicionais. Liste aqui estas propriedes separadas por vírgula. Como em nosso caso queremos adicionar uma data (dividida em dia, mês e ano) e um título, as nossas propriedades adicionais serão: ano, mes, dia, titulo. Um detalhe importante é que quando se quer criar índices hierárquicos com o WI Search a ordem em que as propriedades adicionais são declaradas são levadas em consideração onde a primeira tem maior nível hierárquico (precedência) em relação a segunda que tem maior nível hierárquico em relação a terceira e assim sucessavimente.
Diretório de pesquisa	Indique aqui o caminho onde serão publicados os arquivos que terão seus conteúdos indexados. Geralmente esse diretório será o mesmo diretório a ser configurado como o diretório de upload e download do projeto. Para esse caso esse diretório será **`<projeto>/documentos`**.
Processar conteúdo HTML	Marcando essa opção o processamento de conteúdo HTML será ativado para que as tags que façam parte dos arquivos sejam retiradas e apenas o conteúdo seja indexado. Para nosso caso marque essa opção.
Extensões de arquivos a serem processados	Caso se queira que apenas certos tipos de arquivos sejam processados especifique aqui as extensões desses arquivos separadas por vírgula, se for deixado em branco o WI Search tentará indexar qualquer tipo de arquivo. Aqui vamos dizer que apenas arquivos com extensão **`htm`** ou **`html`** sejam processados, para isso coloque essas extensões separadas por vírgula.
Pesquisar em sub-diretórios	Caso esta opção esteja marcada o mecanismo de indexação também irá pesquisar por arquivos que estejam em algum sub-diretório dentro do diretório especificado no campo Diretório de pesquisa.

Após os campos terem sido preenchidos com os valores correspondentes clique no botão **Indexar** para que a estrutura do índice seja criada.

1.8.2.2 Publicando um Documento

Publicando um Documento

A principal finalidade de um índice é justamente poder indexar conteúdos para que depois possam ser pesquisados de maneira rápida e precisa. A seguir iremos criar a página responsável pela publicação dos documentos, as ações que essa página fará serão o upload do arquivo (através do evento Upload do WebIntegrator) e em seguida a publicação do conteúdo desse arquivo no índice (usando um evento Conector Java através da classe `br.com.itx.modules.search.WIPublisher`).

Para a classe `br.com.itx.modules.search.WIPublisher` fazer a publicação de um documento no índice ela precisa saber de antemão o nome do índice que ela irá atualizar e quais são as propriedades adicionais do documento indexado. Para informar o nome do índice atribua à variável `tmp.indexName` o nome de um índice que já foi criado para o projeto. Para a classe saber qual o arquivo a ser publicado atribua à variável `tmp.arquivo` o nome do arquivo que está sendo publicado representando o conteúdo do documento e caso os documentos desse índice possuam propriedades adicionais crie variáveis temporárias do WebIntegrator com o nome das propriedades adicionais definidas para o índice.

Logo abaixo segue uma sugestão para o modelo da página de publicação para esse estudo de caso, atente que nela estão as variáveis `tmp.indexName`, `tmp.arquivo` e as variáveis temporárias com os nomes das propriedades adicionais.

```
<HTML>
<HEAD>
<TITLE>Publicador de Documentos</TITLE>
</HEAD>
<BODY BGCOLOR="#D5EEE2" TEXT="#000000" LINK="#000000" VLINK="#000080">
<DIV ALIGN="center">
<TABLE BORDER="1" CELLSPACING="0" CELLPADDING="5">
  <TH BGCOLOR="#008080">
    <B><FONT FACE="Arial" COLOR="#FFFFFF" SIZE="3">Publicação de
Documento</FONT></B>
  </TH>
  <TR>
    <TD BGCOLOR="#FFFFFF">
    <FORM ACTION="|wi.page.id|.wsp" METHOD="post"
ENCTYPE="multipart/form-data">
      <!-- variavel que indica o nome do indice que sera atualizado -->
      <INPUT TYPE="hidden" NAME="wi.page.prev" VALUE="|wi.page.id|">
      <INPUT TYPE="hidden" NAME="tmp.indexName" VALUE="modelo">
      <P>
      <FONT FACE="Arial"><STRONG><SMALL>Selecione o
arquivo:<BR></SMALL></STRONG></FONT>
      <!-- variavel que indica o documento a ser publicado -->
      <INPUT TYPE="file" NAME="tmp.arquivo">
      </P>
      <P>
      <FONT
FACE="Arial"><STRONG><SMALL>Título:<BR></SMALL></STRONG></FONT>
      <!-- variavel temporaria que indica o valor que sera indexado
           como sendo parte da propriedade adicional titulo. -->
      <INPUT TYPE="text" NAME="tmp.titulo">
      </P>
      <P ALIGN="left">
      <FONT FACE="Arial"><STRONG><SMALL>Data
(dd/mm/aaaa):<BR></SMALL></STRONG></FONT>
```

```
<!-- variavel temporaria que indica o valor que sera indexado
          como sendo parte da propriedade adicional dia. -->
      <INPUT TYPE="text" NAME="tmp.dia" SIZE="2" MAXLENGTH="2"> /
      <!-- variavel temporaria que indica o valor que sera indexado
          como sendo parte da propriedade adicional mes. -->
      <SELECT NAME="tmp.mes" SIZE="1">
        <OPTION VALUE="01">Janeiro</OPTION>
        <OPTION VALUE="02">Fevereiro</OPTION>
        <OPTION VALUE="03">Março</OPTION>
        <OPTION VALUE="04">Abril</OPTION>
        <OPTION VALUE="05">Maio</OPTION>
        <OPTION VALUE="06">Junho</OPTION>
        <OPTION VALUE="07">Julho</OPTION>
        <OPTION VALUE="08">Agosto</OPTION>
        <OPTION VALUE="09">Setembro</OPTION>
        <OPTION VALUE="10">Outubro</OPTION>
        <OPTION VALUE="11">Novembro</OPTION>
        <OPTION VALUE="12">Dezembro</OPTION>
      </SELECT> /
      <!-- variavel temporaria que indica o valor que sera indexado
          como sendo parte da propriedade adicional ano. -->
      <INPUT TYPE="text" NAME="tmp.ano" SIZE="4" MAXLENGTH="4">
      </P>
      <P>
      <INPUT TYPE="submit" VALUE="Publicar" NAME="tmp.publicar">
      </P>
    </FORM>
    </TD>
  </TR>
</TABLE>
</DIV>
</BODY>
</HTML>
```

Com a página pronta, vamos configurar os eventos Upload e Conector Java. Para criar o evento Upload clique no menu Projeto - Uploads e realize as seguintes configurações em um Upload do tipo `Local`:

Campo	Valor
Identificador	**`UplDoc`**
Diretório	**`|wi.proj.path|/documentos`**. Esse diretório é o local onde será feito o upload do arquivo para que o seu conteúdo seja indexado.
Nome do campo do formulário	**`tmp.arquivo`**
Arquivo	**`|tmp.arquivo|`**

No pós-página da página de publicação coloque este evento Upload recém-criado preenchendo os campos com os seguintes valores:

Campo	Valor
Condição	**`|tmp.publicar| = publicar && |tmp.arquivo| !=`**
Upload	**`UplDoc`**

Também no pré-página coloque agora o evento Conector Java preenchendo os campos com os seguintes valores:

Campo	Valor
Condição	`|tmp.publicar| = publicar && |wi.upl.ok| = true && |tmp.arquivo| !=`
Nome da classe	`br.com.itx.modules.search.WIPublisher`

Com estes passos já criamos uma página de publicação de documentos do nosso índice. Para testar você pode começar a publicar alguns documentos. Para verificar se a publicação está ocorrendo com sucesso o desenvolvedor pode fazer referência à variável **`tmp.publisher.status`** que retornará o nome do arquivo que foi publicado precedido de `OK;`, caso tenha ocorrido algum erro durante a publicação o *`stack trace`* será armazenado na variável **`tmp.publisher.error`** para que o desenvolvedor possa rastrear o que ocorreu.

Se a publicação do documento no índice for efetuada com sucesso a variável **`tmp.publisher.iddoc`** conterá o `ID` sob o qual o documento foi indxado, esse `ID` sempre será único para cada um dos documentos publicados.

1.8.2.3 Pesquisando no Índice

Pesquisando no Índice

Para realizar a busca nos documentos indexados vamos agora criar a página que realiza a busca e a que exibirá os possíveis resultados dessa pesquisa. Crie a página de pesquisa com um formulário HTML e que contenha obrigatoriamente as seguintes variáveis:

Variável	Descição
`tmp.indexName`	indica o nome do índice que será usado para realizar a pesquisa.
`tmp.search.query`	contém a *`query`* de pesquisa. O mecanismo de busca adotado pelo WI Search suporta uma série de recursos tais como pesquisas booleanas, pesquisas por pré-fixos, construção de *`queries`* com uso de parêntesis, etc.

O processador de *`queries`* que vem embutido no WI Search suporta a maioria dos padrões de *`queries`* adotados pelos mecanismos de buscas mais conhecidos, para conhecer as possíveis *`queries`* que podem ser montadas clique aqui. Como exemplo:

free AND "text search" Busca por documentos contendo "free" e a expressão "text search"

+text search Busca por documentos contendo "text" e que preferencialmente

	também contenha "search"
giants -football	Busca por "giants" mas omite os documentos contendo "football"
author:gosling java	Busca por documentos contendo "gosling" na propriedade adicional author e java no conteúdo do documento

Logo abaixo segue uma sugestão para o código-fonte da página de pesquisa.

```
<HTML>
<HEAD>
<TITLE>Formulário de Pesquisa</TITLE>
<META HTTP-EQUIV=PRAGMA CONTENT=NO-CACHE>
<META HTTP-EQUIV=EXPIRES CONTENT=0>
</HEAD>
<BODY BGCOLOR="#D5EEE2">
<FORM ACTION="pesquisa.wsp" METHOD="post">
<INPUT TYPE="hidden" NAME="tmp.indexName" VALUE="modelo">
Expressão:<BR>
<INPUT TYPE="text" NAME="tmp.search.query" size='40'
value="|tmp.search.query|">
<input type="checkbox" value="AND" name="tmp.search.queryOperator"
  |$if(|tmp.search.queryOperator|=AND, checked,)$| />
Com todas as palavras
<br>
<INPUT TYPE="submit" NAME="tmp.procurar" VALUE="Procurar">
<INPUT TYPE="reset" VALUE="Limpar Campos">
</FORM>
|$if(|tmp.procurar|=Procurar,
<P STYLE="FONT: 8pt Verdana\,Arial;">|tmp.resposta|</P>
|grid.resultadoPesquisa|
|grid.resultadoPesquisa.link|
,)$|

</BODY>
</HTML>
```

Note que na página acima aparece um grid Java com os resultados da pesquisa, a classe `br.com.itx.modules.search.SearchGrid` é a responsável pela montagem desse grid Java. Primeiramente vamos criar um grid HTML do tipo JAVA, para isso clique no link Projeto / Grid, escolha a opção Grids HTML e preencha os seguintes valores:

Campo	Valor
Identificador	resultadoPesquisa
Tipo	JAVA

O resultado de uma pesquisa usando a classe `br.com.itx.modules.search.SearchGrid` retorna além de todas as propriedades adicionais definidas para os documentos do índice algumas outras propriedades que são incorporadas automaticamente em cada documento durante a publicação, a saber:

Propriedade	Descrição
iddoc	retorna o identificador único do documento associado durante a publicação.
path	retorna o caminho absoluto do arquivo no servidor.
relativePath	retorna o caminho do diretório do arquivo relativo ao repositório do índice.
absolutePath	retorna o caminho absoluto (completo) do diretório do arquivo no servidor.
fileName	retorna o nome do arquivo.
title	retorna o conteúdo tag TITLE de um arquivo HTML.

Além da classe `br.com.itx.modules.search.SearchGrid` o WISearch também oferece ao desenvolvedor a classe `br.com.itx.modules.search.SearchObjcet` para realizar pesquisas no índice. A diferença é que através da classe `SearchObject` o resultado da pesquisa é estruturado na forma de um objeto do WebIntegrator onde o identificador do objeto é configurado através da variável `tmp.prefix`, o restante do funcionamento é semelhante, inclusive as variáveis disponíveis.

Logo abaixo, segue um exemplo de como poderia ser o modelo do grid a ser adotado para exibir o resultado da pesquisa:

```
<TABLE BORDER="1" CELLSPACING="0" CELLPADDING="3" WIDTH="100%">
<TR>
<TH BGCOLOR="#c0c0c0">Título do documento</TH>
<TH BGCOLOR="#c0c0c0">Data</TH>
</TR>
<TR>
<TD><A HREF="visualizar.wsp?tmp.arquivo=|name|">|titulo|
(|filename|)</A> </TD>
<TD ALIGN="center">|dia|/|mes|/|ano|</TD>
</TR>
<TR>
<TD COLSPAN="2" ALIGN="center">Nenhum documento encontrado!</TD>
</TR>
</TABLE>
```

Definido o grid Java, no pré-página da página de pesquisa coloque um evento Grid Java com os seguintes campos preenchidos:

Campo	Valor
Condição	true
Nome da classe	br.com.itx.modules.search.SearchGrid
Grid	resultadoPesquisa

Para exibir maiores informações sobre a pesquisa feita, vamos criar uma variável que informará a *query* pesquisada, o tempo de duração de pesquisa e a quantidade de documentos que foi encontrada. Para isso, adicione um elemento Gravar com as seguintes definições:

Campo	Valor
Condição	|tmp.search.query| !=
Objetos	tmp.resposta
Se condição verdadeira	A pesquisa por <B>|tmp.search.query|</B> demorou <B>|tmp.search.time|</B> ms para encontrar <B>|tmp.search.count|</B> documento(s)!

O resultado de uma pesquisa feita pelo WISearch, por default, vem ordenado de acordo com a taxa de acerto (*hit*) baseada na *query* que foi realizada a pesquisa mas o desenvolvedor pode alterar esse comportamento para que o resultado venha ordenado por uma das propriedades do documento. Para isso atribua à variável **tmp.searchResult.orderBy** o nome da propriedade cujos valores serão ordenados, caso queira que os valores venham em ordem inversa (do maior para o menor) atribua à variável **tmp.searchResult.reverse** o valor `true`. A variável tmp.searchResult.reverse apenas funciona em conjunto com a variável tmp.searchResult.orderBy e seu valor default é `false`.

No modelo do grid existe um link para uma página chamada *visualizar.wsp*, página essa que pode ser usada para exibir o conteúdo do arquivo. Para isso defina um evento Download do tipo Local com a seguinte configuração:

Campo	Valor
Identificador	downDoc
Diretório	|wi.proj.path|/documentos
Arquivo	|tmp.arquivo| (perceba que no link que chama a página visualizar.wsp a variável tmp.arquivo é passada com o nome do arquivo)

Após isso crie a página *visualizar.wsp* e em seu pré-página coloque o download recém-configurado. Com isso tente fazer algumas pesquisas no índice, a seguir veremos como criar uma página de remoção de documentos do índice.

1.8.2.4 Removendo um Documento

Removendo em Documento

A próxima página que criaremos será responsável pela exclusão das informações dos documentos do índice e do arquivo a depender da escolha feita pelo usuário. Para criarmos a página de exclusão de documentos vamos exibir um *grid* Java com as informações dos documentos que estão disponíveis no índice onde o usuário pode escolher através de *check-*

boxes quais documentos a serem excluídos. Para criação desse *grid* faça os passos semelhantes à criação do *grid* **resultadoPesquisa** nomeando-o como **removerDoc**, como modelo do *grid* use o código HTML que segue abaixo.

```
<table border="1" cellspacing="0" cellpadding="3" width="100%">
<tr>
<th bgcolor="#c0c0c0">Excluir</th>
<th bgcolor="#c0c0c0">Título do documento</th>
<th bgcolor="#c0c0c0">Arquivo</th>
<th bgcolor="#c0c0c0">Data</th>
</tr>
<tr>
<td>
<input type="checkbox" name="id|id|" value="|iddoc|">
<input type="hidden" name="file|id|" value="|name|"
</td>
<td>|title|</td>
<td><a href="visualizar.wsp?tmp.arquivo=|name|">|name|</a></td>
<td>|dia|/|mes|/|ano|</td>
</tr>
<tr>
<td colspan="4" align="center">Nenhum documento encontrado!</td>
</tr>
<tr>
<td colspan="4" align="center"><INPUT TYPE="submit" VALUE="Excluir"
NAME="tmp.excluir"></td>
</tr>
</table>
```

O código do modelo do *grid* listado acima é bem semelhante ao do *grid* **resultadoPesquisa** com alguns detalhes como a criação dos *check-boxes* e o uso das variáveis `id` e `iddoc`. A variável **`id`** representa o sequencial gerado pelo *grid*, enquanto a variável **`iddoc`** indica o sequencial único de um documento do índice. Os valores dessas variáveis serão úteis quando formos informar quais documentos queremos que sejam excluídos. Os campos `HIDDEN` contém o nome do arquivo respectivo ao seu `iddoc`.

A página que irá excluir os documentos do índice que foram escolhidos pelo usuário além do *grid* Java definido acima também fará referência à classe `br.com.itx.modules.search.WIRemover` através de um conector Java, essa é a classe responsável pela remoção dos documentos do índice e ao elemento Remover Arquivo que irá excluir fisicamente os arquivos do diretório de publicação.

Lembre-se que para a montagem do *grid* **removerDoc**, no pré-página da página que irá fazer a remoção de documentos deve ser colocado um elemento *grid* Java fazendo referência à classe `br.com.itx.modules.search.SearchGrid` com a seguinte configuração:

Campo	Valor
Condição	**`true`**
Nome da classe	**`br.com.itx.modules.search.SearchGrid`**
Grid	**`removerDoc`**

No pós-página, através de um conector Java, faça referência à classe `br.com.itx.modules.search.WIRemover`. Para que essa classe saiba quais são os documentos a serem removidos do índice as variáveis **`tmp.idList`** e **`tmp.indexName`** tem que

estar configuradas respectivamente com uma lista dos `iddocs` dos documentos a serem removidos separados por vírgula e o nome do índice que terá excluídos seus documentos. A configuração do pós-página ficaria assim:

Campo	Valor
Condição	**`|tmp.excluir| = excluir && |tmp.idList| !=`**
Nome da classe	**`br.com.itx.modules.search.WIRemover`**

Para finalizar a configuração do pós-página adicione o elemento **Remover Arquivo** que será responsável pela exclusão física dos arquivos selecionados. A configuração ficaria assim:

Campo	Valor
Condição	**`|tmp.excluir| = excluir && |tmp.fileList| !=`**
Máscara	**`|tmp.fileList|`**
Diretório	**`|wi.proj.path|/documentos`**

Para finalizar a construção da página de remoção de documentos do índice logo abaixo segue uma sugestão do código-fonte. Perceba que através do formulário `HTML` da página estão sendo passadas tanto a variável `tmp.indexName` com o nome do índice e a variável `tmp.idList` com a lista de `iddods` dos documentos a serem excluídos que é montada através de uma rotina JavaScript (`doList()`).

```
<HTML><HEAD>
<TITLE>Remoção de documentos</TITLE>
<META HTTP-EQUIV=pragma CONTENT=no-cache>
<META HTTP-EQUIV=expires CONTENT=0>
<SCRIPT>
/**
 * Monta a lista com o(s) IDDoc(s) do(s) documento(s) separados
 * por virgula que serao removidos do indice.
 */
function doList() {
    // variavel q contera a lista de IDDocs dos documentos a serem
removidos
    idList = "";
    // variavel q contera a lista dos nomes de arquivos a serem
removidos
    fileList = "";
    limit = 0;
    if ("|tmp.search.count|" != "") {
        limit = |tmp.search.count|;
    }
    frm = document.forms[0];
    for (i = 0; i < limit; i++) {
        id = "id" + i;
        file = "file" + i;
            if (frm.elements[id].checked) {
            idList += frm.elements[id].value + ",";
            fileList += frm.elements[file].value + ",";
        }
    }
    idList = idList.slice(0, idList.length - 1);
    fileList = fileList.slice(0, fileList.length - 1);
    frm.elements["tmp.idList"].value = idList;
```

```
frm.elements["tmp.fileList"].value = fileList;
    return (list != "");
}
</SCRIPT>
</HEAD>
<BODY BGCOLOR="#ffffff">
<FORM ACTION="|wi.page.name|.wsp" METHOD="post" ONSUBMIT="doList()">
<!--- garante a execução do pós-página que fará a exclusão dos
documentos -->
<INPUT TYPE="hidden" NAME="wi.page.prev" VALUE="|wi.page.id|">

<!--- nome do índice donde serão removidos os documentos -->
<INPUT TYPE="hidden" NAME="tmp.indexName" VALUE="modelo">

<!--- lista de IDDocs dos documentos a serem excluídos -->
<INPUT TYPE="hidden" NAME="tmp.idList" VALUE="">
<!--- lista de nome dos arquivos a serem excluídos -->
<INPUT TYPE="hidden" NAME="tmp.fileList" VALUE="">

<INPUT TYPE="text" NAME="tmp.search.query" VALUE="|tmp.search.query|">
<INPUT TYPE="button" VALUE="Pesquisar" onClick="javascript:submit()">
<!--- listagem dos documentos disponíveis no índice de acordo com
      a query que foi digitada -->
|grid.removerDoc|
</FORM>
</BODY>
</HTML>
```

1.8.2.5 Exibindo a Estrutura Hierárquica do Índice

Exibindo a Estrutura Hierárquica do Índice

Com os passos descritos anteriormente já está disponível um simples porém funcional aplicativo de indexação e busca de documentos. A seguir veremos como exibir uma visão hierárquica em uma estrutura em árvore das informações dos documentos disponíveis no índice.

O índice hierárquico será montado baseado nas propriedades adicionais que foram definidas para o índice. A hierarquia exibida será de acordo com a ordem em que as propriedades adicionais foram listadas na hora da definição do índice. Em nossa aplicação, como definimos as propriedades adicionais **ano**, **mes**, **dia** e **titulo**, então a estrutura a ser gerada se assemelhará com a que segue:

Inicialmente vamos construir um *frameset*, composto de duas colunas onde uma dessas colunas terá duas linhas. Um dos *frames* exibirá um componente **TreeView** que representará a visão hierárquica do índice, um outro *frame* exibirá um formulário de pesquisa e um terceiro *frame* que exibirá o resultado das pesquisas. Para isso crie a página **framepesq.wsp** e coloque o seguinte código:

```
<HTML>
<HEAD>
<TITLE>Índice Hierárquico</TITLE>
<META HTTP-EQUIV="pragma" CONTENT="no-cache">
<META HTTP-EQUIV="expires" CONTENT="0">
</HEAD>
<!-- frames -->
<FRAMESET COLS="23%,*" >
    <FRAME NAME="treeview" SRC="treeview.wsp" MARGINWIDTH="10"
MARGINHEIGHT="10" SCROLLING="auto" FRAMEBORDER="0" NORESIZE>
    <FRAMESET ROWS="21%,*">
        <FRAME NAME="search" SRC="pesquisa.wsp" MARGINWIDTH="10"
MARGINHEIGHT="10" SCROLLING="auto" FRAMEBORDER="0" NORESIZE>
        <FRAME NAME="searchMain" SRC="resultado.wsp" MARGINWIDTH="10"
MARGINHEIGHT="10" SCROLLING="auto" FRAMEBORDER="0">
    </FRAMESET>
</FRAMESET>
</HTML>
```

Note que o código acima faz referência a uma página chamada **treeview.wsp**, será ela que exibirá a estrutura em árvore das informações do índice. Primeiro crie-a e adicione dois eventos em seu pré-página:

Evento	Descrição
Gravar	gravar na variável `tmp.indexName` o nome do índice a partir do qual montará a árvore que no nosso caso é **modelo**.
Conector Java	chamar a classe `br.com.itx.modules.search.WIHierarchicalIndex`

No código-fonte da página que será exibido a estrutura hierárquica do índice tem que ser levado algumas considerações. Primeiro, para que o **TreeView** seja montado e visualizado, o arquivo `treeview.js` terá de ser carregado inserindo a seguinte *tag*:

```
<SCRIPT TYPE="text/javascript" SRC="/wi3/treeview.js"></SCRIPT>
```

Além da tag acima, acrescente outra listando as propriedades do **TreeView**:

```
<SCRIPT>
/*****************************
 * Propriedades do TreeView *
```

```
**************************/
// identificador único para esse TreeView (não altere essa opção)
UniqueID = "JS_TreeView_docu";

// diretório raiz dos links (não altere essa opção)
DocRoot = "";

// diretório das "peles" do TreeView. As opções disponíveis são:
// /wi3/images/grafix/win/
// /wi3/images/grafix/mac/
// /wi3/images/grafix/hlp/
// /wi3/images/grafix/os2/
ImgRoot = "/wi3/images/grafix/win/";

// página que contém o FRAMESET que chama o índice
FrameSet = "";

// largura da imagem (não altere essa opção)
ImgWidth = 14;

// altura da imagem (não altere essa opção)
ImgHeight = 18;

EntryHeight = ImgHeight;

// chave inicial (não altere essa opção)
InitialKey = "";

// cor de fundo da página
CurrPageBG = "#000099";

// cor de frente da página
CurrPageFG = "#FFFFFF";

LinkCurrPage = true;

// texto a ser apresentado como dica do nó raiz do índice
TreeRootHint = "";

// texto a ser apresentado como dica de um link para uma página
NormalPageHint = "";

// texto a ser apresentado como dica de um link
LinkedPageHint = "";

// texto a ser apresentado como dica da figura que fecha um ramo da
árvore
OpenBookHint = "Fechar";

// texto a ser apresentado como dica da figura que abre um ramo da
árvore
ClosedBookHint = "Abrir";

OpenBookStatus = "Fechar sub-lista";

ClosedBookStatus = "Abrir sub-lista";

// mensagem que aparece na barra de status do browser
window.defaultStatus = "Árvore dos arquivos do índice";
```

```
// mensagem a ser exibida quando o índice não fizer parte de um FRAMESET
// e a propriedade checkFrames estiver setada para true
navExplain = "\nThis page normally belongs inside a navigation" + "
frame.\n\nIs it OK to reload the page as designed ?";

// fontes a serem usadas pelo índice
FontFace = "Verdana,Times New Roman,Times,serif";

compactTree = false;

viewMatchCnt = 0;

// indica se visualiza apenas um ramo da árvore
singleBranch = false;

// indica se será feita a verificação se o índice faz parte de um
FRAMESET
checkFrames = false;

//baseHref="_self";

/**
 * método que monta a query da pesquisa de acordo com o link.
 */
function sendQuery(key) {
   var flds = fields.split("^");
   var keys = key.split("^");
   var query = "";
   for (var i = 0; i < flds.length; i++) {
      if (keys[i]) {
         sep = (i == 0) ? "" : " AND ";
         query += sep + flds[i] + ":\"" + keys[i] + ";
      }
   }
   if (top.frames["search"]) {
      top.frames["search"].document.forms[0].target = "searchMain";
      top.frames["search"].document.forms[0]["tmp.search.query"].value =
query;
      top.frames["search"].document.forms[0].submit();
   }
}
</SCRIPT>
```

Por fim, coloque dentro da *tag* <BODY> a referência à variável |`tmp.search.tree`| que é criada pela classe `br.com.itx.modules.search.WIHierarchicalIndex` e que retorna a estrutura em árvore das propriedades dos documentos que fazem parte do índice.

Para mostrar mais uma funcionalidade do WI Search, acrescente agora o recurso de destaque de termos no texto de acordo com o critério de pesquisa. Para isso, na página **`visualizar.wsp`** desative o evento **Download** e acrescente os eventos de **Importar Arquivo** e **Concetor Java**.

Para configurar o evento **Importar Arquivo** faça as seguintes definições:

Campo	Valor
Objeto	`content`
Condição verdadeira	`|wi.proj.path|/documentos/|tmp.arquivo|`

Com a configuração, o conteúdo do arquivo que antes era feito o *download* agora está armazenado na variável `content`. O próximo passo será a configuração da classe Java que irá recuperar o conteúdo dessa variável e destacar os termos. Para isso, adicione um Conector Java com as seguintes configurações:

Campo	Valor
Condição	`|queryDestaque| !=`
Nome da classe	`br.com.itx.modules.search.Highlighter`

Por fim substitua o conteúdo da página visualizar.wsp pela referência à variável `content`, com isso sempre que essa página for solicitada o seu conteúdo corresponderá ao conteúdo do arquivo passado pela variável `tmp.arquivo` com os termos em destaque.

1.8.3 Tutorial de Funções

Tutorial de Funções

Uma função definida pelo usuário no WebIntegrator é uma classe Java que implementa a interface `br.com.itx.integration.InterfaceFunction` e que deve estar devidamente registrada pelo WI Builder.

A interface `br.com.itx.integration.InterfaceFunction` declara apenas o método `execute(Hash wiMap, String[] args)` cujo argumento contém os parâmetros que foram passados para a função. O código-fonte da interface segue abaixo:

```
package br.com.itx.integration;

public interface InterfaceFunction {
    public String execute(WIMap wiMap, String[] args);
}
```

Logo abaixo segue um exemplo de implementação de uma função que retornará o valor passado como parâmetro, caso nenhum ou mais de um parâmetro seja passado a função retornará uma mensagem informando que foi passada uma quantidade incorreta de parâmetros.

```
import br.com.itx.integration.InterfaceFunction;

public class WIMensagem implements InterfaceFunction {
```

```
private static final String MSG = "Oops! Quantidade incorreta de
parâmetros.";

    public String execute(WIMap wiMap, String[] args) {
        return (args.length == 0 || args.length > 1) ? WIMensagem.MSG :
args[0];
    }
}
```

Após compilar a classe Java que implementa a lógica de sua função coloque o arquivo compilado (arquivo com a extensão **`.class`**) no repositório de classes específico de um projeto ou no repositório de classes comum ao *`web container`* que você estiver utilizando. Se você criou a sua função dentro de um pacote (`package`) lembre-se de criar a estrutura de diretórios correspondente.

As funções que forem instaladas no repositório de classes de um projeto apenas poderão ser utilizada no projeto específico onde foi instalada. Já aquelas que forem instaladas no repositório comum ao *`web container`* ficarão disponíveis a todos os projetos.

Para registrar a função acesse o WI Builder e escolha a opção  do menu **Opções** e indique o nome completo da classe e o nome pelo qual ela será chamada dentro dos projetos do WebIntegrator. Para registrar a função acima poderíamos usar os seguintes dados:

Nome:	**`msg`**
Classe:	**`WIMensagem`**

Agora você poderá executar a função `msg` implementada pela classe **WIMensagem** dentro do WebIntegrator bastando digitar `|$msg("<mensagem>")$|`. Sempre que você quiser usar uma função a sintaxe é:

```
|$<nome-da-função>([<parâmetro-#1>,...])$|
```

Caso um dos parâmetro a ser passado para a função contenha vírgula em seu conteúdo use a barra invertida (\) antes do sinal de vírgula para informar ao WI Engine que não se trata da vírgula como separadora de parâmetros e dessa forma evitando que a função receba parâmetros inesperados ou incorretos. Por exemplo, caso quiséssemos passar a mensagem "1, 2, 3, `testando...`" para a função **`msg`** a sintaxe seria:

```
|$msg(1\, 2\, 3\, testando...)$|
```

Junto com a distribuição do WebIntegrator seguem algumas funções já implementadas, são elas:

dateformat	Função para formatação de valores data.
decodeDES	Função que decodifica uma sequência de caracteres usando o algoritmo DES.
encodeDES	Função que codifica uma sequência de caracteres usando o algoritmo DES.
eval	Função que realiza a avaliação de uma expressão numérica.
fileCopy	Função que realiza a cópia de um arquivo.
htmlFilter	Função que remove de um texto as tags HTML nele contidas.
if	Função que realiza um teste de condição e retorna um valor a depender do resultado da avaliação.
md5	Função que retorna o hash de um valor usando o algoritmo MD5.
numberformat	Função para formatação de valores numéricos.
piece	Função que retorna uma substring baseada em um delimitador.
random	Função para geração de números aleatórios.

textformat	Função para formatação de valores texto.
wi.context	Função que realiza um dump no contexto do WebIntegrator.

1.8.3.1 dateformat

dateformat

A função **DateFormat** aceita uma série de parâmetros para realizar formatações em valores datas/horas e está registrada internamente pelo WebIntegrator pelo nome `df`, a sua sintaxe é a seguinte:

```
|$df(<data>, <tipo-de-formatação>, [<parâmetros-de-formatação>])$|
```

A depender do tipo de formatação escolhido pelo desenvolvedor a função poderá usar nenhum, um ou vários parâmetros de formatação.

Para especificar uma máscara para um valor DATA/HORA usa-se uma string que especifique este padrão. Neste padrão, todos os caracteres ASCII são reservados como caracteres de padrão, os quais são definidos a seguir:

Símbolo	Significado	Apresentação	Exemplo
G	designador de era	(Texto)	AD
y	ano	(Número)	1996
M	mês do ano	(Texto & Número)	Julho & 07
d	dia do mês	(Número)	10
h	hora em am/pm (1~12)	(Número)	12
H	hora do dia (0~23)	(Número)	0
m	minutos da hora	(Número)	30
s	segundos do minuto	(Número)	55
S	milisegundos	(Número)	978
E	dia da semana	(Texto)	Quinta-feira
D	dia do ano	(Número)	189
F	dia da semana do mês	(Número)	2 (segunda semana)
w	semana do ano	(Número)	27
W	semana do mês	(Número)	2
a	marcador am/pm	(Texto)	PM
k	hora do dia (1~24)	(Número)	24
K	hora em am/pm (0~11)	(Número)	0
z	Time Zone	(Texto)	Pacific Standard Time
	delimitador de texto	(Delimitador)	
	aspas simples	(Literal)	

A quantidade de um determinado caractere de padrão irá determinar qual será o seu formato.

(Texto): quando for especificado 4 ou mais caracteres de padrão de um determinado símbolo

será usada a forma extendida, menos de 4 caracteres será usada a forma curta ou abreviada se existir alguma.

(Número): o número mínimo de dígitos. Números menores são preenchidos com zero para essa quantidade. O ano é manipulado especialmente; isto é, se a quantidade de y for 2, o ano será truncado para 2 dígitos.

(Texto & Número): três ou mais, use texto, senão use o formato numérico.

Em seguida, há alguns exemplos mostrando como usar esses padrões e quais os resultados.

```
Máscara                              Resultado
--------------                       ---------
yyyy.MM.dd G às hh:mm:ss z      ->>  1996.07.10 AD às 15:08:56 PDT
EEE, d de MMM d, yy             ->>  Qua, 10 de Jul, 96
h:mm a                          ->>  12:08 PM
hh horas a, zzzz                ->>  12 horas PM, Pacific Daylight
Time
"K:mm a, z"                     ->>  0:00 PM, PST
"yyyyy.MMMMM.dd GGG hh:mm aaa"  ->>  1996.Julho.10 AD 12:08 PM
```

Logo abaixo segue uma lista com os tipos de formatações possíveis e seus respectivos parâmetros de formatação.

TIPOS DE FORMATAÇÕES

EXT

Escreve por extenso uma data que esteja no padrão ANSI (yyyy-MM-dd).

Sintaxe:

```
|$df(<data>, EXT)$|
```

Exemplos:

```
|$df(2001-12-20, EXT)$|
```

FMT

Formata uma data para um outro padrão. O desenvolvedor indica qual o formato em que a data se encontra e qual o novo formato que ela assumirá.

Sintaxe:

```
|$df(<data>, FMT, <mascaraAtual>, <mascaraNova>)$|
```

Exemplos:

```
|$df(2001-12-20, FMT, yyyy-MM-dd, dd/MM/yyyy)$|
```

INC

Adiciona uma quantidade especificada de dias à data passada no parâmetro. Tanto a data especificada quanto o resultado estará no formato ANSI (yyyy-MM-dd). Caso o desenvolvedor queira adicionar à data algo diferente a dias, ele poderá utilizar um parâmetro adicional informando a que padrão refere-se o número do incremento.

O padrão do incremento é um daqueles símbolos que a função **DateFormat** suporta, ou seja, M

para meses, d para dias, etc. Se esse parâmetro não for passado a função assume que o número do incremento refere-se a dias.

Sintaxe:

```
|$df(<data>, INC, <número>[, <padrão>])$|
```

Exemplos:

```
|$df(2001-12-20, INC, 10)$|
|$df(2001-10-20, INC, 2, M)$|
```

DEC

Decrementa uma quantidade especificada de dias à data passada no parâmetro. Tanto a data especificada quanto o resultado estarão no formato ANSI (yyyy-MM-dd). Caso o desenvolvedor queira decrementar da data algo diferente a dias, ele poderá utilizar um parâmetro adicional informando a que padrão refere-se o número do decremento.

O padrão do decremento é um daqueles símbolos que a função DateFormat suporta, ou seja, M para meses, d para dias, etc. Se esse parâmetro não for passado a função assume que o número do decremento refere-se a dias.

Sintaxe:

```
|$df(<data>, DEC, <número>[, <padrão>])$|
```

Exemplos:

```
|$df(2001-12-20, DEC, 10)$|
|$df(2001-10-20, DEC, 2, M)$|
```

SUB

Retorna a diferença em dias entre duas datas. As datas passadas como parâmetro deverão estar no formato ANSI (yyyy-MM-dd).

Sintaxe:

```
|$df(<data-mais-nova>, SUB, <data-mais-antiga>)$|
```

Exemplos:

```
|$df(2001-12-20, SUB, 2000-12-20)$|
```

WDAY

Retorna o dia da semana. O primeiro dia da semana é o domingo (1) e o último é o sábado(7).

Sintaxe:

```
|$df(<data>, WDAY, <mascaraAtual>)$|
```

Exemplos:

```
|$df(2001-12-20, WDAY, yyyy-MM-dd)$|
```

1.8.3.2 decodeDES

decodeDES

A função **decodeDES** realiza a decodificação de uma sequência de caracteres usando o algoritmo DES, a sua sintaxe é:

```
|$decodeDES(<sequência-de-caracteres>, <chave-DES>)$|
```

1.8.3.3 encodeDES

encodeDES

A função **encodeDES** realiza a codificação de uma sequência de caracteres usando o algoritmo DES, a sua sintaxe é:

```
|$encodeDES(<sequência-de-caracteres>, <chave-DES>)$|
```

1.8.3.4 eval

eval

A função **eval** realiza a avaliação de um expressão numérica passada como parâmetro. A sua sintaxe é:

```
|$eval(<expressão>[, <máscara-do-resultado>])$|
```

Caso o parâmetro <máscara-do-resultado> seja omitido o resultado da avaliação da expressão numérica será um número real com o ponto (.) como separador da parte inteira da parte fracionária. O padrão de máscara a ser utilizado para formatar a saída dessa função é o mesmo adotado pela função NumberFormat usando a opção FMT.

Exemplos:

```
|$eval(5+3,0.##)$|
```

1.8.3.5 fileCopy

fileCopy

A função **fileCopy** realiza a cópia de um arquivo, a sua sintaxe é:

```
|$fileCopy(<arquivo-de-origem>, <arquivo-de-destino>)$|
```

1.8.3.6 htmlFilter

htmlFilter

A função **htmlFilter** remove de um texto as tags HTML nele contidas, a sua sintaxe é:

```
|$htmlFilter(<texto>)$|
```

1.8.3.7 if

if

A função **if** realiza um teste de condição e retorna um determinado valor a depender do resultado da avaliação. A sintaxe da função é a seguinte:

```
|$if(<condição>, <se-condição-verdadeira>, <se-condição-falsa>)$|
```

Se `<condição>` for avaliada como verdadeira (`TRUE`) a função if irá retornar o valor do parâmetro `<se-condição-verdadeira>` caso contrário retornará o que foi declarado no parâmetro `<se-condição-falsa>`.

1.8.3.8 md5

md5

A função **md5** retorna o *hash* de um certo valor usando o algoritmo `MD5`. A sintaxe da função é a seguinte:

```
|$md5(<valor>)$|
```

O algoritmo `MD5` é recomendado ser usado quando é necessário autenticar dados como por exemplo em validações de *`login`*.

1.8.3.9 numberformat

numberformat

A função **numberformat** serve para formatar números e possui a seguinte sintaxe:

```
|$nf(<número>, <tipo-de-formatação>[, <parâmetros-de-formatação>])$|
```

e a depender do tipo de formatação escolhido pelo usuário, a função poderá usar nenhum, um ou vários parâmetros de formatação.

Tipos de Formatação

CBR

Formata o número passado como parâmetro para o padrão monetário brasileiro.

Sintaxe:
```
|$nf(<número>, CBR)$|
```
Exemplos:
```
|$nf(1234567.89, CBR)$|
```

CLR

Retorna o número passado como parâmtero formatado no padrão númerico removendo-se os pontos e substituindo vírgula por ponto. Se na representação numérica passada como parâmetro houver algum caracter diferente de um número, um ponto (.) ou uma vírgula (,) será retornado vazio.

Sintaxe:
```
|$nf(<número>, CLR)$|
```
Exemplos:
```
|$nf(1.234.567\,89, CLR)$|
```

CUS

Formata o número para o padrão monetário adotado nos Estados Unidos.

Sintaxe:
```
|$nf(<número>, CUS)$|
```
Exemplos:
```
|$nf(1234567.89, CUS)$|
```

EXT

Escreve o extenso de um número inteiro.

Sintaxe:
```
|$nf(<número>, EXT)$|
```
Exemplos:
```
|$nf(1234567, EXT)$|
```

INT

Retorna a parte inteira de um número fracionário.

Sintaxe:
```
|$nf(<número>, INT)$|
```
Exemplos:
```
|$nf(1234567.89, INT)$|
```

FRC

Retorna a parte fracionária de um número fracionário.

Sintaxe:
```
|$nf(<número>, FRC)$|
```
Exemplos:
```
|$nf(1234567.89, FRC)$|
```

FMT

Formata o numero passado como parâmetro para um outro padrão, indicando qual o novo formato que ela assumirá.

Sintaxe:
```
|$nf(<número>, FMT, <novo-padrão>)$|
```
Exemplos:
```
|$nf(123456.78, FMT, ###\,##0.00)$|
```

1.8.3.10 piece

piece

A função **piece** retorna uma *substring* a partir de um texto delimitado pela ocorrência inicial (mas não incluída) e a ocorrência-final de um delimitador no texto, a sua sintaxe é:

```
|$piece(<texto>, <delimitador>, <ocorrência-inicial>[, <ocorrência-final>])$|
```

1.8.3.11 random

random

A função **random** gera números aleatórios onde a quantidade de algarismos do número a ser gerado é indicada no parâmetro da função. O segundo parâmetro da função é opcional e serve para indicar se serão gerados apenas números (`true`) ou se letras também serão inseridas (`false`). A sintaxe da função é a seguinte:

```
|$random(<quantidade-de-algarismos>[,<true | false>])$|
```

1.8.3.12 textformat

textformat

A função **textformat** serve para formatar textos e possui a seguinte sintaxe:

```
|$tf(<texto>, <tipo-de-formatação>[, <parâmetros-de-formatação>])$|
```

e a depender do tipo de formatação escolhido pelo usuário, a função poderá usar nenhum, um ou vários parâmetros de formatação.

Tipos de Formatação

2BR

Substitui `\r\n` por `<br>\r\n`, ideal para se usar nas páginas onde o texto tem mais de uma linha.

Sintaxe:

```
|$tf(<texto>, 2BR)$|
```

Exemplos:

```
|$tf(caos, 2BR)$|
```

REP

Substitui do texto uma sequência de caracteres por uma outra.

Sintaxe:
```
|$tf(<texto>, REP, <sequência-a-ser-trocada>, <sequência-nova>)$|
```

Exemplos:
```
|$tf(cão, REP, c, p)$|
```

TRM

Retira do texto passado como parâmetro os espaços em branco que estão em excesso.

Sintaxe:
```
|$tf(<texto>, TRM)$|
```

Exemplos:
```
|$tf(O  rato    roeu a  roupa do  rei de  Roma  !, TRM)$|
```

FLT

Substitui todas as vogais acentuadas encontradas no texto passado como parâmetro pelas respectivas vogais sem acento.

Sintaxe:
```
|$tf(<texto>, FLT)$|
```

Exemplos:
```
|$tf(Vocês, FLT)$|
```

LWC

Converte todo o texto passado como parâmetro para caixa baixa.

Sintaxe:
```
|$tf(<texto>, LWC)$|
```

Exemplos:
```
|$tf(TEXTO EM CAIXA ALTA, LWC)$|
```

UPC

Converte todo o texto passado como parâmetro para caixa alta.

Sintaxe:
```
|$tf(<texto>, UPC)$|
```

Exemplos:
```
|$tf(texto em caixa baixa, UPC)$|
```

LCF

Converte todo o texto passado como parâmetro para caixa baixa e substitui todas as vogais acentuadas por vogais sem acento.

Sintaxe:

```
|$tf(<texto>, LCF)$|
```

Exemplos:

```
|$tf(TEXTO EM CAIXA ALTA COM ACENTUAÇÃO, LCF)$|
```

UCF

Converte todo o texto passado como parâmetro para caixa alta e substitui todas as vogais acentuadas por vogais sem acento.

Sintaxe:

```
|$tf(<texto>, UCF)$|
```

Exemplos:

```
|$tf(texto em caixa baixa com acentuação, UCF)$|
```

1.8.3.13 wi.context

wi.context

A função **wi.context** realiza um *dump* no contexto do WebIntegrator retornando a lista de variáveis e seus respectivos valores no momento em que ela é executada. A listagem das variáveis é retornada no seguinte padrão:

```
<nome-da-variável>=<valor-da-variável>
```

IMPORTANTE:
possa ser que algumas variáveis tenham em seu conteúdo quebras-de-linha, nestes casos a função `wi.context()` substituirá essas quebras pelos sinais de `\r\n`.

A sintaxe de uso é a seguinte:

```
|$wi.context([<parâmetro>])$|
```

Essa função aceita que seja passado um parâmetro indicando quais as variáveis a serem retornadas e se será feito um *dump* recursivo. Para indicar que um *dump* recursivo seja realizado usa-se o asterisco (*). Veja alguns exemplos que seguem abaixo:

`|$wi.context()$|`	Retorna apenas as variáveis que apenas fazem parte do contexto raiz, ou seja, aquelas que cujo identificador não possue pontos (sem recursividade).

`|$wi.context(*)$|`	Retorna todas as variáveis que fazem parte do contexto do WebIntegrator.
`|$wi.context(wi)$|`	Retorna apenas as variáveis que começam com "wi" (sem recursividade).
`|$wi.context(wi.*)$|`	Retorna todas as variáveis que começam com "wi".
`|$wi.context(wi.s%)$|`	Retorna todas as variáveis do contexto do WebIntegrator que começam com "wi.s".

1.8.4 Tutorial do WIChart

Tutorial do WIChart

O WI Chart, é um conjunto de conectores Java que geram imagens de gráficos baseadas em informações que estão armazenadas em objetos do WebIntegrator. O formato do arquivo da imagem gerada segue o padrão PNG e em virtude disso as páginas que irão exibir as imagens deverão ter a propriedade `Tipo` configurada como **png**.

Independentemente do tipo de gráfico a ser gerado eles sempre possuem algumas propriedades comuns:

`Título do gráfico`: indica o nome do título que será dado ao gráfico.

`Largura`: indica a largura em pixels que a imagem do gráfico irá possuir, o valor *`default`* é de `620`.

`Altura`: indica a altura em pixels que a imagem do gráfico irá possuir, o valor *`default`* é de `460`.

`3D`: indica se o gráfico será desenhado em perspectiva 3D. Esse campo apenas aceita os valores true e false sendo false o seu valor *`default`*.

`Nível de opacidade do gráfico`: indica o nível de opacidade que o gráfico terá. O nível de opacidade irá influenciar na intensidade da transparência que o gráfico irá ter, ou seja, quanto menor for o nível de opacidade maior será a transparência. Os valores vão de `1.0f` a `0.0f` sendo `1.0f` o valor *`default`*.

`Cor de fundo da imagem (hexadecimal)`: indica a cor de fundo em formato hexadecimal que a imagem do gráfico irá possuir.

`Objeto (dataset)`: indica o identificador do objeto do WebIntegrator que servirá como dataset

para o gráfico. Por *dataset* entenda como sendo o conjunto de informações que irá popular o gráfico.

A depender do tipo de gráfico as informações dentro do objeto que representam o *dataset* devem estar dispostas de uma maneira padrão a fim de que o gráfico possa ser populado corretamente.

As duas formas aceitáveis são a disposição das informações em linha ou em coluna sendo que em alguns tipos de gráficos apenas uma delas é aceitável. Para saber qual o formato aceito pelo gráfico leia a documentação específica do gráfico.

Os tipos de gráficos atualmente disponíveis são:

- Gráfico em barras
- Gráfico em fatias (torta)
- Gráfico de linhas
- Gráfico de área

1.8.4.1 Gráfico de barras

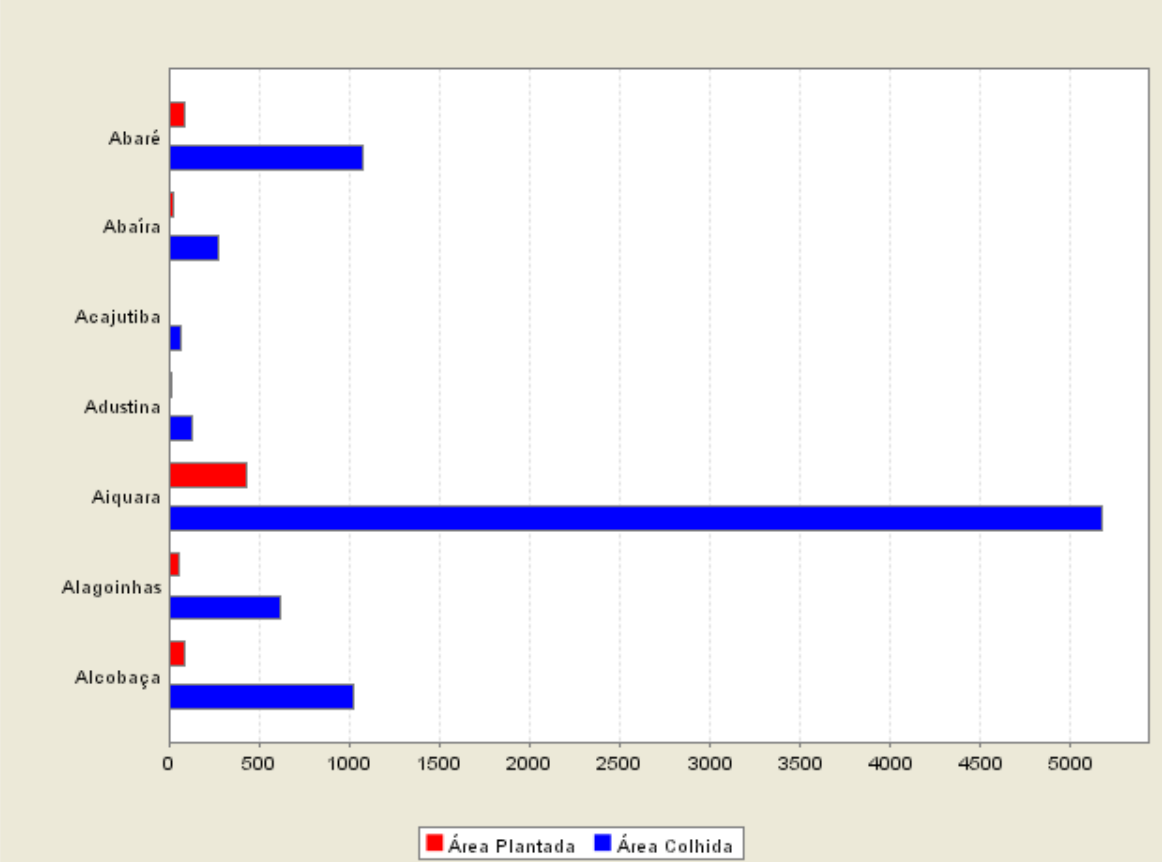
Gráfico de barras

O nome do conector responsável pela geração de gráficos de barra é `br.com.itx.modules.chart.BarChart`. O gráfico de barras gerado por esse conector assume que o objeto que servirá como *dataset* contém a informaçõa de uma ou mais categorias sendo que cada uma dessas categorias pode possuir uma ou mais séries de dados.

O conector `BarChart` assume que o objeto que ele usará para popular o gráfico esteja com as informações das categorias dispostas ou em linha ou em coluna.

Quando a disposição dos dados das categorias estão em linha (ver exemplo abaixo) o conector `BarChart` faz o seguinte procedimento: os valores da primeira coluna serão usados como rótulos das categorias e os nomes das colunas a partir da segunda coluna serão usados como rótulos para a legenda das séries. Perceba que nesse caso os dados para cada uma das categorias estão na linha do nome da categoria.

Por exemplo, o gráfico em barras abaixo foi gerado a partir de um objeto onde as categorias estão sendo representadas pelos municípios Abaré, Abaíra, Acajutiba, Adustina, Aiquara, Alagoinhas e Alcobaça e para cada um desses municípios (categorias) há uma série de dados indicada pelos rótulos "`Área plantada`" e "`Área colhida`".



O objeto usado para gerar o gráfico acima possui a estrutura exibida abaixo. Perceba que os dados das séries para cada uma das categorias (municípios) estão dispostos em linha e os nomes das colunas foram usados como legenda.

Município	Área Plantada	Área Colhida
Abaré	89.7500000000	1077
Abaíra	30.2222222222	272
Acajutiba	5.7500000000	69
Adustina	16.6666666667	127
Aiquara	430.8333333333	5170
Alagoinhas	51.6666666667	620
Alcobaça	85.5000000000	1026

Então um objeto a ser usado como *`dataset`* para geração de gráfico de barra com disposição dos dados das categorias em linha deve ter o seguinte padrão:

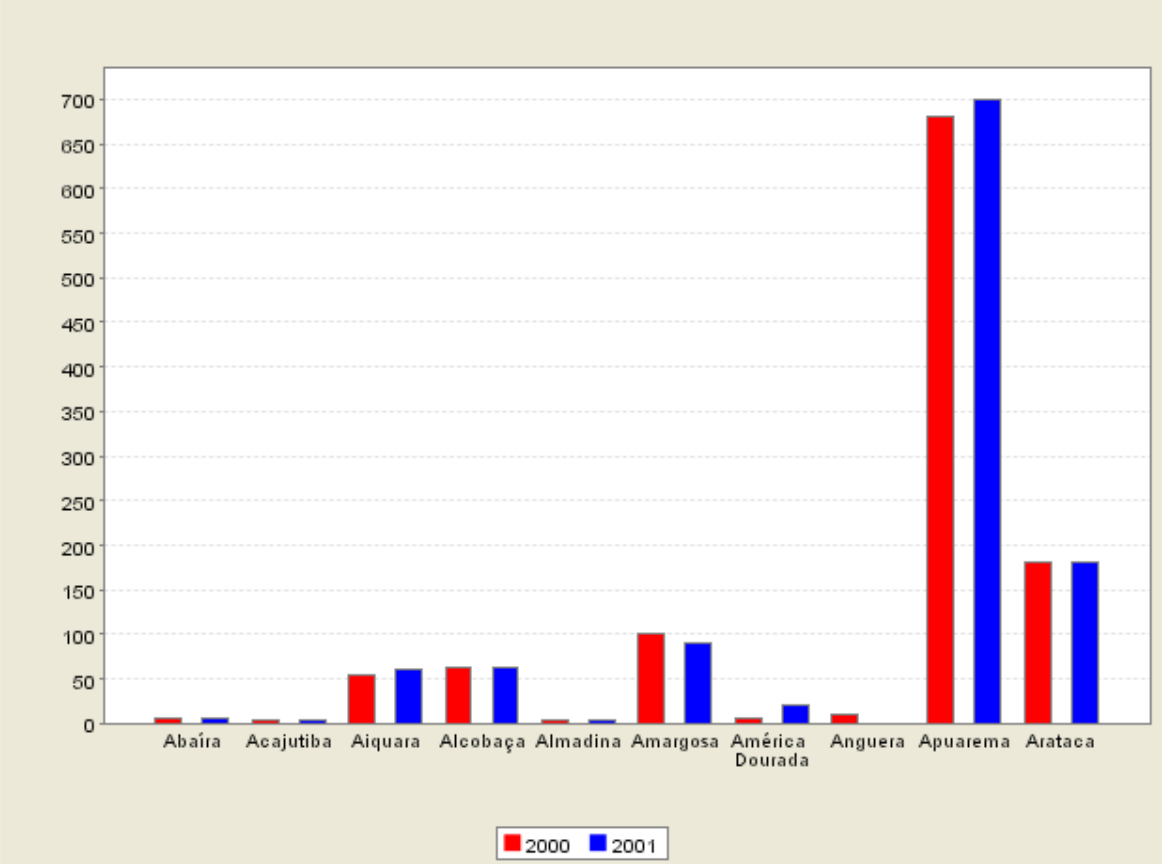
1. Os valores da primeira coluna do objeto contêm os nomes dos rótulos para as categorias.

2. Os nomes das colunas a partir da segunda coluna do objeto indicam os nomes dos rótulos das séries das categorias. É com base nesses nomes que será gerada a legenda para o gráfico.

3. Os valores das séries para cada uma das categorias deverão estar dispostos numa única linha.

Quando a disposição dos dados das categorias estão em coluna (ver exemplo abaixo) o conector `BarChart` faz o seguinte procedimento: os valores da primeira coluna serão usados como rótulos das categorias, os valores da segunda coluna serão usados como rótulos das séries e os valores da terceira coluna serão usados como os dados correspondente ao par (categoria, série) formado pelo valor das duas colunas anteriores.

Por exemplo, o gráfico de barra abaixo foi gerado a partir de um objeto onde as categorias estão sendo representadas pelos municípios listados na primeira coluna e para cada um desses municípios (categorias) há uma série de dados indicada pelos anos onde se calcula a área colhida.



O objeto usado para gerar o gráfico acima possui a estrutura exibida abaixo. Perceba que os dados das séries para cada uma das categorias (municípios) estão dispostos nas colunas.

Município	Ano	Área Colhida
Abaíra	2000	6
Abaíra	2001	6
Acajutiba	2000	3
Acajutiba	2001	3
Aiquara	2000	55
Aiquara	2001	60
Alcobaça	2000	62
Alcobaça	2001	64
Almadina	2000	3
Almadina	2001	3
Amargosa	2000	101
Amargosa	2001	90
América Dourada	2000	5
América Dourada	2001	20
Anguera	2000	10
Apuarema	2000	680
Apuarema	2001	700
Arataca	2000	182
Arataca	2001	182

Então um objeto a ser usado como *dataset* para geração de gráfico de barra com disposição dos dados das categorias em coluna deve ter o seguinte padrão:

1. Os valores da primeira coluna do objeto contêm os nomes dos rótulos para as categorias.
2. Os valores da segunda coluna do obejto contêm os nomes dos rótulos para as séries.
3. Os valores da terceira coluna do objeto contêm os valores dos dados correspondente ao par (categoria, série) formado pelas coluna anteriores.

O conector `BarChart` assume como padrão que os dados das categorias estão dispostos em linha no objeto, mas esse valor pode ser alterado setando a propriedade `tmp.chart.bar.categoryDatasetDisposition` através do campo `Disposição dos dados das categorias no objeto`.

Além das propriedades comuns a todos os gráficos, os gráficos em barra possuem algumas outras propriedades adicionais.

`Posição das barras`: indica a posição na qual serão geradas as barras. Os valores possíveis são `vertical` e `horizontal`, sendo `vertical` o valor *`default`*.

`Empilhar barras`: indica se as barras que compõem uma categoria serão empilhadas em uma

única barra. Os valores possíveis são `true` e `false`, sendo `false` o valor *`default`*.

`Rótulo do eixo X`: indica o nome do rótulo que será dado ao eixo X.

`Rótulo do eixo Y`: indica o nome do rótulo que será dado ao eixo Y.

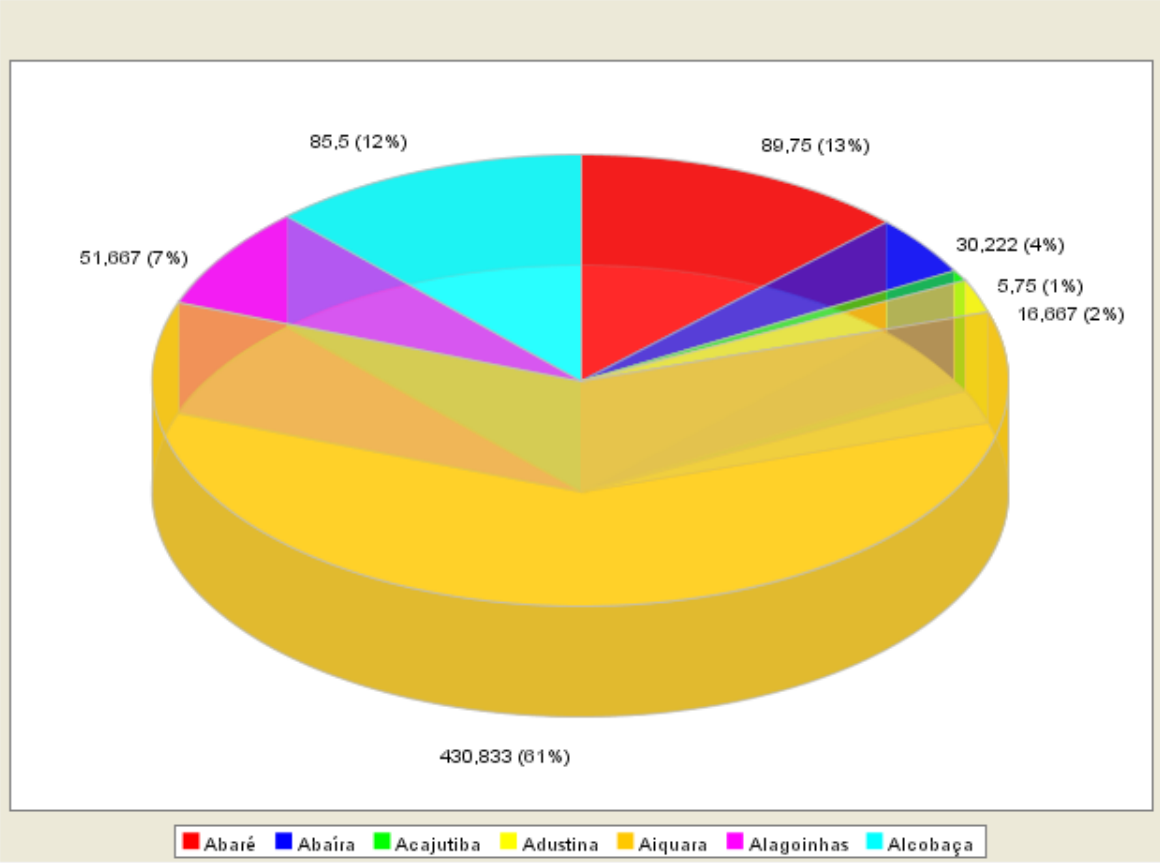
`Disposição dos dados das categorias no objeto`: indica como estão dispostos os dados de cada uma das categorias dentro do objeto. Os valores possíveis são: `line`, indicando que os dados das categorias estão dispostos em linha no objeto; e `column`, indicando que os dados das categorias estão dispostos em coluna no obejto. O valor *`default`* é `line`.

1.8.4.2 Gráfico em fatias (torta)

Gráfico em fatias (torta)

O nome do conector responsável pela geração de gráficos em fatias é `br.com.itx.modules.chart.PieChart`. O gráfico em fatias gerado por esse conector assume que o objeto que servirá como *`dataset`* contém a informação de cada uma das fatias (seções) dispostas uma por linha, ou seja, cada linha do objeto deve conter o nome da sessão e o seu valor correspondente.

Ao contrário do conector BarChart, o conector `PieChart` espera que o objeto a ser utilizado como *`dataset`* disponha os dados de tal maneira que cada linha do objeto contenha as informações das seções do gráfico onde na primeira coluna estão dispostos os nomes das seções e na segunda coluna os valores correspondentes. Como exemplo veja o gráfico a seguir.



O objeto usado para gerar o gráfico acima possui a estrutura exibida abaixo. Perceba que os dados e os nomes de cada uma das fatias estão dispotos um por linha.

Município	Área Plantada
Abaré	89.7500000000
Abaíra	30.2222222222
Acajutiba	5.7500000000
Adustina	16.6666666667
Aiquara	430.8333333333
Alagoinhas	51.6666666667
Alcobaça	85.5000000000

Então um objeto a ser usado como dataset para geração de gráfico de fatias deve ter o seguinte padrão:

1. Os valores da primeira coluna do objeto contêm os nomes das seções.

2. Os valores da segunda coluna do obejto contêm os valores das seções.

Além das propriedades comuns a todos os gráficos, os gráficos em fatias possuem uma propriedade bastante peculiar:

`Modo de exibição dos rótulos das seções`: indica o modo de exibição dos rótulos de cada seção do gráfico, e pode estar disposto segundo uma das especificações, conforme descrição abaixo:

- **no**: indica que não será exibido um rótulo
- **name**: indica que será exibido um rótulo apenas com o nome
- **value**: indica que será exibido um rótulo contendo apenas o valor
- **percent**: indica que será exibido apenas a porcentagem no rótulo
- **name_and_value**: inidica a exibição exibe nome e valor
- **name_and_percent**: exibe nome e porcentagem
- **value_and_percent**: exibe valor e porcentagem

1.8.4.3 Gráfico de linhas

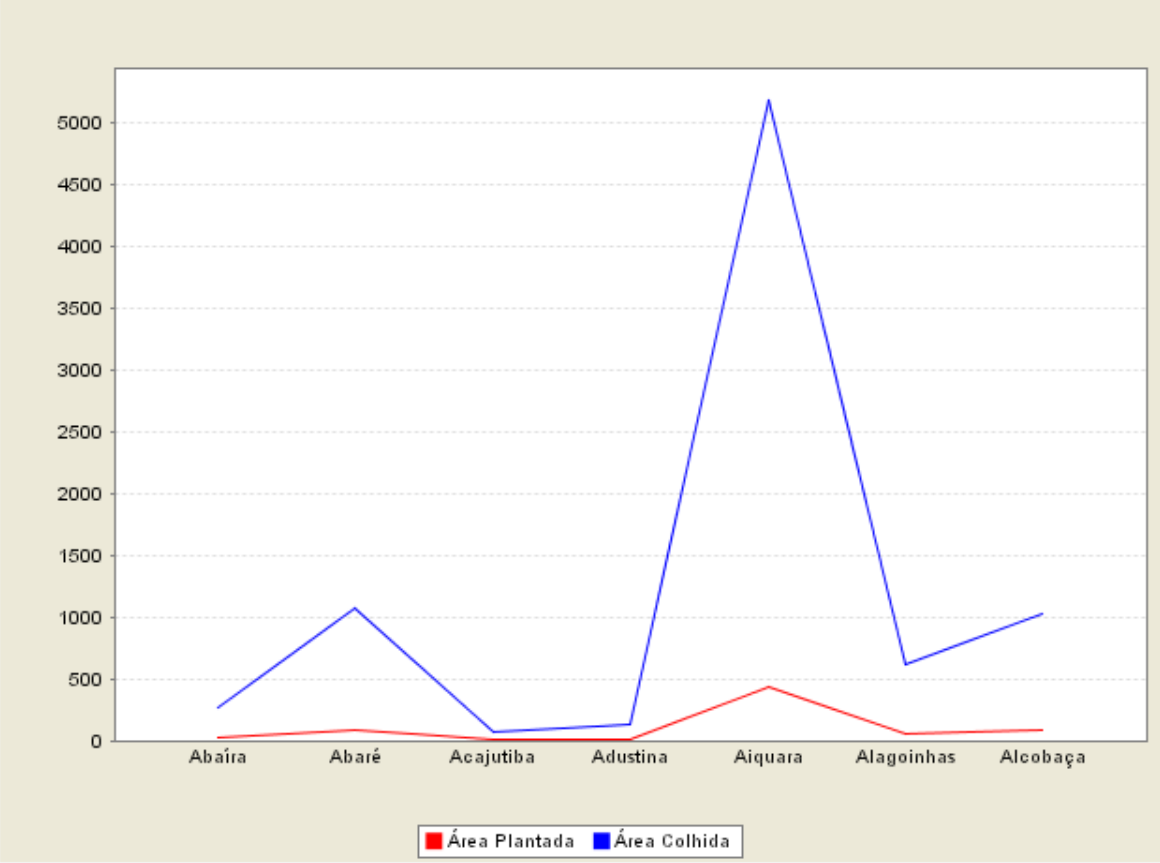
Gráfico em linhas

O nome do conector responsável pela geração de gráficos de linhas é `br.com.itx.modules.chart.LineChart`. O gráfico de linhas gerado por esse conector assume que o objeto que servirá como *`dataset`* contém a informaçõa de uma ou mais categorias sendo que cada uma dessas categorias pode possuir uma ou mais séries de dados.

O conector `LineChart` assume que o objeto que ele usará para popular o gráfico esteja com as informações das categorias dispostas ou em linha ou em coluna.

Quando a disposição dos dados das categorias estão em linha (ver exemplo abaixo) o conector `LineChart` faz o seguinte procedimento: os valores da primeira coluna serão usados como rótulos das categorias e os nomes das colunas a partir da segunda coluna serão usados como rótulos para a legenda das séries. Perceba que nesse caso os dados para cada uma das categorias estão na linha do nome da categoria.

Por exemplo, o gráfico de linhas abaixo foi gerado a partir de um objeto onde as categorias estão sendo representadas pelos municípios Abaré, Abaíra, Acajutiba, Adustina, Aiquara, Alagoinhas e Alcobaça e para cada um desses municípios (categorias) há uma série de dados indicada pelos rótulos "`Área plantada`" e "`Área colhida`".



O objeto usado para gerar o gráfico acima possui a estrutura exibida abaixo. Perceba que os dados das séries para cada uma das categorias (municípios) estão dispostos em linha e os nomes das colunas foram usados como legenda.

Município	Área Plantada	Área Colhida
Abaré	89.7500000000	1077
Abaíra	30.2222222222	272
Acajutiba	5.7500000000	69
Adustina	16.6666666667	127
Aiquara	430.8333333333	5170
Alagoinhas	51.6666666667	620
Alcobaça	85.5000000000	1026

Então um objeto a ser usado como dataset para geração de gráfico de linhas com disposição dos dados das categorias em linha deve ter o seguinte padrão:

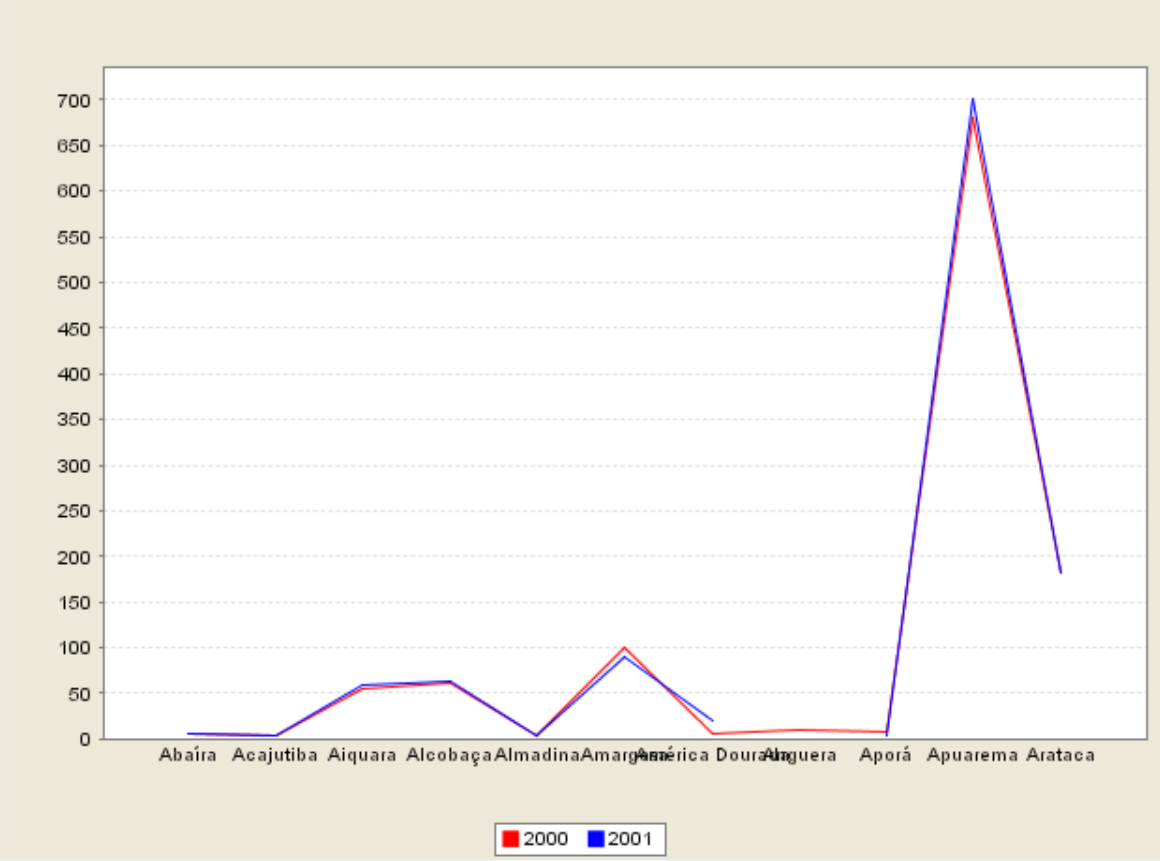
1. Os valores da primeira coluna do objeto contêm os nomes dos rótulos para as categorias.

2. Os nomes das colunas a partir da segunda coluna do objeto indicam os nomes dos rótulos das séries das categorias. É com base nesses nomes que será gerada a legenda para o gráfico.

3. Os valores das séries para cada uma das categorias deverão estar dispostos numa única linha.

Quando a disposição dos dados das categorias estão em coluna (ver exemplo abaixo) o conector `LineChart` faz o seguinte procedimento: os valores da primeira coluna serão usados como rótulos das categorias, os valores da segunda coluna serão usados como rótulos das séries e os valores da terceira coluna serão usados como os dados correspondente ao par (categoria, série) formado pelo valor das duas colunas anteriores.

Por exemplo, o gráfico de linhas abaixo foi gerado a partir de um objeto onde as categorias estão sendo representadas pelos municípios listados na primeira coluna e para cada um desses municípios (categorias) há uma série de dados indicada pelos anos onde se calcula a área colhida.



O objeto usado para gerar o gráfico acima possui a estrutura exibida abaixo. Perceba que os dados das séries para cada uma das categorias (municípios) estão dispostos nas colunas.

Município	Ano	Área Colhida
Abaíra	2000	6
Abaíra	2001	6
Acajutiba	2000	3
Acajutiba	2001	3
Aiquara	2000	55
Aiquara	2001	60
Alcobaça	2000	62
Alcobaça	2001	64
Almadina	2000	3
Almadina	2001	3
Amargosa	2000	101
Amargosa	2001	90
América Dourada	2000	5
América Dourada	2001	20
Anguera	2000	10
Apuarema	2000	680
Apuarema	2001	700
Arataca	2000	182
Arataca	2001	182

Então um objeto a ser usado como dataset para geração de gráfico de linhas com disposição dos dados das categorias em coluna deve ter o seguinte padrão:

1. Os valores da primeira coluna do objeto contêm os nomes dos rótulos para as categorias.

2. Os valores da segunda coluna do obejto contêm os nomes dos rótulos para as séries.

3. Os valores da terceira coluna do objeto contêm os valores dos dados correspondente ao par (categoria, série) formado pelas coluna anteriores.

O conector `LineChart` assume como padrão que os dados das categorias estão dispostos em linha no objeto, mas esse valor pode ser alterado setando a propriedade `tmp.chart.bar.categoryDatasetDisposition` através do campo `Disposição dos dados das categorias no objeto`.

Além das propriedades comuns a todos os gráficos, os gráficos de linhas possuem algumas outras propriedades adicionais.

`Orientação do plot`: indica a orientação que o *`plot`* utilizará para desenhar o gráfico. Os valores possíveis são `vertical` e `horizontal`, sendo vertical o valor *`default`*.

`Rótulo do eixo X`: indica o nome do rótulo que será dado ao eixo X.

`Rótulo do eixo Y`: indica o nome do rótulo que será dado ao eixo Y.

`Disposição dos dados das categorias no objeto`: indica como estão dispostos os dados de cada uma das categorias dentro do objeto. Os valores possíveis são `line`, indicando que os dados das categorias estão dispostos em linha no objeto, e `column`, indicando que os dados das categorias estão dispostos em coluna no obejto. O valor *`default`* é `line`.

1.8.4.4 Gráfico de área

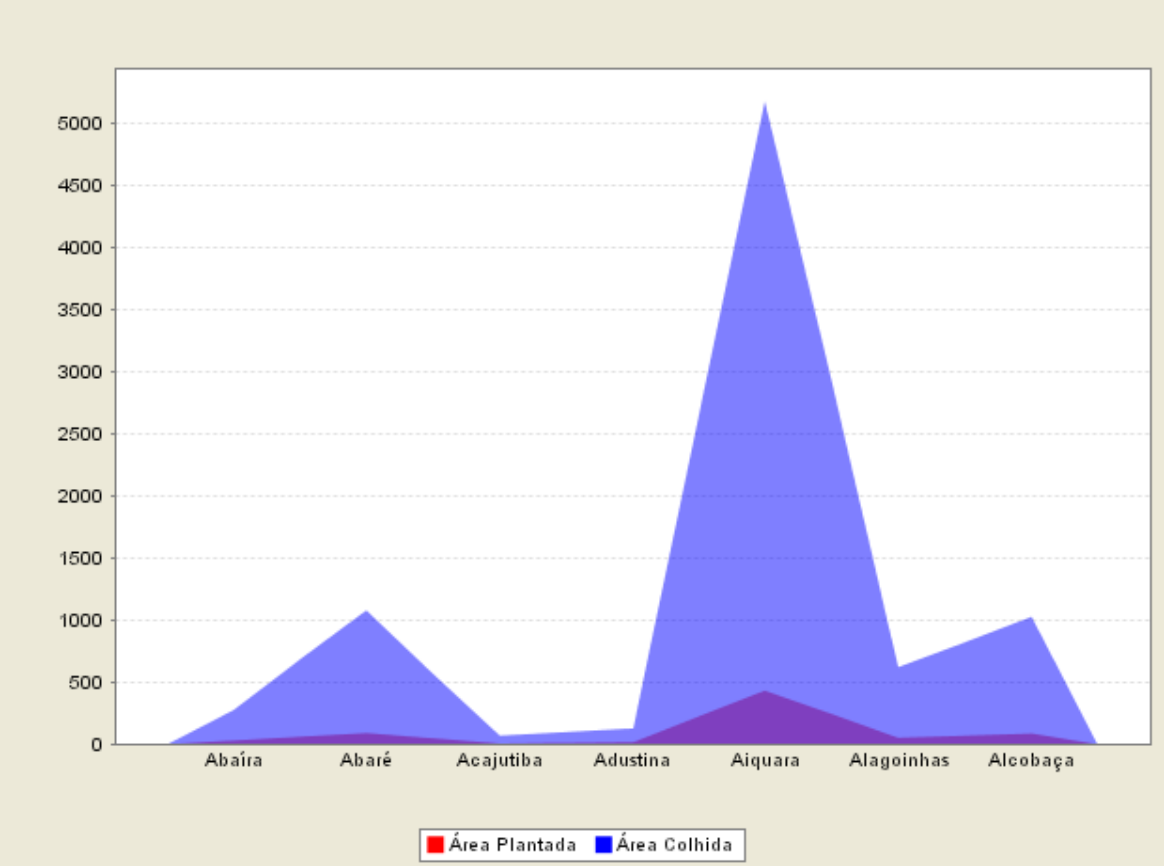
Gráfico de área

O nome do conector responsável pela geração de gráficos de área é `br.com.itx.modules.chart.AreaChart`. O gráfico de área gerado por esse conector assume que o objeto que servirá como dataset contém a informaçõa de uma ou mais categorias sendo que cada uma dessas categorias pode possuir uma ou mais séries de dados.

O conector `AreaChart` assume que o objeto que ele usará para popular o gráfico esteja com as informações das categorias dispostas ou em linha ou em coluna.

Quando a disposição dos dados das categorias estão em linha (ver exemplo abaixo) o conector `AreaChart` faz o seguinte procedimento: os valores da primeira coluna serão usados como rótulos das categorias e os nomes das colunas a partir da segunda coluna serão usados como rótulos para a legenda das séries. Perceba que nesse caso os dados para cada uma das categorias estão na linha do nome da categoria.

Por exemplo, o gráfico de linhas abaixo foi gerado a partir de um objeto onde as categorias estão sendo representadas pelos municípios Abaré, Abaíra, Acajutiba, Adustina, Aiquara, Alagoinhas e Alcobaça e para cada um desses municípios (categorias) há uma série de dados indicada pelos rótulos "`Área plantada`" e "`Área colhida`".



O objeto usado para gerar o gráfico acima possui a estrutura exibida abaixo. Perceba que os dados das séries para cada uma das categorias (municípios) estão dispostos em linha e os nomes das colunas foram usados como legenda.

Município	Área Plantada	Área Colhida
Abaré	89.7500000000	1077
Abaíra	30.2222222222	272
Acajutiba	5.7500000000	69
Adustina	16.6666666667	127
Aiquara	430.8333333333	5170
Alagoinhas	51.6666666667	620
Alcobaça	85.5000000000	1026

Então um objeto a ser usado como *`dataset`* para geração de gráfico de linhas com disposição dos dados das categorias em linha deve ter o seguinte padrão:

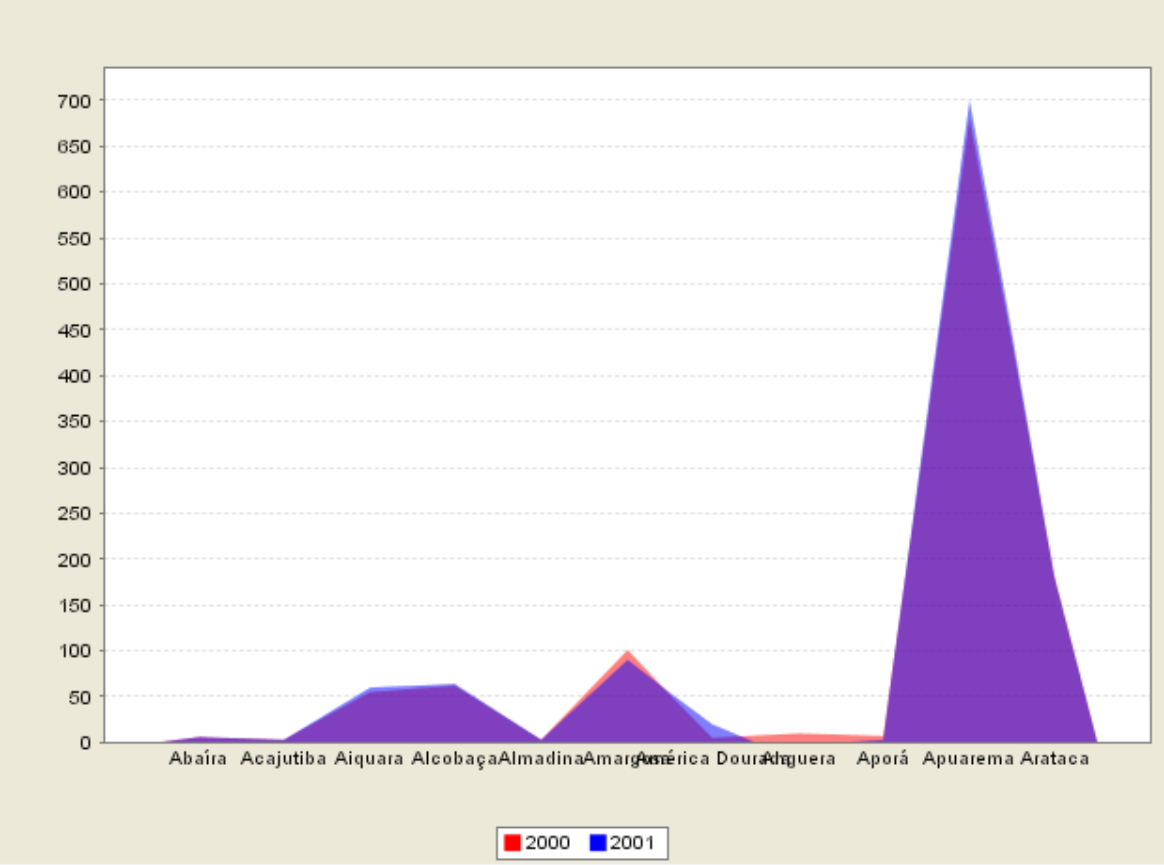
1. Os valores da primeira coluna do objeto contêm os nomes dos rótulos para as categorias.

2. Os nomes das colunas a partir da segunda coluna do objeto indicam os nomes dos rótulos das séries das categorias. É com base nesses nomes que será gerada a legenda para o gráfico.

3. Os valores das séries para cada uma das categorias deverão estar dispostos numa única linha.

Quando a disposição dos dados das categorias estão em coluna (ver exemplo abaixo) o conector `AreaChart` faz o seguinte procedimento: os valores da primeira coluna serão usados como rótulos das categorias, os valores da segunda coluna serão usados como rótulos das séries e os valores da terceira coluna serão usados como os dados correspondente ao par (categoria, série) formado pelo valor das duas colunas anteriores.

Por exemplo, o gráfico de linhas abaixo foi gerado a partir de um objeto onde as categorias estão sendo representadas pelos municípios listados na primeira coluna e para cada um desses municípios (categorias) há uma série de dados indicada pelos anos onde se calcula a área colhida.



O objeto usado para gerar o gráfico acima possui a estrutura exibida abaixo. Perceba que os dados das séries para cada uma das categorias (municípios) estão dispostos nas colunas.

Município	Ano	Área Colhida
Abaíra	2000	6
Abaíra	2001	6
Acajutiba	2000	3
Acajutiba	2001	3
Aiquara	2000	55
Aiquara	2001	60
Alcobaça	2000	62
Alcobaça	2001	64
Almadina	2000	3
Almadina	2001	3
Amargosa	2000	101
Amargosa	2001	90
América Dourada	2000	5
América Dourada	2001	20
Anguera	2000	10
Apuarema	2000	680
Apuarema	2001	700
Arataca	2000	182
Arataca	2001	182

Então um objeto a ser usado como dataset para geração de gráfico de linhas com disposição dos dados das categorias em coluna deve ter o seguinte padrão:

1. Os valores da primeira coluna do objeto contêm os nomes dos rótulos para as categorias.
2. Os valores da segunda coluna do obejto contêm os nomes dos rótulos para as séries.
3. Os valores da terceira coluna do objeto contêm os valores dos dados correspondente ao par (categoria, série) formado pelas coluna anteriores.

O conector `AreaChart` assume como padrão que os dados das categorias estão dispostos em linha no objeto, mas esse valor pode ser alterado setando a propriedade `tmp.chart.bar.categoryDatasetDisposition` através do campo `Disposição dos dados das categorias no objeto`.

Além das propriedades comuns a todos os gráficos, os gráficos de área possuem algumas outras propriedades adicionais.

`Orientação do plot`: indica a orientação que o *`plot`* utilizará para desenhar o gráfico. Os valores possíveis são `vertical` e `horizontal`, sendo `vertical` o valor *`default`*.

`Rótulo do eixo X`: indica o nome do rótulo que será dado ao eixo X.

`Rótulo do eixo Y`: indica o nome do rótulo que será dado ao eixo Y.

`Disposição dos dados das categorias no objeto`: indica como estão dispostos os dados de cada uma das categorias dentro do objeto. Os valores possíveis são `line`, indicando que os dados das categorias estão dispostos em linha no objeto, e `column`, indicando que os dados das categorias estão dispostos em coluna no obejto. O valor *`default`* é `line`.

1.9 Como...

Como...

Aqui o desenvolvedor encontrará um passo a passo para diversos elementos do WebIntegrator.

1.9.1 ...usar a Bridge Jdbc para Ldap

...usar a Bridge Jdbc para Ldap

Neste passo-a-passo será mostrado como o desenvolvedor poderá utilizar a Bridge JDBC Ldap para acessar um servidor de diretórios (Active Directory, Apache DS, etc) usando o protocolo Ldap através de pseudo comandos SQL.

O documentação da Bridge Jdbc Ldap pode ser encontrada no site myvd.sourcefourge.net através do link http://myvd.sourceforge.net/bridge.html enquando o download pode ser feito clicando http://myvd.sourceforge.net/jdbcldap.html.

Nos testes foram utilizados o sevidor de diretórios Apache Directory Server (http://directory.apache.org/apacheds/1.5/) e Apache Directory Studio (http://directory.apache.org/studio/), mas nada impede que outros servidores de diretórios sejam utilizados a título do Active Directory da Microsoft.

Antes de iniciar os teste usando o Apache DS foi necessário importar o arquivo example.ldif que já acompanha o produto usando o Apache Studio.

Para que o WI reconheça a Bridge Jdbc Jdap é necessário o download da mesma no link mencionado acima e a colocação do arquivo jdbcLdap.jar na pasta lib do Tomcat 6 ou shared/lib

do Tomcat 5.

Ao criar uma conexão com o danco de dados deve-se selecionar a Bridge Jdbc Ldap a qual não estará com *, preencher no alias **//localhost:10389**, no campo usuário colocar **uid=admin,ou=system** e no campo senha o valor **secret**.

Antes de efetivamente começar a escrever os pseudos sqls deve-se enterder o 3 tipos de escopos que a Brigde Jdbc Ldap pode utilizar:

- **objectScope**: será recuperado apenas os dados do nó especificado
(baixo uso pois os dados retornados são bastante limitados)
Ex1: select * from objectScope;dc=example,dc=com
Ex2: select * from objectScope;ou=Users,dc=example,dc=com
Ex3: select * from objectScope;uid=aeinstein,ou=Users,dc=example,dc=com

- **oneLevelScope**: será recuperado os dados filhos do nó especificado
(muito uso pois retorna dados estruturalmente muito parecidos com um tabela relacional)
Ex1: select * from oneLevelScope;dc=example,dc=com
Ex2: select * from oneLevelScope;ou=users,dc=example,dc=com
Ex3: select * from oneLevelScope;ou=Users,dc=example,dc=com where uid=?|tmp.user|

- **subTreeScope**: será recuperado os dados do nó especificado e todos os filhos recursivamente
(baixo uso pois os dados retornados representam uma estrutura hierárquica díficil de trabalhar com o modelo relacional)

OBS: Conforme vimos nos exemplos acima em quase 100% dos casos iremos trabalhar com o **oneLevelScope.**
E caso o desenvolvedor deseje ele pode ser especificado como escopo padrão no alias do BD para não precisar ser informando nos SQLs.
Ex: //localhost:10389?SEARCH_SCOPE:=oneLevelScope

Segue alguns exemplos de SQL:

- Que pode ser usado no login do projeto que não usa MD5
SELECT * FROM oneLevelScope;ou=Users,dc=example,dc=com WHERE uid=?|tmp.user| and userpassword=?|tmp.pass|

- Que pode ser usado no login do projeto que usa MD5
SELECT userpassword, uid, cn, givenname FROM oneLevelScope;ou=Users,dc=example,dc=com WHERE uid = ?|tmp.user|
Nesse exemplo as senhas armazenadas em userpassword precisam estar no formato MD5

- Um exemplo de SQL com texto fixo
select * from oneLevelScope;ou=Users,dc=example,dc=com where uid='aeinstein'

- O download BD de uma imagem armazenada no formato Base64 no servidor de diretórios
select jpegphoto from oneLevelScope;ou=Users,dc=example,dc=com where uid='aeinstein'
É necessário marcar na definição do download a opção base64 e colocar em tipo o valor jpg que é o formato que está armazenado.

A Bridge Jdbc Ldap também suporta os comando insert, update e delete de forma muito semelhante a um banco de dados relacional, e maiores detalhes podem ser obtidos na documentação do produto. Segue alguns exemplos de update:

- Altera o nome do usuário aeinstein
UPDATE objectScope;uid=aeinstein,ou=Users,dc=example,dc=com SET givenname = 'AlbertX'

OBS: A implementação 2.1 da Bridge Jdbc Ldap contém algum bug que só está aceitando inserts, updates e deletes usando o objectScope.
Funciona: UPDATE objectScope;uid=|tmp.user|,ou=Users,dc=example,dc=com SET givenname = 'AlbertX'
Deveria Funcionar: UPDATE oneLevelScope;ou=Users,dc=example,dc=com SET givenname = 'AlbertX' where uid='|tmp.user|'

- Altera, adiciona ou remove propriedades diretamente pelo acesso Ldap
UPDATE ENTRY objectScope;uid=aeinstein,ou=Users,dc=example,dc=com DO REPLACE SET userPassword='secret'
OBS: Pela documentação da Bridge Jdbc Ldap apesar dessa forma não ser padrão Sql é a forma mais poderosa de manipular as propriedades Ldap.

1.9.2 ...usar o WI_Event

...usar o WI_Event

Neste passo-a-passo será explicado como o desenvolvedor poderá utilizar o WI_Event para fazer a interação do cliente (browser) com o servidor sem a necessidade de submter a página. Como estudo de caso vamos popular uma combo-box dinamicamente, neste exemplo vamos criar uma página com uma combo-box que terá seus valores populados de acordo com um critério de pesqusia a ser digitado pelo usuário:

Para a montagem desse exemplo primeiramente crie um banco de dados que possua um esquema SQL semelhante ao que segue abaixo.

```
CREATE TABLE uf (
  sigla varchar(2) ,
  nome varchar(30)
);

INSERT INTO uf VALUES("AC","Acre");
INSERT INTO uf VALUES("AL","Alagoas");
INSERT INTO uf VALUES("AM","Amazonas");
INSERT INTO uf VALUES("AP","Amapá");
INSERT INTO uf VALUES("BA","Bahia");
INSERT INTO uf VALUES("CE","Ceará");
INSERT INTO uf VALUES("DF","Distrito Federal");
INSERT INTO uf VALUES("GO","Goiás");
INSERT INTO uf VALUES("MA","Maranhão");
INSERT INTO uf VALUES("MG","Minas Gerais");
INSERT INTO uf VALUES("MS","Mato Grosso do Sul");
INSERT INTO uf VALUES("MT","Mato Grosso");
INSERT INTO uf VALUES("PB","Paraíba");
```

```
INSERT INTO uf VALUES("PE","Pernambuco");
INSERT INTO uf VALUES("PI","Piauí");
INSERT INTO uf VALUES("PR","Paraná");
INSERT INTO uf VALUES("RJ","Rio de Janeiro");
INSERT INTO uf VALUES("RO","Rondônia");
INSERT INTO uf VALUES("RR","Roraima");
INSERT INTO uf VALUES("RS","Rio Grande do Sul");
INSERT INTO uf VALUES("SC","Santa Catarina");
INSERT INTO uf VALUES("SE","Sergipe");
INSERT INTO uf VALUES("SP","São Paulo");
INSERT INTO uf VALUES("TO","Tocantins");
```

Após a criação da base de dados crie um projeto pelo WI_Builder e defina o banco de dados criado como o banco de dados do projeto. Primeiramente vamos definir um Evento do tipo SELECT (clique em Projeto e em seguida na opção Events escolhendo o tipo Selects) definindo o identificador como uf, escolha o banco de dados que possui a estrutura de tabela definida acima e no campo SQL digite a seguinte instrução:

```
SELECT * FROM uf WHERE nome LIKE '|tmp.criterio|%'
```

Para uma página poder referenciar aos Eventos definidos no WI_Builder é necessário que essa página caregue um arquivo JavaScript que representa a biblioteca de funções do WI_Event. Para inserir a chamada à biblioteca do WI_Event numa página de um projeto usando o WIzard de páginas basta selecionar a opção <HEAD> e no campo Link JS adicionar o valor /|wi.proj.id|/js/wievent.js, o campo Link JS suporta que a definição de vários arquivos JS desde que cada chamada aos arquivos JS estejam separadas por vírgula.

Para inserir a chamada à biblioteca do WI_Engine diretamente no código-fonte de uma página do projeto insira o trecho de código HTML que segue abaixo dentro da tag HEAD do documento HTML que representa a página.

```
<SCRIPT TYPE="text/javascript"
SRC="/|wi.proj.id|/js/wievent.js"></SCRIPT>
```

No arquivo wievent.js pode ser vista um resumo dos métodos suportados pelo WIEvent.

A biblioteca de funções do WIEvent disponibiliza o método utilitário populateCombo() para ajudar o desenvolvedor na montagem de combos dinâmica.

Logo abaixo segue o código-fonte da página que irá popular irá popular dinamicamente a combo-box.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Populando uma combo-box dinamicamente</title>
<!-- arquivo JS da biblioteca de funções de um objeto WIEvent -->
<SCRIPT TYPE="text/javascript"
SRC="/|wi.proj.id|/js/wievent.js"></SCRIPT>
<script>
function popularCombo() {
  //instancio um objeto do tipo WIEvent para que eu possa interagir
  //com um componente WI_Event definido no WI_Builder.
  var evt = new WIEvent();
  //crio uma variável "tmp.criterio" na sessão do WebIntegrator com
  //o valor que foi digitado no campo "criterio" do formulário.
  //Lembre-se que na definição do componente WI_Event pelo WI_Builder
  //foi feita a referência a uma variável "tmp.criterio" no comando SQL.
```

```
evt.writeobj("tmp.criterio", document.forms[0]["criterio"].value);
  //Uso a função populate.Combo com os seguintes parâmetros: "uf"
  //é o nome do evento definido no WI_Builder,
document.forms[0]["tmp.estados"]
  //é o nome da combo definida na página, "sigla" indica o nome da
  //coluna que será usada para popular os valores da combo e "sigla"
  //indica o nome da coluna que será usada para popular os textos.
  evt.populateCombo("uf", document.forms[0]["tmp.estados"], "sigla",
"nome");
}
</script>
</head>
<body>
<form>
Critério de pesquisa: <input type="text" name="criterio" value="">
<input type="button" value="Popular Combo" onclick="popularCombo()">
<br>
<select name="tmp.estados"></select>
</form>
</body>
</html>
```

A população dinâmica de uma combo é apenas um dos exemplos de usabilidade do componente WI_Event, outras funcionalidades podem ser desenvolvidas a depender das necessidades que venham a surgir para um determinado projeto. Como regras para utilizar o WI_Event adote os seguintes pontos:

- a página que for interagir com um componente WI_Event do projeto ou apenas utilizar o objeto WIEvent deve carregar a biblioteca de funções JavaScript que se encontra em /|wi.proj.id|/js/wievent.js.
- instancie um objeto do tipo WIEvent.
  ...
  evt = new WIEvent();
  ...
- execute o método desejado.

Exemplos:

Para dar alerts dos estados que iniciam com o prefixo desejado:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<SCRIPT TYPE="text/javascript"
SRC="/|wi.proj.id|/js/wievent.js"></SCRIPT>
<script>
function myalerts() {
  var evt = new WIEvent();
  evt.debug = true; // se desejar ver o debug da comunicação do WIEvent
  evt.writeobj("tmp.criterio", document.forms[0]["criterio"].value);
  evt.selectdb("uf");
  while (evt.next() > -1) {
    alert(evt.column("nome"));
  }
}
</script>
</head>
<body>
```

```
<form>
Critério de pesquisa: <input type="text" name="criterio" value="">
<input type="button" value="Dar alerts" onclick="myalerts()">
</form>
</body>
</html>
```

Para recuperar numa caixa de texto o nome do estado dado o código do mesmo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<SCRIPT TYPE="text/javascript"
SRC="/|wi.proj.id|/js/wievent.js"></SCRIPT>
<script>
function acheNome() {
  var evt = new WIEvent();
  evt.writeobj("tmp.sigla", document.forms[0]["sigla"].value);
  evt.setInputValue("acheNome", document.forms[0]["nome"], "nome")
}
</script>
</head>
<body>
<form>
Sigla: <input type="text" name="sigla" value=""><br>
Nome: <input type="text" name="nome" value=""><br>
<input type="button" value="Ache Nome" onclick="acheNome()">
</form>
</body>
</html>
```

OBS: Para funcionar precisa ser criado o WIEvent do tipo Select com o seguinte comando sql: SELECT * FROM uf WHERE sigla = '|tmp.sigla|'

1.9.3 ...adicionar outras taglibs aos arquivos jsp

...adicionar outras taglibs aos arquivos jsp

O desenvolvedor pode definir outras taglibs para serem adicionadas aos arquivos jsp, bastando para isso criar no WEB-INF do projeto um arquivo taglibs.txt e colocar os registros desejados. Ao modificar o arquivo é necesário regerar todos os arquivos jsp.

Veja exemplo do arquivo taglibs.txt:

```
<%@ taglib prefix='x' uri='http://java.sun.com/jsp/jstl/core' %>
<%@ taglib prefix='y' uri='http://java.sun.com/jsp/jstl/core' %>
<%@ taglib prefix='z' uri='/wicore' %>
```

1.9.4 ...configurar o arquivo JDBC_Drivers.xml

...configurar o arquivo JDBC_Drivers.xml

O arquivo `JDBC_Drivers.xml` é um arquivo com estrutura XML que contém as definições das possíveis conexões com bancos de dados que podem ser feitas pelo WebIntegrator. O principal objetivo desse arquivo é poder adicionar novos tipos de bancos de dados ao WebIntegrator extendo e adaptando o produto às necessidades do usuário.

Para adicionar uma entrada de um novo driver JDBC a esse arquivo basta editá-lo informando o nome completo da classe do driver que implementa a interface `Driver` da API JDBC, a string de conexão usada por esse driver, uma descrição para ser exibida na definição de banco de dados no **WI Builder**, um identificador único para essa configuração que será usado internamente pelo **WI Engine** e opcionalmente uma query de validação a ser usada para validar se uma conexão do pool ainda corresponde com uma conexão ativa no banco de dados.

As informações sobre a classe do driver (CLASS) e da URL de string de conexão (URL) são dependentes do driver JDBC que se estiver configurando e deverão estar disponíveis na documentação do driver, a `query` de validação (VALIDATIONQUERY) quando configurada deverá retornar como resultado no mínimo uma linha válida, as outras duas informações são de escolha própria do usuário.

Como exemplo suponha que se queira possibilitar que o WebIntegrator acesse bases de dados em **HSQL**. Primeiramente deve-se fazer o download do respectivo driver JDBC a partir do site oficial do fabricante e instalá-lo num local adequado, por "local adequado" entenda como sendo um diretório que faça parte do CLASSPATH da JVM e que também esteja acessível às classes do pacote do WebIntegrator. Em ambientes com instalações baseadas no Tomcat recomendamos que a instalação dos drivers JDBC seja feita em `<TOMCAT_HOME>/common/lib`.

A documentação do `HSQL 1.7.1b` informa que o nome completo da classe que implementa a interface `Driver` da API JDBC é `org.hsqldb.jdbcDriver` e a string de conexão deve ser `jdbc:hsqldb:hsql://<servidor>:<porta>`. Com essas informações já se pode criar a entrada no arquivo `drivers.def` que pode ser semelhante com a que segue abaixo.

```
...
<JDBC ID="hsqldb">
  <DESCRIPTION>HSQLDB</DESCRIPTION>
  <CLASS>org.hsqldb.jdbcDriver</CLASS>
  <URL>jdbc:hsqldb:hsql:|alias|</URL>
  <VALIDATIONQUERY>SELECT 1</VALIDATIONQUERY>/
</JDBC>
...
```

Perceba que a definição da string de conexão feita na tag URL faz referência a uma variável chamada **`alias`**. Essa variável será substituída pelo que o usuário definir no campo Alias na definição de banco de dados completando assim a string de conexão a ser usada pelo driver

JDBC para se conectar com o banco de dados.

Algumas implementações de drivers JDBC recomendam que os mesmos sejam inicializados ou registrados anteriormente, para casos como esses a tag CLASS aceita que seja passado ou o atributo NEWINSTANCE ou o atributo REGISTERDRIVE. Para drivers JDBC que recomendem o uso do método newInstance() basta adicionar o atributo NEWINSTANCE com o valor **ON**, para drivers JDBC que recomendem o registro prévio através do método registerDrive() basta adicionar o atributo REGISTERDRIVER com o valor **ON**.

1.9.5 ...criar um novo elemento para uso no WIzard

...criar um novo elemento para uso no WIzard

A ferramenta WIzard do WebIntegrator oferece a oportunidade do desenvolvedor poder acrescentar novos elementos a serem usados numa página WSP criados pelo próprio desenvolvedor. Esses elementos não são nada mais além do que trechos de código HTML que serão incluídos na página onde algumas propriedades poderão ser preenchidas de acordo com os valores que elas foram definidas através do WIzard.

Um elemento criado pelo desenvolvedor poderá estar disponível para todos os projetos que fazem parte de uma instalação do WebIntegrator ou apenas para um projeto específico e além do mais eles também podem estar associados a um tipo específico de dado de um dos bancos de dados do projeto, desta maneira sempre que o WIzard montar uma página baseada em um Objeto de pré-página ele tentará usar aqueles componentes que estiverem associados a um tipo específico de acordo com os valores do Objeto.

Como um exemplo inicial tente definir um elemento que servirá como entrada para valores de data dividido em três campos separado pelo símbolo da barra (/) onde um dos campos indica o dia, o outro o mês e o última campo indicando o ano. Para isso, acesse o WIzard e na janela de Propriedades clique no rótulo do campo Tipo. Na nova janela que se abre digite o nome que o componente terá no campo Nome. O código-fonte desse elemento pode ser algo semelhante com o que se segue logo abaixo:

```
<table>
<tr>
<td nowrap>Data</td>
<td nowrap><input type="text" name="tmp.data.dia" size="2" value="|tmp.data.dia|">/
<input size="2" type="text" name="tmp.data.mes" value="|tmp.data.mes|">/
<input type="text" size="4" name="tmp.data.ano" value="|tmp.data.ano|"></td>
</tr>
</table>
```

```
<div>
  <label>|wiz[1].Titulo={label}| |$if(|wiz[2].Requerido={required}|,*)$|</label><br>
  <input type="text" name="|wiz.{name}|" value="|wiz.Valor={value}|"
        size="|wiz[3].Tamanho={size}|" maxlength="|wiz[4].Limite={maxlength}|">
</div>
|wiznewline|
```

Se o elemento for salvo com a opção **Global** marcada ele estará disponível para todos os projetos do WebIntegrator e no campo `Tipo` da janela de Propriedades seu nome virá precedido do símbolo **$**, caso contrário o elemento apenas estará disponível no projeto em que ele foi criado e no campo `Tipo` seu nome virá precedido do símbolo **#**.

O elemento acima ilustra como pode ser o código-fonte de um elemento porém ele ainda não oferece a oportunidade do desenvolvedor poder definir certos valores em tempo de design da página. Para acrescentar essa característica basta fazer referência a variáveis que irão fazer parte do ambiente do WIzard para que o elemento possa ser configurado de acordo com o que o desenvolvedor queira.

Para criar variáveis no ambiente do WIzard basta que no código-fonte do elemento o desenvolvedor referencie variáveis que comecem com o prefixo **`wiz.`**, o complemento do nome da variável servirá como o nome da propriedade que será exibida na janele de Propriedades do WIzard. A sintaxe a ser usada é a seguinte:

```
|wiz.<nome-da-propriedade>|
```

Como exemplo altere o código-fonte do elemento digitado acima para que o rótulo do campo possa ser digitado pelo desenvolvedor em tempo de design da página. O código-fonte pode ser o que segue abaixo:

```
<table>
<tr>
<td nowrap>|wiz.Rotulo|</td>
<td nowrap><input type="text" size="|wiz.TamCampoDia=2|" name="tmp.data.dia"
value="|tmp.data.dia|">/
<input type="text" size="|wiz.TamCampoMes=2|" name="tmp.data.mes"
value="|tmp.data.mes|">/
<input type="text" size="|wiz.TamCampoAno=4|" name="tmp.data.ano"
value="|tmp.data.ano|"></td>
</tr>
</table>
```

A alteração feita acima faz com que o elemento possua agora uma propriedade chamada **Rótulo** que estará disponível para ser configurada através da janela de propriedade do WIzard quando um desenvolvedor fizer uso desse elemento numa página WSP.

Durante a definição de uma propriedade o desenvolvedor também poderá definir algum valor padrão que elá irá assumir. Para isso basta que na definição da propriedade no código-fonte do elemento o desenvolvedor atribua esse valor usando a seguinte sintaxe:

```
|wiz.<nome-da-propriedade>=<valor-padrão>|
```

Para ilustar essa possibilidade altere novamente o código-fonte do elemento exemplificado aqui oferecendo agora a chance do desenvolvedor poder definir qual o tamanho que os campos de entrada de dados terão mas já tendo eles valores pré-configurados.

```
<table>
<tr>
<td nowrap>|wiz.Rotulo|</td>
<td nowrap><input type="text" size="|wiz.TamCampoDia=2|" name="tmp.data.dia"
value="|tmp.data.dia|">/
<input type="text" size="|wiz.TamCampoMes=2|" name="tmp.data.mes"
value="|tmp.data.mes|">/
<input type="text" size="|wiz.TamCampoAno=4|" name="tmp.data.ano"
value="|tmp.data.ano|"></td>
</tr>
</table>
```

O elemento representado pelo código-fonte acima possui agora quatro propriedades que podem ser configuradas pelo WIzard: **Rotulo**, **TamCampoDia**, **TamCampoMes** e **TamCampoAno**, sendo que as três últimas propriedades, respectivamente, já possuem valores pré-configurados.

A princípio as propriedades de um elemento quando são exibidas na janela de Propriedades do WIzard estão ordenadas alfabeticamente em ordem crescente de acordo com o nome da propriedade, mas o desenvolvedor poderá definir a ordem na qual as propriedades irão aparecer no WIzard bastando que após o prefixo **`wiz`** ele informe entre sinais de colchetes a sequência que a propriedade irá assumir. A sintaxe é a seguinte:

```
|wiz[<sequência>].<nome-da-propriedade>|
```

O código-fonte abaixo altera novamente o elemento para que se possa definir a sequência que as propriedades irão aparecer no WIzard.

```
<table>
<tr>
<td nowrap>|wiz[1].Rotulo|</td>
<td nowrap><input type="text" size="|wiz[2].TamCampoDia=2|" name="tmp.data.dia"
value="|tmp.data.dia|">/
<input type="text" size="|wiz[3].TamCampoMes=2|" name="tmp.data.mes"
value="|tmp.data.mes|">/
<input type="text" size="|wiz[4].TamCampoAno=4|" name="tmp.data.ano"
value="|tmp.data.ano|"></td>
</tr>
</table>
```

Uma outra possibilidade que pode ser feita para configurar qual valor uma determinada propriedade pode ter é o desenvolvedor poder especificar um determinado grupo de valores que ela poderá ter. Para isso basta que o desenvolvedor liste os possíveis valores da propriedade separados por vírgula definindo-os como valores-padrões dela. A sintaxe é a seguinte:

```
|wiz.<nome-da-propriedade>.combo=<valor#1>[,<valor#2>,...]|
```

Note a presença do sufixo `combo`, é ele que indica ao WIzard que essa propriedade possui um grupo de valores e que os mesmos serão exibidos numa combo-box limitando assim as opções de escolha que o desenvolvedor da página poderá ter. Caso queira o desenvolvedor também poderá indicar qual dos valores possíveis será o valor-padrão bastando que para isso delimite o valor entre os sinais de chaves. Como exemplo altere o código-fonte do elemento acrescentando a propriedade de poder alinhar verticalmente através dos valores `bottom`, `middle` ou `top`.

```
<table>
<tr>
<td nowrap style="vertical-align:
|wiz.AlinhamentoVertical.combo={bottom},middle,top|;">|wiz[1].Rotulo|</td>
<td nowrap><input type="text" size="|wiz[2].TamCampoDia=2|" name="tmp.data.dia"
value="|tmp.data.dia|">/
<input type="text" size="|wiz[3].TamCampoMes=2|" name="tmp.data.mes"
value="|tmp.data.mes|">/
<input type="text" size="|wiz[4].TamCampoAno=4|" name="tmp.data.ano"
value="|tmp.data.ano|"></td>
</tr>
</table>
```

O desenvolvedor ainda poderá atribuir a cada um dos valores da combo um texto a fim de torná-los mais explicativos, para isso coloca-se o texto que deverá estar associado seguido do valor entre os sinais de **[** e **]**. A funcionalidade dos sinais de chaves também vale aqui, ou seja, delimitando um par texto-valor entre chaves esse será o valor defaul do componente.

Alterando o código-fonte do elemento para que ao invés de exibir os valores `bottom`, `middle` e `top` sejam exibidos os textos Abaixo, Meio e Acima ficaria assim:

```
<table>
<tr>
<td nowrap style="vertical-align:
|wiz.AlinhamentoVertical.combo={Abaixo[bottom]},Meio[middle],Acima[top]|;">|wiz[1].Rotulo|</td
>
<td nowrap><input type="text" size="|wiz[2].TamCampoDia=2|" name="tmp.data.dia"
value="|tmp.data.dia|">/
<input type="text" size="|wiz[3].TamCampoMes=2|" name="tmp.data.mes"
value="|tmp.data.mes|">/
<input type="text" size="|wiz[4].TamCampoAno=4|" name="tmp.data.ano"
value="|tmp.data.ano|"></td>
</tr>
</table>
```

Além das combos de valores que podem ser definidas pelo próprio desenvolvedor também poderão ser feitas referencias às combos do próprio WIzard que listam as classes CSS e as funções JavaScript disponíveis para uma página.

Para referenciar a combo com a listagem das classes CSS disponíveis utilize a seguinte sintaxe:

```
|wiz.<nome-da-propriedade>.combo=wiz.CSSCombo|
```

Para fazer referência à combo que lista as funções JavaScript disponíveis para uma página use a sintaxe a seguie:

```
|wiz.<nome-da-propriedade>.combo=wiz.JSCombo|
```

Essas combos também suportam a funcionalidade de definição de um valor padrão já vir selecionado. Para isso basta colocar o valor que se deseje vir selecionado entre chaves logo após a referência a combo. A sintaxe ficaria assim:

```
|wiz.<nome-da-propriedade>.combo=wiz.CSSCombo|wizJSCombo[{valor-
                                padrão}]|
```

Além da possibilidade de referenciar as combos de CSS e funções JavaScript disponíveis pelo WIzard o desenvolvedor poderá referenciar as variáveis `wiz.{name}`, `wiz.{label}`, `wiz.{size}` e `wiz.{value}` para, respectivamente, recuperarem o nome que foi definido para esse componente na página, o rótulo que foi dado à coluna, o tamanho do campo da coluna e o valor padrão que essa coluna possui. As três últimas variáveis só terão utilidades com elementos do WIzard que estejam vinculados com algum tipo de dado de banco de dados pois os mesmos poderão ser usados para recuperar informações de uma coluna de uma tabela do banco.

Um outro recurso que pode ser adicionado aos componentes é ofercer ao desenvolvedor que irá utilizá-los uma dica (`hint`) do que cada uma das propriedades faz. Para definir o texto da dica das propriedades de um componente basta colocar dentro de um comentário HTML, no próprio código-fonte do componente, as propriedades com o texto da dica. Tomando como exemplo o componente aqui desenvolvido, a declaração dos textos das dicas ficaria assim:

```
<!--
|wiztitle.AlinhamentoVertical=Alinhamento vertical do campo.|
|wiztitle.Rotulo=O texto do rótulo.|
|wiztitle.TamCampoDia=Tamanho do campo dia.|
|wiztitle.TamCampoMes=Tamanho do campo mês.|
|wiztitle.TamCampoAno=Tamanho do campo ano.|
-->
<table>
<tr>
<td nowrap style="vertical-align:
|wiz.AlinhamentoVertical.combo={Abaixo[bottom]},Meio[middle],Acima[top]|;">|wiz[1].Rotulo|</td
>
<td nowrap><input type="text" size="|wiz[2].TamCampoDia=2|" name="tmp.data.dia"
value="|tmp.data.dia|">/
<input type="text" size="|wiz[3].TamCampoMes=2|" name="tmp.data.mes"
value="|tmp.data.mes|">/
<input type="text" size="|wiz[4].TamCampoAno=4|" name="tmp.data.ano"
value="|tmp.data.ano|"></td>
</tr>
</table>
```

Para desassociar um componente de um tipo de dados no banco o desenvolvedor deve selecionar o componente, com a tecla <ctrl> apertada desmarcar o tipo associado e gravar a alteração.

1.9.6 ...depurar uma aplicação no WebIntegrator usando a função wi.context()

...depurar uma aplicação no WebIntegrator usando a função wi.context()

Neste passo-a-passo será explicado como o desenvolvedor poderá usar a função `wi.context()` para realizar debugs nas variáveis das aplicações criadas usando o WebIntegrator. O que a função `wi.context()` faz é um *dump* no contexto do WebIntegrator

retornando a lista de variáveis e seus respectivos valores no momento em que ela é executada. A listagem das variáveis é retornada no seguinte padrão:

```
<nome-da-variável>=<valor-da-variável>
```

 **IMPORTANTE:**

Possa ser que algumas variáveis tenham em seu conteúdo quebras-de-linha, nestes casos a função `wi.context()` substituirá essas quebras pelos sinais de **`\r\n`**.

Por ser uma função, `wi.context()` tem que estar entre o sinal de `|$` e `$|` e o valor retornado deverá ser armazenado em uma variável. Um outro detalhe é que ela também suporta que seja passado um parâmetro indicando quais as variáveis a serem retornadas e se será feito um *dump* recursivo. Para indicar que um *dump* recursivo seja realizado usa-se o asterisco `(*)`. Veja alguns exemplos que seguem abaixo:

`|$wi.context()$|`	Retorna apenas as variáveis que apenas fazem parte do contexto raiz, ou seja, aquelas que cujo identificador não possue pontos.
`|$wi.context(*)$|`	Retorna todas as variáveis que fazem parte do contexto do WebIntegrator.
`|$wi.context(wi)$|`	Retorna apenas as variáveis que começam com wi.
`|$wi.context(wi.*)$|`	Retorna todas as variáveis que começam com o identificador **`wi`**.

Para demonstrarmos a funcionalidade prática dessa função vamos criar uma página de *debug* onde o usuário entra com o nome de uma variável ou uma máscara adequada para retornar o valor desejado. Esta página bastará ter apenas um formulário HTML com um campo-texto onde o usuário entre com o nome ou máscara da variável. No Pré-Página dessa página vamos colocar um elemento **Gravar** criando a variável **`tmp.contexto`** e em seu conteúdo gravaremos o resultado da chamada à função `|$wi.context()$|` passando como parâmetro um possível valor que foi digitado pelo usuário. Como código-fonte da página pode-se usar o modelo que segue abaixo:

```
<HTML>
<HEAD>
<TITLE>|wi.page.title|</TITLE>
</HEAD>

<BODY>
<form action="|wi.page.name|.wsp" method="post" >
<input type="text" name="tmp.var">
<input type="submit" name="tmp.btn" value="Debug">
</form>
<pre>
|tmp.context|
</pre>
</BODY>
</HTML>
```

No Pré-Página coloque um elemento **Gravar** com as seguintes definições:

Campo	Valor
Condição	`|tmp.btn| = debug`
Objetos	`tmp.context`
Se condição verdadeira	`|$wi.context(\|tmp.var\|)$|`

1.9.7 ...editar o layout de uma página WSP fora do WI_Builder

...editar o layout de uma página WSP fora do WI Builder

O **WI Builder** permite que o desenvolvedor edite o layout das páginas WSP de um projeto tanto pelo WIzard de páginas quanto pelo simples editor acessível nas tela de definição da própria página porém como fazer caso o desenvolvedor queira editar os conteúdos dessas páginas em editores de terceiros.

Para cada página criada pelo **WI Builder** será criada uma página JSP diretamente no diretório do projeto, um arquivo `.xml` com as definições da página em `<WEBAPP_PATH>/WEB-INF/definitions/pages` e um outro arquivo `.xml` representando o layout da página em `<WEBAPP_PATH>/WEB-INF/definitions/layouts`.

Nenhum dos três arquivos citados acima devem ser manipulados diretamente pelo desenvolvedor através de ferramentas de terceiros e fora do ambiente do **WI Builder**, principalmente as páginas JSP, pois as mesmas são geradas automaticamente pelo **WI Builder** baseadas nas definições dos layouts.

Para que você possa editar o layout de uma página WSP fora do ambiente do **WI Builder**

primeiramente você precisa acessar a tela de definição da página e em seguida clicar no ícone  . Na tela de edição, clique no botão  . Nesse momento será criado um arquivo **`.html`** em `<WEBAPP_PATH>/WEB-INF/definitions/layouts`.

Será editando esse arquivo `.html` que o desenvolvedor poderá estar editando o layout de uma página WSP. Vale notar que após as edições das páginas feitas por programas externos é necessário efetuar um recarregamento do projeto para que as correspondentes páginas JSP sejam geradas baseadas nos layouts.

1.9.8 ...funciona o WIzard de grids

...funciona o WIzard de grids

A idéia do WIzard de grids assim como do WIzard de páginas é facilitar o trabalho do desenvolvedor na criação e manutenção de modelos simples de grids utilizando o mesmo mecanismo dos componentes do WIzard. O WIzard de grids vê o modelo simples de um grid dividos em partes específicas e cada uma dessas partes recebe um nome especial definindo assim um estilo para cada parte do grid.

O modelo simples de um grid é formado por um código de uma tabela HTML, como exemplo vamos segue o modelo simples que segue abaixo.

```
<!-- Cabeçalho (início) -->
<table border="1">
<tr>
<!-- Cabeçalho (fim) -->

<!-- Célula do Cabeçalho (ínicio) -->
<th>Célula #1</th>
<!-- Célula do Cabeçalho (fim) -->

<!-- Detalhe (início) -->
</tr>
<tr>
<!-- Detalhe (fim) -->

<!-- Célula do Detalhe (início) -->
<td>|dado1|</td>
<!-- Célula do Detalhe (fim) -->

<!-- Sem Registros (início) -->
</tr>
<tr>
<td>Nenhum registro retornado</td>
</tr>
```

```
<!-- Sem Registros (fim) -->

<!-- Rodapé (início) -->
</table>
<!-- Rodapé (fim) -->
```

O modelo simples acima está definido para um grid onde a segunda linha da tabela será iteragida com os dados a serem populados. Os trechos que se encontram destacados são justamente as partes do estilo do grid a ser criado. Um estilo de grid é composto pelas várias partes que irão compor o estilo, e os nomes deverão ser no formato `grid.<nome-do-estilo>.<parte>`, onde `<parte>` deve ser: `head`, `headCell`, `detail`, `detailCell`, `noRegister` e `foot`.

Cada parte de um estilo de grid é definido semelhantemente a um componente do WIzard sendo que esses componentes também poderão ser de projeto ou globais havendo uma tela específica para a sua edição.

Na página de definição do grid há uma combo "Estilo", onde serão mostrados os estilos globais (prefixo **$**) e de projeto (prefixo #). Selecionando-se o estilo e marcando-se o checkbox "Gerar", o grid será gerado utilizando-se os componentes previamente definidos.

Uma vez gerado o grid, os componentes poderão facilmente ser editados. Vale lembrar que os parâmetros mostrados são aqueles definidos nos componentes no formato `|wiz.Nome=default|`, que poderão ser colocados de acordo com o grau de customização que se deseja dar ao componente.

1.9.9 ...montar um grid HTML do tipo Java

...montar um grid HTML do tipo Java

Neste passo-a-passo será explicado como o desenvolvedor poderá montar um grid HTML do tipo Java a fim de demonstrar a funcionalidade desse tipo de grid que pode ser empregado quando se quer exibir dados de uma forma tabular, ou seja, em linhas e colunas onde a fonte desses dados não é suportada nativamente pelos componentes grids disponíveis no WebIntegrator.

Para a montagem desse grid é necessário que o desenvolvedor tenha algum nível de conhecimento na linguagem de programação Java. Como exemplo vamos montar um grid que irá listar as propriedades do sistema onde a máquina virtual Java está sendo executada. Essa informação pode ser recuperada usando o método `getProperties()` da classe `System`.

A primeira coisa a ser feita é criarmos uma classe Java que implemente a interface `br.com.itx.integration.InterfaceGrid` colocando no método `execute()` a lógica que irá popular a estrutura de dados a ser passada para o **WI Engine** que se encarregará de montar o grid.

O código-fonte da classe segue logo abaixo com os comentários explicando a sua lógica:

```
import br.com.itx.integration.InterfaceGrid;
import java.util.Hashtable;
import java.util.HashMap;
import java.util.Properties;
import java.util.Enumeration;
import br.com.itx.util.Hash;
import br.com.itx.integration.DatabaseAliases;

public class TesteInterfaceGrid implements InterfaceGrid {
  public HashMap[] execute(Hash context, DatabaseAliases databases) {
    // recupera as informações do sistema
    Properties props = System.getProperties();

    // instancia a estrutura de dados que será usada pelo WI Engine
    // para a montagem do grid
    Hashtable[] ht = new Hashtable[props.size()];

    Enumeration enum = props.keys();
    for (int i = 0; enum.hasMoreElements(); i++) {
      // cada índice do array representa uma linha de dados que pode
      // ser exibida no grid
      ht[i] = new Hashtable();
      String key = (String) enum.nextElement();

      // as chaves das tabelas hashes correspondem às variáveis que
      // serão referenciadas no modelo do grid
      ht[i].put("chave", key);
      ht[i].put("valor", props.getProperty(key));
    }
    return ht;
  }

  public int returnType() {
    // sempre retornará todos os índices do array em caso de navegação
    // do grid
    return InterfaceGrid.COMPLETE;
  }
}
```

Sempre que um desenvolvedor quiser implementar o seu próprio grid ele terá de criar uma classe Java que implemente a *`interface`* `br.com.itx.integration.InterfaceGrid`. Esta *`interface`* é composta de dois métodos: o `execute(Hash, Databases)` que, como explicado logo acima, conterá a lógica da montagem do array de objetos `Hashtable;` e o método `returnType()` que indica, em caso de navegação do grid, como os índices do array serão retornados. Há três maneiras dos índices serem retornados:

COMPLETE	Retorna todos os elementos do array independentemente do índice inicial que esteja sendo usado para a exibição dos dados no grid.
HAS_MORE_ROWS	Retorna os elementos do array a partir do índice inicial que esteja sendo usado para a exibição dos dados no grid, ou seja, dependendo do tamanho do array, poderá ou não haver mais registros a serem exibidos.
NO_MORE_ROWS	Retorna apenas a quantidade de elementos a serem exibidas no grid, ou seja, caso o campo Quantidade do grid esteja definido com o valor 10 então serão retornado apenas 10 índices.

Compile a classe acima e coloque o arquivo compilado (arquivo com extensão .class) em um dos repositórios de classes que seja visível à sua aplicação web. Caso o desenvolvedor queira criar classes Java que façam parte de um pacote (usando a palavra-chave package), lembre-se que ao fazer o deployment dessas classes criar a estrutura de diretórios correspondente a estrutura de nomeação dos pacotes.

Após ter compilada a classe e feito o deployment vamos configurar a página. Primeiro defina um grid HTML do tipo Java (clique em Projeto, depois no ícone  e em seguida clique no link Grids HTML). Para definir um grid HTML do tipo Java basta apenas preencher o campo Identificador com o nome pelo qual referenciaremos o grid e escolher a opção JAVA no campo Tipo, neste exemplo os outros campos podem ser deixados em branco.

Depois de ter clicado no botão 

clique no link Editar para que possamos editar o modelo do nosso grid. Como um exemplo para o modelo veja o código-fonte abaixo:

```
<table border="1" cellspacing="0">
<tr>
<th>Chave</th>
<th>Valor</th>
</tr>
<!--- linha que faz referência às chaves das tabelas hashes
      montada pela classe TesteInterfaceGrid -->
<tr>
<td>|chave|</td>
<td>|valor| </td>
</tr>
</table>
```

Por fim crie uma página e no pré-página coloque um elemento Grid Java onde no campo Nome da classe preencha com o nome de uma classe que implementa a *interface* `br.com.itx.integration.InterfaceGrid` que no nosso caso é **TesteInterfaceGrid** e, no campo Grid, escolha o grid Java que corresponde a essa classe. No código-fonte da página faça referência ao grid digitando `|grid.<identificador>|` substituindo `<identificador>` pelo identificador correspondente ao grid.

1.9.10 ...montar uma barra de navegação de grid

...montar uma barra de navegação de grid

Neste passo-a-passo será explicado como o desenvolvedor poderá montar uma barra de navegação com botões que levem aos primeiros, próximo, anterior e últimos registro, semelhante às que são encontradas em componentes Delphi ou Visual Basic. Para montar essa barra de navegação serão feitas algumas personalizações em algumas propriedades que os grids possuem, para saber mais detalhes sobre as variáveis disponíveis para um grid acesse o tópico da Ajuda Variáveis do grid.

Para a montagem da barra de navegação de um grid precisaremos de:

- um grid (obviamente!) que esteja com o campo Quantidade definido com algum valor válido o suficiente para gerar uma navegação pelos registros retornados.
- imagens que serão usadas para a montagem da barra de navegação.

	Imagem para os últimos registros.
	Imagem para os primeiros registros.
	Imagem para os próximos registros.
	Imagem para os registros anteriores.

Como o exemplo dos códigos aqui exibidos fazem referência á `grid.<identificador>` lembre-se de substituir `<identificador>` pelo identificador correspondente ao grid para o qual você quer montar a barra de navegação, ou seja, caso você queira montar a barra de navegação para um grid cujo identificador seja `meuGrid` então substitua `<identificador>` por `meuGrid`.

Um grid do WebIntegrator oferece uma série de propriedades sendo que algumas dessas podem ter seus valores alterados, as que serão alteradas aqui para a montagem da barra de navegação serão:

`grid.<identificador>txtBack`	Esta propriedade tanto pode ser usada para exibir o texto do link que aponta para os registros anteriores quanto pode ser usada para que o desenvolvedor personalize este texto atribuindo algum valor a ela. Por default, o valor dessa propriedade é **Anterior**.
`grid.<identificador>txtGo`	Esta propriedade tanto pode ser usada para exibir o texto do link que aponta para os próximos registros quanto pode ser usada para que o desenvolvedor personalize este texto atribuindo algum valor a ela. Por default, o valor dessa propriedade é **Próximo**.
`grid.<identificador>txtFirst`	Esta propriedade tanto pode ser usada para exibir o texto do link que aponta para os primeiros registros quanto pode ser usada para que o desenvolvedor personalize este texto atribuindo algum valor a ela. Por default, o valor dessa propriedade é **Primeiro**.
`grid.<identificador>txtLast`	Esta propriedade tanto pode ser usada para exibir o texto do link que aponta para os últimos registros quanto pode ser usada para que o desenvolvedor personalize este texto atribuindo algum valor a ela. Por default, o valor dessa propriedade é **Último**.
`grid.<identificador>txtMid`	Esta propriedade tanto pode ser usada para exibir o que será exibido como separador dos links para os anteriores e os próximos registros quanto pode ser usada para que o desenvolvedor personalize o que será exibido atribuindo algum valor a ela. Por default, o valor dessa propriedade é **-**.

As propriedades `txtBack`, `txtGo`, `txtLast` e `txtFirst` são usadas internamente pelo **WI Engine** para a montagem das propriedades `linkBack`, `linkGo`, `linkLast` e `linkFirst`, respectivamente. As duas primeiras propriedades são usadas internamente para a montagem do link de navegação de um grid, que é retornado pela propriedade link. Já a propriedade `linkFull` retorna um link de navegação contendo todas as propriedades de navegação incluindo os links para os primeiros e últimos resultados.

Para começar a montagem da barra de navegação primeiramente copie as imagens que serão usadas para um diretório images do projeto. Na página que exibirá o grid adicione um elemento **Gravar** com a seguinte configuração:

Campo	Valor
Objetos	grid.<**identificador**>.txtGo, grid.<**identificador**>.txtBack, grid.<**identificador**>.txtMid, grid.<**identificador**>.txtFirst, grid.<**identificador**>.txtLast
Se condição verdadeira	<img src="/|wi.proj.id|/images/next.jpg" border="0" title="Próximo">,<img src="/|wi.proj.id|/images/previous.jpg" border="0" title="Anterior">, ,<img src="/|wi.proj.id|/images/first.jpg" border="0" title="Primeiros">,<img src="/|wi.proj.id|/images/last.jpg" border="0" title="Últimos"

IMPORTANTE:

Lembre-se de substituir as ocorrências de **`<identificador>`** pelo identificador correspondente ao grid que se esteja montando a barra de navegação.

Pronto, a nossa barra de navegação já está configurada! Agora basta fazer a referência na página ao grid (`|grid.<identificador>|`) e ao link de navegação (`|grid.<identificador>.linkFull|`). Para que a montagem do link de navegação sejá correto é necessário que a propriedade `rowCount` do grid retorne um valor válido e esse resultado é dependente do driver JDBC que estiver sendo utilizado.

1.9.11 ...registrar classes Java como plug-in

...registrar classes Java como plug-in

O WebIntegrator oferece uma série de componentes configuráveis que ajuda o desenvolvedor no momento da implementação das regras de negócio da aplicação, tais como execução de comandos de consulta e atualização em SGBD, envio de e-mail, download, etc. Além dessas funcionalidades é oferecido ao desenvolvedor a possibilidade de criar seus próprios componentes de regras de negócio e integrá-los com o WebIntegrator através do uso de conectores Java, grids Java ou funções.

Para facilitar o processo de integração e uso de componentes baseados em conectores Java, grids Java ou funções foi idealizado o conceito de plug-in. Um plug-in consiste de um arquivo **`.jar`** onde podem estar empacotados num mesmo arquivo: conectores Java, grids Java ou funções, que serão automaticamente cadastrados no WI Builder de acordo com as informações de configuração presentes no arquivo `plugins.xml` que, obrigatoriamente, também deverá estar presente no arquivo **`.jar`**.

Não é necessário fazer algum tipo de modificação no código-fonte dos conectores Java, grids Java ou funções já existentes para que os mesmos possam ser cadastrados como plug-in, basta empacotá-los em um arquivo **`.jar`** e prover um arquivo `plugins.xml`.

Para exemplificar o registro de um arquivo de plug-ins vamos assumir que possuímos as seguintes classes a serem publicadas:

Tipo	Nome da classe Java	
Função	**br.com.xpto.webintegrator.function.MyFunctionRandom**	
Conector Java	**br.com.xpto.webintegrator.connector.MyConnector**	
Conector Java de pós-página	**br.com.xpto.webintegrator.connector.MyPosConnector**	
Grid Java	**br.com.xpto.webintegrator.grid.MyJavaGrid**	

Primeiramente vamos cadastrar a classe `br.com.xpto.webintegrator.function.MyFunctionRandom` como sendo a função `myrandom`. Toda classe Java quando registrada como função do WebIntegrator possui um **`id`** pelo qual essa classe poderá ser referenciada num projeto do WebIntegrator. O arquivo `plugins.xml` para a definição dessa função ficaria assim:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<REGISTER>
  <NAMESPACE>xpto-plugins</NAMESPACE>
  <FUNCTIONS>
    <FUNCTION ID="myrandom"
CLASS="br.com.xpto.webintegrator.function.MyFunctionRandom"/>
  </FUNCTIONS>
</REGISTER>
```

Vejamos o que essas linhas querem dizer, a tag `REGISTER` é o nó-raiz do arquivo `plugins.xml`. A tag `NAMESPACE` indica o identificador sob o qual as classes Java cadastradas por esse plug-in ficaram associadas, cada arquivo **`.jar`** deve garantir que em seu arquivo `plugins.xml` a tag `NAMESPACE` esteja definida com um identificador único para aquele arquivo.

A tag `FUNCTIONS` agrupa o registro de cada uma das funções que devem ser feitos através da

tag FUNCTION que possui os atributos ID e CLASS que informam, respectivamente, qual o **id** da função e qual a classe Java que está associada àquele **id**. Para fazer o registro de mais uma função bastaria definir uma outra tag FUNCTION dentro da tag FUNCTIONS. Todos os atributos da tag FUNCTION são de preenchimento obrigatório.

Uma característica interessante dos plug-ins é que os conectores ou grids Java registrados como tal aparecem listados como componentes de pré-página ou pós-página nas telas do WI Builder. O registro de um conector Java é feito através da tag PLUGIN onde são informados o título do conector, a classe Java, os parâmetros de entrada e os parâmetros de saída. Para o registro do conector `br.com.xpto.webintegrator.connector.MyConnector` o arquivo `plugins.xml` ficaria assim:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<REGISTER>
  <NAMESPACE>xpto-plugins</NAMESPACE>
  <FUNCTIONS>
    <FUNCTION ID="myrandom"
CLASS="br.com.xpto.webintegrator.function.MyFunctionRandom"/>
  </FUNCTIONS>
  <PLUGINS>
    <PLUGIN PID="MyConnector">
      <TITLE>Meu Conector</TITLE>
      <CLASS>br.com.xpto.webintegrator.connector.MyConnector</CLASS>
      <TYPE>CONNECTOR</TYPE>
      <PARAMETERS>
        <IN>
          <PARAMETER ID="tmp.param.in">
            <DESCRIPTION>Parâmetro de Entrada</DESCRIPTION>
            <HINT>[DICA] Parâmetro de entrada do conector Meu
Conector</HINT>
            <VALUE>|tmp.param.in|</VALUE>
          </PARAMETER>
        </IN>
        <OUT>
          <PARAMETER ID="tmp.param.out">
            <DESCRIPTION>Parâmetro de saída do conector Meu
Conector</DESCRIPTION>
          </PARAMETER>
        </OUT>
      </PARAMETERS>
    </PLUGIN>
  </PLUGINS>
</REGISTER>
```

Alguns detalhes da configuração acima merecem ser destacados. Primeiramente toda classe Java configurada como plug-in deve possuir um **id** único dentro do seu *namespace* que deve ser configurado através do atributo PID da tag PLUGIN. A tag TITLE contém o texto que será exibido na listagem de componentes do WI Builder que está associado a esse conector. A tag CLASS indica a classe Java do conector. A tag TYPE serve para indicar o tipo de plug-in que se está cadastrando, neste caso um conector, sendo que os valores possíveis são CONNECTOR, para indicar o registro de uma clase Java que seja um conector, e JAVAGRID para indicar o registro de uma classe Java que seja um grid Java. Os possíveis parâmetros de entrada e de saída que um plug-in venha a ter é feito através da tag PARAMETERS por meio das tags IN e OUT respectivamente.

O conector Java acima foi cadastrado de tal maneira que ele poderá ser utilizado tanto no pré-página quanto no pós-página, se existir algum conector a ser cadastrado que possua a restrição

de apenas poder ser executado ou no pré-página ou no pós-página o desenvolvedor poderá utilizar a tag USEDIN indicando em qual evento de página ele será executado. Se a execução apenas puder ocorrer em um pré-página o conteúdo da tag dever ser PRE, se apenas puder ser executado em um pós-página o conteúdo deve ser POS.

A configuração do arquivo `plugins.xml` que segue abaixo inclui o cadastro do conector `br.com.xpto.webintegrator.connector.MyPosConnector` que apenas pode ser executado num pós-página.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<REGISTER>
  <NAMESPACE>xpto-plugins</NAMESPACE>
  <FUNCTIONS>
    <FUNCTION ID="myrandom"
CLASS="br.com.xpto.webintegrator.function.MyFunctionRandom"/>
  </FUNCTIONS>
  <PLUGINS>
    <PLUGIN PID="MyConnector">
      <TITLE>Meu Conector</TITLE>
      <CLASS>br.com.xpto.webintegrator.connector.MyConnector</CLASS>
      <TYPE>CONNECTOR</TYPE>
      <PARAMETERS>
        <IN>
          <PARAMETER ID="tmp.param.in">
            <DESCRIPTION>Parâmetro de Entrada</DESCRIPTION>
            <HINT>[DICA] Parâmetro de entrada do conector Meu
Conector</HINT>
            <VALUE>|tmp.param.in|</VALUE>
          </PARAMETER>
        </IN>
        <OUT>
          <PARAMETER ID="tmp.param.out">
            <DESCRIPTION>Parâmetro de saída do conector Meu
Conector</DESCRIPTION>
          </PARAMETER>
        </OUT>
      </PARAMETERS>
    </PLUGIN>
    <PLUGIN PID="MyPosConnector">
      <TITLE>Meu Conector de Pós-Página</TITLE>
      <CLASS>br.com.xpto.webintegrator.connector.MyPosConnector</CLASS>
      <TYPE>CONNECTOR</TYPE>
      <USEDIN>POS</USEDIN>
      <PARAMETERS>
        <IN>
          <PARAMETER ID="tmp.param.in">
            <DESCRIPTION>Parâmetro de Entrada</DESCRIPTION>
            <HINT>[DICA] Parâmetro de entrada do conector Meu Conector
de Pós-Página</HINT>
            <VALUE>|tmp.param.in|</VALUE>
          </PARAMETER>
        </IN>
        <OUT>
          <PARAMETER ID="tmp.param.out">
            <DESCRIPTION>Parâmetro de saída do conector Meu Conector de
Pós-Página</DESCRIPTION>
          </PARAMETER>
        </OUT>
```

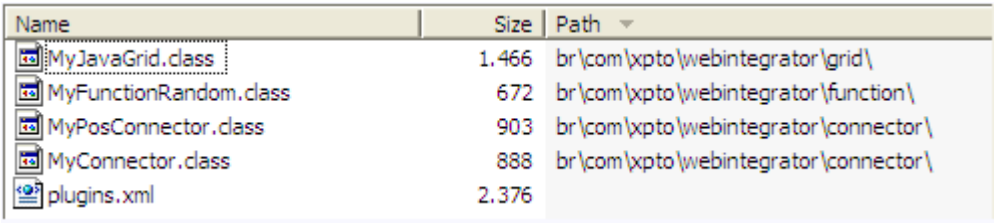
```
</PARAMETERS>
    </PLUGIN>
  </PLUGINS>
</REGISTER>
```

O registro de classes Java que representem um grid Java é semelhante ao registro de conectores Java com a particularidade que o valor da tag `TYPE` deve ser `JAVAGRID`. O outro detalhe é que como todo componente Grid Java apenas é executado em eventos de pré-página o desenvolvedor não precisará definir a tag `USEDIN` com o valor `PRE` pois isso é feito automaticmanete pelo sistema registrador de plug-ins. Logo abaixo segue o exemplo de como ficaria o nosso arquivo `plugins.xml` ao inserirmos a definição do grid Java `br.com.xpto.webintegrator.grid.MyJavaGrid`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<REGISTER>
  <NAMESPACE>xpto-plugins</NAMESPACE>
  <FUNCTIONS>
    <FUNCTION ID="myrandom"
CLASS="br.com.xpto.webintegrator.function.MyFunctionRandom"/>
  </FUNCTIONS>
  <PLUGINS>
    <PLUGIN PID="MyConnector">
      <TITLE>Meu Conector</TITLE>
      <CLASS>br.com.xpto.webintegrator.connector.MyConnector</CLASS>
      <TYPE>CONNECTOR</TYPE>
      <PARAMETERS>
        <IN>
          <PARAMETER ID="tmp.param.in">
            <DESCRIPTION>Parâmetro de Entrada</DESCRIPTION>
            <HINT>[DICA] Parâmetro de entrada do conector Meu
Conector</HINT>
            <VALUE>|tmp.param.in|</VALUE>
          </PARAMETER>
        </IN>
        <OUT>
          <PARAMETER ID="tmp.param.out">
            <DESCRIPTION>Parâmetro de saída do conector Meu
Conector</DESCRIPTION>
          </PARAMETER>
        </OUT>
      </PARAMETERS>
    </PLUGIN>
    <PLUGIN PID="MyPosConnector">
      <TITLE>Meu Conector de Pós-Página</TITLE>
      <CLASS>br.com.xpto.webintegrator.connector.MyPosConnector</CLASS>
      <TYPE>CONNECTOR</TYPE>
      <USEDIN>POS</USEDIN>
      <PARAMETERS>
        <IN>
          <PARAMETER ID="tmp.param.in">
            <DESCRIPTION>Parâmetro de Entrada</DESCRIPTION>
            <HINT>[DICA] Parâmetro de entrada do conector Meu Conector
de Pós-Página</HINT>
            <VALUE>|tmp.param.in|</VALUE>
          </PARAMETER>
        </IN>
        <OUT>
          <PARAMETER ID="tmp.param.out">
```

```
<DESCRIPTION>Parâmetro de saída do conector Meu Conector de Pós-
Página</DESCRIPTION>
          </PARAMETER>
        </OUT>
      </PARAMETERS>
    </PLUGIN>
    <PLUGIN PID="MyJavaGrid">
      <TITLE>Meu Grid Java</TITLE>
      <CLASS>br.com.xpto.webintegrator.grid.MyJavaGrid</CLASS>
      <TYPE>JAVAGRID</TYPE>
      <PARAMETERS>
        <IN>
          <PARAMETER ID="tmp.param.in">
            <DESCRIPTION>Parâmetro de Entrada</DESCRIPTION>
            <HINT>[DICA] Parâmetro de entrada do grid Java Meu Grid
Java</HINT>
            <VALUE>|tmp.param.in|</VALUE>
          </PARAMETER>
        </IN>
        <OUT>
          <PARAMETER ID="tmp.param.error">
            <DESCRIPTION>Parâmetro de saída do grid Java Meu Grid
Java</DESCRIPTION>
          </PARAMETER>
        </OUT>
      </PARAMETERS>
    </PLUGIN>
  </PLUGINS>
</REGISTER>
```

Para fazer a publicação dessas classes como plug-ins empacote-as em um arquivo com extensão `.jar` juntamente com o arquivo `plugins.xml` sendo que esse arquivo deverá estar na raiz do arquivo JAR. O conteúdo do arquivo JAR para o nosso estudo de caso seria o que está sendo exibido abaixo:



Name	Size	Path
MyJavaGrid.class	1.466	br\com\xpto\webintegrator\grid\
MyFunctionRandom.class	672	br\com\xpto\webintegrator\function\
MyPosConnector.class	903	br\com\xpto\webintegrator\connector\
MyConnector.class	888	br\com\xpto\webintegrator\connector\
plugins.xml	2.376	

No **WI Builder** acesse as opções do menu Opções e em seguida clique no ícone de cadastro de plug-ins identificado pela imagem  . No campo Arquivo JAR indique o caminho do arquivo JAR que representa os plug-ins a serem instalados, a opção Global quando marcada indica que os plug-ins a serem instalados estarão disponíveis para todos os projetos que forem desenvolvidos pelo WI Builder caso você queira limitar a disponibilidade dos plug-ins a apenas alguns dos projetos disponíveis deverá ser a instalação especificamente em cada um dos projetos.

Após a instalação ou remoção de um arquivo de plug-ins é altamente recomendado que a(s) aplicação(ões) web a ele relacionada(s) seja(m) reiniciada(s).

1.9.12 ...usar o SingleSignOn em projetos

...usar o SingleSignOn em projetos

O SingleSignOn é um recurso que permite compartilhar o login entre projetos. Com esse recurso o desenvolvedor pode criar um projeto para ser o que efetivamente realiza o login e os demais projetos podem se utilizar das variáveis exportadas por ele sem necessitar que o usuário da aplicação tenha que ficar se logando para cada projeto.

Para facilitar o entendimento vamos identificar o projeto que faz o login como *projeto pai* enquanto os demais projetos como *projeto filho.*

Antes de começar a efetiva adequação dos projetos para utilizar o SingleSignOn é necessário copiar no common/lib do tomcat a biblioteca *wi-singlesignonrepository.jar* que encontra-se dentro do WEB-INF/lib dos projetos, removê-las nos projetos envolvidos e reiniciar o tomcat.

Passos para utilizar o SingleSignOn:

- No *projeto pai* deve ser colocado no pré-página da página principal (aquela que entra depois de logado) o elemento Atribuir SingleSignOn, e nele o desenvolvedor pode informar quais variáveis deseja exportar para os projetos filhos. (Ex: pvt.login., pvt.nome, etc). As variáveis tanto podem ser específicas como tmp.nome e grupos como pvt.login. e separadas por vírgula.

- Nos *projetos filhos* o login deve estar apenas ativado e com a página de login informada. A página de login do projeto filho deve ter o conteúdo abaixo para remeter para a página de login do projeto pai.

```
<script>
location="/[projeto_pai]/[pagina_login].wsp?tmp.gotoproj=|wi.proj.prev|&tmp.gotopage=|wi.page.prev|";
</script>
```

Pronto, feito esses passos passa a existir uma interligação entre o login do projeto pai e o dos projetos filhos.

1.9.13 ...configurar o login para lembrar a página que foi chamada

...configurar o login para lembrar a página que foi chamada

O login padrão de um projeto WI sempre entra na mesma página principal informada num projeto, mas o desenvolvedor pode configurar o projeto para lembrar a página que foi chamada e ao logar entrar nela.

No pré-página da pagina login criar um gravar em tmp.gotoproj, tmp.gotopage se |tmp.gotopage| = vazio o valor |wi.proj.prev|, |wi.page.prev|. Se está sendo usado SingleSignOn isso só deve ser feito no pré-página do projeto que efetivamente faz o login.

No formulário de login criar um elemento oculto (hidden) tmp.gotoproj com o valor |tmp.gotoproj| e tmp.gotopage com o valor |tmp.gotopage|.

No pré-página da página principal (aquela que entra depois de logado) deve ser colocado um desvio caso |tmp.gotopage| != vazio para |wi.server.url|/|tmp.gotoproj|/|tmp.gotopage|.wsp. Se está sendo usado SingleSignOn isso deve ser feito após o Atribuir SingleSignOn.

1.9.14 ...fazer busca em mensagens usando Imap

...fazer busca em mensagens usando Imap

O protocolo Imap permite que o desenvolvedor faça busca nas mensagens de uma determinada pasta utilizando critérios pré-estabelecidos.

Para facilitar a geração de um grid a partir de um uma busca o WI conta com a funcionalidade

Para gerar um grid contendo o resultado de uma busca numa pasta IMAP de um servidor pode ser utilizada a opção Java de Banco de Dados e no comando sql coloca-se `br.com.itx.database.impl.ResultSetImapSearch:[id_servidor],[pasta],[titulo],[corpo],[de]` onde os parâmetros tanto podem estar fixos como ser variável entre pipes.

id_servidor = o id do servidor a ser utilizado.
pasta = a pasta onde a busca deve ser feita.
titulo = o texto completo que deve conter no título (não é obrigatório).
corpo = o texto completo que deve conter no corpo da mensagem (não é obrigatório).
de = o email do remetente (não é obrigatório).

OBS: Uma busca pode ser testada no Explorar BD e caso seja necessário passar as variáveis do usuário e senha da conexão pode se colocar como se fosse parâmetros adicionais da classe, pois a classe irá desconsiderá-las e o Explorar BD irá permitir entrar uma valor para os campos.

1.9.15 ...usar o wi.token

...usar o wi.token

É comum algumas aplicações web requererem que se controle ou que se evite que o usuário possa utilizar alguns recursos do browser como voltar ou atualizar as páginas, podendo causar o processamento dos dados de uma mesma página repetidas vezes. Como exemplo: suponha uma aplicação web que onde o usuário acessa uma página que realiza uma operação de débito em uma conta-corrente bancária, e que não seja oferecido um controle transacional sobre esta página, se o usuário pedir através do browser para que essa página seja atualizada, uma nova solicitação de débito será realizada sem que necessariamente o usuário tenha pedido isso conscientemente.

Para oferecer um controle transacional sobre esses tipos de ação o WebIntegrator gera um **token** (identificador) que servirá como um validador de página. Automaticamente, o **WI Engine** gera para cada página WSP solicitada, um token de transação único identificado na variável `wi.token`, este token será usado para garantir e reforçar uma única requisição para uma transação em particular.

Para que o controle seja feito basta referenciar essa variável (`wi.token`) na página que terá sua transação controlada. Quando os dados da página são enviados para o servidor, o valor do token também deverá ser enviado pela página como conteúdo de uma variável chamada **`wi.token`**, isso pode ser feito criando um campo no formulário do tipo `HIDDEN` cujo atributo `NAME` seja `wi.token` e o atributo VALUE referencie `|wi.token|`. No lado servidor o valor da variável `wi.token` é comparado com o token que está armazenado na sessão do usuário, caso os valores coincidam, é atribuída à variável `wi.token.ok` o valor **`true`**, se os valores não coincidirem a variável `wi.token.ok` terá seu valor igual a **`false`**.

As páginas criadas pelo WIzard já referenciam automaticamente a variável `wi.token`, portanto estando elas prontas para terem um controle transacional. Uma maneira para implementar esse controle é colocar um evento no pós-página da página que será controlada para que faça um desvio para uma outra página se o token não for válido ou, em termos de condição, se `|wi.token.ok| != true`.

1.9.16 ...utilizar a Renderização Parcial

...utilizar a Renderização Parcial

A Renderização Parcial permite que um evento javascript dispare a atualização de apenas uma dada área na tela.

Exemplo de formulário:

```
<form name="myform" method="post">
<input type="text" name="tmp.ini" value="|tmp.ini|"
onkeyup="rerenderSubmit(myform,'grid_pessoas')">
<div id="grid_pessoas">
|grid.pessoas|
</div>
</form>
```

Funções javascript disponibilizadas:

```
rerenderSubmit(form, rerender, action, doPos, msg)
Envia um formulário e atualiza uma área
- form é o objeto formulário a ser enviado
- rerender são as áreas a serem atualizadas separadas por vírgula
- action é um valor que deve ser atribuído ao campo de id tmp.action
- doPos indica se o pós página da página deve ser executado
- msg serve para ser colocada uma mensagem de confirmação antes de
efetuar a operação

OBS: o campo action exige que o formulário tenha <input type="hidden"
name="tmp.action">

rerenderLink(link, rerender, action, doPos, msg)
Acessa um link e atualiza uma área
- link é o objeto link a ser acessado
- rerender são as áreas a serem atualizadas separadas por vírgula
- action é um valor que deve ser atribuído ao campo de id tmp.action
- doPos indica se o pós página da página deve ser executado
- msg serve para ser colocada uma mensagem de confirmação antes de
efetuar a operação

A área rerenderIndicator pose ser usada para colocar um icone animado
ajax ou uma mensagem de aguarde.
Ex:
<div id="rerenderIndicator" style="display: none; position:absolute;
top:0px; right:0px">
<img src="/|wi.proj.id|/ajax.gif">
</div>
```

1.9.17 ...fazer o debug da lógica de um pré ou pós página

...fazer o debug da lógica de um pré ou pós página

É comum em algumas situações ser necessário fazer uma análise detalhada da lógica que um pré ou pós página está executando vendo como se encontrava o contexto de variáveis antes e depois de sua execução. Para realizar essa tarefa o desenvolvedor pode gravar uma variável pvt.debug.core ou tmp.debug.core com o valor *true* que irá gerar o debug no diretório de logs do projeto. Caso o desenvolvedor não deseje que todas as variáveis sejam exportadas ele pode usar pvt.core.filter para definir o que deve ser exportado.

Quando uma requisição é feita ao WI e existe a variável pvt.debug.core com valor true é então criada uma pasta descrita abaixo:
debug-[hora:minuto:segundo da requisição]-[id da sessão]

Dentro da pasta haverá um arquivo conforme descrito abaixo para cada elemento de lógica do pré ou pós página processado:
element-[hora:minuto:segundo da execução do elemento]-[tipo do elemento de lógica]_[1 ou 2].txt
onde o txt terminado por 1 informa as variáveis antes da lógica ser executada e 2 depois de ser executada.

Dentro do arquivo txt pode ser visto como título a descrição do componente de lógica do WI e o conteúdo de todas as variáveis do contexto.

1.9.18 ...incluir o registro de taglibs externas

...incluir o registro de taglibs externas

Caso o desenvolvedor deseje fazer uso de uma taglib que não pertence nativamente ao WI ele pode configurar para o builder ao gerar os jps já inclua automaticamente essa taglib. Para isso basta criar um arquivo chamado taglibs.txt no WEB-INF do projeto com as linhas que devem ser incluidas automaticamente.

Exemplo do conteudo do arquivo taglibs.txt:

```
<%@ taglib prefix='cache' uri='http://jakarta.apache.org/taglibs/cache-1.0'%>
```

```
<%@ taglib prefix='bsf' uri='http://jakarta.apache.org/taglibs/bsf-
1.0'%>
```

OBS: O arquivo jar correspondente a taglib a ser utilizada precisa ser colocado no WEB-INF/lib do projeto.

1.9.19 ...utilizar o Apache Tiles

...utilizar o Apache Tiles

O Apache Tiles é uma API que permite que o desenvolvedor crie páginas modularizadas a partir da composição de partes pré-definidas. Além de facilitar o particionamento do layout do projeto, o Tiles tem amplo suporte para criação de portais e de perspectivas, isto é, o usuário final definir a forma que deseja diagramar o site (ex: menu superior ou laterar em função da escolha do usuário).

Layout em tempo de execução

Para facilitar o entendimento monte o exemplo abaixo.

Para maiores detalhes consulte a documentação no site:
http://tiles.apache.org/2.1/framework/index.html

OBS: Todas as páginas com exceção da que será chamada por url pode estar com segurança ativa ou até mesmo como página de sistema que é o ideal.

Crie uma página tiles_frame1.wsp com o código abaixo:

```
<html>
 <head>
  <title><tiles:getAsString name="title"/></title>
 </head>
 <body>
  <table width="100%" height="100%">
    <tr>
     <td bgcolor="yellow">
      <tiles:insertAttribute name="menu"/>
     </td>
    </tr>
    <tr>
     <td bgcolor="gray">
```

```
<tiles:insertAttribute name="body"/>
      </td>
     </tr>
   </table>
 </body>
</html>
```

Crie uma página tiles_frame2.wsp com o código abaixo:

```
<html>
 <head>
  <title><tiles:getAsString name="title"/></title>
 </head>
 <body>
  <table width="100%" height="100%">
    <tr>
     <td bgcolor="green">
      <tiles:insertAttribute name="menu"/>
     </td>
     <td bgcolor="gray">
      <tiles:insertAttribute name="body"/>
     </td>
    </tr>
  </table>
 </body>
</html>
```

Crie uma página tiles_modelo.wsp com o código abaixo:

```
<wi:set var="template" value="/tiles_frame1.wsp"/>
<wi:set var="template" value="/tiles_frame2.wsp" test="|tmp.modelo|=2"/>
<tiles:insertTemplate template="${template}">
 <tiles:putAttribute name="title" value="WSP Tiles"/>
 <tiles:putAttribute name="menu" value="/menu.wsp"/>
</tiles:insertTemplate>
```

Crie uma página menu.wsp com o código abaixo:

```
<p>menu</p>
```

Crie uma página teste.wsp com o código abaixo:

```
<tiles:insertTemplate template="/tiles_modelo.wsp">
 <tiles:putAttribute name="body" cascade="true">
  corpo da página
 </tiles:putAttribute>
</tiles:insertTemplate>
```

Agora é só executar:
http://servidor:porta/projeto/teste.wsp
http://servidor:porta/projeto/teste.wsp?tmp.modelo=2

Layout definido em arquivo XML

Caso o desenvolvedor deseje a montagem do modelo pode estar definida no arquivo tiles.xml dentro da pasta WEB-INF do projeto conforme abaixo:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
       "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">
<tiles-definitions>
  <definition name="modelo" template="/tiles_frame1.wsp">
    <put-attribute name="title" value="XML Tiles"/>
    <put-attribute name="menu" value="/menu.wsp"/>
  </definition>
</tiles-definitions>
```

OBS: Qualquer alteração nesse arquivo necessita a recarga da aplicação web por isso essa forma não é recomendada.

Quando se usa definição em XML a página teste.wsp deve ser criada da forma a seguir:

```
<tiles:insertDefinition name="modelo">
  <tiles:putAttribute name="body">
    corpinho da página
  </tiles:putAttribute>
</tiles:insertDefinition>
```

1.10 Dicas e Truques

Dicas e Truques

1.10.1 Verificando se uma página está em modo de design do WIzard

Verificando se uma página está em modo de design do WIzard

Podem aparecer situações em que seja necessário distinguir se uma página WSP está sendo

exibida no modo de design do WIzard ou se está sendo executada diretamente pelo **WI Engine**. Tendo essa necessidade você poderá fazer uso da variável designMode que se encontra disponível em qualquer página WSP que tenha sido criada pelo WIzard.

Como um exemplo bem simples acesse uma página WSP através do WIzard, acrescente um elemento do tipo Genérico e em seu campo Texto coloque o seguinte trecho de código:

```
<script>
msg = (window.designMode ? "Esta página está sendo exibida pelo WIzard!"
: "Esta página está sendo exibida pelo WI_Engine!");
alert(msg);
</script>
```

Ao ser exibida pelo WIzard a página irá mostrar a mensagem "`Esta página está sendo exibida pelo WIzard!`" e quando for montada pelo **WI Engine**, ou seja, quando for chamada diretamente por uma URL será mostrada a mesagem "`Esta página está sendo exibida pelo WI Engine!`".

1.10.2 Executando uma ação ao carregar uma página WSP

Executando uma ação ao carregar uma página WSP

Se você quiser definir uma função JavaScript no atributo `ONLOAD` da tag `BODY` de uma página WSP gerada pelo WIzard insiram um elemento do tipo Genérico na página WSP e na propriedade Texto desse elemento coloque o seguinte trecho de código:

```
<script>
window.onload = function() {init(); <seu-código>};
</script>
```

A parte indicada por **`<seu-código>`** pode ser substituída por qualquer código JavaScript inclusive chamada a outras funções. A função `init()` é uma função usada pelas páginas geradas pelo WIzard para uma pré-formatação dos campos de um formulário então recomendamos que ela seja a primeira instrução a ser executada antes de qualquer código.

1.11 Links

Links

- Site oficial do WebIntegrator
- Site oficial do Apache Tomcat
- Site oficial da Tecnologia Java